Pyomo Documentation

Release 5.1

Pyomo

Nov 21, 2017

Contents

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

CHAPTER 1

Getting Started

Installation, pyomo command, a simple example, etc

```
>>> print('Hello World')
Hello World
```

CHAPTER 2

Tutorial

2.1 Overview

Pyomo includes a diverse set of optimization capabilities for formulating and analyzing optimization models. Pyomo supports the formulation and analysis of mathematical models for complex optimization applications. This capability is commonly associated with algebraic modeling languages (AMLs), which support the description and analysis of mathematical models with a high-level language. Although most AMLs are implemented in custom modeling languages, Pyomo's modeling objects are embedded within Python, a full-featured high-level programming language that contains a rich set of supporting libraries.

Pyomo has also proven an effective framework for developing high-level optimization and analysis tools. It is easy to develop Python scripts that use Pyomo as a part of a complex analysis workflow. Additionally, Pyomo includes a variety of optimization solvers for stochastic programming, dynamic optimization with differential algebraic equations, mathematical programming with equilibrium conditions, and more! Increasingly, Pyomo is integrating functionality that is normally associated with an optimization solver library.

2.2 Concrete vs Abstract Models

2.3 Modeling Components

2.4 Pyomo Command

```
>>> print('Hello World')
Hello World
```

CHAPTER 3

Core Pyomo Components

Detailed component descriptions...

```
>>> print('Hello World')
Hello World
```

CHAPTER 4

Scripting

Scripting examples...

```
>>> print('Hello World')
Hello World
```

CHAPTER 5

Modeling Extensions

5.1 Bilevel Programming

TODO

```
>>> print('Hello World')
Hello World
```

5.2 Dynamic Optimization with pyomo.DAE

The pyomo.DAE modeling extension allows users to incorporate systems of differential algebraic equations (DAE)s in a Pyomo model. The modeling components in this extension are able to represent ordinary or partial differential equations. The differential equations do not have to be written in a particular format and the components are flexible enough to represent higher-order derivatives or mixed partial derivatives. Pyomo.DAE also includes model transformations which use simultaneous discretization approaches to transform a DAE model into an algebraic model. Finally, pyomo.DAE includes utilities for simulating DAE models and initializing dynamic optimization problems.

5.2.1 Modeling Components

Pyomo.DAE introduces three new modeling components to Pyomo:

pyomo.dae.ContinuousSet	Represents a bounded continuous domain
pyomo.dae.DerivativeVar	Represents derivatives in a model and defines how a

Continued on next page

Table 5.1 – continued from previous page

pyomo.dae.Integral	Represents an integral over a continuous domain

As will be shown later, differential equations can be declared using using these new modeling components along with the standard Pyomo `Var` and `Constraint` components.

ContinuousSet

This component is used to define continuous bounded domains (for example 'spatial' or 'time' domains). It is similar to a Pyomo `Set` component and can be used to index things like variables and constraints. Any number of *ContinuousSets* can be used to index a component and components can be indexed by both `Sets` and *ContinuousSets* in arbitrary order.

In the current implementation, models with *ContinuousSet* components may not be solved until every *ContinuousSet* has been discretized. Minimally, a *ContinuousSet* must be initialized with two numeric values representing the upper and lower bounds of the continuous domain. A user may also specify additional points in the domain to be used as finite element points in the discretization.

class pyomo.dae.**ContinuousSet**(**args*, ***kwds*)

Represents a bounded continuous domain

Minimally, this set must contain two numeric values defining the bounds of a continuous range. Discrete points of interest may be added to the continuous set. A continuous set is one dimensional and may only contain numerical values.

Parameters

- **initialize** (*list*) – Default discretization points to be included
- **bounds** (*tuple*) – The bounding points for the continuous domain. The bounds will be included as discrete points in the `ContinuousSet` but will not be used to restrict points added to the `ContinuousSet` through the 'initialize' argument, a data file, or the add() method

_changed

boolean – This keeps track of whether or not the ContinuousSet was changed during discretization. If the user specifies all of the needed discretization points before the discretization then there is no need to go back through the model and reconstruct things indexed by the `ContinuousSet`

_fe

list – This is a sorted list of the finite element points in the `ContinuousSet`. i.e. this list contains all the discrete points in the `ContinuousSet` that are not collocation points. Points that are both finite element points and collocation points will be included in this list.

_discretization_info

dict – This is a dictionary which contains information on the discretization transformation which has been applied to the `ContinuousSet`.

construct(*values=None*)

Constructs a `ContinuousSet` component

get_changed()

Returns flag indicating if the `ContinuousSet` was changed during discretization

Returns "True" if additional points were added to the *ContinuousSet* while applying a discretization scheme

Returns

Return type *boolean*

get_discretization_info()
Returns a *dict* with information on the discretization scheme that has been applied to the ContinuousSet.

Returns

Return type *dict*

get_finite_elements()
Returns the finite element points

If the *ContinuousSet* has been discretizaed using a collocation scheme, this method will return a list of the finite element discretization points but not the collocation points within each finite element. If the *ContinuousSet* has not been discretized or a finite difference discretization was used, this method returns a list of all the discretization points in the *ContinuousSet*.

Returns

Return type *list* of *floats*

get_lower_element_boundary(*point*)
Returns the first finite element point that is less than or equal to 'point'

Parameters **point** (*float*) –

Returns

Return type float

get_upper_element_boundary(*point*)
Returns the first finite element point that is greater or equal to 'point'

Parameters **point** (*float*) –

Returns

Return type float

set_changed(*newvalue*)
Sets the _changed flag to 'newvalue'

Parameters **newvalue** (*boolean*) –

The following code snippet shows examples of declaring a *ContinuousSet* component on a concrete Pyomo model:

```
Required imports
>>> from pyomo.environ import *
>>> from pyomo.dae import *

>>> model = ConcreteModel()

Declaration by providing bounds
>>> model.t = ContinuousSet(bounds=(0,5))

Declaration by initializing with desired discretization points
>>> model.x = ContinuousSet(initialize=[0,1,2,5])
```

Note: A *ContinuousSet* may not be constructed unless at least two numeric points are provided to bound the continuous domain.

The following code snippet shows an example of declaring a *ContinuousSet* component on an abstract Pyomo model using the example data file.

```
set t := 0 0.5 2.25 3.75 5;
```

```
Required imports
>>> from pyomo.environ import *
>>> from pyomo.dae import *

>>> model = AbstractModel()

The ContinuousSet below will be initialized using the points
in the data file when a model instance is created.
>>> model.t = ContinuousSet()
```

Note: If a separate data file is used to initialize a *ContinuousSet*, it is done using the 'set' command and not 'continuousset'

Note: Most valid ways to declare and initialize a `Set` can be used to declare and initialize a *ContinuousSet*. See the documentation for `Set` for additional options.

Warning: Be careful using a *ContinuousSet* as an implicit index in an expression, i.e. `sum(m.v[i] for i in m.myContinuousSet)`. The expression will be generated using the discretization points contained in the *ContinuousSet* at the time the expression was constructed and will not be updated if additional points are added to the set during discretization.

Note: *ContinuousSet* components are always ordered (sorted) therefore the `first()` and `last()` `Set` methods can be used to access the lower and upper boundaries of the *ContinuousSet* respectively

DerivativeVar

class pyomo.dae.**DerivativeVar**(*sVar*, ***kwds*)

Represents derivatives in a model and defines how a `Var` is differentiated

The *DerivativeVar* component is used to declare a derivative of a `Var`. The constructor accepts a single positional argument which is the `Var` that's being differentiated. A `Var` may only be differentiated with respect to a *ContinuousSet* that it is indexed by. The indexing sets of a *DerivativeVar* are identical to those of the `Var` it is differentiating.

Parameters

- **sVar** (pyomo.environ.Var) – The variable being differentiated
- **wrt** (pyomo.dae.ContinuousSet or tuple) – Equivalent to *withrespectto* keyword argument. The *ContinuousSet* that the derivative is being taken with respect to. Higher order derivatives are represented by including the *ContinuousSet* multiple times in the tuple sent to this keyword. i.e. `wrt=(m.t, m.t)` would be the second order derivative with respect to `m.t`

get_continuousset_list()

Return the a list of `ContinuousSet` components the derivative is being taken with respect to.

Returns

Return type *list*

get_derivative_expression()
Returns the current discretization expression for this derivative or creates an access function to its `Var` the first time this method is called. The expression gets built up as the discretization transformations are sequentially applied to each `ContinuousSet` in the model.

get_state_var()
Return the `Var` that is being differentiated.

Returns

Return type `Var`

is_fully_discretized()
Check to see if all the *ContinuousSets* this derivative is taken with respect to have been discretized.

Returns

Return type *boolean*

set_derivative_expression(*expr*)
Sets''_expr'', an expression representing the discretization equations linking the `DerivativeVar` to its state `Var`

The code snippet below shows examples of declaring *DerivativeVar* components on a Pyomo model. In each case, the variable being differentiated is supplied as the only positional argument and the type of derivative is specified using the 'wrt' (or the more verbose 'withrespectto') keyword argument. Any keyword argument that is valid for a Pyomo `Var` component may also be specified.

```
Required imports
>>> from pyomo.environ import *
>>> from pyomo.dae import *

>>> model = ConcreteModel()
>>> model.s = Set(initialize=['a','b'])
>>> model.t = ContinuousSet(bounds=(0,5))
>>> model.l = ContinuousSet(bounds=(-10,10))

>>> model.x = Var(model.t)
>>> model.y = Var(model.s,model.t)
>>> model.z = Var(model.t,model.l)

Declare the first derivative of model.x with respect to model.t
>>> model.dxdt = DerivativeVar(model.x, withrespectto=model.t)

Declare the second derivative of model.y with respect to model.t
Note that this DerivativeVar will be indexed by both model.s and model.t
>>> model.dydt2 = DerivativeVar(model.y, wrt=(model.t,model.t))

Declare the partial derivative of model.z with respect to model.l
Note that this DerivativeVar will be indexed by both model.t and model.l
>>> model.dzdl = DerivativeVar(model.z, wrt=(model.l), initialize=0)

Declare the mixed second order partial derivative of model.z with respect
to model.t and model.l and set bounds
>>> model.dz2 = DerivativeVar(model.z, wrt=(model.t, model.l), bounds=(-10, 10))
```

Note: The 'initialize' keyword argument will initialize the value of a derivative and is **not** the same as specifying an initial condition. Initial or boundary conditions should be specified using a `Constraint` or `ConstraintList` or by fixing the value of a `Var` at a boundary point.

5.2.2 Declaring Differential Equations

A differential equations is declared as a standard Pyomo `Constraint` and is not required to have any particular form. The following code snippet shows how one might declare an ordinary or partial differential equation.

```
Required imports
>>> from pyomo.environ import *
>>> from pyomo.dae import *

>>> model = ConcreteModel()
>>> model.s = Set(initialize=['a', 'b'])
>>> model.t = ContinuousSet(bounds=(0, 5))
>>> model.l = ContinuousSet(bounds=(-10, 10))

>>> model.x = Var(model.s, model.t)
>>> model.y = Var(model.t, model.l)
>>> model.dxdt = DerivativeVar(model.x, wrt=model.t)
>>> model.dydt = DerivativeVar(model.y, wrt=model.t)
>>> model.dydl2 = DerivativeVar(model.y, wrt=(model.l, model.l))

An ordinary differential equation
>>> def _ode_rule(m, s, t):
...     if t == 0:
...         return Constraint.Skip
...     return m.dxdt[s, t] == m.x[s, t]**2
>>> model.ode = Constraint(model.s, model.t, rule=_ode_rule)

A partial differential equation
>>> def _pde_rule(m, t, l):
...     if t == 0 or l == m.l.first() or l == m.l.last():
...         return Constraint.Skip
...     return m.dydt[t, l] == m.dydl2[t, l]
>>> model.pde = Constraint(model.t, model.l, rule=_pde_rule)
```

By default, a `Constraint` declared over a *`ContinuousSet`* will be applied at every discretization point contained in the set. Often a modeler does not want to enforce a differential equation at one or both boundaries of a continuous domain. This may be addressed explicitly in the `Constraint` declaration using `Constraint.Skip` as shown above. Alternatively, the desired constraints can be deactivated just before the model is sent to a solver as shown below.

```
>>> def _ode_rule(m, s, t):
...     return m.dxdt[s, t] == m.x[s, t]**2
>>> model.ode = Constraint(model.s, model.t, rule=_ode_rule)

>>> def _pde_rule(m, t, l):
...     return m.dydt[t, l] == m.dydl2[t, l]
>>> model.pde = Constraint(model.t, model.l, rule=_pde_rule)

Declare other model components and apply a discretization transformation
...
```

```
Deactivate the differential equations at certain boundary points
>>> for con in model.ode[:, model.t.first()]:
...     con.deactivate()

>>> for con in model.pde[0, :]:
...     con.deactivate()

>>> for con in model.pde[:, model.l.first()]:
...     con.deactivate()

>>> for con in model.pde[:, model.l.last()]:
...   con.deactivate()

Solve the model
...
```

Note: If you intend to use the pyomo.DAE *Simulator* on your model then you **must** use **constraint deactivation** instead of **constraint skipping** in the differential equation rule.

5.2.3 Declaring Integrals

Warning: The *Integral* component is still under development and considered a prototype. It currently includes only basic functionality for simple integrals. We welcome feedback on the interface and functionality but **we do not recommend using it** on general models. Instead, integrals should be reformulated as differential equations.

class pyomo.dae.**Integral**(**args*, ***kwds*)

Represents an integral over a continuous domain

The *Integral* component can be used to represent an integral taken over the entire domain of a *ContinuousSet*. Once every *ContinuousSet* in a model has been discretized, any integrals in the model will be converted to algebraic equations using the trapezoid rule. Future development will include more sophisticated numerical integration methods.

Parameters

- ***args** – Every indexing set needed to evaluate the integral expression
- **wrt** (*ContinuousSet*) – The continuous domain over which the integral is being taken
- **rule** (*function*) – Function returning the expression being integrated

get_differentialset()

Return the *ContinuousSet* the integral is being taken over

Declaring an *Integral* component is similar to declaring an Expression component. A simple example is shown below:

```
>>> model = ConcreteModel()
>>> model.time = ContinuousSet(bounds=(0,10))
>>> model.X = Var(model.time)
>>> model.scale = Param(initialize=1E-3)

>>> def _intX(m,t):
...     return m.X[t]
```

```
>>> model.intX = Integral(model.time,wrt=model.time,rule=_intX)

>>> def _obj(m):
...     return m.scale*m.intX
>>> model.obj = Objective(rule=_obj)
```

Notice that the positional arguments supplied to the *Integral* declaration must include all indices needed to evaluate the integral expression. The integral expression is defined in a function and supplied to the 'rule' keyword argument. Finally, a user must specify a *ContinuousSet* that the integral is being evaluated over. This is done using the 'wrt' keyword argument.

Note: The *ContinuousSet* specified using the 'wrt' keyword argument must be explicitly specified as one of the indexing sets (meaning it must be supplied as a positional argument). This is to ensure consistency in the ordering and dimension of the indexing sets

---

After an *Integral* has been declared, it can be used just like a Pyomo `Expression` component and can be included in constraints or the objective function as shown above.

If an *Integral* is specified with multiple positional arguments, i.e. multiple indexing sets, the final component will be indexed by all of those sets except for the *ContinuousSet* that the integral was taken over. In other words, the *ContinuousSet* specified with the 'wrt' keyword argument is removed from the indexing sets of the *Integral* even though it must be specified as a positional argument. This should become more clear with the following example showing a double integral over the *ContinuousSet* components `model.t1` and `model.t2`. In addition, the expression is also indexed by the `Set` `model.s`. The mathematical representation and implementation in Pyomo are shown below:

$$\sum_{s} \int_{t_2} \int_{t_1} X(t_1, t_2, s)\, dt_1\, dt_2$$

```
>>> model = ConcreteModel()
>>> model.t1 = ContinuousSet(bounds=(0, 10))
>>> model.t2 = ContinuousSet(bounds=(-1, 1))
>>> model.s = Set(initialize=['A', 'B', 'C'])

>>> model.X = Var(model.t1, model.t2, model.s)

>>> def _intX1(m, t1, t2, s):
...     return m.X[t1, t2, s]
>>> model.intX1 = Integral(model.t1, model.t2, model.s, wrt=model.t1,
...                        rule=_intX1)

>>> def _intX2(m, t2, s):
...     return m.intX1[t2, s]
>>> model.intX2 = Integral(model.t2, model.s, wrt=model.t2, rule=_intX2)

>>> def _obj(m):
...     return sum(m.intX2[k] for k in m.s)
>>> model.obj = Objective(rule=_obj)
```

5.2.4 Discretization Transformations

Before a Pyomo model with *DerivativeVar* or *Integral* components can be sent to a solver it must first be sent through a discretization transformation. These transformations approximate any derivatives or integrals in

the model by using a numerical method. The numerical methods currently included in pyomo.DAE discretize the continuous domains in the problem and introduce equality constraints which approximate the derivatives and integrals at the discretization points. Two families of discretization schemes have been implemented in pyomo.DAE, Finite Difference and Collocation. These schemes are described in more detail below.

Note: The schemes described here are for derivatives only. All integrals will be transformed using the trapezoid rule.

The user must write a Python script in order to use these discretizations, they have not been tested on the pyomo command line. Example scripts are shown below for each of the discretization schemes. The transformations are applied to Pyomo model objects which can be further manipulated before being sent to a solver. Examples of this are also shown below.

Finite Difference Transformation

This transformation includes implementations of several finite difference methods. For example, the Backward Difference method (also called Implicit or Backward Euler) has been implemented. The discretization equations for this method are shown below:

$$
\begin{aligned}
&\text{Given :}\\
&\tfrac{dx}{dt} = f(t,x), \quad x(t_0) = x_0\\
&\text{discretize } t \text{ and } x \text{ such that}\\
&x(t_0 + kh) = x_k\\
&x_{k+1} = x_k + h * f(t_{k+1}, x_{k+1})\\
&t_{k+1} = t_k + h
\end{aligned}
$$

where h is the step size between discretization points or the size of each finite element. These equations are generated automatically as `Constraints` when the backward difference method is applied to a Pyomo model.

There are several discretization options available to a `dae.finite_difference` transformation which can be specified as keyword arguments to the `.apply_to()` function of the transformation object. These keywords are summarized below:

Keyword arguments for applying a finite difference transformation:

'nfe' The desired number of finite element points to be included in the discretization. The default value is 10.

'wrt' Indicates which *`ContinuousSet`* the transformation should be applied to. If this keyword argument is not specified then the same scheme will be applied to every *`ContinuousSet`* .

'scheme' Indicates which finite difference method to apply. Options are 'BACKWARD', 'CENTRAL', or 'FORWARD'. The default scheme is the backward difference method.

If the existing number of finite element points in a *`ContinuousSet`* is less than the desired number, new discretization points will be added to the set. If a user specifies a number of finite element points which is less than the number of points already included in the *`ContinuousSet`* then the transformation will ignore the specified number and proceed with the larger set of points. Discretization points will never be removed from a *`ContinousSet`* during the discretization.

The following code is a Python script applying the backward difference method. The code also shows how to add a constraint to a discretized model.

```
Discretize model using Backward Difference method
>>> discretizer = TransformationFactory('dae.finite_difference')
>>> discretizer.apply_to(model,nfe=20,wrt=model.time,scheme='BACKWARD')

Add another constraint to discretized model
>>> def _sum_limit(m):
```

```
...     return sum(m.x1[i] for i in m.time) <= 50
>>> model.con_sum_limit = Constraint(rule=_sum_limit)

Solve discretized model
>>> solver = SolverFactory('ipopt')
>>> results = solver.solve(model)
```

Collocation Transformation

This transformation uses orthogonal collocation to discretize the differential equations in the model. Currently, two types of collocation have been implemented. They both use Lagrange polynomials with either Gauss-Radau roots or Gauss-Legendre roots. For more information on orthogonal collocation and the discretization equations associated with this method please see chapter 10 of the book "Nonlinear Programming: Concepts, Algorithms, and Applications to Chemical Processes" by L.T. Biegler.

The discretization options available to a `dae.collocation` transformation are the same as those described above for the finite difference transformation with different available schemes and the addition of the 'ncp' option.

Additional keyword arguments for collocation discretizations:

'scheme' The desired collocation scheme, either 'LAGRANGE-RADAU' or 'LAGRANGE-LEGENDRE'. The default is 'LAGRANGE-RADAU'.

'ncp' The number of collocation points within each finite element. The default value is 3.

Note: If the user's version of Python has access to the package Numpy then any number of collocation points may be specified, otherwise the maximum number is 10.

Note: Any points that exist in a *ContinuousSet* before discretization will be used as finite element boundaries and not as collocation points. The locations of the collocation points cannot be specified by the user, they must be generated by the transformation.

The following code is a Python script applying collocation with Lagrange polynomials and Radau roots. The code also shows how to add an objective function to a discretized model.

```
Discretize model using Radau Collocation
>>> discretizer = TransformationFactory('dae.collocation')
>>> discretizer.apply_to(model,nfe=20,ncp=6,scheme='LAGRANGE-RADAU')

Add objective function after model has been discretized
>>> def obj_rule(m):
...     return sum((m.x[i]-m.x_ref)**2 for i in m.time)
>>> model.obj = Objective(rule=obj_rule)

Solve discretized model
>>> solver = SolverFactory('ipopt')
>>> results = solver.solve(model)
```

Restricting Optimal Control Profiles

When solving an optimal control problem a user may want to restrict the number of degrees of freedom for the control input by forcing, for example, a piecewise constant profile. Pyomo.DAE provides the

`reduce_collocation_points` function to address this use-case. This function is used in conjunction with the `dae.collocation` discretization transformation to reduce the number of free collocation points within a finite element for a particular variable.

class `pyomo.dae.plugins.colloc.`**Collocation_Discretization_Transformation**

reduce_collocation_points (*instance*, *var=None*, *ncp=None*, *contset=None*)

This method will add additional constraints to a model to reduce the number of free collocation points (degrees of freedom) for a particular variable.

Parameters

- **instance** (*Pyomo model*) – The discretized Pyomo model to add constraints to
- **var** (`pyomo.environ.Var`) – The Pyomo variable for which the degrees of freedom will be reduced
- **ncp** (*int*) – The new number of free collocation points for *var*. Must be less that the number of collocation points used in discretizing the model.
- **contset** (`pyomo.dae.ContinuousSet`) – The *ContinuousSet* that was discretized and for which the *var* will have a reduced number of degrees of freedom

An example of using this function is shown below:

```
>>> discretizer = TransformationFactory('dae.collocation')
>>> discretizer.apply_to(model, nfe=10, ncp=6)
>>> model = discretizer.reduce_collocation_points(model,
...                                               var=model.u,
...                                               ncp=1,
...                                               contset=model.time)
```

In the above example, the `reduce_collocation_points` function restricts the variable `model.u` to have only **1** free collocation point per finite element, thereby enforcing a piecewise constant profile. Fig. 5.1 shows the solution profile before and after appling the `reduce_collocation_points` function.

Applying Multiple Discretization Transformations

Discretizations can be applied independently to each *ContinuousSet* in a model. This allows the user great flexibility in discretizing their model. For example the same numerical method can be applied with different resolutions:

```
>>> discretizer = TransformationFactory('dae.finite_difference')
>>> discretizer.apply_to(model,wrt=model.t1,nfe=10)
>>> discretizer.apply_to(model,wrt=model.t2,nfe=100)
```

This also allows the user to combine different methods. For example, applying the forward difference method to one *ContinuousSet* and the central finite difference method to another *ContinuousSet*:

```
>>> discretizer = TransformationFactory('dae.finite_difference')
>>> discretizer.apply_to(model,wrt=model.t1,scheme='FORWARD')
>>> discretizer.apply_to(model,wrt=model.t2,scheme='CENTRAL')
```

In addition, the user may combine finite difference and collocation discretizations. For example:

```
>>> disc_fe = TransformationFactory('dae.finite_difference')
>>> disc_fe.apply_to(model,wrt=model.t1,nfe=10)
>>> disc_col = TransformationFactory('dae.collocation')
>>> disc_col.apply_to(model,wrt=model.t2,nfe=10,ncp=5)
```

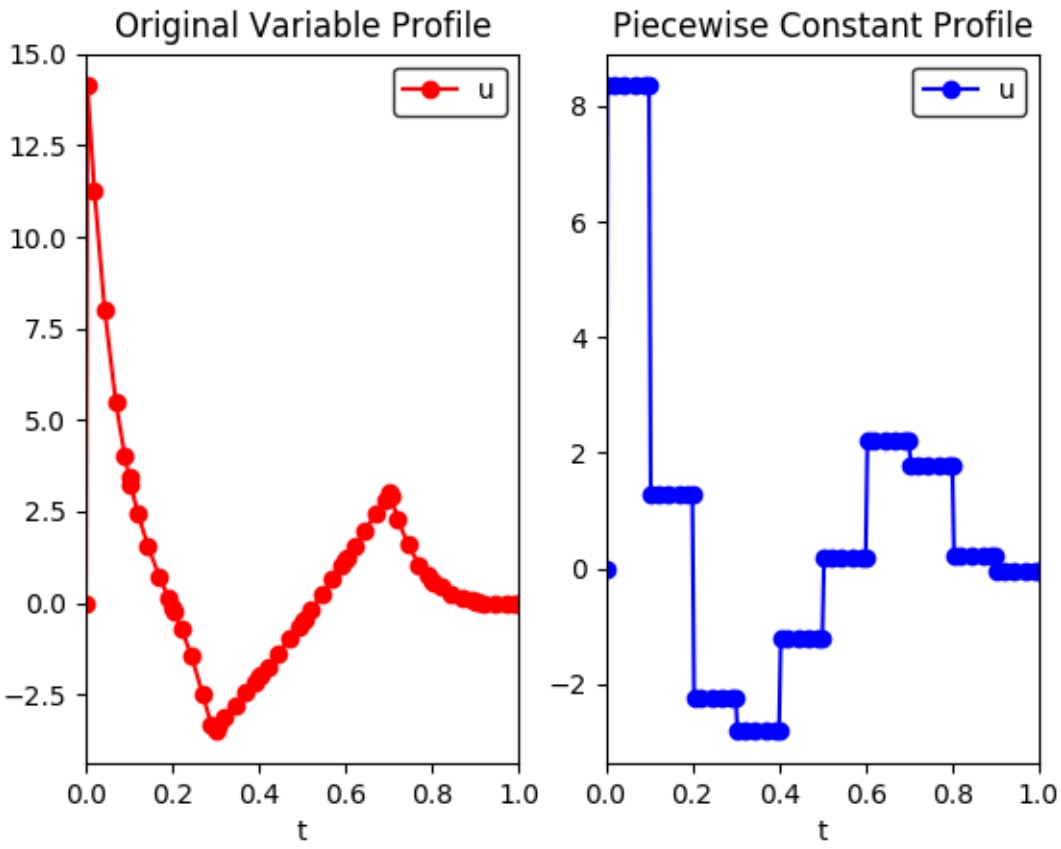

Fig. 5.1: (left) Profile before applying the `reduce_collocation_points` function (right) Profile after applying the function, restricting `model.u` to have a piecewise constant profile.

If the user would like to apply the same discretization to all *ContinuousSet* components in a model, just specify the discretization once without the 'wrt' keyword argument. This will apply that scheme to all *ContinuousSet* components in the model that haven't already been discretized.

Custom Discretization Schemes

A transformation framework along with certain utility functions has been created so that advanced users may easily implement custom discretization schemes other than those listed above. The transformation framework consists of the following steps:

1. Specify Discretization Options
2. Discretize the ContinuousSet(s)
3. Update Model Components
4. Add Discretization Equations
5. Return Discretized Model

If a user would like to create a custom finite difference scheme then they only have to worry about step (4) in the framework. The discretization equations for a particular scheme have been isolated from of the rest of the code for implementing the transformation. The function containing these discretization equations can be found at the top of the source code file for the transformation. For example, below is the function for the forward difference method:

```
def _forward_transform(v,s):
"""
Applies the Forward Difference formula of order O(h) for first derivatives
"""
    def _fwd_fun(i):
        tmp = sorted(s)
        idx = tmp.index(i)
        return 1/(tmp[idx+1]-tmp[idx])*(v(tmp[idx+1])-v(tmp[idx]))
    return _fwd_fun
```

In this function, 'v' represents the continuous variable or function that the method is being applied to. 's' represents the set of discrete points in the continuous domain. In order to implement a custom finite difference method, a user would have to copy the above function and just replace the equation next to the first return statement with their method.

After implementing a custom finite difference method using the above function template, the only other change that must be made is to add the custom method to the 'all_schemes' dictionary in the `dae.finite_difference` class.

In the case of a custom collocation method, changes will have to be made in steps (2) and (4) of the transformation framework. In addition to implementing the discretization equations, the user would also have to ensure that the desired collocation points are added to the ContinuousSet being discretized.

5.2.5 Dynamic Model Simulation

The pyomo.dae Simulator class can be used to simulate systems of ODEs and DAEs. It provides an interface to integrators available in other Python packages.

Note: The pyomo.dae Simulator does not include integrators directly. The user must have at least one of the supported Python packages installed in order to use this class.

---

class `pyomo.dae.`**`Simulator`**(*m*, *package='scipy'*)

Simulator objects allow a user to simulate a dynamic model formulated using pyomo.dae.

Parameters

- **m** (*Pyomo Model*) – The Pyomo model to be simulated should be passed as the first argument
- **package** (*string*) – The Python simulator package to use. Currently 'scipy' and 'casadi' are the only supported packages

get_variable_order(*vartype=None*)

This function returns the ordered list of differential variable names. The order corresponds to the order being sent to the integrator function. Knowing the order allows users to provide initial conditions for the differential equations using a list or map the profiles returned by the simulate function to the Pyomo variables.

Parameters **vartype** (*string* or None) – Optional argument for specifying the type of variables to return the order for. The default behavior is to return the order of the differential variables. 'time-varying' will return the order of all the time-dependent algebraic variables identified in the model. 'algebraic' will return the order of algebraic variables used in the most recent call to the simulate function. 'input' will return the order of the time-dependent algebraic variables that were treated as inputs in the most recent call to the simulate function.

Returns

Return type *list*

initialize_model()

This function will initialize the model using the profile obtained from simulating the dynamic model.

simulate(*numpoints=None*, *tstep=None*, *integrator=None*, *varying_inputs=None*, *initcon=None*, *integrator_options=None*)

Simulate the model. Integrator-specific options may be specified as keyword arguments and will be passed on to the integrator.

Parameters

- **numpoints** (*int*) – The number of points for the profiles returned by the simulator. Default is 100
- **tstep** (*int or float*) – The time step to use in the profiles returned by the simulator. This is not the time step used internally by the integrators. This is an optional parameter that may be specified in place of 'numpoints'.
- **integrator** (*string*) – The string name of the integrator to use for simulation. The default is 'lsoda' when using Scipy and 'idas' when using CasADi
- **varying_inputs** (pyomo.environ.Suffix) – A Suffix object containing the piecewise constant profiles to be used for certain time-varying algebraic variables.
- **initcon** (*list of floats*) – The initial conditions for the the differential variables. This is an optional argument. If not specified then the simulator will use the current value of the differential variables at the lower bound of the ContinuousSet for the initial condition.
- **integrator_options** (*dict*) – Dictionary containing options that should be passed to the integrator. See the documentation for a specific integrator for a list of valid options.

Returns The first return value is a 1D array of time points corresponding to the second return value which is a 2D array of the profiles for the simulated differential and algebraic variables.

Return type numpy array, numpy array

Note: Any keyword options supported by the integrator may be specified as keyword options to the simulate function and will be passed to the integrator.

Supported Simulator Packages

The Simulator currently includes interfaces to SciPy and CasADi. ODE simulation is supported in both packages however, DAE simulation is only supported by CasADi. A list of available integrators for each package is given below. Please refer to the SciPy and CasADi documentation directly for the most up-to-date information about these packages and for more information about the various integrators and options.

SciPy Integrators:

- **'vode'** : Real-valued Variable-coefficient ODE solver, options for non-stiff and stiff systems
- **'zvode'** : Complex-values Variable-coefficient ODE solver, options for non-stiff and stiff systems
- **'lsoda'** : Real-values Variable-coefficient ODE solver, automatic switching of algorithms for non-stiff or stiff systems
- **'dopri5'** : Explicit runge-kutta method of order (4)5 ODE solver
- **'dop853'** : Explicit runge-kutta method of order 8(5,3) ODE solver

CasADi Integrators:

- **'cvodes'** : CVodes from the Sundials suite, solver for stiff or non-stiff ODE systems
- **'idas'** : IDAS from the Sundials suite, DAE solver
- **'collocation'** : Fixed-step implicit runge-kutta method, ODE/DAE solver
- **'rk'** : Fixed-step explicit runge-kutta method, ODE solver

Using the Simulator

We now show how to use the Simulator to simulate the following system of ODEs:

$$\frac{d\theta}{dt} = \omega$$
$$\frac{d\omega}{dt} = -b * \omega - c * sin(\theta)$$

We begin by formulating the model using pyomo.DAE

```
>>> m = ConcreteModel()

>>> m.t = ContinuousSet(bounds=(0.0, 10.0))

>>> m.b = Param(initialize=0.25)
>>> m.c = Param(initialize=5.0)

>>> m.omega = Var(m.t)
>>> m.theta = Var(m.t)

>>> m.domegadt = DerivativeVar(m.omega, wrt=m.t)
>>> m.dthetadt = DerivativeVar(m.theta, wrt=m.t)

Setting the initial conditions
>>> m.omega[0].fix(0.0)
>>> m.theta[0].fix(3.14 - 0.1)
```

```
>>> def _diffeq1(m, t):
...     return m.domegadt[t] == -m.b * m.omega[t] - m.c * sin(m.theta[t])
>>> m.diffeq1 = Constraint(m.t, rule=_diffeq1)

>>> def _diffeq2(m, t):
...     return m.dthetadt[t] == m.omega[t]
>>> m.diffeq2 = Constraint(m.t, rule=_diffeq2)
```

Notice that the initial conditions are set by *fixing* the values of `m.omega` and `m.theta` at t=0 instead of being specified as extra equality constraints. Also notice that the differential equations are specified without using `Constraint.Skip` to skip enforcement at t=0. The Simulator cannot simulate any constraints that contain if-statements in their construction rules.

To simulate the model you must first create a Simulator object. Building this object prepares the Pyomo model for simulation with a particular Python package and performs several checks on the model to ensure compatibility with the Simulator. Be sure to read through the list of limitations at the end of this section to understand the types of models supported by the Simulator.

```
>>> sim = Simulator(m, package='scipy')
```

After creating a Simulator object, the model can be simulated by calling the simulate function. Please see the API documentation for the *Simulator* for more information about the valid keyword arguments for this function.

```
>>> tsim, profiles = sim.simulate(numpoints=100, integrator='vode')
```

The `simulate` function returns numpy arrays containing time points and the corresponding values for the dynamic variable profiles.

***Simulator Limitations*:**

- Differential equations must be first-order and separable
- Model can only contain a single ContinuousSet
- Can't simulate constraints with if-statements in the construction rules
- Need to provide initial conditions for dynamic states by setting the value or using fix()

Specifying Time-Varing Inputs

The *Simulator* supports simulation of a system of ODE's or DAE's with time-varying parameters or control inputs. Time-varying inputs can be specified using a Pyomo `Suffix`. We currently only support piecewise constant profiles. For more complex inputs defined by a continuous function of time we recommend adding an algebraic variable and constraint to your model.

The profile for a time-varying input should be specified using a Python dictionary where the keys correspond to the switching times and the values correspond to the value of the input at a time point. A `Suffix` is then used to associate this dictionary with the appropriate `Var` or `Param` and pass the information to the *Simulator*. The code snippet below shows an example.

```
>>> m = ConcreteModel()

>>> m.t = ContinuousSet(bounds=(0.0, 20.0))

Time-varying inputs
>>> m.b = Var(m.t)
>>> m.c = Param(m.t, default=5.0)
```

```
>>> m.omega = Var(m.t)
>>> m.theta = Var(m.t)

>>> m.domegadt = DerivativeVar(m.omega, wrt=m.t)
>>> m.dthetadt = DerivativeVar(m.theta, wrt=m.t)

Setting the initial conditions
>>> m.omega[0] = 0.0
>>> m.theta[0] = 3.14 - 0.1

>>> def _diffeq1(m, t):
...     return m.domegadt[t] == -m.b[t] * m.omega[t] - \
...                             m.c[t] * sin(m.theta[t])
>>> m.diffeq1 = Constraint(m.t, rule=_diffeq1)

>>> def _diffeq2(m, t):
...     return m.dthetadt[t] == m.omega[t]
>>> m.diffeq2 = Constraint(m.t, rule=_diffeq2)

Specifying the piecewise constant inputs
>>> b_profile = {0: 0.25, 15: 0.025}
>>> c_profile = {0: 5.0, 7: 50}

Declaring a Pyomo Suffix to pass the time-varying inputs to the Simulator
>>> m.var_input = Suffix(direction=Suffix.LOCAL)
>>> m.var_input[m.b] = b_profile
>>> m.var_input[m.c] = c_profile

Simulate the model using scipy
>>> sim = Simulator(m, package='scipy')
>>> tsim, profiles = sim.simulate(numpoints=100,
...                               integrator='vode',
...                               varying_inputs=m.var_input)
```

Note: The Simulator does not support multi-indexed inputs (i.e. if `m.b` in the above example was indexed by another set besides `m.t`)

5.2.6 Dynamic Model Initialization

Providing a good initial guess is an important factor in solving dynamic optimization problems. There are several model initialization tools under development in pyomo.DAE to help users initialize their models. These tools will be documented here as they become available.

From Simulation

The *Simulator* includes a function for initializing discretized dynamic optimization models using the profiles returned from the simulator. An example using this function is shown below

```
Simulate the model using scipy
>>> sim = Simulator(m, package='scipy')
>>> tsim, profiles = sim.simulate(numpoints=100, integrator='vode',
...                               varying_inputs=m.var_input)
```

```
Discretize the model using Orthogonal Collocation
>>> discretizer = TransformationFactory('dae.collocation')
>>> discretizer.apply_to(m, nfe=10, ncp=3)

Initialize the discretized model using the simulator profiles
>>> sim.initialize_model()
```

Note: A model must be simulated before it can be initialized using this function

5.3 Stochastic Programming

TODO

```
>>> print('Hello World')
Hello World
```

5.4 Generalized Disjunctive Programming

TODO

```
>>> print('Hello World')
Hello World
```

5.5 Stochastic Programming

To express a stochastic program in PySP, the user specifies both the deterministic base model and the scenario tree model with associated uncertain parameters. Both concrete and abstract model representations are supported.

Given the deterministic and scenario tree models, PySP provides multiple paths for the solution of the corresponding stochastic program. One alternative involves forming the extensive form and invoking an appropriate deterministic solver for the entire problem once. For more complex stochastic programs, we provide a generic implementation of Rockafellar and Wets' Progressive Hedging algorithm, with additional specializations for approximating mixed-integer stochastic programs as well as other decomposition methods. By leveraging the combination of a high-level programming language (Python) and the embedding of the base deterministic model in that language (Pyomo), we are able to provide completely generic and highly configurable solver implementations.

CHAPTER 6

Persistent Solvers

The purpose of the persistent solver interfaces is to efficiently notify the solver of incremental changes to a Pyomo model. The persistent solver interfaces create and store model instances from the Python API for the corresponding solver. For example, the *GurobiPersistent* class maintaints a pointer to a gurobipy Model object. Thus, we can make small changes to the model and notify the solver rather than recreating the entire model using the solver Python API (or rewriting an entire model file - e.g., an lp file) every time the model is solved.

Warning: Users are responsible for notifying persistent solver interfaces when changes to a model are made!

6.1 Using Persistent Solvers

The first step in using a persistent solver is to create a Pyomo model as usual.

```
>>> import pyomo.environ as pe
>>> m = pe.ConcreteModel()
>>> m.x = pe.Var()
>>> m.y = pe.Var()
>>> m.obj = pe.Objective(expr=m.x**2 + m.y**2)
>>> m.c = pe.Constraint(expr=m.y >= -2*m.x + 5)
```

You can create an instance of a persistent solver through the SolverFactory.

```
>>> opt = pe.SolverFactory('gurobi_persistent')
```

This returns an instance of `GurobiPersistent`. Now we need to tell the solver about our model.

```
>>> opt.set_instance(m)
```

This will create a gurobipy Model object and include the appropriate variables and constraints. We can now solve the model.

```
>>> results = opt.solve()
```

We can also add or remove variables, constraints, blocks, and objectives. For example,

```
>>> m.c2 = pe.Constraint(expr=m.y >= m.x)
>>> opt.add_constraint(m.c2)
```

This tells the solver to add one new constraint but otherwise leave the model unchanged. We can now resolve the model.

```
>>> results = opt.solve()
```

To remove a component, simply call the corresponding remove method.

```
>>> opt.remove_constraint(m.c2)
>>> del m.c2
>>> results = opt.solve()
```

If a pyomo component is replaced with another component with the same name, the first component must be removed from the solver. Otherwise, the solver will have multiple components. For example, the following code will run without error, but the solver will have an extra constraint. The solver will have both y >= -2*x + 5 and y <= x, which is not what was intended!

```
>>> m = pe.ConcreteModel()
>>> m.x = pe.Var()
>>> m.y = pe.Var()
>>> m.c = pe.Constraint(expr=m.y >= -2*m.x + 5)
>>> opt = pe.SolverFactory('gurobi_persistent')
>>> opt.set_instance(m)
>>> # WRONG:
>>> del m.c
>>> m.c = pe.Constraint(expr=m.y <= m.x)
>>> opt.add_constraint(m.c)
```

The correct way to do this is:

```
>>> m = pe.ConcreteModel()
>>> m.x = pe.Var()
>>> m.y = pe.Var()
>>> m.c = pe.Constraint(expr=m.y >= -2*m.x + 5)
>>> opt = pe.SolverFactory('gurobi_persistent')
>>> opt.set_instance(m)
>>> # Correct:
>>> opt.remove_constraint(m.c)
>>> del m.c
>>> m.c = pe.Constraint(expr=m.y <= m.x)
>>> opt.add_constraint(m.c)
```

Warning: Components removed from a pyomo model must be removed from the solver instance by the user.

Additionally, unexpected behavior may result if a component is modified before being removed.

```
>>> m = pe.ConcreteModel()
>>> m.b = pe.Block()
>>> m.b.x = pe.Var()
```

```
>>> m.b.y = pe.Var()
>>> m.b.c = pe.Constraint(expr=m.b.y >= -2*m.b.x + 5)
>>> opt = pe.SolverFactory('gurobi_persistent')
>>> opt.set_instance(m)
>>> m.b.c2 = pe.Constraint(expr=m.b.y <= m.b.x)
>>> # ERROR: The constraint referenced by m.b.c2 does not
>>> # exist in the solver model.
>>> opt.remove_block(m.b)
```

In most cases, the only way to modify a component is to remove it from the solver instance, modify it with Pyomo, and then add it back to the solver instance. The only exception is with variables. Variables may be modified and then updated with with solver:

```
>>> m = pe.ConcreteModel()
>>> m.x = pe.Var()
>>> m.y = pe.Var()
>>> m.obj = pe.Objective(expr=m.x**2 + m.y**2)
>>> m.c = pe.Constraint(expr=m.y >= -2*m.x + 5)
>>> opt = pe.SolverFactory('gurobi_persistent')
>>> opt.set_instance(m)
>>> m.x.setlb(1.0)
>>> opt.update_var(m.x)
```

6.2 Persistent Solver Performance

In order to get the best performance out of the persistent solvers, use the "save_results" flag:

```
>>> import pyomo.environ as pe
>>> m = pe.ConcreteModel()
>>> m.x = pe.Var()
>>> m.y = pe.Var()
>>> m.obj = pe.Objective(expr=m.x**2 + m.y**2)
>>> m.c = pe.Constraint(expr=m.y >= -2*m.x + 5)
>>> opt = pe.SolverFactory('gurobi_persistent')
>>> opt.set_instance(m)
>>> results = opt.solve(save_results=False)
```

Note that if the "save_results" flag is set to False, then the following is not supported.

```
>>> results = opt.solve(save_results=False, load_solutions=False)
>>> if results.solver.termination_condition == TerminationCondition.optimal:
...     m.solutions.load_from(results)
```

However, the following will work:

```
>>> results = opt.solve(save_results=False, load_solutions=False)
>>> if results.solver.termination_condition == TerminationCondition.optimal:
...     opt.load_vars()
```

Additionally, a subset of variable values may be loaded back into the model:

```
>>> results = opt.solve(save_results=False, load_solutions=False)
>>> if results.solver.termination_condition == TerminationCondition.optimal:
...     opt.load_vars(m.x)
```

CHAPTER 7

Library Reference

Pyomo is being increasingly used as a library to support Python scripts. This section describes library APIs for key elements of Pyomo.

7.1 Kernel Library Reference

Low-level Interfaces:

7.1.1 Base Object Storage Interface

class pyomo.core.kernel.component_interface.**IActiveObject**
Bases: object

Interface for objects that support activate/deactivate semantics.

This class is abstract.

activate(**args*, ***kwds*)
Set the active attribute to True

active
A boolean indicating whether or not this object is active.

deactivate(**args*, ***kwds*)
Set the active attribute to False

class pyomo.core.kernel.component_interface.**ICategorizedObject**
Bases: object

Interface for objects that maintain a weak reference to a parent storage object and have a category type.

This class is abstract. It assumes any derived class declares the attributes below at the class or instance level (with or without __slots__):

_ctype
The objects category type.

_parent
A weak reference to the object's parent or `None`.

_is_categorized_object
bool – A flag used to indicate the class is an instance of ICategorized object. This is a workaround for the slow behavior of isinstance on classes that use abc.ABCMeta as a metaclass.

_is_component
bool – A flag used to indicate that the class is an instance of IComponent. This is a workaround for the slow behavior of isinstance on classes that use abc.ABCMeta as a metaclass.

_is_container
bool – A flag used to indicate that the class is an instance IComponentContainer. This is a workaround for the slow behavior of isinstance on classes that use abc.ABCMeta as a metaclass.

getname (*fully_qualified=False*, *name_buffer={}*, *convert=<type 'str'>*)
Dynamically generates a name for this object.

Parameters

- **fully_qualified** (*bool*) – Generate a full name by iterating through all anscestor containers. Default is `False`.
- **convert** (*function*) – A function that converts a storage key into a string representation. Default is the built-in function str.

Returns If a parent exists, this method returns a string representing the name of the object in the context of its parent; otherwise (if no parent exists), this method returns `None`.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

local_name
The object's local name within the context of its parent. Alias for *obj.getname(fully_qualified=False)*.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

name
The object's fully qualified name. Alias for *obj.getname(fully_qualified=True)*.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

parent
The object's parent

parent_block
The first ancestor block above this object

root_block
The root storage block above this object

class pyomo.core.kernel.component_interface.**IComponent**
Bases: *pyomo.core.kernel.component_interface.ICategorizedObject*

Interface for components that can be stored inside objects of type IComponentContainer.

This class is abstract, but it partially implements the ICategorizedObject interface by defining the following attributes:

_is_component
True

_is_container
False

class pyomo.core.kernel.component_interface.**IComponentContainer**
Bases: *pyomo.core.kernel.component_interface.ICategorizedObject*

Interface for containers of components or other containers.

This class is abstract, but it partially implements the ICategorizedObject interface by defining the following attributes:

_is_component
False

_is_container
True

child(**args*, ***kwds*)
Returns a child of this container given a storage key.

child_key(**args*, ***kwds*)
Returns the lookup key associated with a child of this container.

children(**args*, ***kwds*)
A generator over the children of this container.

components(**args*, ***kwds*)
A generator over the set of components stored under this container.

postorder_traversal(**args*, ***kwds*)
A generator over all descendents in postfix order.

preorder_traversal(**args*, ***kwds*)
A generator over all descendents in prefix order.

preorder_visit(**args*, ***kwds*)
Visit all descendents in prefix order.

class pyomo.core.kernel.component_interface.**_ActiveComponentContainerMixin**
Bases: *pyomo.core.kernel.component_interface.IActiveObject*

To be used as an additional base class in IComponentContainer implementations to add fuctionality for activating and deactivating the container and its children.

Note: This class is abstract. It assumes any derived class declares the attributes below at the class or instance level (with or without __slots__):

Attributes:

_active (int): A integer that keeps track of the number of active children stored under this container.

_decrement_active()
This method must be called any time an active is child removed or any time an existing child's active status changes from `True` to `False`.

_increment_active()
This method must be called any time a new active child is added or any time an existing child's active status changes from `False` to `True`.

activate(*_from_parent_=False*)
Activate this container. All children of this container will be activated and the active flag on all ancestors of this container will be set to `True`.

active
The active status of this container.

deactivate(*_from_parent_=False*)
Deactivate this container and all of its children.

class pyomo.core.kernel.component_interface.**_ActiveComponentMixin**
Bases: *pyomo.core.kernel.component_interface.IActiveObject*

To be used as an additional base class in IComponent implementations to add fuctionality for activating and deactivating the component.

Any container that stores implementations of this type should use _ActiveComponentContainerMixin as a base class.

This class is abstract. It assumes any derived class declares the attributes below at the class or instance level (with or without __slots__):

_active
bool – A boolean indicating whethor or not this component is active.

activate(*_from_parent_=False*)
Activate this component.

active
The active status of this container.

deactivate(*_from_parent_=False*)
Deactivate this component.

class pyomo.core.kernel.component_interface.**_SimpleContainerMixin**
Bases: `object`

A partial implementation of the IComponentContainer interface for implementations that store a single component category.

Complete implementations need to set the _ctype property at the class level and declare the remaining required abstract properties of the IComponentContainer base class.

Note that this implementation allows nested storage of other *IComponentContainer* implementations that are defined with the same ctype.

_prepare_for_add(*obj*)
This method must be called any time a new child is inserted into this container.

_prepare_for_delete(*obj*)
This method must be called any time a new child is removed from this container.

components(*active=None*, *return_key=False*)
Generates an efficient traversal of all components stored under this container. Components are leaf nodes in a storage tree (not containers themselves, except for blocks).

Parameters

- **active** (`True`/`None`) – Set to `True` to indicate that only active objects should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the local storage key of the object within its parent and the object itself. By default, only the objects are returned.

Returns iterator of objects or (key,object) tuples

generate_names(*active=None*, *descend_into=True*, *convert=<type 'str'>*, *prefix=''*)

Generate a container of fully qualified names (up to this container) for objects stored under this container.

Parameters

- **active** (`True`/`None`) – Set to `True` to indicate that only active components should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **descend_into** (*bool*) – Indicates whether or not to include subcomponents of any container objects that are not components. Default is `True`.
- **convert** (*function*) – A function that converts a storage key into a string representation. Default is str.
- **prefix** (*str*) – A string to prefix names with.

Returns A component map that behaves as a dictionary mapping component objects to names.

postorder_traversal(*active=None*, *return_key=False*, *root_key=None*)

Generates a postorder traversal of the storage tree.

Parameters

- **active** (`True`/`None`) – Set to `True` to indicate that only active objects should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the local storage key of the object within its parent and the object itself. By default, only the objects are returned.
- **root_key** – The key to return with this object. Ignored when `return_key` is `False`.

Returns iterator of objects or (key,object) tuples

preorder_traversal(*active=None*, *return_key=False*, *root_key=None*)

Generates a preorder traversal of the storage tree.

Parameters

- **active** (`True`/`None`) – Set to `True` to indicate that only active objects should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the local storage key of the object within its parent and the object itself. By default, only the objects are returned.

- **root_key** – The key to return with this object. Ignored when `return_key` is `False`.

Returns iterator of objects or (key,object) tuples

preorder_visit(*visit*, *active=None*, *include_key=False*, *root_key=None*)
Visits each node in the storage tree using a preorder traversal.

Parameters

- **visit** – A function that is called on each node in the storage tree. When the `include_key` keyword is `False`, the function signature should be *visit(node) -> [True|False]*. When the `include_key` keyword is `True`, the function signature should be *visit(key,node) -> [True|False]*. When the return value of the function evaluates to to `True`, this indicates that the traversal should continue with the children of the current node; otherwise, the traversal does not go below the current node.
- **active** (`True`/`None`) – Set to `True` to indicate that only active objects should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **include_key** (*bool*) – Set to `True` to indicate that 2 arguments should be passed to the visit function, with the first being the local storage key of the object within its parent and the second being the object itself. By default, only the objects are passed to the function.
- **root_key** – The key to pass with this object. Ignored when `include_key` is `False`.

Container Interfaces:

7.1.2 Blocks

class pyomo.core.kernel.component_block.**IBlockStorage**
Bases: *pyomo.core.kernel.component_interface.IComponent*, *pyomo.core.kernel.component_interface.IComponentContainer*, *pyomo.core.kernel.component_interface._ActiveComponentContainerMixin*

A container that stores multiple types.

This class is abstract, but it partially implements the `ICategorizedObject` interface by defining the following attributes:

_is_component
`True`

_is_container
`True`

activate(*_from_parent_=False*)
Activate this container. All children of this container will be activated and the active flag on all ancestors of this container will be set to `True`.

active
The active status of this container.

child(**args*, ***kwds*)
Returns a child of this container given a storage key.

child_key(**args*, ***kwds*)
Returns the lookup key associated with a child of this container.

clone()
Clones this block. Returns a new block with whose parent pointer is set to None. Any components encountered that are descendents of this block will be deepcopied, otherwise a reference to the original component is retained.

deactivate(*_from_parent_=False*)
Deactivate this container and all of its children.

getname(*fully_qualified=False*, *name_buffer={}*, *convert=<type 'str'>*)
Dynamically generates a name for this object.

Parameters

- **fully_qualified** (*bool*) – Generate a full name by iterating through all anscestor containers. Default is False.
- **convert** (*function*) – A function that converts a storage key into a string representation. Default is the built-in function str.

Returns If a parent exists, this method returns a string representing the name of the object in the context of its parent; otherwise (if no parent exists), this method returns None.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

local_name
The object's local name within the context of its parent. Alias for *obj.getname(fully_qualified=False)*.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

name
The object's fully qualified name. Alias for *obj.getname(fully_qualified=True)*.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

parent
The object's parent

parent_block
The first ancestor block above this object

postorder_traversal(**args*, ***kwds*)
A generator over all descendents in postfix order.

preorder_traversal(**args*, ***kwds*)
A generator over all descendents in prefix order.

preorder_visit(**args*, ***kwds*)
Visit all descendents in prefix order.

root_block
The root storage block above this object

class pyomo.core.kernel.component_block.**block**
Bases: pyomo.core.kernel.component_block._block_base, *pyomo.core.kernel.component_block.IBlockStorage*

An implementation of the *IBlockStorage* interface.

activate(*shallow=True*, *descend_into=False*, *_from_parent_=False*)
Activates this block.

Parameters

- **shallow** (*bool*) – If False, all children of the block will be activated. By default, the active status of children are not changed.
- **descend_into** (*bool*) – Indicates whether or not to perform the same action on sub-blocks. The default is False, as a shallow operation on the top-level block is sufficient.

active
The active status of this container.

blocks(*active=None*, *descend_into=True*)
Generates a traversal of all blocks associated with this one (including itself). This method yields identical behavior to calling the components() method with ctype=Block, except that this block is included (as the first item in the generator).

child(*key*)
Get the child object associated with a given storage key for this container.

Raises KeyError – if the argument is not a storage key for any children of this container

child_key(*child*)
Get the lookup key associated with a child of this container.

Raises ValueError – if the argument is not a child of this container

children(*ctype=<object object>*, *return_key=False*)
Iterate over the children of this block.

Parameters

- **ctype** – Indicate the type of children to iterate over. The default value indicates that all types should be included.
- **return_key** (*bool*) – Set to True to indicate that the return type should be a 2-tuple consisting of the child storage key and the child object. By default, only the child objects are returned.

Returns iterator of objects or (key,object) tuples

clone()
Clones this block. Returns a new block with whose parent pointer is set to None. Any components encountered that are descendents of this block will be deepcopied, otherwise a reference to the original component is retained.

collect_ctypes(*active=None*, *descend_into=True*)
Count all object category types stored on or under this block.

Parameters

- **active** (True/None) – Set to True to indicate that only active categorized objects should be counted. The default value of None indicates that all categorized objects (including those that have been deactivated) should be counted. *Note*: This flag is ignored for any objects that do not have an active flag.

- **descend_into** (*bool*) – Indicates whether or not category types should be counted on sub-blocks. Default is `True`.

Returns set of category types

components(*ctype=<object object>*, *active=None*, *return_key=False*, *descend_into=True*)

Generates an efficient traversal of all components stored under this block. Components are leaf nodes in a storage tree (not containers themselves, except for blocks).

Parameters

- **ctype** – Indicate the type of components to include. The default value indicates that all types should be included.
- **active** (`True`/`None`) – Set to `True` to indicate that only active objects should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the local storage key of the object within its parent and the object itself. By default, only the objects are returned.
- **descend_into** (*bool*) – Indicates whether or not to include components on sub-blocks. Default is `True`.

Returns iterator of objects or (key,object) tuples

deactivate(*shallow=True*, *descend_into=False*, *_from_parent_=False*)

Deactivates this block.

Parameters

- **shallow** (*bool*) – If `False`, all children of the block will be deactivated. By default, the active status of children are not changed, but they become effectively inactive for anything above this block.
- **descend_into** (*bool*) – Indicates whether or not to perform the same action on sub-blocks. The default is `False`, as a shallow operation on the top-level block is sufficient.

generate_names(*ctype=<object object>*, *active=None*, *descend_into=True*, *convert=<type 'str'>*, *prefix=''*)

Generate a container of fully qualified names (up to this block) for objects stored under this block.

This function is useful in situations where names are used often, but they do not need to be dynamically regenerated each time.

Parameters

- **ctype** – Indicate the type of components to include. The default value indicates that all types should be included.
- **active** (`True`/`None`) – Set to `True` to indicate that only active components should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **descend_into** (*bool*) – Indicates whether or not to include components on sub-blocks. Default is `True`.
- **convert** (*function*) – A function that converts a storage key into a string representation. Default is str.
- **prefix** (*str*) – A string to prefix names with.

Returns A component map that behaves as a dictionary mapping component objects to names.

getname (*fully_qualified=False*, *name_buffer={}*, *convert=<type 'str'>*)
Dynamically generates a name for this object.

Parameters

- **fully_qualified** (*bool*) – Generate a full name by iterating through all anscestor containers. Default is `False`.
- **convert** (*function*) – A function that converts a storage key into a string representation. Default is the built-in function str.

Returns If a parent exists, this method returns a string representing the name of the object in the context of its parent; otherwise (if no parent exists), this method returns `None`.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

load_solution (*solution*, *allow_consistent_values_for_fixed_vars=False*, *comparison_tolerance_for_fixed_vars=1e-05*)
Load a solution.

Parameters

- **solution** – A `pyomo.opt.Solution` object with a symbol map. Optionally, the solution can be tagged with a default variable value (e.g., 0) that will be applied to those variables in the symbol map that do not have a value in the solution.
- **allow_consistent_values_for_fixed_vars** – Indicates whether a solution can specify consistent values for variables that are fixed.
- **comparison_tolerance_for_fixed_vars** – The tolerance used to define whether or not a value in the solution is consistent with the value of a fixed variable.

local_name
The object's local name within the context of its parent. Alias for *obj.getname(fully_qualified=False)*.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

name
The object's fully qualified name. Alias for *obj.getname(fully_qualified=True)*.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

parent
The object's parent

parent_block
The first ancestor block above this object

postorder_traversal (*ctype=<object object>*, *active=None*, *include_all_parents=True*, *return_key=False*, *root_key=None*)

Generates a postorder traversal of the storage tree. This includes all components and all component containers (optionally) matching the requested type.

Parameters

- **ctype** – Indicate the type of components to include. The default value indicates that all types should be included.
- **active** (`True`/`None`) – Set to `True` to indicate that only active objects should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **include_all_parents** (*bool*) – Indicates if all parent containers (such as blocks and simple block containers) should be included in the traversal even when the `ctype` keyword is set to something that is not Block. Default is `True`.
- **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the local storage key of the object within its parent and the object itself. By default, only the objects are returned.
- **root_key** – The key to return with this object. Ignored when `return_key` is `False`.

Returns iterator of objects or (key,object) tuples

preorder_traversal(*ctype=<object object>*, *active=None*, *include_all_parents=True*, *return_key=False*, *root_key=None*)

Generates a preorder traversal of the storage tree. This includes all components and all component containers (optionally) matching the requested type.

Parameters

- **ctype** – Indicate the type of components to include. The default value indicates that all types should be included.
- **active** (`True`/`None`) – Set to `True` to indicate that only active objects should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **include_all_parents** (*bool*) – Indicates if all parent containers (such as blocks and simple block containers) should be included in the traversal even when the `ctype` keyword is set to something that is not Block. Default is `True`.
- **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the local storage key of the object within its parent and the object itself. By default, only the objects are returned.
- **root_key** – The key to return with this object. Ignored when `return_key` is `False`.

Returns iterator of objects or (key,object) tuples

preorder_visit(*visit*, *ctype=<object object>*, *active=None*, *include_all_parents=True*, *include_key=False*, *root_key=None*)

Visits each node in the storage tree using a preorder traversal. This includes all components and all component containers (optionally) matching the requested type.

Parameters

- **visit** – A function that is called on each node in the storage tree. When the `include_key` keyword is `False`, the function signature should be *visit(node) -> [True|False]*. When the `include_key` keyword is `True`, the function signature should be *visit(key,node) -> [True|False]*. When the return value of the function evaluates to to

True, this indicates that the traversal should continue with the children of the current node; otherwise, the traversal does not go below the current node.

- **ctype** – Indicate the type of components to include. The default value indicates that all types should be included.
- **active** (True/None) – Set to True to indicate that only active objects should be included. The default value of None indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **include_all_parents** (*bool*) – Indicates if all parent containers (such as blocks and simple block containers) should be included in the traversal even when the ctype keyword is set to something that is not Block. Default is True.
- **include_key** (*bool*) – Set to True to indicate that 2 arguments should be passed to the visit function, with the first being the local storage key of the object within its parent and the second being the object itself. By default, only the objects are passed to the function.
- **root_key** – The key to pass with this object. Ignored when include_key is False.

root_block
The root storage block above this object

write(*filename*, *format=None*, *_solver_capability=None*, *_called_by_solver=False*, ***kwds*)
Write the model to a file, with a given format.

Parameters

- **filename** (*str*) – The name of the file to write.
- **format** – The file format to use. If this is not specified, the file format will be inferred from the filename suffix.
- ****kwds** – Additional keyword options passed to the model writer.

Returns a SymbolMap

class pyomo.core.kernel.component_block.**block_dict**(**args*, ***kwds*)
Bases: *pyomo.core.kernel.component_dict.ComponentDict*, *pyomo.core.kernel.component_interface._ActiveComponentContainerMixin*

A dict-style container for blocks.

activate(*_from_parent_=False*)
Activate this container. All children of this container will be activated and the active flag on all ancestors of this container will be set to True.

active
The active status of this container.

child(*key*)
Get the child object associated with a given storage key for this container.

Raises KeyError – if the argument is not a storage key for any children of this container

child_key(*child*)
Get the lookup key associated with a child of this container.

Raises ValueError – if the argument is not a child of this container

children(*return_key=False*)
Iterate over the children of this container.

Parameters **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the child storage key and the child object. By default, only the child objects are returned.

Returns iterator of objects or (key,object) tuples

clear() → None. Remove all items from D.

components(*active=None*, *return_key=False*)
Generates an efficient traversal of all components stored under this container. Components are leaf nodes in a storage tree (not containers themselves, except for blocks).

Parameters

- **active** (`True`/`None`) – Set to `True` to indicate that only active objects should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the local storage key of the object within its parent and the object itself. By default, only the objects are returned.

Returns iterator of objects or (key,object) tuples

deactivate(*_from_parent_=False*)
Deactivate this container and all of its children.

generate_names(*active=None*, *descend_into=True*, *convert=<type 'str'>*, *prefix=''*)
Generate a container of fully qualified names (up to this container) for objects stored under this container.

Parameters

- **active** (`True`/`None`) – Set to `True` to indicate that only active components should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **descend_into** (*bool*) – Indicates whether or not to include subcomponents of any container objects that are not components. Default is `True`.
- **convert** (*function*) – A function that converts a storage key into a string representation. Default is str.
- **prefix** (*str*) – A string to prefix names with.

Returns A component map that behaves as a dictionary mapping component objects to names.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

getname(*fully_qualified=False*, *name_buffer={}*, *convert=<type 'str'>*)
Dynamically generates a name for this object.

Parameters

- **fully_qualified** (*bool*) – Generate a full name by iterating through all anscestor containers. Default is `False`.
- **convert** (*function*) – A function that converts a storage key into a string representation. Default is the built-in function str.

Returns If a parent exists, this method returns a string representing the name of the object in the context of its parent; otherwise (if no parent exists), this method returns `None`.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

items() → list of D's (key, value) pairs, as 2-tuples

iteritems() → an iterator over the (key, value) items of D

iterkeys() → an iterator over the keys of D

itervalues() → an iterator over the values of D

keys() → list of D's keys

local_name
The object's local name within the context of its parent. Alias for *obj.getname(fully_qualified=False)*.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

name
The object's fully qualified name. Alias for *obj.getname(fully_qualified=True)*.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

parent
The object's parent

parent_block
The first ancestor block above this object

pop($k[, d]$) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

postorder_traversal(*active=None*, *return_key=False*, *root_key=None*)
Generates a postorder traversal of the storage tree.

Parameters

- **active** (`True`/`None`) – Set to `True` to indicate that only active objects should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the local storage key of the object within its parent and the object itself. By default, only the objects are returned.
- **root_key** – The key to return with this object. Ignored when `return_key` is `False`.

Returns iterator of objects or (key,object) tuples

preorder_traversal(*active=None*, *return_key=False*, *root_key=None*)
Generates a preorder traversal of the storage tree.

Parameters

- **active** (`True`/`None`) – Set to `True` to indicate that only active objects should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the local storage key of the object within its parent and the object itself. By default, only the objects are returned.
- **root_key** – The key to return with this object. Ignored when `return_key` is `False`.

Returns iterator of objects or (key,object) tuples

preorder_visit(*visit*, *active=None*, *include_key=False*, *root_key=None*)
Visits each node in the storage tree using a preorder traversal.

Parameters

- **visit** – A function that is called on each node in the storage tree. When the `include_key` keyword is `False`, the function signature should be *visit(node) -> [True|False]*. When the `include_key` keyword is `True`, the function signature should be *visit(key,node) -> [True|False]*. When the return value of the function evaluates to to `True`, this indicates that the traversal should continue with the children of the current node; otherwise, the traversal does not go below the current node.
- **active** (`True`/`None`) – Set to `True` to indicate that only active objects should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **include_key** (*bool*) – Set to `True` to indicate that 2 arguments should be passed to the visit function, with the first being the local storage key of the object within its parent and the second being the object itself. By default, only the objects are passed to the function.
- **root_key** – The key to pass with this object. Ignored when `include_key` is `False`.

root_block
The root storage block above this object

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → list of D's values

class pyomo.core.kernel.component_block.**block_list**(**args*, ***kwds*)
Bases: *pyomo.core.kernel.component_list.ComponentList*, *pyomo.core.kernel.component_interface._ActiveComponentContainerMixin*

A list-style container for blocks.

activate(*_from_parent_=False*)
Activate this container. All children of this container will be activated and the active flag on all ancestors of this container will be set to `True`.

active
The active status of this container.

append(*value*)
S.append(object) – append object to the end of the sequence

child(*key*)
Get the child object associated with a given storage key for this container.

Raises `KeyError` – if the argument is not a storage key for any children of this container

child_key(*child*)
Get the lookup key associated with a child of this container.

Raises `ValueError` – if the argument is not a child of this container

children(*return_key=False*)
Iterate over the children of this container.

Parameters **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the child storage key and the child object. By default, only the child objects are returned.

Returns iterator of objects or (key,object) tuples

components(*active=None*, *return_key=False*)
Generates an efficient traversal of all components stored under this container. Components are leaf nodes in a storage tree (not containers themselves, except for blocks).

Parameters

- **active** (`True`/`None`) – Set to `True` to indicate that only active objects should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the local storage key of the object within its parent and the object itself. By default, only the objects are returned.

Returns iterator of objects or (key,object) tuples

count(*value*) → integer – return number of occurrences of value

deactivate(*_from_parent_=False*)
Deactivate this container and all of its children.

extend(*values*)
S.extend(iterable) – extend sequence by appending elements from the iterable

generate_names(*active=None*, *descend_into=True*, *convert=<type 'str'>*, *prefix=''*)
Generate a container of fully qualified names (up to this container) for objects stored under this container.

Parameters

- **active** (`True`/`None`) – Set to `True` to indicate that only active components should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **descend_into** (*bool*) – Indicates whether or not to include subcomponents of any container objects that are not components. Default is `True`.
- **convert** (*function*) – A function that converts a storage key into a string representation. Default is str.
- **prefix** (*str*) – A string to prefix names with.

Returns A component map that behaves as a dictionary mapping component objects to names.

getname (*fully_qualified=False*, *name_buffer={}*, *convert=<type 'str'>*)
Dynamically generates a name for this object.

Parameters

- **fully_qualified** (*bool*) – Generate a full name by iterating through all anscestor containers. Default is `False`.
- **convert** (*function*) – A function that converts a storage key into a string representation. Default is the built-in function str.

Returns If a parent exists, this method returns a string representing the name of the object in the context of its parent; otherwise (if no parent exists), this method returns `None`.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

index (*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

insert (*i*, *item*)
S.insert(index, object) – insert object before index

local_name
The object's local name within the context of its parent. Alias for *obj.getname(fully_qualified=False)*.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

name
The object's fully qualified name. Alias for *obj.getname(fully_qualified=True)*.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

parent
The object's parent

parent_block
The first ancestor block above this object

pop ([*index*]) → item – remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

postorder_traversal (*active=None*, *return_key=False*, *root_key=None*)
Generates a postorder traversal of the storage tree.

Parameters

- **active** (`True`/`None`) – Set to `True` to indicate that only active objects should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.

- **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the local storage key of the object within its parent and the object itself. By default, only the objects are returned.
- **root_key** – The key to return with this object. Ignored when `return_key` is `False`.

Returns iterator of objects or (key,object) tuples

preorder_traversal(*active=None*, *return_key=False*, *root_key=None*)
Generates a preorder traversal of the storage tree.

Parameters

- **active** (`True`/`None`) – Set to `True` to indicate that only active objects should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the local storage key of the object within its parent and the object itself. By default, only the objects are returned.
- **root_key** – The key to return with this object. Ignored when `return_key` is `False`.

Returns iterator of objects or (key,object) tuples

preorder_visit(*visit*, *active=None*, *include_key=False*, *root_key=None*)
Visits each node in the storage tree using a preorder traversal.

Parameters

- **visit** – A function that is called on each node in the storage tree. When the `include_key` keyword is `False`, the function signature should be *visit(node) -> [True|False]*. When the `include_key` keyword is `True`, the function signature should be *visit(key,node) -> [True|False]*. When the return value of the function evaluates to to `True`, this indicates that the traversal should continue with the children of the current node; otherwise, the traversal does not go below the current node.
- **active** (`True`/`None`) – Set to `True` to indicate that only active objects should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **include_key** (*bool*) – Set to `True` to indicate that 2 arguments should be passed to the visit function, with the first being the local storage key of the object within its parent and the second being the object itself. By default, only the objects are passed to the function.
- **root_key** – The key to pass with this object. Ignored when `include_key` is `False`.

remove(*value*)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

reverse()
S.reverse() – reverse *IN PLACE*

root_block
The root storage block above this object

class pyomo.core.kernel.component_block.**block_tuple**(**args*, ***kwds*)
Bases: *pyomo.core.kernel.component_tuple.ComponentTuple*, *pyomo.core.kernel.component_interface._ActiveComponentContainerMixin*

A tuple-style container for blocks.

activate(*_from_parent_=False*)
Activate this container. All children of this container will be activated and the active flag on all ancestors of this container will be set to `True`.

active
The active status of this container.

child(*key*)
Get the child object associated with a given storage key for this container.

Raises `KeyError` – if the argument is not a storage key for any children of this container

child_key(*child*)
Get the lookup key associated with a child of this container.

Raises `ValueError` – if the argument is not a child of this container

children(*return_key=False*)
Iterate over the children of this container.

Parameters **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the child storage key and the child object. By default, only the child objects are returned.

Returns iterator of objects or (key,object) tuples

components(*active=None*, *return_key=False*)
Generates an efficient traversal of all components stored under this container. Components are leaf nodes in a storage tree (not containers themselves, except for blocks).

Parameters

- **active** (`True`/`None`) – Set to `True` to indicate that only active objects should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the local storage key of the object within its parent and the object itself. By default, only the objects are returned.

Returns iterator of objects or (key,object) tuples

count(*value*) → integer – return number of occurrences of value

deactivate(*_from_parent_=False*)
Deactivate this container and all of its children.

generate_names(*active=None*, *descend_into=True*, *convert=<type 'str'>*, *prefix=''*)
Generate a container of fully qualified names (up to this container) for objects stored under this container.

Parameters

- **active** (`True`/`None`) – Set to `True` to indicate that only active components should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **descend_into** (*bool*) – Indicates whether or not to include subcomponents of any container objects that are not components. Default is `True`.

- **convert** (*function*) – A function that converts a storage key into a string representation. Default is str.
- **prefix** (*str*) – A string to prefix names with.

Returns A component map that behaves as a dictionary mapping component objects to names.

getname (*fully_qualified=False*, *name_buffer={}*, *convert=<type 'str'>*)
Dynamically generates a name for this object.

Parameters

- **fully_qualified** (*bool*) – Generate a full name by iterating through all anscestor containers. Default is `False`.
- **convert** (*function*) – A function that converts a storage key into a string representation. Default is the built-in function str.

Returns If a parent exists, this method returns a string representing the name of the object in the context of its parent; otherwise (if no parent exists), this method returns `None`.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

index (*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

local_name
The object's local name within the context of its parent. Alias for *obj.getname(fully_qualified=False)*.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

name
The object's fully qualified name. Alias for *obj.getname(fully_qualified=True)*.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

parent
The object's parent

parent_block
The first ancestor block above this object

postorder_traversal (*active=None*, *return_key=False*, *root_key=None*)
Generates a postorder traversal of the storage tree.

Parameters

- **active** (`True`/`None`) – Set to `True` to indicate that only active objects should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.

- **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the local storage key of the object within its parent and the object itself. By default, only the objects are returned.
- **root_key** – The key to return with this object. Ignored when `return_key` is `False`.

Returns iterator of objects or (key,object) tuples

preorder_traversal(*active=None*, *return_key=False*, *root_key=None*)
Generates a preorder traversal of the storage tree.

Parameters

- **active** (`True`/`None`) – Set to `True` to indicate that only active objects should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the local storage key of the object within its parent and the object itself. By default, only the objects are returned.
- **root_key** – The key to return with this object. Ignored when `return_key` is `False`.

Returns iterator of objects or (key,object) tuples

preorder_visit(*visit*, *active=None*, *include_key=False*, *root_key=None*)
Visits each node in the storage tree using a preorder traversal.

Parameters

- **visit** – A function that is called on each node in the storage tree. When the `include_key` keyword is `False`, the function signature should be *visit(node) -> [True|False]*. When the `include_key` keyword is `True`, the function signature should be *visit(key,node) -> [True|False]*. When the return value of the function evaluates to to `True`, this indicates that the traversal should continue with the children of the current node; otherwise, the traversal does not go below the current node.
- **active** (`True`/`None`) – Set to `True` to indicate that only active objects should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **include_key** (*bool*) – Set to `True` to indicate that 2 arguments should be passed to the visit function, with the first being the local storage key of the object within its parent and the second being the object itself. By default, only the objects are passed to the function.
- **root_key** – The key to pass with this object. Ignored when `include_key` is `False`.

root_block
The root storage block above this object

class pyomo.core.kernel.component_block.**tiny_block**
Bases: pyomo.core.kernel.component_block._block_base, *pyomo.core.kernel.component_block.IBlockStorage*

A memory efficient block for storing a small number of child components.

activate(*shallow=True*, *descend_into=False*, *_from_parent_=False*)
Activates this block.

Parameters

- **shallow** (*bool*) – If `False`, all children of the block will be activated. By default, the active status of children are not changed.
- **descend_into** (*bool*) – Indicates whether or not to perform the same action on sub-blocks. The default is `False`, as a shallow operation on the top-level block is sufficient.

active
The active status of this container.

blocks(*active=None*, *descend_into=True*)
Generates a traversal of all blocks associated with this one (including itself). This method yields identical behavior to calling the components() method with ctype=Block, except that this block is included (as the first item in the generator).

child(*key*)
Get the child object associated with a given storage key for this container.

Raises `KeyError` – if the argument is not a storage key for any children of this container

child_key(*child*)
Get the lookup key associated with a child of this container.

Raises `ValueError` – if the argument is not a child of this container

children(*ctype=<object object>*, *return_key=False*)
Iterate over the children of this block.

Parameters

- **ctype** – Indicate the type of children to iterate over. The default value indicates that all types should be included.
- **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the child storage key and the child object. By default, only the child objects are returned.

Returns iterator of objects or (key,object) tuples

clone()
Clones this block. Returns a new block with whose parent pointer is set to `None`. Any components encountered that are descendents of this block will be deepcopied, otherwise a reference to the original component is retained.

collect_ctypes(*active=None*, *descend_into=True*)
Count all object category types stored on or under this block.

Parameters

- **active** (`True`/`None`) – Set to `True` to indicate that only active categorized objects should be counted. The default value of `None` indicates that all categorized objects (including those that have been deactivated) should be counted. *Note*: This flag is ignored for any objects that do not have an active flag.
- **descend_into** (*bool*) – Indicates whether or not category types should be counted on sub-blocks. Default is `True`.

Returns set of category types

components(*ctype=<object object>*, *active=None*, *return_key=False*, *descend_into=True*)
Generates an efficient traversal of all components stored under this block. Components are leaf nodes in a storage tree (not containers themselves, except for blocks).

Parameters

- **ctype** – Indicate the type of components to include. The default value indicates that all types should be included.
- **active** (True/None) – Set to True to indicate that only active objects should be included. The default value of None indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **return_key** (*bool*) – Set to True to indicate that the return type should be a 2-tuple consisting of the local storage key of the object within its parent and the object itself. By default, only the objects are returned.
- **descend_into** (*bool*) – Indicates whether or not to include components on sub-blocks. Default is True.

Returns iterator of objects or (key,object) tuples

deactivate(*shallow=True*, *descend_into=False*, *_from_parent_=False*)

Deactivates this block.

Parameters

- **shallow** (*bool*) – If False, all children of the block will be deactivated. By default, the active status of children are not changed, but they become effectively inactive for anything above this block.
- **descend_into** (*bool*) – Indicates whether or not to perform the same action on sub-blocks. The default is False, as a shallow operation on the top-level block is sufficient.

generate_names(*ctype=<object object>*, *active=None*, *descend_into=True*, *convert=<type 'str'>*, *prefix=''*)

Generate a container of fully qualified names (up to this block) for objects stored under this block.

This function is useful in situations where names are used often, but they do not need to be dynamically regenerated each time.

Parameters

- **ctype** – Indicate the type of components to include. The default value indicates that all types should be included.
- **active** (True/None) – Set to True to indicate that only active components should be included. The default value of None indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **descend_into** (*bool*) – Indicates whether or not to include components on sub-blocks. Default is True.
- **convert** (*function*) – A function that converts a storage key into a string representation. Default is str.
- **prefix** (*str*) – A string to prefix names with.

Returns A component map that behaves as a dictionary mapping component objects to names.

getname(*fully_qualified=False*, *name_buffer={}*, *convert=<type 'str'>*)

Dynamically generates a name for this object.

Parameters

- **fully_qualified** (*bool*) – Generate a full name by iterating through all anscestor containers. Default is False.

- **convert** (*function*) – A function that converts a storage key into a string representation. Default is the built-in function str.

Returns If a parent exists, this method returns a string representing the name of the object in the context of its parent; otherwise (if no parent exists), this method returns `None`.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

load_solution(*solution*, *allow_consistent_values_for_fixed_vars=False*, *comparison_tolerance_for_fixed_vars=1e-05*)

Load a solution.

Parameters

- **solution** – A `pyomo.opt.Solution` object with a symbol map. Optionally, the solution can be tagged with a default variable value (e.g., 0) that will be applied to those variables in the symbol map that do not have a value in the solution.
- **allow_consistent_values_for_fixed_vars** – Indicates whether a solution can specify consistent values for variables that are fixed.
- **comparison_tolerance_for_fixed_vars** – The tolerance used to define whether or not a value in the solution is consistent with the value of a fixed variable.

local_name

The object's local name within the context of its parent. Alias for *obj.getname(fully_qualified=False)*.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

name

The object's fully qualified name. Alias for *obj.getname(fully_qualified=True)*.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

parent

The object's parent

parent_block

The first ancestor block above this object

postorder_traversal(*ctype=<object object>*, *active=None*, *include_all_parents=True*, *return_key=False*, *root_key=None*)

Generates a postorder traversal of the storage tree. This includes all components and all component containers (optionally) matching the requested type.

Parameters

- **ctype** – Indicate the type of components to include. The default value indicates that all types should be included.
- **active** (`True`/`None`) – Set to `True` to indicate that only active objects should be included. The default value of `None` indicates that all components (including those that

have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.

- **include_all_parents** (*bool*) – Indicates if all parent containers (such as blocks and simple block containers) should be included in the traversal even when the `ctype` keyword is set to something that is not Block. Default is `True`.
- **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the local storage key of the object within its parent and the object itself. By default, only the objects are returned.
- **root_key** – The key to return with this object. Ignored when `return_key` is `False`.

Returns iterator of objects or (key,object) tuples

preorder_traversal(*ctype=<object object>*, *active=None*, *include_all_parents=True*, *return_key=False*, *root_key=None*)

Generates a preorder traversal of the storage tree. This includes all components and all component containers (optionally) matching the requested type.

Parameters

- **ctype** – Indicate the type of components to include. The default value indicates that all types should be included.
- **active** (`True`/`None`) – Set to `True` to indicate that only active objects should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **include_all_parents** (*bool*) – Indicates if all parent containers (such as blocks and simple block containers) should be included in the traversal even when the `ctype` keyword is set to something that is not Block. Default is `True`.
- **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the local storage key of the object within its parent and the object itself. By default, only the objects are returned.
- **root_key** – The key to return with this object. Ignored when `return_key` is `False`.

Returns iterator of objects or (key,object) tuples

preorder_visit(*visit*, *ctype=<object object>*, *active=None*, *include_all_parents=True*, *include_key=False*, *root_key=None*)

Visits each node in the storage tree using a preorder traversal. This includes all components and all component containers (optionally) matching the requested type.

Parameters

- **visit** – A function that is called on each node in the storage tree. When the `include_key` keyword is `False`, the function signature should be *visit(node) -> [True|False]*. When the `include_key` keyword is `True`, the function signature should be *visit(key,node) -> [True|False]*. When the return value of the function evaluates to to `True`, this indicates that the traversal should continue with the children of the current node; otherwise, the traversal does not go below the current node.
- **ctype** – Indicate the type of components to include. The default value indicates that all types should be included.
- **active** (`True`/`None`) – Set to `True` to indicate that only active objects should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.

- **include_all_parents** (*bool*) – Indicates if all parent containers (such as blocks and simple block containers) should be included in the traversal even when the ctype keyword is set to something that is not Block. Default is True.
- **include_key** (*bool*) – Set to True to indicate that 2 arguments should be passed to the visit function, with the first being the local storage key of the object within its parent and the second being the object itself. By default, only the objects are passed to the function.
- **root_key** – The key to pass with this object. Ignored when include_key is False.

root_block
The root storage block above this object

write(*filename*, *format=None*, *_solver_capability=None*, *_called_by_solver=False*, ***kwds*)
Write the model to a file, with a given format.

Parameters

- **filename** (*str*) – The name of the file to write.
- **format** – The file format to use. If this is not specified, the file format will be inferred from the filename suffix.
- ****kwds** – Additional keyword options passed to the model writer.

Returns a SymbolMap

7.1.3 Tuple-like Object Storage

class pyomo.core.kernel.component_tuple.**ComponentTuple**(**args*)
Bases: *pyomo.core.kernel.component_interface._SimpleContainerMixin*, *pyomo.core.kernel.component_interface.IComponentContainer*, _abcoll.Sequence

A partial implementation of the IComponentContainer interface that presents tuple-like storage functionality.

Complete implementations need to set the _ctype property at the class level, declare the remaining required abstract properties of the IComponentContainer base class, and declare a slot or attribute named _data.

Note that this implementation allows nested storage of other IComponentContainer implementations that are defined with the same ctype.

__delattr__
x.__delattr__('name') <==> del x.name

__format__()
default object formatter

__getattribute__
x.__getattribute__('name') <==> x.name

__hash__

__metaclass__
alias of ABCMeta

__new__(*S*, ...) → a new object with type S, a subtype of T

__reduce__()
helper for pickle

__reduce_ex__()
helper for pickle

__repr__

__setattr__
x.__setattr__('name', value) <==> x.name = value

__sizeof__() → int
size of object in memory, in bytes

__str__()
Convert this object to a string by first attempting to generate its fully qualified name. If the object does not have a name (because it does not have a parent, then a string containing the class name is returned.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

__weakref__
list of weak references to the object (if defined)

child(*key*)
Get the child object associated with a given storage key for this container.

Raises KeyError – if the argument is not a storage key for any children of this container

child_key(*child*)
Get the lookup key associated with a child of this container.

Raises ValueError – if the argument is not a child of this container

children(*return_key=False*)
Iterate over the children of this container.

Parameters **return_key** (*bool*) – Set to True to indicate that the return type should be a 2-tuple consisting of the child storage key and the child object. By default, only the child objects are returned.

Returns iterator of objects or (key,object) tuples

components(*active=None*, *return_key=False*)
Generates an efficient traversal of all components stored under this container. Components are leaf nodes in a storage tree (not containers themselves, except for blocks).

Parameters

- **active** (True/None) – Set to True to indicate that only active objects should be included. The default value of None indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **return_key** (*bool*) – Set to True to indicate that the return type should be a 2-tuple consisting of the local storage key of the object within its parent and the object itself. By default, only the objects are returned.

Returns iterator of objects or (key,object) tuples

count(*value*) → integer – return number of occurrences of value

generate_names(*active=None*, *descend_into=True*, *convert=<type 'str'>*, *prefix=''*)
Generate a container of fully qualified names (up to this container) for objects stored under this container.

Parameters

- **active** (`True`/`None`) – Set to `True` to indicate that only active components should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **descend_into** (*bool*) – Indicates whether or not to include subcomponents of any container objects that are not components. Default is `True`.
- **convert** (*function*) – A function that converts a storage key into a string representation. Default is str.
- **prefix** (*str*) – A string to prefix names with.

Returns A component map that behaves as a dictionary mapping component objects to names.

getname (*fully_qualified=False*, *name_buffer={}*, *convert=<type 'str'>*)
Dynamically generates a name for this object.

Parameters

- **fully_qualified** (*bool*) – Generate a full name by iterating through all anscestor containers. Default is `False`.
- **convert** (*function*) – A function that converts a storage key into a string representation. Default is the built-in function str.

Returns If a parent exists, this method returns a string representing the name of the object in the context of its parent; otherwise (if no parent exists), this method returns `None`.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

index (*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

local_name
The object's local name within the context of its parent. Alias for *obj.getname(fully_qualified=False)*.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

name
The object's fully qualified name. Alias for *obj.getname(fully_qualified=True)*.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

parent
The object's parent

parent_block
The first ancestor block above this object

postorder_traversal (*active=None*, *return_key=False*, *root_key=None*)
Generates a postorder traversal of the storage tree.

Parameters

- **active** (True/None) – Set to True to indicate that only active objects should be included. The default value of None indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **return_key** (*bool*) – Set to True to indicate that the return type should be a 2-tuple consisting of the local storage key of the object within its parent and the object itself. By default, only the objects are returned.
- **root_key** – The key to return with this object. Ignored when return_key is False.

Returns iterator of objects or (key,object) tuples

preorder_traversal(*active=None*, *return_key=False*, *root_key=None*)
Generates a preorder traversal of the storage tree.

Parameters

- **active** (True/None) – Set to True to indicate that only active objects should be included. The default value of None indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **return_key** (*bool*) – Set to True to indicate that the return type should be a 2-tuple consisting of the local storage key of the object within its parent and the object itself. By default, only the objects are returned.
- **root_key** – The key to return with this object. Ignored when return_key is False.

Returns iterator of objects or (key,object) tuples

preorder_visit(*visit*, *active=None*, *include_key=False*, *root_key=None*)
Visits each node in the storage tree using a preorder traversal.

Parameters

- **visit** – A function that is called on each node in the storage tree. When the include_key keyword is False, the function signature should be *visit(node) -> [True|False]*. When the include_key keyword is True, the function signature should be *visit(key,node) -> [True|False]*. When the return value of the function evaluates to to True, this indicates that the traversal should continue with the children of the current node; otherwise, the traversal does not go below the current node.
- **active** (True/None) – Set to True to indicate that only active objects should be included. The default value of None indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **include_key** (*bool*) – Set to True to indicate that 2 arguments should be passed to the visit function, with the first being the local storage key of the object within its parent and the second being the object itself. By default, only the objects are passed to the function.
- **root_key** – The key to pass with this object. Ignored when include_key is False.

root_block
The root storage block above this object

7.1.4 List-like Object Storage

class pyomo.core.kernel.component_list.**ComponentList**(**args*)
Bases: *pyomo.core.kernel.component_tuple.ComponentTuple*, _abcoll.MutableSequence

A partial implementation of the IComponentContainer interface that presents list-like storage functionality.

Complete implementations need to set the _ctype property at the class level, declare the remaining required abstract properties of the IComponentContainer base class, and declare a slot or attribute named _data.

Note that this implementation allows nested storage of other IComponentContainer implementations that are defined with the same ctype.

__delattr__
x.__delattr__('name') <==> del x.name

__format__()
default object formatter

__getattribute__
x.__getattribute__('name') <==> x.name

__hash__

__metaclass__
alias of ABCMeta

__new__(*S*, ...) → a new object with type S, a subtype of T

__reduce__()
helper for pickle

__reduce_ex__()
helper for pickle

__repr__

__setattr__
x.__setattr__('name', value) <==> x.name = value

__sizeof__() → int
size of object in memory, in bytes

__str__()
Convert this object to a string by first attempting to generate its fully qualified name. If the object does not have a name (because it does not have a parent, then a string containing the class name is returned.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

__weakref__
list of weak references to the object (if defined)

append(*value*)
S.append(object) – append object to the end of the sequence

child(*key*)
Get the child object associated with a given storage key for this container.

Raises KeyError – if the argument is not a storage key for any children of this container

child_key(*child*)
Get the lookup key associated with a child of this container.

Raises ValueError – if the argument is not a child of this container

children(*return_key=False*)
Iterate over the children of this container.

Parameters **return_key** (*bool*) – Set to True to indicate that the return type should be a 2-tuple consisting of the child storage key and the child object. By default, only the child objects are returned.

Returns iterator of objects or (key,object) tuples

components(*active=None*, *return_key=False*)
Generates an efficient traversal of all components stored under this container. Components are leaf nodes in a storage tree (not containers themselves, except for blocks).

Parameters

- **active** (True/None) – Set to True to indicate that only active objects should be included. The default value of None indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **return_key** (*bool*) – Set to True to indicate that the return type should be a 2-tuple consisting of the local storage key of the object within its parent and the object itself. By default, only the objects are returned.

Returns iterator of objects or (key,object) tuples

count(*value*) → integer – return number of occurrences of value

extend(*values*)
S.extend(iterable) – extend sequence by appending elements from the iterable

generate_names(*active=None*, *descend_into=True*, *convert=<type 'str'>*, *prefix=''*)
Generate a container of fully qualified names (up to this container) for objects stored under this container.

Parameters

- **active** (True/None) – Set to True to indicate that only active components should be included. The default value of None indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **descend_into** (*bool*) – Indicates whether or not to include subcomponents of any container objects that are not components. Default is True.
- **convert** (*function*) – A function that converts a storage key into a string representation. Default is str.
- **prefix** (*str*) – A string to prefix names with.

Returns A component map that behaves as a dictionary mapping component objects to names.

getname(*fully_qualified=False*, *name_buffer={}*, *convert=<type 'str'>*)
Dynamically generates a name for this object.

Parameters

- **fully_qualified** (*bool*) – Generate a full name by iterating through all anscestor containers. Default is False.

- **convert** (*function*) – A function that converts a storage key into a string representation. Default is the built-in function str.

Returns If a parent exists, this method returns a string representing the name of the object in the context of its parent; otherwise (if no parent exists), this method returns `None`.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

index (*value*[, *start*[, *stop*]]) → integer – return first index of value.
Raises ValueError if the value is not present.

insert (*i*, *item*)
S.insert(index, object) – insert object before index

local_name
The object's local name within the context of its parent. Alias for *obj.getname(fully_qualified=False)*.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

name
The object's fully qualified name. Alias for *obj.getname(fully_qualified=True)*.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

parent
The object's parent

parent_block
The first ancestor block above this object

pop ([*index*]) → item – remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

postorder_traversal (*active=None*, *return_key=False*, *root_key=None*)
Generates a postorder traversal of the storage tree.

Parameters

- **active** (`True`/`None`) – Set to `True` to indicate that only active objects should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the local storage key of the object within its parent and the object itself. By default, only the objects are returned.
- **root_key** – The key to return with this object. Ignored when `return_key` is `False`.

Returns iterator of objects or (key,object) tuples

preorder_traversal (*active=None*, *return_key=False*, *root_key=None*)
Generates a preorder traversal of the storage tree.

Parameters

- **active** (`True`/`None`) – Set to `True` to indicate that only active objects should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the local storage key of the object within its parent and the object itself. By default, only the objects are returned.
- **root_key** – The key to return with this object. Ignored when `return_key` is `False`.

Returns iterator of objects or (key,object) tuples

preorder_visit(*visit*, *active=None*, *include_key=False*, *root_key=None*)
Visits each node in the storage tree using a preorder traversal.

Parameters

- **visit** – A function that is called on each node in the storage tree. When the `include_key` keyword is `False`, the function signature should be *visit(node) -> [True|False]*. When the `include_key` keyword is `True`, the function signature should be *visit(key,node) -> [True|False]*. When the return value of the function evaluates to to `True`, this indicates that the traversal should continue with the children of the current node; otherwise, the traversal does not go below the current node.
- **active** (`True`/`None`) – Set to `True` to indicate that only active objects should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **include_key** (*bool*) – Set to `True` to indicate that 2 arguments should be passed to the visit function, with the first being the local storage key of the object within its parent and the second being the object itself. By default, only the objects are passed to the function.
- **root_key** – The key to pass with this object. Ignored when `include_key` is `False`.

remove(*value*)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

reverse()
S.reverse() – reverse *IN PLACE*

root_block
The root storage block above this object

7.1.5 Dict-like Object Storage

class `pyomo.core.kernel.component_dict.`**ComponentDict**(**args*, ***kwds*)
Bases: *pyomo.core.kernel.component_interface._SimpleContainerMixin*, *pyomo.core.kernel.component_interface.IComponentContainer*, `_abcoll.MutableMapping`

A partial implementation of the IComponentContainer interface that presents dict-like storage functionality.

Complete implementations need to set the _ctype property at the class level, declare the remaining required abstract properties of the IComponentContainer base class, and declare a slot or attribute named _data.

Note that this implementation allows nested storage of other IComponentContainer implementations that are defined with the same ctype.

The optional keyword 'ordered' can be set to `True`/`False` to enable/disable the use of an OrderedDict as the underlying storage dictionary (default is `True`).

__delattr__
x.__delattr__('name') <==> del x.name

__format__()
default object formatter

__getattribute__
x.__getattribute__('name') <==> x.name

__metaclass__
alias of `ABCMeta`

__new__(*S*, ...) → a new object with type S, a subtype of T

__reduce__()
helper for pickle

__reduce_ex__()
helper for pickle

__repr__

__setattr__
x.__setattr__('name', value) <==> x.name = value

__sizeof__() → int
size of object in memory, in bytes

__str__()
Convert this object to a string by first attempting to generate its fully qualified name. If the object does not have a name (because it does not have a parent, then a string containing the class name is returned.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

__weakref__
list of weak references to the object (if defined)

child(*key*)
Get the child object associated with a given storage key for this container.

Raises `KeyError` – if the argument is not a storage key for any children of this container

child_key(*child*)
Get the lookup key associated with a child of this container.

Raises `ValueError` – if the argument is not a child of this container

children(*return_key=False*)
Iterate over the children of this container.

Parameters **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the child storage key and the child object. By default, only the child objects are returned.

Returns iterator of objects or (key,object) tuples

clear() → None. Remove all items from D.

components(*active=None*, *return_key=False*)
Generates an efficient traversal of all components stored under this container. Components are leaf nodes in a storage tree (not containers themselves, except for blocks).

Parameters

- **active** (`True`/`None`) – Set to `True` to indicate that only active objects should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the local storage key of the object within its parent and the object itself. By default, only the objects are returned.

Returns iterator of objects or (key,object) tuples

generate_names(*active=None*, *descend_into=True*, *convert=<type 'str'>*, *prefix=''*)
Generate a container of fully qualified names (up to this container) for objects stored under this container.

Parameters

- **active** (`True`/`None`) – Set to `True` to indicate that only active components should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **descend_into** (*bool*) – Indicates whether or not to include subcomponents of any container objects that are not components. Default is `True`.
- **convert** (*function*) – A function that converts a storage key into a string representation. Default is str.
- **prefix** (*str*) – A string to prefix names with.

Returns A component map that behaves as a dictionary mapping component objects to names.

get(*k*[, *d*]) → D[k] if k in D, else d. d defaults to None.

getname(*fully_qualified=False*, *name_buffer={}*, *convert=<type 'str'>*)
Dynamically generates a name for this object.

Parameters

- **fully_qualified** (*bool*) – Generate a full name by iterating through all anscestor containers. Default is `False`.
- **convert** (*function*) – A function that converts a storage key into a string representation. Default is the built-in function str.

Returns If a parent exists, this method returns a string representing the name of the object in the context of its parent; otherwise (if no parent exists), this method returns `None`.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

items() → list of D's (key, value) pairs, as 2-tuples

iteritems() → an iterator over the (key, value) items of D

iterkeys() → an iterator over the keys of D

itervalues() → an iterator over the values of D

keys() → list of D's keys

local_name
The object's local name within the context of its parent. Alias for *obj.getname(fully_qualified=False)*.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

name
The object's fully qualified name. Alias for *obj.getname(fully_qualified=True)*.

> **Warning:** Name generation can be slow. See the generate_names method, found on most containers, for a way to generate a static set of component names.

parent
The object's parent

parent_block
The first ancestor block above this object

pop(*k*[, *d*]) → v, remove specified key and return the corresponding value.
If key is not found, d is returned if given, otherwise KeyError is raised.

popitem() → (k, v), remove and return some (key, value) pair
as a 2-tuple; but raise KeyError if D is empty.

postorder_traversal(*active=None*, *return_key=False*, *root_key=None*)
Generates a postorder traversal of the storage tree.

Parameters

- **active** (`True`/`None`) – Set to `True` to indicate that only active objects should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the local storage key of the object within its parent and the object itself. By default, only the objects are returned.
- **root_key** – The key to return with this object. Ignored when `return_key` is `False`.

Returns iterator of objects or (key,object) tuples

preorder_traversal(*active=None*, *return_key=False*, *root_key=None*)
Generates a preorder traversal of the storage tree.

Parameters

- **active** (`True`/`None`) – Set to `True` to indicate that only active objects should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the local storage key of the object within its parent and the object itself. By default, only the objects are returned.
- **root_key** – The key to return with this object. Ignored when `return_key` is `False`.

Returns iterator of objects or (key,object) tuples

preorder_visit(*visit*, *active=None*, *include_key=False*, *root_key=None*)
Visits each node in the storage tree using a preorder traversal.

Parameters

- **visit** – A function that is called on each node in the storage tree. When the `include_key` keyword is `False`, the function signature should be *visit(node) -> [True|False]*. When the `include_key` keyword is `True`, the function signature should be *visit(key,node) -> [True|False]*. When the return value of the function evaluates to to `True`, this indicates that the traversal should continue with the children of the current node; otherwise, the traversal does not go below the current node.
- **active** (`True`/`None`) – Set to `True` to indicate that only active objects should be included. The default value of `None` indicates that all components (including those that have been deactivated) should be included. *Note*: This flag is ignored for any objects that do not have an active flag.
- **include_key** (*bool*) – Set to `True` to indicate that 2 arguments should be passed to the visit function, with the first being the local storage key of the object within its parent and the second being the object itself. By default, only the objects are passed to the function.
- **root_key** – The key to pass with this object. Ignored when `include_key` is `False`.

root_block
The root storage block above this object

setdefault(*k*[, *d*]) → D.get(k,d), also set D[k]=d if k not in D

update([*E*], ***F*) → None. Update D from mapping/iterable E and F.
If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v

values() → list of D's values

Modeling Objects:

7.1.6 Variables

Summary

pyomo.core.kernel.component_variable.variable([...])	A decision variable
pyomo.core.kernel.component_variable.variable_tuple(...)	A tuple-style container for variables.
pyomo.core.kernel.component_variable.create_variable_tuple(...)	Generates a full `variable_tuple`.
pyomo.core.kernel.component_variable.variable_list(...)	A list-style container for variables.
pyomo.core.kernel.component_variable.create_variable_list(...)	Generates a full `variable_list`.
pyomo.core.kernel.component_variable.variable_dict(...)	A dict-style container for variables.
pyomo.core.kernel.component_variable.create_variable_dict(...)	Generates a full `variable_dict`.

Member Documentation

class pyomo.core.kernel.component_variable.**variable**(*domain_type=None*, *domain=None*, *lb=None*, *ub=None*, *value=None*, *fixed=False*)

Bases: pyomo.core.kernel.component_variable.IVariable

A decision variable

Decision variables are used in objectives and constraints to define an optimization problem.

Parameters

- **domain_type** – Sets the domain type of the variable. Must be one of RealSet or IntegerSet. Can be updated later by assigning to the domain_type property. The default value of None is equivalent to RealSet, unless the domain keyword is used.
- **domain** – Sets the domain of the variable. This updates the domain_type, lb, and ub properties of the variable. The default value of None implies that this keyword is ignored. This keyword can not be used in combination with the domain_type keyword.
- **lb** – Sets the lower bound of the variable. Can be updated later by assigning to the lb property on the variable. Default is None, which is equivalent to -inf.
- **ub** – Sets the upper bound of the variable. Can be updated later by assigning to the ub property on the variable. Default is None, which is equivalent to +inf.
- **value** – Sets the value of the variable. Can be updated later by assigning to the value property on the variable. Default is None.
- **fixed** (*bool*) – Sets the fixed status of the variable. Can be updated later by assigning to the fixed property or by calling the fix() method. Default is False.

Examples

```
>>> # A continuous variable with infinite bounds
>>> x = pmo.variable()
>>> # A binary variable
>>> x = pmo.variable(domain=pmo.Binary)
>>> # Also a binary variable
>>> x = pmo.variable(domain_type=pmo.IntegerSet, lb=0, ub=1)
```

domain
Set the domain of the variable. This method updates the domain_type property and overwrites the lb and ub properties with the domain bounds.

domain_type
The domain type of the variable (RealSet or IntegerSet)

fixed
The fixed status of the variable

lb
The lower bound of the variable

stale
The stale status of the variable

ub
The upper bound of the variable

value
The value of the variable

class pyomo.core.kernel.component_variable.**variable_tuple**(**args*, ***kwds*)
Bases: *pyomo.core.kernel.component_tuple.ComponentTuple*

A tuple-style container for variables.

pyomo.core.kernel.component_variable.**create_variable_tuple**(*size*, **args*, ***kwds*)
Generates a full variable_tuple.

Parameters

- **size** (*int*) – The number of objects to place in the variable_tuple.
- **type** – The object type to populate the container with. Must have the same ctype as variable_tuple. Default: variable
- ***args** – arguments used to construct the objects placed in the container.
- ****kwds** – keywords used to construct the objects placed in the container.

Returns class:'variable_tuple'

Return type a fully populated

class pyomo.core.kernel.component_variable.**variable_list**(**args*, ***kwds*)
Bases: *pyomo.core.kernel.component_list.ComponentList*

A list-style container for variables.

pyomo.core.kernel.component_variable.**create_variable_list**(*size*, **args*, ***kwds*)
Generates a full variable_list.

Parameters

- **size** (*int*) – The number of objects to place in the variable_list.
- **type** – The object type to populate the container with. Must have the same ctype as variable_list. Default: variable
- ***args** – arguments used to construct the objects placed in the container.
- ****kwds** – keywords used to construct the objects placed in the container.

Returns a fully populated variable_list

class pyomo.core.kernel.component_variable.**variable_dict**(**args*, ***kwds*)
Bases: *pyomo.core.kernel.component_dict.ComponentDict*

A dict-style container for variables.

pyomo.core.kernel.component_variable.**create_variable_dict**(*keys*, **args*, ***kwds*)
Generates a full variable_dict.

Parameters

- **keys** – The set of keys to used to populate the variable_dict.
- **type** – The object type to populate the container with. Must have the same ctype as variable_dict. Default: variable
- ***args** – arguments used to construct the objects placed in the container.
- ****kwds** – keywords used to construct the objects placed in the container.

Returns a fully populated variable_dict

7.1.7 Constraint

Summary

pyomo.core.kernel.component_constraint.constraint([...])	A general algebraic constraint
pyomo.core.kernel.component_constraint.linear_constraint([...])	A linear constraint
pyomo.core.kernel.component_constraint.constraint_tuple(...)	A tuple-style container for constraints.
pyomo.core.kernel.component_constraint.constraint_list(...)	A list-style container for constraints.
pyomo.core.kernel.component_constraint.constraint_dict(...)	A dict-style container for constraints.
pyomo.core.kernel.component_matrix_constraint.matrix_constraint(A)	A container for constraints of the form L <= Ax <= b.

Member Documentation

class pyomo.core.kernel.component_constraint.**constraint**(*expr=None*, *body=None*, *lb=None*, *ub=None*, *rhs=None*)

Bases: pyomo.core.kernel.component_constraint._MutableBoundsConstraintMixin, pyomo.core.kernel.component_constraint.IConstraint

A general algebraic constraint

Algebraic constraints store relational expressions composed of linear or nonlinear functions involving decision variables.

Parameters

- **expr** – Sets the relational expression for the constraint. Can be updated later by assigning to the expr property on the constraint. When this keyword is used, values for the body, lb, ub, and rhs attributes are automatically determined based on the relational expression type. Default value is None.
- **body** – Sets the body of the constraint. Can be updated later by assigning to the body property on the constraint. Default is None. This keyword should not be used in combination with the expr keyword.
- **lb** – Sets the lower bound of the constraint. Can be updated later by assigning to the lb property on the constraint. Default is None, which is equivalent to -inf. This keyword should not be used in combination with the expr keyword.
- **ub** – Sets the upper bound of the constraint. Can be updated later by assigning to the ub property on the constraint. Default is None, which is equivalent to +inf. This keyword should not be used in combination with the expr keyword.
- **rhs** – Sets the right-hand side of the constraint. Can be updated later by assigning to the rhs property on the constraint. The default value of None implies that this keyword is ignored. Otherwise, use of this keyword implies that the equality property is set to True. This keyword should not be used in combination with the expr keyword.

Examples

```
>>> # A decision variable used to define constraints
>>> x = pmo.variable()
>>> # An upper bound constraint
>>> c = pmo.constraint(0.5*x <= 1)
>>> # (equivalent form)
>>> c = pmo.constraint(body=0.5*x, ub=1)
>>> # A range constraint
>>> c = pmo.constraint(lb=-1, body=0.5*x, ub=1)
>>> # An nonlinear equality constraint
>>> c = pmo.constraint(x**2 == 1)
>>> # (equivalent form)
>>> c = pmo.constraint(body=x**2, rhs=1)
```

body
The body of the constraint

expr
The full constraint expression –

- lb <= body <= ub: for range constraints
- lb <= body: for lower bounding constraints
- ub >= body: for upper bounding constraints
- body == rhs: for equality constraints

class pyomo.core.kernel.component_constraint.**linear_constraint**(*variables=None*, *coefficients=None*, *terms=None*, *lb=None*, *ub=None*, *rhs=None*)

Bases: pyomo.core.kernel.component_constraint._MutableBoundsConstraintMixin, pyomo.core.kernel.component_constraint.IConstraint

A linear constraint

A linear constraint stores a linear relational expression defined by a list of variables and coefficients. This class can be used to reduce build time and memory for an optimization model. It also increases the speed at which the model can be output to a solver.

Parameters

- **variables** (*list*) – Sets the list of variables in the linear expression defining the body of the constraint. Can be updated later by assigning to the `variables` property on the constraint.
- **coefficients** (*list*) – Sets the list of coefficients for the variables in the linear expression defining the body of the constraint. Can be updated later by assigning to the `coefficients` property on the constraint.
- **terms** (*list*) – An alternative way of initializing the `variables` and `coefficients` lists using an iterable of (variable, coefficient) tuples. Can be updated later by assigning to the `terms` property on the constraint. This keyword should not be used in combination with the `variables` or `coefficients` keywords.
- **lb** – Sets the lower bound of the constraint. Can be updated later by assigning to the `lb` property on the constraint. Default is `None`, which is equivalent to `-inf`.

- **ub** – Sets the upper bound of the constraint. Can be updated later by assigning to the `ub` property on the constraint. Default is `None`, which is equivalent to `+inf`.
- **rhs** – Sets the right-hand side of the constraint. Can be updated later by assigning to the `rhs` property on the constraint. The default value of `None` implies that this keyword is ignored. Otherwise, use of this keyword implies that the `equality` property is set to `True`.

Examples

```
>>> # Decision variables used to define constraints
>>> x = pmo.variable()
>>> y = pmo.variable()
>>> # An upper bound constraint
>>> c = pmo.constraint(variables=[x,y], coefficients=[1,2], ub=1)
>>> # (equivalent form)
>>> c = pmo.constraint(terms=[(x,1), (y,2)], ub=1)
>>> # (equivalent form using a general constraint)
>>> c = pmo.constraint(x + 2*y <= 1)
```

body
The body of the constraint

terms
An iterator over the terms in the body of this constraint as (variable, coefficient) tuples

class pyomo.core.kernel.component_constraint.**constraint_tuple**(**args*, ***kwds*)
Bases: *pyomo.core.kernel.component_tuple.ComponentTuple*, *pyomo.core.kernel.component_interface._ActiveComponentContainerMixin*

A tuple-style container for constraints.

class pyomo.core.kernel.component_constraint.**constraint_list**(**args*, ***kwds*)
Bases: *pyomo.core.kernel.component_list.ComponentList*, *pyomo.core.kernel.component_interface._ActiveComponentContainerMixin*

A list-style container for constraints.

class pyomo.core.kernel.component_constraint.**constraint_dict**(**args*, ***kwds*)
Bases: *pyomo.core.kernel.component_dict.ComponentDict*, *pyomo.core.kernel.component_interface._ActiveComponentContainerMixin*

A dict-style container for constraints.

class pyomo.core.kernel.component_matrix_constraint.**matrix_constraint**(*A*, *lb=None*, *ub=None*, *rhs=None*, *variable_order=None*, *sparse=True*)
Bases: *pyomo.core.kernel.component_constraint.constraint_tuple*

A container for constraints of the form L <= Ax <= b.

Parameters

- **A** – A scipy sparse matrix or 2D numpy array (always copied)

- **lb** – A scalar or array with the same number of rows as A that is set to the lower bound of the constraints
- **ub** – A scalar or array with the same number of rows as A that is set to the upper bound of the constraints
- **rhs** – A scalar or array with the same number of rows as A that is set to the right-hand side the constraints (implies equality constraints)
- **variable_order** – A list with the same number of columns as A that stores the variable associated with each column
- **sparse** – Indicates whether or not sparse storage (CSR format) should be used to store A. Default is `True`.

equality
The array of boolean entries indicating the indices that are equality constraints

lb
The array of constraint lower bounds

lslack
Lower slack (body - lb)

rhs
The array of constraint right-hand sides. Can be set to a scalar or a numpy array of the same dimension. This property can only be read when the equality property is `True` on every index. Assigning to this property implicitly sets the equality property to `True` on every index.

slack
min(lslack, uslack)

sparse
Boolean indicating whether or not the underlying matrix uses sparse storage

ub
The array of constraint upper bounds

uslack
Upper slack (ub - body)

variable_order
The list of variables associated with the columns of the constraint matrix

7.1.8 Parameters

Summary

pyomo.core.kernel.component_parameter.parameter([value])	A placeholder for a mutable, numeric value.
pyomo.core.kernel.component_parameter.parameter_tuple(...)	A tuple-style container for parameters.
pyomo.core.kernel.component_parameter.parameter_list(...)	A list-style container for parameters.
pyomo.core.kernel.component_parameter.parameter_dict(...)	A dict-style container for parameters.

Member Documentation

class pyomo.core.kernel.component_parameter.**parameter**(*value=None*)
Bases: pyomo.core.kernel.component_parameter.IParameter

A placeholder for a mutable, numeric value.

value
The value of the paramater

class pyomo.core.kernel.component_parameter.**parameter_tuple**(**args*, ***kwds*)
Bases: *pyomo.core.kernel.component_tuple.ComponentTuple*

A tuple-style container for parameters.

class pyomo.core.kernel.component_parameter.**parameter_list**(**args*, ***kwds*)
Bases: *pyomo.core.kernel.component_list.ComponentList*

A list-style container for parameters.

class pyomo.core.kernel.component_parameter.**parameter_dict**(**args*, ***kwds*)
Bases: *pyomo.core.kernel.component_dict.ComponentDict*

A dict-style container for parameters.

7.1.9 Objectives

Summary

pyomo.core.kernel.component_objective.objective([...])	An optimization objective.
pyomo.core.kernel.component_objective.objective_tuple(...)	A tuple-style container for objectives.
pyomo.core.kernel.component_objective.objective_list(...)	A list-style container for objectives.
pyomo.core.kernel.component_objective.objective_dict(...)	A dict-style container for objectives.

Member Documentation

class pyomo.core.kernel.component_objective.**objective**(*expr=None*, *sense=1*)
Bases: pyomo.core.kernel.component_objective.IObjective

An optimization objective.

class pyomo.core.kernel.component_objective.**objective_tuple**(**args*, ***kwds*)
Bases: *pyomo.core.kernel.component_tuple.ComponentTuple*, *pyomo.core.kernel.component_interface._ActiveComponentContainerMixin*

A tuple-style container for objectives.

class pyomo.core.kernel.component_objective.**objective_list**(**args*, ***kwds*)
Bases: *pyomo.core.kernel.component_list.ComponentList*, *pyomo.core.kernel.component_interface._ActiveComponentContainerMixin*

A list-style container for objectives.

class pyomo.core.kernel.component_objective.**objective_dict**(**args*, ***kwds*)

Bases: *pyomo.core.kernel.component_dict.ComponentDict*, *pyomo.core.kernel.component_interface._ActiveComponentContainerMixin*

A dict-style container for objectives.

7.1.10 Expressions

Summary

pyomo.core.kernel.component_expression.expression([expr])	A named, mutable expression.
pyomo.core.kernel.component_expression.expression_tuple(...)	A tuple-style container for expressions.
pyomo.core.kernel.component_expression.expression_list(...)	A list-style container for expressions.
pyomo.core.kernel.component_expression.expression_dict(...)	A dict-style container for expressions.

Member Documentation

class pyomo.core.kernel.component_expression.**expression**(*expr=None*)
Bases: pyomo.core.kernel.component_expression.IExpression

A named, mutable expression.

class pyomo.core.kernel.component_expression.**expression_tuple**(**args*, ***kwds*)
Bases: *pyomo.core.kernel.component_tuple.ComponentTuple*

A tuple-style container for expressions.

class pyomo.core.kernel.component_expression.**expression_list**(**args*, ***kwds*)
Bases: *pyomo.core.kernel.component_list.ComponentList*

A list-style container for expressions.

class pyomo.core.kernel.component_expression.**expression_dict**(**args*, ***kwds*)
Bases: *pyomo.core.kernel.component_dict.ComponentDict*

A dict-style container for expressions.

7.1.11 Special Ordered Sets

Summary

pyomo.core.kernel.component_sos.sos(variables)	A Special Ordered Set of type n.
pyomo.core.kernel.component_sos.sos1(variables)	A Special Ordered Set of type 1.
pyomo.core.kernel.component_sos.sos2(variables)	A Special Ordered Set of type 2.
pyomo.core.kernel.component_sos.sos_tuple(...)	A tuple-style container for Special Ordered Sets.

Continued on next page

Table 7.6 – continued from previous page

pyomo.core.kernel.component_sos.sos_list(...)	A list-style container for Special Ordered Sets.
pyomo.core.kernel.component_sos.sos_dict(...)	A dict-style container for Special Ordered Sets.

Member Documentation

class pyomo.core.kernel.component_sos.**sos**(*variables*, *weights=None*, *level=1*)
Bases: pyomo.core.kernel.component_sos.ISOS

A Special Ordered Set of type n.

pyomo.core.kernel.component_sos.**sos1**(*variables*, *weights=None*)
A Special Ordered Set of type 1.

This is an alias for sos(..., level=1)

pyomo.core.kernel.component_sos.**sos2**(*variables*, *weights=None*)
A Special Ordered Set of type 2.

This is an alias for sos(..., level=2).

class pyomo.core.kernel.component_sos.**sos_tuple**(**args*, ***kwds*)
Bases: *pyomo.core.kernel.component_tuple.ComponentTuple*, *pyomo.core.kernel.component_interface._ActiveComponentContainerMixin*

A tuple-style container for Special Ordered Sets.

class pyomo.core.kernel.component_sos.**sos_list**(**args*, ***kwds*)
Bases: *pyomo.core.kernel.component_list.ComponentList*, *pyomo.core.kernel.component_interface._ActiveComponentContainerMixin*

A list-style container for Special Ordered Sets.

class pyomo.core.kernel.component_sos.**sos_dict**(**args*, ***kwds*)
Bases: *pyomo.core.kernel.component_dict.ComponentDict*, *pyomo.core.kernel.component_interface._ActiveComponentContainerMixin*

A dict-style container for Special Ordered Sets.

7.1.12 Suffixes

pyomo.core.kernel.component_suffix.**export_suffix_generator**(*blk*, *datatype=<object object>*, *active=None*, *descend_into=True*, *return_key=False*)
Generates an efficient traversal of all suffixes that have been declared for exporting data.

Parameters

- **blk** – A block object.
- **datatype** – Restricts the suffixes included in the returned generator to those matching the provided suffix datatype.
- **active** (True/None) – Set to True to indicate that only active suffixes should be included. The default value of None indicates that all suffixes (including those that have been deactivated) should be included.

- **descend_into** (*bool*) – Indicates whether or not to include suffixes on sub-blocks. Default is `True`.
- **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the local storage key of the suffix within its parent and the suffix itself. By default, only the suffixes are returned.

Returns iterator of suffixes or (key,suffix) tuples

pyomo.core.kernel.component_suffix.**import_suffix_generator**(*blk*, *datatype=<object object>*, *active=None*, *descend_into=True*, *return_key=False*)

Generates an efficient traversal of all suffixes that have been declared for importing data.

Parameters

- **blk** – A block object.
- **datatype** – Restricts the suffixes included in the returned generator to those matching the provided suffix datatype.
- **active** (`True`/`None`) – Set to `True` to indicate that only active suffixes should be included. The default value of `None` indicates that all suffixes (including those that have been deactivated) should be included.
- **descend_into** (*bool*) – Indicates whether or not to include suffixes on sub-blocks. Default is `True`.
- **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the local storage key of the suffix within its parent and the suffix itself. By default, only the suffixes are returned.

Returns iterator of suffixes or (key,suffix) tuples

pyomo.core.kernel.component_suffix.**local_suffix_generator**(*blk*, *datatype=<object object>*, *active=None*, *descend_into=True*, *return_key=False*)

Generates an efficient traversal of all suffixes that have been declared local data storage.

Parameters

- **blk** – A block object.
- **datatype** – Restricts the suffixes included in the returned generator to those matching the provided suffix datatype.
- **active** (`True`/`None`) – Set to `True` to indicate that only active suffixes should be included. The default value of `None` indicates that all suffixes (including those that have been deactivated) should be included.
- **descend_into** (*bool*) – Indicates whether or not to include suffixes on sub-blocks. Default is `True`.
- **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the local storage key of the suffix within its parent and the suffix itself. By default, only the suffixes are returned.

Returns iterator of suffixes or (key,suffix) tuples

class pyomo.core.kernel.component_suffix.**suffix**(**args*, ***kwds*)

Bases: pyomo.core.kernel.component_map.ComponentMap, *pyomo.core.kernel.*

component_interface.IComponent, *pyomo.core.kernel.component_interface._ActiveComponentMixin*

A container for storing extraneous model data that can be imported to or exported from a solver.

datatype
Return the suffix datatype.

direction
Return the suffix direction.

export_enabled
Returns `True` when this suffix is enabled for export to solvers.

import_enabled
Returns `True` when this suffix is enabled for import from solutions.

pyomo.core.kernel.component_suffix.**suffix_generator**(*blk*, *datatype=<object object>*, *active=None*, *descend_into=True*, *return_key=False*)

Generates an efficient traversal of all suffixes that have been declared.

Parameters

- **blk** – A block object.
- **datatype** – Restricts the suffixes included in the returned generator to those matching the provided suffix datatype.
- **active** (`True`/`None`) – Set to `True` to indicate that only active suffixes should be included. The default value of `None` indicates that all suffixes (including those that have been deactivated) should be included.
- **descend_into** (*bool*) – Indicates whether or not to include suffixes on sub-blocks. Default is `True`.
- **return_key** (*bool*) – Set to `True` to indicate that the return type should be a 2-tuple consisting of the local storage key of the suffix within its parent and the suffix itself. By default, only the suffixes are returned.

Returns iterator of suffixes or (key,suffix) tuples

7.1.13 Piecewise Function Library

Modules

Single-variate Piecewise Functions

Summary

pyomo.core.kernel.component_piecewise.transforms.piecewise(...)	Models a single-variate piecewise linear function.
pyomo.core.kernel.component_piecewise.transforms.PiecewiseLinearFunction(...)	A piecewise linear function

Continued on next page

Table 7.7 – continued from previous page

pyomo.core.kernel.component_piecewise.transforms.TransformedPiecewiseLinearFunction(f)	Base class for transformed piecewise linear functions
pyomo.core.kernel.component_piecewise.transforms.piecewise_convex(...)	Simple convex piecewise representation
pyomo.core.kernel.component_piecewise.transforms.piecewise_sos2(...)	Discrete SOS2 piecewise representation
pyomo.core.kernel.component_piecewise.transforms.piecewise_dcc(...)	Discrete DCC piecewise representation
pyomo.core.kernel.component_piecewise.transforms.piecewise_cc(...)	Discrete CC piecewise representation
pyomo.core.kernel.component_piecewise.transforms.piecewise_mc(...)	Discrete MC piecewise representation
pyomo.core.kernel.component_piecewise.transforms.piecewise_inc(...)	Discrete INC piecewise representation
pyomo.core.kernel.component_piecewise.transforms.piecewise_dlog(...)	Discrete DLOG piecewise representation
pyomo.core.kernel.component_piecewise.transforms.piecewise_log(...)	Discrete LOG piecewise representation

Member Documentation

pyomo.core.kernel.component_piecewise.transforms.**piecewise**(*breakpoints*, *values*, *input=None*, *output=None*, *bound='eq'*, *repn='sos2'*, *validate=True*, *simplify=True*, *equal_slopes_tolerance=1e-06*, *require_bounded_input_variable=True*, *require_variable_domain_coverage=True*)

Models a single-variate piecewise linear function.

This function takes a list breakpoints and function values describing a piecewise linear function and transforms this input data into a block of variables and constraints that enforce a piecewise linear relationship between an input variable and an output variable. In the general case, this transformation requires the use of discrete decision variables.

Parameters

- **breakpoints** (*list*) – The list of breakpoints of the piecewise linear function. This can be a list of numbers or a list of objects that store mutable data (e.g., mutable parameters). If mutable data is used validation might need to be disabled by setting the `validate` keyword to `False`. The list of breakpoints must be in non-decreasing order.
- **values** (*list*) – The values of the piecewise linear function corresponding to the breakpoints.
- **input** – The variable constrained to be the input of the piecewise linear function.
- **output** – The variable constrained to be the output of the piecewise linear function.

- **bound** (*str*) – The type of bound to impose on the output expression. Can be one of:
  - 'lb': y <= f(x)
  - 'eq': y = f(x)
  - 'ub': y >= f(x)
- **repn** (*str*) – The type of piecewise representation to use. Choices are shown below (+ means step functions are supported)
  - 'sos2': standard representation using sos2 constraints (+)
  - 'dcc': disaggregated convex combination (+)
  - 'dlog': logarithmic disaggregated convex combination (+)
  - 'cc': convex combination (+)
  - 'log': logarithmic branching convex combination (+)
  - 'mc': multiple choice
  - 'inc': incremental method (+)
- **validate** (*bool*) – Indicates whether or not to perform validation of the input data. The default is `True`. Validation can be performed manually after the piecewise object is created by calling the `validate()` method. Validation should be performed any time the inputs are changed (e.g., when using mutable parameters in the breakpoints list or when the input variable changes).
- **simplify** (*bool*) – Indicates whether or not to attempt to simplify the piecewise representation to avoid using discrete variables. This can be done when the feasible region for the output variable, with respect to the piecewise function and the bound type, is a convex set. Default is `True`. Validation is required to perform simplification, so this keyword is ignored when the `validate` keyword is `False`.
- **equal_slopes_tolerance** (*float*) – Tolerance used check if consecutive slopes are nearly equal. If any are found, validation will fail. Default is 1e-6. This keyword is ignored when the `validate` keyword is `False`.
- **require_bounded_input_variable** (*bool*) – Indicates if the input variable is required to have finite upper and lower bounds. Default is `True`. Setting this keyword to `False` can be used to allow general expressions to be used as the input in place of a variable. This keyword is ignored when the `validate` keyword is `False`.
- **require_variable_domain_coverage** (*bool*) – Indicates if the function domain (defined by the endpoints of the breakpoints list) needs to cover the entire domain of the input variable. Default is `True`. Ignored for any bounds of variables that are not finite, or when the input is not assigned a variable. This keyword is ignored when the `validate` keyword is `False`.

Returns a block that stores any new variables, constraints, and other components used by the piecewise representation

Return type *TransformedPiecewiseLinearFunction*

class pyomo.core.kernel.component_piecewise.transforms.**PiecewiseLinearFunction**(*breakpoints*, *values*, *validate=True*, ***kwds*)

Bases: `object`

A piecewise linear function

Piecewise linear functions are defined by a list of breakpoints and a list function values corresponding to each breakpoint. The function value between breakpoints is implied through linear interpolation.

Parameters

- **breakpoints** (*list*) – The list of function breakpoints.
- **values** (*list*) – The list of function values (one for each breakpoint).
- **validate** (*bool*) – Indicates whether or not to perform validation of the input data. The default is `True`. Validation can be performed manually after the piecewise object is created by calling the `validate()` method. Validation should be performed any time the inputs are changed (e.g., when using mutable parameters in the breakpoints list).
- ****kwds** – Additional keywords are passed to the `validate()` method when the `validate` keyword is `True`; otherwise, they are ignored.

__call__(*x*)
Evaluates the piecewise linear function at the given point using interpolation

breakpoints
The set of breakpoints used to defined this function

validate(*equal_slopes_tolerance=1e-06*)
Validate this piecewise linear function by verifying various properties of the breakpoints and values lists (e.g., that the list of breakpoints is nondecreasing).

Parameters **equal_slopes_tolerance** (*float*) – Tolerance used check if consecutive slopes are nearly equal. If any are found, validation will fail. Default is 1e-6.

Returns a function characterization code (see `util.characterize_function()`)

Return type int

Raises `PiecewiseValidationError` – if validation fails

values
The set of values used to defined this function

class pyomo.core.kernel.component_piecewise.transforms.**TransformedPiecewiseLinearFunction**(*f*, *input=None*, *output=None*, *bound=None*, *validate=None*, ***kwds*)

Bases: *pyomo.core.kernel.component_block.tiny_block*

Base class for transformed piecewise linear functions

A transformed piecewise linear functions is a block of variables and constraints that enforce a piecewise linear relationship between an input variable and an output variable.

Parameters

- **f** (`PiecewiseLinearFunction`) – The piecewise linear function to transform.
- **input** – The variable constrained to be the input of the piecewise linear function.

- **output** – The variable constrained to be the output of the piecewise linear function.
- **bound** (*str*) – The type of bound to impose on the output expression. Can be one of:
  - 'lb': y <= f(x)
  - 'eq': y = f(x)
  - 'ub': y >= f(x)
- **validate** (*bool*) – Indicates whether or not to perform validation of the input data. The default is `True`. Validation can be performed manually after the piecewise object is created by calling the `validate()` method. Validation should be performed any time the inputs are changed (e.g., when using mutable parameters in the breakpoints list or when the input variable changes).
- ****kwds** – Additional keywords are passed to the `validate()` method when the `validate` keyword is `True`; otherwise, they are ignored.

__call__(*x*)
Evaluates the piecewise linear function at the given point using interpolation

bound
The bound type assigned to the piecewise relationship ('lb','ub','eq').

breakpoints
The set of breakpoints used to defined this function

input
The expression that stores the input to the piecewise function. The returned object can be updated by assigning to its `expr` attribute.

output
The expression that stores the output of the piecewise function. The returned object can be updated by assigning to its `expr` attribute.

validate(*equal_slopes_tolerance=1e-06*, *require_bounded_input_variable=True*, *require_variable_domain_coverage=True*)
Validate this piecewise linear function by verifying various properties of the breakpoints, values, and input variable (e.g., that the list of breakpoints is nondecreasing).

Parameters

- **equal_slopes_tolerance** (*float*) – Tolerance used check if consecutive slopes are nearly equal. If any are found, validation will fail. Default is 1e-6.
- **require_bounded_input_variable** (*bool*) – Indicates if the input variable is required to have finite upper and lower bounds. Default is `True`. Setting this keyword to `False` can be used to allow general expressions to be used as the input in place of a variable.
- **require_variable_domain_coverage** (*bool*) – Indicates if the function domain (defined by the endpoints of the breakpoints list) needs to cover the entire domain of the input variable. Default is `True`. Ignored for any bounds of variables that are not finite, or when the input is not assigned a variable.

Returns a function characterization code (see `util.characterize_function()`)

Return type int

Raises `PiecewiseValidationError` – if validation fails

values
The set of values used to defined this function

class pyomo.core.kernel.component_piecewise.transforms.**piecewise_convex**(**args*, ***kwds*)

Bases: *pyomo.core.kernel.component_piecewise.transforms.TransformedPiecewiseLinearFunction*

Simple convex piecewise representation

Expresses a piecewise linear function with a convex feasible region for the output variable using a simple collection of linear constraints.

validate(***kwds*)

Validate this piecewise linear function by verifying various properties of the breakpoints, values, and input variable (e.g., that the list of breakpoints is nondecreasing).

See base class documentation for keyword descriptions.

class pyomo.core.kernel.component_piecewise.transforms.**piecewise_sos2**(**args*, ***kwds*)

Bases: *pyomo.core.kernel.component_piecewise.transforms.TransformedPiecewiseLinearFunction*

Discrete SOS2 piecewise representation

Expresses a piecewise linear function using the SOS2 formulation.

validate(***kwds*)

Validate this piecewise linear function by verifying various properties of the breakpoints, values, and input variable (e.g., that the list of breakpoints is nondecreasing).

See base class documentation for keyword descriptions.

class pyomo.core.kernel.component_piecewise.transforms.**piecewise_dcc**(**args*, ***kwds*)

Bases: *pyomo.core.kernel.component_piecewise.transforms.TransformedPiecewiseLinearFunction*

Discrete DCC piecewise representation

Expresses a piecewise linear function using the DCC formulation.

validate(***kwds*)

Validate this piecewise linear function by verifying various properties of the breakpoints, values, and input variable (e.g., that the list of breakpoints is nondecreasing).

See base class documentation for keyword descriptions.

class pyomo.core.kernel.component_piecewise.transforms.**piecewise_cc**(**args*, ***kwds*)

Bases: *pyomo.core.kernel.component_piecewise.transforms.TransformedPiecewiseLinearFunction*

Discrete CC piecewise representation

Expresses a piecewise linear function using the CC formulation.

validate(***kwds*)

Validate this piecewise linear function by verifying various properties of the breakpoints, values, and input variable (e.g., that the list of breakpoints is nondecreasing).

See base class documentation for keyword descriptions.

class pyomo.core.kernel.component_piecewise.transforms.**piecewise_mc**(**args*, ***kwds*)

Bases: *pyomo.core.kernel.component_piecewise.transforms.TransformedPiecewiseLinearFunction*

Discrete MC piecewise representation

Expresses a piecewise linear function using the MC formulation.

validate(***kwds*)
Validate this piecewise linear function by verifying various properties of the breakpoints, values, and input variable (e.g., that the list of breakpoints is nondecreasing).

See base class documentation for keyword descriptions.

class pyomo.core.kernel.component_piecewise.transforms.**piecewise_inc**(**args*, ***kwds*)
Bases: *pyomo.core.kernel.component_piecewise.transforms.TransformedPiecewiseLinearFunction*

Discrete INC piecewise representation

Expresses a piecewise linear function using the INC formulation.

validate(***kwds*)
Validate this piecewise linear function by verifying various properties of the breakpoints, values, and input variable (e.g., that the list of breakpoints is nondecreasing).

See base class documentation for keyword descriptions.

class pyomo.core.kernel.component_piecewise.transforms.**piecewise_dlog**(**args*, ***kwds*)
Bases: *pyomo.core.kernel.component_piecewise.transforms.TransformedPiecewiseLinearFunction*

Discrete DLOG piecewise representation

Expresses a piecewise linear function using the DLOG formulation. This formulation uses logarithmic number of discrete variables in terms of number of breakpoints.

validate(***kwds*)
Validate this piecewise linear function by verifying various properties of the breakpoints, values, and input variable (e.g., that the list of breakpoints is nondecreasing).

See base class documentation for keyword descriptions.

class pyomo.core.kernel.component_piecewise.transforms.**piecewise_log**(**args*, ***kwds*)
Bases: *pyomo.core.kernel.component_piecewise.transforms.TransformedPiecewiseLinearFunction*

Discrete LOG piecewise representation

Expresses a piecewise linear function using the LOG formulation. This formulation uses logarithmic number of discrete variables in terms of number of breakpoints.

validate(***kwds*)
Validate this piecewise linear function by verifying various properties of the breakpoints, values, and input variable (e.g., that the list of breakpoints is nondecreasing).

See base class documentation for keyword descriptions.

Multi-variate Piecewise Functions

Summary

pyomo.core.kernel.component_piecewise.transforms_nd.piecewise_nd(...)	Models a multi-variate piecewise linear function.
pyomo.core.kernel.component_piecewise.transforms_nd.PiecewiseLinearFunctionND(...)	A multi-variate piecewise linear function
pyomo.core.kernel.component_piecewise.transforms_nd.TransformedPiecewiseLinearFunctionND(f)	Base class for transformed multi-variate piecewise
pyomo.core.kernel.component_piecewise.transforms_nd.piecewise_nd_cc(...)	Discrete CC multi-variate piecewise representation

Member Documentation

pyomo.core.kernel.component_piecewise.transforms_nd.**piecewise_nd**(*tri*, *values*, *input=None*, *output=None*, *bound='eq'*, *repn='cc'*)

Models a multi-variate piecewise linear function.

This function takes a D-dimensional triangulation and a list of function values associated with the points of the triangulation and transforms this input data into a block of variables and constraints that enforce a piecewise linear relationship between an D-dimensional vector of input variable and a single output variable. In the general case, this transformation requires the use of discrete decision variables.

Parameters

- **tri** (*scipy.spatial.Delaunay*) – A triangulation over the discretized variable domain. Can be generated using a list of variables using the utility function `util.generate_delaunay()`. Required attributes:
  - points: An (npoints, D) shaped array listing the D-dimensional coordinates of the discretization points.
  - simplices: An (nsimplices, D+1) shaped array of integers specifying the D+1 indices of the points vector that define each simplex of the triangulation.
- **values** (*numpy.array*) – An (npoints,) shaped array of the values of the piecewise function at each of coordinates in the triangulation points array.
- **input** – A D-length list of variables or expressions bound as the inputs of the piecewise function.
- **output** – The variable constrained to be the output of the piecewise linear function.
- **bound** (*str*) – The type of bound to impose on the output expression. Can be one of:
  - 'lb': y <= f(x)
  - 'eq': y = f(x)
  - 'ub': y >= f(x)
- **repn** (*str*) – The type of piecewise representation to use. Can be one of:
  - 'cc': convex combination

Returns a block containing any new variables, constraints, and other components used by the piecewise representation

Return type *TransformedPiecewiseLinearFunctionND*

class pyomo.core.kernel.component_piecewise.transforms_nd.**PiecewiseLinearFunctionND**(*tri*, *values*, *validate=True*, ***kwds*)

Bases: object

A multi-variate piecewise linear function

Multi-varite piecewise linear functions are defined by a triangulation over a finite domain and a list of function values associated with the points of the triangulation. The function value between points in the triangulation is implied through linear interpolation.

Parameters

- **tri** (*scipy.spatial.Delaunay*) – A triangulation over the discretized variable domain. Can be generated using a list of variables using the utility function util.generate_delaunay(). Required attributes:
  - points: An (npoints, D) shaped array listing the D-dimensional coordinates of the discretization points.
  - simplices: An (nsimplices, D+1) shaped array of integers specifying the D+1 indices of the points vector that define each simplex of the triangulation.
- **values** (*numpy.array*) – An (npoints,) shaped array of the values of the piecewise function at each of coordinates in the triangulation points array.

__call__(*x*)

Evaluates the piecewise linear function using interpolation. This method supports vectorized function calls as the interpolation process can be expensive for high dimensional data.

For the case when a single point is provided, the argument x should be a (D,) shaped numpy array or list, where D is the dimension of points in the triangulation.

For the vectorized case, the argument x should be a (n,D)-shaped numpy array.

triangulation

The triangulation over the domain of this function

values

The set of values used to defined this function

class pyomo.core.kernel.component_piecewise.transforms_nd.**TransformedPiecewiseLinearFunctionN**

Bases: *pyomo.core.kernel.component_block.tiny_block*

Base class for transformed multi-variate piecewise linear functions

A transformed multi-variate piecewise linear functions is a block of variables and constraints that enforce a piecewise linear relationship between an vector input variables and a single output variable.

Parameters

- **f** (PiecewiseLinearFunctionND) – The multi-variate piecewise linear function to transform.
- **input** – The variable constrained to be the input of the piecewise linear function.
- **output** – The variable constrained to be the output of the piecewise linear function.
- **bound** (*str*) – The type of bound to impose on the output expression. Can be one of:
  - 'lb': y <= f(x)
  - 'eq': y = f(x)
  - 'ub': y >= f(x)

__call__(*x*)

Evaluates the piecewise linear function using interpolation. This method supports vectorized function calls as the interpolation process can be expensive for high dimensional data.

For the case when a single point is provided, the argument x should be a (D,) shaped numpy array or list, where D is the dimension of points in the triangulation.

For the vectorized case, the argument x should be a (n,D)-shaped numpy array.

bound

The bound type assigned to the piecewise relationship ('lb','ub','eq').

input

The tuple of expressions that store the inputs to the piecewise function. The returned objects can be updated by assigning to their expr attribute.

output

The expression that stores the output of the piecewise function. The returned object can be updated by assigning to its expr attribute.

triangulation

The triangulation over the domain of this function

values

The set of values used to defined this function

class pyomo.core.kernel.component_piecewise.transforms_nd.**piecewise_nd_cc**(**args*, ***kwds*)

Bases: *pyomo.core.kernel.component_piecewise.transforms_nd.TransformedPiecewiseLinearFunctionND*

Discrete CC multi-variate piecewise representation

Expresses a multi-variate piecewise linear function using the CC formulation.

Utilities for Piecewise Functions

exception pyomo.core.kernel.component_piecewise.util.**PiecewiseValidationError**

Bases: exceptions.Exception

An exception raised when validation of piecewise linear functions fail.

pyomo.core.kernel.component_piecewise.util.**characterize_function**(*breakpoints*, *values*)

Characterizes a piecewise linear function described by a list of breakpoints and function values.

Parameters

- **breakpoints** (*list*) – The list of breakpoints of the piecewise linear function. It is assumed that the list of breakpoints is in non-decreasing order.
- **values** (*list*) – The values of the piecewise linear function corresponding to the breakpoints.

Returns a function characterization code and the list of slopes.

Return type (int, list)

Note: The function characterization codes are

- 1: affine
- 2: convex
- 3: concave
- 4: step
- 5: other

If the function has step points, some of the slopes may be `None`.

---

pyomo.core.kernel.component_piecewise.util.**generate_delaunay**(*variables*, *num=10*, ***kwds*)

Generate a Delaunay triangulation of the D-dimensional bounded variable domain given a list of D variables.

Requires numpy and scipy.

Parameters

- **variables** – A list of variables, each having a finite upper and lower bound.
- **num** (*int*) – The number of grid points to generate for each variable (default=10).
- ****kwds** – All additional keywords are passed to the scipy.spatial.Delaunay constructor.

Returns A scipy.spatial.Delaunay object.

pyomo.core.kernel.component_piecewise.util.**generate_gray_code**(*nbits*)

Generates a Gray code of nbits as list of lists

pyomo.core.kernel.component_piecewise.util.**is_constant**(*vals*)

Checks if a list of points is constant

pyomo.core.kernel.component_piecewise.util.**is_nondecreasing**(*vals*)

Checks if a list of points is nondecreasing

pyomo.core.kernel.component_piecewise.util.**is_nonincreasing**(*vals*)

Checks if a list of points is nonincreasing

pyomo.core.kernel.component_piecewise.util.**is_positive_power_of_two**(*x*)

Checks if a number is a nonzero and positive power of 2

pyomo.core.kernel.component_piecewise.util.**log2floor**(*n*)

Computes the exact value of floor(log2(n)) without using floating point calculations. Input argument must be a positive integer.

7.2 AML Library Reference

Under construction...

7.3 Solver Interfaces

7.3.1 GurobiPersistent

Methods

GurobiPersistent.activate()	Register this plugin with all interfaces that it implements.
GurobiPersistent.add_block(block)	Add a Pyomo Block to the solver's model.
GurobiPersistent.add_constraint(con)	Add a constraint to the solver's model.
GurobiPersistent.set_objective(obj)	Set the solver's objective.
GurobiPersistent.add_sos_constraint(con)	Add an SOS constraint to the solver's model (if supported).
GurobiPersistent.add_var(var)	Add a variable to the solver's model.
GurobiPersistent.alias(name[, doc, subclass])	This function is used to declare aliases that can be used by a factory for constructing plugin instances.
GurobiPersistent.available([exception_flag])	True if the solver is available.
GurobiPersistent.deactivate()	Unregister this plugin with all interfaces that it implements.
GurobiPersistent.disable()	Disable this plugin
GurobiPersistent.enable()	Enable this plugin
GurobiPersistent.enabled()	Return value indicating if this plugin is enabled
GurobiPersistent.has_capability(cap)	Returns a boolean value representing whether a solver supports a specific feature.
GurobiPersistent.has_instance()	True if set_instance has been called and this solver interface has a pyomo model and a solver model.
GurobiPersistent.implements(interface[, ...])	Can be used in the class definition of *Plugin* subclasses to declare the extension points that are implemented by this interface class.
GurobiPersistent.load_vars([vars_to_load])	Load the values from the solver's variables into the corresponding pyomo variables.
GurobiPersistent.problem_format()	Returns the current problem format.
GurobiPersistent.remove_block(block)	Remove a block from the solver's model.
GurobiPersistent.remove_constraint(con)	Remove a constraint from the solver's model.
GurobiPersistent.remove_sos_constraint(con)	Remove an SOS constraint from the solver's model.
GurobiPersistent.remove_var(var)	Remove a variable from the solver's model.
GurobiPersistent.reset()	Reset the state of the solver
GurobiPersistent.results_format()	Returns the current results format.
GurobiPersistent.set_callback(name[, ...])	Set the callback function for a named callback.
GurobiPersistent.set_instance(model, **kwds)	This method is used to translate the Pyomo model provided to an instance of the solver's Python model.
GurobiPersistent.set_problem_format(format)	Set the current problem format (if it's valid) and update the results format to something valid for this problem format.
GurobiPersistent.set_results_format(format)	Set the current results format (if it's valid for the current problem format).
GurobiPersistent.solve(*args, **kwds)	Solve the model.
GurobiPersistent.update_var(var)	Update a variable in the solver's model.
GurobiPersistent.version()	Returns a 4-tuple describing the solver executable version.
GurobiPersistent.write(filename)	Write the model to a file (e.g., and lp file).

class pyomo.solvers.plugins.solvers.gurobi_persistent.**GurobiPersistent**(***kwds*)

Bases: pyomo.solvers.plugins.solvers.persistent_solver.PersistentSolver, pyomo.solvers.plugins.solvers.gurobi_direct.GurobiDirect

A class that provides a persistent interface to Gurobi. Direct solver interfaces do not use any file io. Rather, they interface directly with the python bindings for the specific solver. Persistent solver interfaces are similar except that they "remember" their model. Thus, persistent solver interfaces allow incremental changes to the solver model (e.g., the gurobi python model or the cplex python model). Note that users are responsible for notifying the persistent solver interfaces when changes are made to the corresponding pyomo model.

Keyword Arguments

- **model** (*ConcreteModel*) – Passing a model to the constructor is equivalent to calling the set_instance mehtod.
- **type** (*str*) – String indicating the class type of the solver instance.
- **name** (*str*) – String representing either the class type of the solver instance or an assigned name.
- **doc** (*str*) – Documentation for the solver
- **options** (*dict*) – Dictionary of solver options

activate()
Register this plugin with all interfaces that it implements.

add_block(*block*)
Add a Pyomo Block to the solver's model. This will keep any existing model components intact.

Parameters **block** (*Block*) –

add_constraint(*con*)
Add a constraint to the solver's model. This will keep any existing model components intact.

Parameters **con** (*Constraint*) –

add_sos_constraint(*con*)
Add an SOS constraint to the solver's model (if supported). This will keep any existing model components intact.

Parameters **con** (*SOSConstraint*) –

add_var(*var*)
Add a variable to the solver's model. This will keep any existing model components intact.

Parameters **var** (*Var*) – The variable to add to the solver's model.

alias(*name*, *doc=None*, *subclass=False*)
This function is used to declare aliases that can be used by a factory for constructing plugin instances.

When the subclass option is True, then subsequent calls to alias() with this class name are ignored, because they are assumed to be due to subclasses of the original class declaration.

available(*exception_flag=True*)
True if the solver is available.

deactivate()
Unregister this plugin with all interfaces that it implements.

disable()
Disable this plugin

enable()
Enable this plugin

enabled()
Return value indicating if this plugin is enabled

has_capability(*cap*)
Returns a boolean value representing whether a solver supports a specific feature. Defaults to 'False' if the solver is unaware of an option. Expects a string.

Example: # prints True if solver supports sos1 constraints, and False otherwise print(solver.has_capability('sos1')

prints True is solver supports 'feature', and False otherwise print(solver.has_capability('feature')

Parameters **cap** (*str*) – The feature

Returns **val** – Whether or not the solver has the specified capability.

Return type bool

has_instance()
True if set_instance has been called and this solver interface has a pyomo model and a solver model.

Returns **tmp**

Return type bool

implements(*interface*, *inherit=None*, *namespace=None*, *service=False*)
Can be used in the class definition of *Plugin* subclasses to declare the extension points that are implemented by this interface class.

load_duals(*cons_to_load=None*)
Load the duals into the 'dual' suffix. The 'dual' suffix must live on the parent model.

Parameters **cons_to_load** (*list of Constraint*) –

load_rc(*vars_to_load*)
Load the reduced costs into the 'rc' suffix. The 'rc' suffix must live on the parent model.

Parameters **vars_to_load** (*list of Var*) –

load_slacks(*cons_to_load=None*)
Load the values of the slack variables into the 'slack' suffix. The 'slack' suffix must live on the parent model.

Parameters **cons_to_load** (*list of Constraint*) –

load_vars(*vars_to_load=None*)
Load the values from the solver's variables into the corresponding pyomo variables.

Parameters **vars_to_load** (*list of Var*) –

problem_format()
Returns the current problem format.

remove_block(*block*)
Remove a block from the solver's model. This will keep any other model components intact.

WARNING: Users must call remove_block BEFORE modifying the block.

Parameters **block** (*Block*) –

remove_constraint(*con*)
Remove a constraint from the solver's model. This will keep any other model components intact.

Parameters **con** (*Constraint*) –

remove_sos_constraint(*con*)
Remove an SOS constraint from the solver's model. This will keep any other model components intact.

Parameters **con** (*SOSConstraint*) –

remove_var(*var*)
Remove a variable from the solver's model. This will keep any other model components intact.

Parameters **var** (*Var*) –

reset()
Reset the state of the solver

results_format()
Returns the current results format.

set_callback(*name*, *callback_fn=None*)
Set the callback function for a named callback.

A call-back function has the form:

def fn(solver, model): pass

where 'solver' is the native solver interface object and 'model' is a Pyomo model instance object.

set_instance(*model*, ***kwds*)
This method is used to translate the Pyomo model provided to an instance of the solver's Python model. This discards any existing model and starts from scratch.

Parameters **model** (*ConcreteModel*) – The pyomo model to be used with the solver.

Keyword Arguments

- **symbolic_solver_labels** (*bool*) – If True, the solver's components (e.g., variables, constraints) will be given names that correspond to the Pyomo component names.
- **skip_trivial_constraints** (*bool*) – If True, then any constraints with a constant body will not be added to the solver model. Be careful with this. If a trivial constraint is skipped then that constraint cannot be removed from a persistent solver (an error will be raised if a user tries to remove a non-existent constraint).
- **output_fixed_variable_bounds** (*bool*) – If False then an error will be raised if a fixed variable is used in one of the solver constraints. This is useful for catching bugs. Ordinarily a fixed variable should appear as a constant value in the solver constraints. If True, then the error will not be raised.

set_objective(*obj*)
Set the solver's objective. Note that, at least for now, any existing objective will be discarded. Other than that, any existing model components will remain intact.

Parameters **obj** (*Objective*) –

set_problem_format(*format*)
Set the current problem format (if it's valid) and update the results format to something valid for this problem format.

set_results_format(*format*)
Set the current results format (if it's valid for the current problem format).

solve(**args*, ***kwds*)
Solve the model.

Keyword Arguments

- **suffixes** (*list of str*) – The strings should represnt suffixes support by the solver. Examples include 'dual', 'slack', and 'rc'.
- **options** (*dict*) – Dictionary of solver options. See the solver documentation for possible solver options.

- **warmstart** (*bool*) – If True, the solver will be warmstarted.
- **keepfiles** (*bool*) – If True, the solver log file will be saved.
- **logfile** (*str*) – Name to use for the solver log file.
- **load_solutions** (*bool*) – If True and a solution exists, the solution will be loaded into the Pyomo model.
- **report_timing** (*bool*) – If True, then timing information will be printed.
- **tee** (*bool*) – If True, then the solver log will be printed.

update_var(*var*)
Update a variable in the solver's model. This will update bounds, fix/unfix the variable as needed, and update the variable type.

Parameters **var** (*Var*) –

version()
Returns a 4-tuple describing the solver executable version.

write(*filename*)
Write the model to a file (e.g., and lp file).

Parameters **filename** (*str*) – Name of the file to which the model should be written.

7.3.2 CPLEXPersistent

class pyomo.solvers.plugins.solvers.cplex_persistent.**CPLEXPersistent**(***kwds*)
Bases: pyomo.solvers.plugins.solvers.persistent_solver.PersistentSolver, pyomo.solvers.plugins.solvers.cplex_direct.CPLEXDirect

A class that provides a persistent interface to Cplex. Direct solver interfaces do not use any file io. Rather, they interface directly with the python bindings for the specific solver. Persistent solver interfaces are similar except that they "remember" their model. Thus, persistent solver interfaces allow incremental changes to the solver model (e.g., the gurobi python model or the cplex python model). Note that users are responsible for notifying the persistent solver interfaces when changes are made to the corresponding pyomo model.

Keyword Arguments

- **model** (*ConcreteModel*) – Passing a model to the constructor is equivalent to calling the set_instance mehtod.
- **type** (*str*) – String indicating the class type of the solver instance.
- **name** (*str*) – String representing either the class type of the solver instance or an assigned name.
- **doc** (*str*) – Documentation for the solver
- **options** (*dict*) – Dictionary of solver options

activate()
Register this plugin with all interfaces that it implements.

add_block(*block*)
Add a Pyomo Block to the solver's model. This will keep any existing model components intact.

Parameters **block** (*Block*) –

add_constraint(*con*)
Add a constraint to the solver's model. This will keep any existing model components intact.

Parameters **con** (*Constraint*) –

add_sos_constraint(*con*)
Add an SOS constraint to the solver's model (if supported). This will keep any existing model components intact.

Parameters **con** (*SOSConstraint*) –

add_var(*var*)
Add a variable to the solver's model. This will keep any existing model components intact.

Parameters **var** (*Var*) –

alias(*name*, *doc=None*, *subclass=False*)
This function is used to declare aliases that can be used by a factory for constructing plugin instances.

When the subclass option is True, then subsequent calls to alias() with this class name are ignored, because they are assumed to be due to subclasses of the original class declaration.

available(*exception_flag=True*)
True if the solver is available.

deactivate()
Unregister this plugin with all interfaces that it implements.

disable()
Disable this plugin

enable()
Enable this plugin

enabled()
Return value indicating if this plugin is enabled

has_capability(*cap*)
Returns a boolean value representing whether a solver supports a specific feature. Defaults to 'False' if the solver is unaware of an option. Expects a string.

Example: # prints True if solver supports sos1 constraints, and False otherwise print(solver.has_capability('sos1')

prints True is solver supports 'feature', and False otherwise print(solver.has_capability('feature')

Parameters **cap** (*str*) – The feature

Returns **val** – Whether or not the solver has the specified capability.

Return type bool

has_instance()
True if set_instance has been called and this solver interface has a pyomo model and a solver model.

Returns **tmp**

Return type bool

implements(*interface*, *inherit=None*, *namespace=None*, *service=False*)
Can be used in the class definition of *Plugin* subclasses to declare the extension points that are implemented by this interface class.

load_duals(*cons_to_load=None*)
Load the duals into the 'dual' suffix. The 'dual' suffix must live on the parent model.

Parameters **cons_to_load** (*list of Constraint*) –

load_rc(*vars_to_load*)
Load the reduced costs into the 'rc' suffix. The 'rc' suffix must live on the parent model.

Parameters **vars_to_load** (*list of Var*) –

load_slacks(*cons_to_load=None*)
Load the values of the slack variables into the 'slack' suffix. The 'slack' suffix must live on the parent model.

Parameters **cons_to_load** (*list of Constraint*) –

load_vars(*vars_to_load=None*)
Load the values from the solver's variables into the corresponding pyomo variables.

Parameters **vars_to_load** (*list of Var*) –

problem_format()
Returns the current problem format.

remove_block(*block*)
Remove a block from the solver's model. This will keep any other model components intact.

WARNING: Users must call remove_block BEFORE modifying the block.

Parameters **block** (*Block*) –

remove_constraint(*con*)
Remove a constraint from the solver's model. This will keep any other model components intact.

Parameters **con** (*Constraint*) –

remove_sos_constraint(*con*)
Remove an SOS constraint from the solver's model. This will keep any other model components intact.

Parameters **con** (*SOSConstraint*) –

remove_var(*var*)
Remove a variable from the solver's model. This will keep any other model components intact.

Parameters **var** (*Var*) –

reset()
Reset the state of the solver

results_format()
Returns the current results format.

set_callback(*name*, *callback_fn=None*)
Set the callback function for a named callback.

A call-back function has the form:

def fn(solver, model): pass

where 'solver' is the native solver interface object and 'model' is a Pyomo model instance object.

set_instance(*model*, ***kwds*)
This method is used to translate the Pyomo model provided to an instance of the solver's Python model. This discards any existing model and starts from scratch.

Parameters **model** (*ConcreteModel*) – The pyomo model to be used with the solver.

Keyword Arguments

- **symbolic_solver_labels** (*bool*) – If True, the solver's components (e.g., variables, constraints) will be given names that correspond to the Pyomo component names.

- **skip_trivial_constraints** (*bool*) – If True, then any constraints with a constant body will not be added to the solver model. Be careful with this. If a trivial constraint is skipped then that constraint cannot be removed from a persistent solver (an error will be raised if a user tries to remove a non-existent constraint).
- **output_fixed_variable_bounds** (*bool*) – If False then an error will be raised if a fixed variable is used in one of the solver constraints. This is useful for catching bugs. Ordinarily a fixed variable should appear as a constant value in the solver constraints. If True, then the error will not be raised.

set_objective(*obj*)
Set the solver's objective. Note that, at least for now, any existing objective will be discarded. Other than that, any existing model components will remain intact.

Parameters **obj** (*Objective*) –

set_problem_format(*format*)
Set the current problem format (if it's valid) and update the results format to something valid for this problem format.

set_results_format(*format*)
Set the current results format (if it's valid for the current problem format).

solve(**args*, ***kwds*)
Solve the model.

Keyword Arguments

- **suffixes** (*list of str*) – The strings should represnt suffixes support by the solver. Examples include 'dual', 'slack', and 'rc'.
- **options** (*dict*) – Dictionary of solver options. See the solver documentation for possible solver options.
- **warmstart** (*bool*) – If True, the solver will be warmstarted.
- **keepfiles** (*bool*) – If True, the solver log file will be saved.
- **logfile** (*str*) – Name to use for the solver log file.
- **load_solutions** (*bool*) – If True and a solution exists, the solution will be loaded into the Pyomo model.
- **report_timing** (*bool*) – If True, then timing information will be printed.
- **tee** (*bool*) – If True, then the solver log will be printed.

update_var(*var*)
Update a variable in the solver's model. This will update bounds, fix/unfix the variable as needed, and update the variable type.

Parameters **var** (*Var*) –

version()
Returns a 4-tuple describing the solver executable version.

write(*filename*, *filetype=''*)
Write the model to a file (e.g., and lp file).

Parameters

- **filename** (*str*) – Name of the file to which the model should be written.
- **filetype** (*str*) – The file type (e.g., lp).

CHAPTER 8

Problem Reference

Examples of Pyomo models for different types of problems ...

```
>>> print('Hello World')
Hello World
```

CHAPTER 9

Indices and Tables

- genindex
- modindex
- search

CHAPTER 10

Pyomo Resources

The Pyomo home page provides resources for Pyomo users:

- http://pyomo.org

Pyomo development is hosted at GitHub:

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

See the Pyomo Forum for online discussions of Pyomo:

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

Python Module Index

p

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

R

S

T

U

V

W