<table>
<thead>
<tr>
<th>Chapter</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  Getting Started</td>
<td>3</td>
</tr>
<tr>
<td>2  Tutorial</td>
<td>5</td>
</tr>
<tr>
<td>3  Core Pyomo Components</td>
<td>9</td>
</tr>
<tr>
<td>4  Managing Data in Pyomo Models</td>
<td>11</td>
</tr>
<tr>
<td>5  Scripting</td>
<td>41</td>
</tr>
<tr>
<td>6  Modeling Extensions</td>
<td>43</td>
</tr>
<tr>
<td>7  Developer Reference</td>
<td>61</td>
</tr>
<tr>
<td>8  Persistent Solvers</td>
<td>85</td>
</tr>
<tr>
<td>9  Library Reference</td>
<td>89</td>
</tr>
<tr>
<td>10 Problem Reference</td>
<td>191</td>
</tr>
<tr>
<td>11 Third-Party Contributions</td>
<td>193</td>
</tr>
<tr>
<td>12 Bibliography</td>
<td>195</td>
</tr>
<tr>
<td>13 Indices and Tables</td>
<td>197</td>
</tr>
<tr>
<td>14 Pyomo Resources</td>
<td>199</td>
</tr>
<tr>
<td>Bibliography</td>
<td>201</td>
</tr>
<tr>
<td>Python Module Index</td>
<td>203</td>
</tr>
</tbody>
</table>
Pyomo is a Python-based, open-source optimization modeling language with a diverse set of optimization capabilities.
1.1 Installation

Pyomo currently supports the following versions of Python:

- CPython: 2.7, 3.4, 3.5, 3.6

1.1.1 Using CONDA

We recommend installation with conda, which is included with the Anaconda distribution of Python. If you have a different Python distribution, then you can install miniconda using pip:

You can install Pyomo in your system Python installation by executing the following in a shell:

Pyomo also has conditional dependencies on a variety of third-party Python packages. These can also be installed with conda:

Optimization solvers are not installed with Pyomo, but some open source optimization solvers can be installed with conda as well:

1.1.2 Using PIP

The standard utility for installing Python packages is pip. You can install Pyomo in your system Python installation by executing the following in a shell:

1.2 A Simple Example

TODO pyomo command, importing pyomo, using NEOS, etc
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

The primary modeling components defined by Pyomo are Param, Var, Constraint, Objective, Set, and Block. Each of these may be indexed.

```python
>>> import pyomo.environ as pe
>>> m = pe.ConcreteModel()
>>> m.a = pe.Set(initialize=[1, 2, 3])
>>> m.x = pe.Var(m.a, initialize=0, bounds=(-10,10))
>>> m.y = pe.Var(m.a)
>>> def c_rule(m, i):
...     return m.x[i] >= m.y[i]
>>> m.c = pe.Constraint(m.a, rule=c_rule)
```
The index specifies the set of allowable members of the component. In the case of \texttt{Var}, the constructor will automatically create variables for each member of the index. Other components (like \texttt{Constraint<pyomo.core.base.constraint.Constraint>}) leverage a rule, which is called by the constructor for every member of the index, the return value of which dictates whether or not to create the corresponding modeling object. Beyond facilitating the construction of large structured models, discrete indexing sets provide error checking, ensuring that requested modeling objects are allowed by the index:

```python
>>> m.z = pe.Var()
>>> m.c[4] = m.x[1] == 5 * m.z
Traceback (most recent call last):
  ... 
KeyError: "Index '4' is not valid for indexed component 'c'"
```

This helps prevent many common mistakes. To add new objects to a component, the new index must first be added to the underlying index set:

```python
>>> m.a.add(4)
>>> m.c[4] = m.x[1] == 5 * m.z
```

However, it is sometimes useful to allow a more flexible form of indexing using non-iterable sets. For example, an indexed component may be made to behave like a dictionary by indexing it using the \texttt{Any} set. This set admits any hashable object as a member.

```python
>>> m.c2 = pe.Constraint(pe.Any)
>>> m.c2[1] = m.x[1] == 5 * m.z
>>> m.c2[8] = m.x[2] == m.z * m.y[2]
>>> m.c2.pprint()
c2 : Size=3, Index=Any, Active=True
  Key : Lower : Body : Upper : Active
  1 : -Inf : y[1] - x[1] : 0.0 : True
```

\textbf{Note:} It is important that the component construction not iterate over the non-iterable set. For most components, simply omitting the \texttt{rule=} argument is sufficient. \texttt{Var} requires the \texttt{dense=False} argument so that the constructor does not iterate over the non-iterable set.

```python
>>> m.v = pe.Var(pe.Any, dense=False)
>>> m.v.pprint()
v : Size=2, Index=Any
  Key : Lower : Value : Upper : Fixed : Stale : Domain
  1 : None : None : None : False : True : Reals
  2 : None : None : None : False : True : Reals
>>> m.c2.pprint()
c2 : Size=3, Index=Any, Active=True
  Key : Lower : Body : Upper : Active
  1 : 0.0 : x[1] - 5*z : 0.0 : True
```
The following illustrates how to use `Any` with Blocks.

```python
>>> m.b = pe.Block(pe.Any)
>>> m.b['foo1'].x = pe.Var()
>>> m.b['foo1'].y = pe.Var()
>>> m.b['foo1'].c = pe.Constraint(expr=m.b['foo1'].x >= 5*m.b['foo1'].y)
>>> m.b[1].x = pe.Var()
```

## 2.4 Pyomo Command

```python
>>> print('Hello World')
Hello World
```
Core Pyomo Components

Detailed component descriptions . . .

```python
>>> print('Hello World')
Hello World
```
There are roughly three ways of using data to construct a Pyomo model:

1. use standard Python data objects,
2. initialize a model with data loaded with a `DataPortal` object, and
3. load model data from a Pyomo data command file.

Standard Python data objects include native Python data types (e.g. lists, sets, and dictionaries) as well as standard data formats like NumPy arrays and Pandas data frames. Standard Python data objects can be used to define constant values in a Pyomo model, and they can be used to initialize `Set` and `Param` components. However, initializing `Set` and `Param` components in this manner provides few advantages over direct use of standard Python data objects. (An import exception is that components indexed by `Set` objects use less memory than components indexed by native Python data.)

The `DataPortal` class provides a generic facility for loading data from disparate sources. A `DataPortal` object can load data in a consistent manner, and this data can be used to simply initialize all `Set` and `Param` components in a model. `DataPortal` objects can be used to initialize both concrete and abstract models in a uniform manner, which is important in some scripting applications. But in practice, this capability is only necessary for abstract models, whose data components are initialized after being constructed. (In fact, all abstract data components in an abstract model are loaded from `DataPortal` objects.)

Finally, Pyomo data command files provide a convenient mechanism for initializing `Set` and `Param` components with a high-level data specification. Data command files can be used with both concrete and abstract models, though in a different manner. Data command files are parsed using a `DataPortal` object, which must be done explicitly for a concrete model. However, abstract models can load data from a data command file directly, after the model is constructed. Again, this capability is only necessary for abstract models, whose data components are initialized after being constructed.

The following sections provide more detail about how data can be used to initialize Pyomo models.
4.1 Using Standard Data Types

4.1.1 Defining Constant Values

In many cases, Pyomo models can be constructed without Set and Param data components. Native Python data types class can be simply used to define constant values in Pyomo expressions. Consequently, Python sets, lists and dictionaries can be used to construct Pyomo models, as well as a wide range of other Python classes.

TODO

More examples here: set, list, dict, numpy, pandas.

4.1.2 Initializing Set and Parameter Components

The Set and Param components used in a Pyomo model can also be initialized with standard Python data types. This enables some modeling efficiencies when manipulating sets (e.g. when re-using sets for indices), and it supports validation of set and parameter data values. The Set and Param components are initialized with Python data using the code{initialize} option.

Set Components

In general, Set components can be initialized with iterable data. For example, simple sets can be initialized with:

- list, set and tuple data:

```
model.A = Set(initialize=[2,3,5])
model.B = Set(initialize=set([2,3,5]))
model.C = Set(initialize=(2,3,5))
```

- generators:

```
model.D = Set(initialize=range(9))
model.E = Set(initialize=(i for i in model.B if i%2 == 0))
```

- numpy arrays:

```
f = numpy.array([2, 3, 5])
model.F = Set(initialize=f)
```

Sets can also be indirectly initialized with functions that return native Python data:

```
def g(model):
    return [2,3,5]
model.G = Set(initialize=g)
```

Indexed sets can be initialized with dictionary data where the dictionary values are iterable data:

```
H_init = {}
H_init[2] = [1,3,5]
H_init[3] = [2,4,6]
H_init[4] = [3,5,7]
model.H = Set([2,3,4],initialize=H_init)
```
Parameter Components

When a parameter is a single value, then a Param component can be simply initialized with a value:

```python
code
model.a = Param(initialize=1.1)
```

More generally, Param components can be initialized with dictionary data where the dictionary values are single values:

```python
code
model.b = Param([1,2,3], initialize={1:1, 2:2, 3:3})
```

Parameters can also be indirectly initialized with functions that return native Python data:

```python
code
def c(model):
    return {1:1, 2:2, 3:3}
model.c = Param([1,2,3], initialize=c)
```

4.2 Data Command Files

**Note:** The discussion and presentation below are adapted from Chapter 6 of the “Pyomo Book” [PyomoBook2Ed]. The discussion of the DataPortal class uses these same examples to illustrate how data can be loaded into Pyomo models within Python scripts (see the Data Portals section).

4.2.1 Model Data

Pyomo’s data command files employ a domain-specific language whose syntax closely resembles the syntax of AMPL’s data commands [AMPL]. A data command file consists of a sequence of commands that either (a) specify set and parameter data for a model, or (b) specify where such data is to be obtained from external sources (e.g. table files, CSV files, spreadsheets and databases).

The following commands are used to declare data:

- The **set** command declares set data.
- The **param** command declares a table of parameter data, which can also include the declaration of the set data used to index the parameter data.
- The **table** command declares a two-dimensional table of parameter data.
- The **load** command defines how set and parameter data is loaded from external data sources, including ASCII table files, CSV files, XML files, YAML files, JSON files, ranges in spreadsheets, and database tables.

The following commands are also used in data command files:

- The **include** command specifies a data command file that is processed immediately.
- The **data** and **end** commands do not perform any actions, but they provide compatibility with AMPL scripts that define data commands.
- The **namespace** keyword allows data commands to be organized into named groups that can be enabled or disabled during model construction.

The following data types can be represented in a data command file:

- **Numeric value**: Any Python numeric value (e.g. integer, float, scientific notation, or boolean).
• **Simple string**: A sequence of alpha-numeric characters.

• **Quoted string**: A simple string that is included in a pair of single or double quotes. A quoted string can include quotes within the quoted string.

Numeric values are automatically converted to Python integer or floating point values when a data command file is parsed. Additionally, if a quoted string can be interpreted as a numeric value, then it will be converted to Python numeric types when the data is parsed. For example, the string “100” is converted to a numeric value automatically.

**Warning:** Pyomo data commands do *not* exactly correspond to AMPL data commands. The `set` and `param` commands are designed to closely match AMPL’s syntax and semantics, though these commands only support a subset of the corresponding declarations in AMPL. However, other Pyomo data commands are not generally designed to match the semantics of AMPL.

**Note:** Pyomo data commands are terminated with a semicolon, and the syntax of data commands does not depend on whitespace. Thus, data commands can be broken across multiple lines – newlines and tab characters are ignored – and data commands can be formatted with whitespace with few restrictions.

### 4.2.2 The set Command

#### Simple Sets

The `set` data command explicitly specifies the members of either a single set or an array of sets, i.e., an indexed set. A single set is specified with a list of data values that are included in this set. The formal syntax for the `set` data command is:

```python
set <setname> := [<value>] ... ;
```

A set may be empty, and it may contain any combination of numeric and non-numeric string values. For example, the following are valid `set` commands:

```python
# An empty set
set A := ;

# A set of numbers
set A := 1 2 3;

# A set of strings
set B := north south east west;

# A set of mixed types
set C :=
0
-1.0e+10
'foo bar'
infinity
"100"
;
```
Sets of Tuple Data

The set data command can also specify tuple data with the standard notation for tuples. For example, suppose that set \( A \) contains 3-tuples:

```py
model.A = Set(dimen=3)
```

The following set data command then specifies that \( A \) is the set containing the tuples \((1, 2, 3)\) and \((4, 5, 6)\):

```py
set A := (1,2,3) (4,5,6) ;
```

Alternatively, set data can simply be listed in the order that the tuple is represented:

```py
set A := 1 2 3 4 5 6 ;
```

Obviously, the number of data elements specified using this syntax should be a multiple of the set dimension.

Sets with 2-tuple data can also be specified in a matrix denoting set membership. For example, the following set data command declares 2-tuples in \( A \) using plus (+) to denote valid tuples and minus (-) to denote invalid tuples:

```py
set A : A1 A2 A3 A4 :=
  1 + - - +
  2 + - + -
  3 - + - - ;
```

This data command declares the following five 2-tuples: ('A1',1), ('A1',2), ('A2',3), ('A3',2), and ('A4',1).

Finally, a set of tuple data can be concisely represented with tuple templates that represent a slice of tuple data. For example, suppose that the set \( A \) contains 4-tuples:

```py
model.A = Set(dimen=4)
```

The following set data command declares groups of tuples that are defined by a template and data to complete this template:

```py
set A :=
  (1,2,*4) A B
  (*2,*4) A B C D ;
```

A tuple template consists of a tuple that contains one or more asterisk (*) symbols instead of a value. These represent indices where the tuple value is replaced by the values from the list of values that follows the tuple template. In this example, the following tuples are in set \( A \):

\[
\begin{align*}
(1, 2, 'A', 4) \\
(1, 2, 'B', 4) \\
('A', 2, 'B', 4) \\
('C', 2, 'D', 4)
\end{align*}
\]

Set Arrays

The set data command can also be used to declare data for a set array. Each set in a set array must be declared with a separate set data command with the following syntax:

```py
set <set-name>[<index>] := [<value>] ... ;
```
Because set arrays can be indexed by an arbitrary set, the index value may be a numeric value, a non-numeric string value, or a comma-separated list of string values.

Suppose that a set \( \text{A} \) is used to index a set \( \text{B} \) as follows:

```python
model.A = Set()
model.B = Set(model.A)
```

Then set \( \text{B} \) is indexed using the values declared for set \( \text{A} \):

```python
set A := 1 aaa 'a b';
set B[1] := 0 1 2;
set B[aaa] := aa bb cc;
set B['a b'] := 'aa bb cc';
```

### 4.2.3 The `param` Command

Simple or non-indexed parameters are declared in an obvious way, as shown by these examples:

```python
param A := 1.4;
param B := 1;
param C := abc;
param D := true;
param E := 1.0e+04;
```

Parameters can be defined with numeric data, simple strings and quoted strings. Note that parameters cannot be defined without data, so there is no analog to the specification of an empty set.

#### One-dimensional Parameter Data

Most parameter data is indexed over one or more sets, and there are a number of ways the `param` data command can be used to specify indexed parameter data. One-dimensional parameter data is indexed over a single set. Suppose that the parameter \( \text{B} \) is a parameter indexed by the set \( \text{A} \):

```python
model.A = Set()
model.B = Param(model.A)
```

A `param` data command can specify values for \( \text{B} \) with a list of index-value pairs:

```python
set A := a c e;
param B := a 10 c 30 e 50;
```

Because whitespace is ignored, this example data command file can be reorganized to specify the same data in a tabular format:

```python
set A := a c e;
param B :=
a 10
c 30
e 50
```

---

16 Chapter 4. Managing Data in Pyomo Models
Multiple parameters can be defined using a single `param` data command. For example, suppose that parameters $B$, $C$, and $D$ are one-dimensional parameters all indexed by the set $A$:

```python
model.A = Set()
model.B = Param(model.A)
model.C = Param(model.A)
model.D = Param(model.A)
```

Values for these parameters can be specified using a single `param` data command that declares these parameter names followed by a list of index and parameter values:

```text
set A := a c e;

param : B C D :=
a 10 -1 1.1
c 30 -3 3.3
e 50 -5 5.5
;
```

The values in the `param` data command are interpreted as a list of sublists, where each sublist consists of an index followed by the corresponding numeric value.

Note that parameter values do not need to be defined for all indices. For example, the following data command file is valid:

```text
set A := a c e g;

param : B C D :=
a 10 -1 1.1
c 30 -3 3.3
e 50 -5 5.5
;
```

The index $g$ is omitted from the `param` command, and consequently this index is not valid for the model instance that uses this data. More complex patterns of missing data can be specified using the period (.) symbol to indicate a missing value. This syntax is useful when specifying multiple parameters that do not necessarily have the same index values:

```text
set A := a c e;

param : B C D :=
a . -1 1.1
c 30 . 3.3
e 50 -5 .
;
```

This example provides a concise representation of parameters that share a common index set while using different index values.

Note that this data file specifies the data for set $A$ twice: (1) when $A$ is defined and (2) implicitly when the parameters are defined. An alternate syntax for `param` allows the user to concisely specify the definition of an index set along with associated parameters:

```text
param : A : B C D :=
a 10 -1 1.1
c 30 -3 3.3
e 50 -5 5.5
;
```

4.2. Data Command Files
Finally, we note that default values for missing data can also be specified using the `default` keyword:

```py
set A := a c e;
param B default 0.0 :=
c 30
e 50
;
```

Note that default values can only be specified in `param` commands that define values for a single parameter.

### Multi-Dimensional Parameter Data

Multi-dimensional parameter data is indexed over either multiple sets or a single multi-dimensional set. Suppose that parameter `B` is a parameter indexed by set `A` that has dimension 2:

```py
model.A = Set(dimen=2)
model.B = Param(model.A)
```

The syntax of the `param` data command remains essentially the same when specifying values for `B` with a list of index and parameter values:

```py
set A := a 1 c 2 e 3;
param B :=
a 1 10
c 2 30
e 3 50;
```

Missing and default values are also handled in the same way with multi-dimensional index sets:

```py
set A := a 1 c 2 e 3;
param B default 0 :=
a 1 10
c 2 .
e 3 50;
```

Similarly, multiple parameters can defined with a single `param` data command. Suppose that parameters `B`, `C`, and `D` are parameters indexed over set `A` that has dimension 2:

```py
model.A = Set(dimen=2)
model.B = Param(model.A)
model.C = Param(model.A)
model.D = Param(model.A)
```

These parameters can be defined with a single `param` command that declares the parameter names followed by a list of index and parameter values:

```py
set A := a 1 c 2 e 3;
param : B C D :=
a 1 10 -1 1.1
c 2 30 -3 3.3
e 3 50 -5 5.5
;
```
Similarly, the following `param` data command defines the index set along with the parameters:

```
param : A : B C D :=
a 1 10 -1 1.1
c 2 30 -3 3.3
e 3 50 -5 5.5
;
```

The `param` command also supports a matrix syntax for specifying the values in a parameter that has a 2-dimensional index. Suppose parameter `B` is indexed over set `A` that has dimension 2:

```
model.A = Set(dimen=2)
model.B = Param(model.A)
```

The following `param` command defines a matrix of parameter values:

```
set A := 1 a 1 c 1 e 2 a 2 c 2 e 3 a 3 c 3 e;

param B : a c e :=
  1 1 2 3
  2 4 5 6
  3 7 8 9
;
```

Additionally, the following syntax can be used to specify a transposed matrix of parameter values:

```
set A := 1 a 1 c 1 e 2 a 2 c 2 e 3 a 3 c 3 e;

param B (tr) : 1 2 3 :=
a 1 4 7
c 2 5 8
e 3 6 9
;
```

This functionality facilitates the presentation of parameter data in a natural format. In particular, the transpose syntax may allow the specification of tables for which the rows comfortably fit within a single line. However, a matrix may be divided column-wise into shorter rows since the line breaks are not significant in Pyomo data commands.

For parameters with three or more indices, the parameter data values may be specified as a series of slices. Each slice is defined by a template followed by a list of index and parameter values. Suppose that parameter `B` is indexed over set `A` that has dimension 4:

```
model.A = Set(dimen=4)
model.B = Param(model.A)
```

The following `param` command defines a matrix of parameter values with multiple templates:

```
set A := (a,1,a,1) (a,2,a,2) (b,1,b,1) (b,2,b,2);

param B :=
  [*,1,*] a a 10 b b 20
  [*,2,*] a a 30 b b 40
;
```

The `B` parameter consists of four values: $B[a, 1, a, 1] = 10$, $B[b, 1, b, 1] = 20$, $B[a, 2, a, 2] = 30$, and $B[b, 2, b, 2] = 40$. 

### 4.2. Data Command Files

19
4.2.4 The table Command

The table data command explicitly specifies a two-dimensional array of parameter data. This command provides a more flexible and complete data declaration than is possible with a param declaration. The following example illustrates a simple table command that declares data for a single parameter:

```
table M(A) :
  A  B M N :=
  A1 B1 4.3 5.3
  A2 B2 4.4 5.4
  A3 B3 4.5 5.5
;
```

The parameter M is indexed by column A, which must be pre-defined unless declared separately (see below). The column labels are provided after the colon and before the colon-equal (:=). Subsequently, the table data is provided. The syntax is not sensitive to whitespace, so the following is an equivalent table command:

```
table M(A) :
  A  B M N :=
  A1 B1 4.3 5.3 A2 B2 4.4 5.4 A3 B3 4.5 5.5;
```

Multiple parameters can be declared by simply including additional parameter names. For example:

```
table M(A) N(A,B) :
  A  B M N :=
  A1 B1 4.3 5.3
  A2 B2 4.4 5.4
  A3 B3 4.5 5.5
;
```

This example declares data for the M and N parameters, which have different indexing columns. The indexing columns represent set data, which is specified separately. For example:

```
table A={A} Z={A,B} M(A) N(A,B) :
  A  B M N :=
  A1 B1 4.3 5.3
  A2 B2 4.4 5.4
  A3 B3 4.5 5.5
;
```

This example declares data for the M and N parameters, along with the A and Z indexing sets. The correspondence between the index set Z and the indices of parameter N can be made more explicit by indexing N by Z:

```
table A={A} Z={A,B} M(A) N(Z) :
  A  B M N :=
  A1 B1 4.3 5.3
  A2 B2 4.4 5.4
  A3 B3 4.5 5.5
;
```

Set data can also be specified independent of parameter data:

```
table Z={A,B} Y={M,N} :
  A  B M N :=
  A1 B1 4.3 5.3
  A2 B2 4.4 5.4
  A3 B3 4.5 5.5
;
```
Warning: If a table command does not explicitly indicate the indexing sets, then these are assumed to be initialized separately. A table command can separately initialize sets and parameters in a Pyomo model, and there is no presumed association between the data that is initialized. For example, the table command initializes a set $Z$ and a parameter $M$ that are not related:

```plaintext
table Z={A,B} M(A):
  A  B  M  N :=
  A1 B1 4.3 5.3
  A2 B2 4.4 5.4
  A3 B3 4.5 5.5
;
```

Finally, simple parameter values can also be specified with a table command:

```plaintext
table pi := 3.1416 ;
```

The previous examples considered examples of the table command where column labels are provided. The table command can also be used without column labels. For example, the first example can be revised to omit column labels as follows:

```plaintext
table columns=4 M(1)={3} :=
  A1 B1 4.3 5.3
  A2 B2 4.4 5.4
  A3 B3 4.5 5.5
;
```

The columns=4 is a keyword-value pair that defines the number of columns in this table; this must be explicitly specified in tables without column labels. The default column labels are integers starting from 1; the labels are columns 1, 2, 3, and 4 in this example. The $M$ parameter is indexed by column 1. The braces syntax declares the column where the $M$ data is provided.

Similarly, set data can be declared referencing the integer column labels:

```plaintext
table columns=4 A={1} Z={1,2} M(1)={3} N(1,2)={4} :=
  A1 B1 4.3 5.3
  A2 B2 4.4 5.4
  A3 B3 4.5 5.5
;
```

Declared set names can also be used to index parameters:

```plaintext
table columns=4 A={1} Z={1,2} M(A)={3} N(Z)={4} :=
  A1 B1 4.3 5.3
  A2 B2 4.4 5.4
  A3 B3 4.5 5.5
;
```

Finally, we compare and contrast the table and param commands. Both commands can be used to declare parameter and set data, and both commands can be used to declare a simple parameter. However, there are some important differences between these data commands:

- The param command can declare a single set that is used to index one or more parameters. The table command can declare data for any number of sets, independent of whether they are used to index parameter data.
- The param command can declare data for multiple parameters only if they share the same index set. The table command can declare data for any number of parameters that are may be indexed separately.

4.2. Data Command Files 21
• The table syntax unambiguously describes the dimensionality of indexing sets. The param command must be interpreted with a model that provides the dimension of the indexing set.

This last point provides a key motivation for the table command. Specifically, the table command can be used to reliably initialize concrete models using Pyomo’s DataPortal class. By contrast, the param command can only be used to initialize concrete models with parameters that are indexed by a single column (i.e., a simple set).

### 4.2.5 The load Command

The load command provides a mechanism for loading data from a variety of external tabular data sources. This command loads a table of data that represents set and parameter data in a Pyomo model. The table consists of rows and columns for which all rows have the same length, all columns have the same length, and the first row represents labels for the column data.

The load command can load data from a variety of different external data sources:

- **TAB File**: A text file format that uses whitespace to separate columns of values in each row of a table.
- **CSV File**: A text file format that uses comma or other delimiters to separate columns of values in each row of a table.
- **XML File**: An extensible markup language for documents and data structures. XML files can represent tabular data.
- **Excel File**: A spreadsheet data format that is primarily used by the Microsoft Excel application.
- **Database**: A relational database.

This command uses a data manager that coordinates how data is extracted from a specified data source. In this way, the load command provides a generic mechanism that enables Pyomo models to interact with standard data repositories that are maintained in an application-specific manner.

#### Simple Load Examples

The simplest illustration of the load command is specifying data for an indexed parameter. Consider the file Y.tab:

```
A Y
A1 3.3
A2 3.4
A3 3.5
```

This file specifies the values of parameter Y which is indexed by set A. The following load command loads the parameter data:

```python
load Y.tab : [A] Y;
```

The first argument is the filename. The options after the colon indicate how the table data is mapped to model data. Option [A] indicates that set A is used as the index, and option Y indicates the parameter that is initialized.

Similarly, the following load command loads both the parameter data as well as the index set A:

```python
```

The difference is the specification of the index set, A=[A], which indicates that set A is initialized with the index loaded from the ASCII table file.

Set data can also be loaded from a ASCII table file that contains a single column of data:
The format option must be specified to denote the fact that the relational data is being interpreted as a set:

```
load A.tab format=set : A;
```

Note that this allows for specifying set data that contains tuples. Consider file C.tab:

```
A  B
A1 1
A1 2
A1 3
A2 1
A2 2
A2 3
A3 1
A3 2
A3 3
```

A similar load syntax will load this data into set C:

```
load C.tab format=set : C;
```

Note that this example requires that C be declared with dimension two.

### Load Syntax Options

The syntax of the load command is broken into two parts. The first part ends with the colon, and it begins with a filename, database URL, or DSN (data source name). Additionally, this first part can contain option value pairs. The following options are recognized:

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>format</td>
<td>A string that denotes how the relational table is interpreted</td>
</tr>
<tr>
<td>password</td>
<td>The password that is used to access a database</td>
</tr>
<tr>
<td>query</td>
<td>The query that is used to request data from a database</td>
</tr>
<tr>
<td>range</td>
<td>The subset of a spreadsheet that is requestedindex{spreadsheet}</td>
</tr>
<tr>
<td>user</td>
<td>The user name that is used to access the data source</td>
</tr>
<tr>
<td>using</td>
<td>The data manager that is used to process the data source</td>
</tr>
<tr>
<td>table</td>
<td>The database table that is requested</td>
</tr>
</tbody>
</table>

The format option is the only option that is required for all data managers. This option specifies how a relational table is interpreted to represent set and parameter data. If the using option is omitted, then the filename suffix is used to select the data manager. The remaining options are specific to spreadsheets and relational databases (see below).

The second part of the load command consists of the specification of column names for indices and data. The remainder of this section describes different specifications and how they define how data is loaded into a model. Suppose file ABCD.tab defines the following relational table:

```
A  B  C  D
A1 B1 1 10
A2 B2 2 20
A3 B3 3 30
```
There are many ways to interpret this relational table. It could specify a set of 4-tuples, a parameter indexed by 3-tuples, two parameters indexed by 2-tuples, and so on. Additionally, we may wish to select a subset of this table to initialize data in a model. Consequently, the load command provides a variety of syntax options for specifying how a table is interpreted.

A simple specification is to interpret the relational table as a set:

```
load ABCD.tab format=set : Z ;
```

Note that $Z$ is a set in the model that the data is being loaded into. If this set does not exist, an error will occur while loading data from this table.

Another simple specification is to interpret the relational table as a parameter with indexed by 3-tuples:

```
load ABCD.tab : [A,B,C] D ;
```

Again, this requires that $D$ be a parameter in the model that the data is being loaded into. Additionally, the index set for $D$ must contain the indices that are specified in the table. The load command also allows for the specification of the index set:

```
load ABCD.tab : Z=[A,B,C] D ;
```

This specifies that the index set is loaded into the $Z$ set in the model. Similarly, data can be loaded into another parameter than what is specified in the relational table:

```
load ABCD.tab : Z=[A,B,C] Y=D ;
```

This specifies that the index set is loaded into the $Z$ set and that the data in the $D$ column in the table is loaded into the $Y$ parameter.

This syntax allows the load command to provide an arbitrary specification of data mappings from columns in a relational table into index sets and parameters. For example, suppose that a model is defined with set $Z$ and parameters $Y$ and $W$:

```python
model.Z = Set()
model.Y = Param(model.Z)
model.W = Param(model.Z)
```

Then the following command defines how these data items are loaded using columns $B$, $C$ and $D$:

```
load ABCD.tab : Z=[B] Y=D W=C;
```

When the using option is omitted the data manager is inferred from the filename suffix. However, the filename suffix does not always reflect the format of the data it contains. For example, consider the relational table in the file `ABCD.txt`:

```
A,B,C,D
A1,B1,1,10
A2,B2,2,20
A3,B3,3,30
```

We can specify the using option to load from this file into parameter $D$ and set $Z$:

```
load ABCD.txt using=csv : Z=[A,B,C] D ;
```

Note: The data managers supported by Pyomo can be listed with the `pyomo help` subcommand:
pyomo help --data-managers

The following data managers are supported in Pyomo 5.1:

```
Pyomo Data Managers
---------------------
csv              CSV file interface
dat              Pyomo data command file interface
json             JSON file interface
pymysql          pymysql database interface
pyodbc           pyodbc database interface
pypypyodbc       pypypyodbc database interface
sqlite3          sqlite3 database interface
tab              TAB file interface
xls              Excel XLS file interface
xlsb             Excel XLSB file interface
xlsx             Excel XLSX file interface
xml              XML file interface
yaml             YAML file interface
```

**Interpreting Tabular Data**

By default, a table is interpreted as columns of one or more parameters with associated index columns. The `format` option can be used to specify other interpretations of a table:

| `array` | The table is a matrix representation of a two dimensional parameter. |
| `param` | The data is a simple parameter value. |
| `set` | Each row is a set element. |
| `set_array` | The table is a matrix representation of a set of 2-tuples. |
| `transposed_array` | The table is a transposed matrix representation of a two dimensional parameter. |

We have previously illustrated the use of the `set` format value to interpret a relational table as a set of values or tuples. The following examples illustrate the other format values.

A table with a single value can be interpreted as a simple parameter using the `param` format value. Suppose that `Z.tab` contains the following table:

```
1.1
```
The following load command then loads this value into parameter $p$:

```
load Z.tab format=param: p;
```

Sets with 2-tuple data can be represented with a matrix format that denotes set membership. The `set_array` format value interprets a relational table as a matrix that defines a set of 2-tuples where + denotes a valid tuple and – denotes an invalid tuple. Suppose that $D.tab$ contains the following relational table:

```
<table>
<thead>
<tr>
<th>B</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>+</td>
</tr>
</tbody>
</table>
```

Then the following load command loads data into set $B$:

```
load D.tab format=set_array: B;
```

This command declares the following 2-tuples: (‘A1’,1), (‘A2’,2), and (‘A3’,3).

Parameters with 2-tuple indices can be interpreted with a matrix format where rows and columns are different indices. Suppose that $U.tab$ contains the following table:

```
<table>
<thead>
<tr>
<th>I</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
</tr>
</thead>
<tbody>
<tr>
<td>I1</td>
<td>1.3</td>
<td>2.3</td>
<td>3.3</td>
</tr>
<tr>
<td>I2</td>
<td>1.4</td>
<td>2.4</td>
<td>3.4</td>
</tr>
<tr>
<td>I3</td>
<td>1.5</td>
<td>2.5</td>
<td>3.5</td>
</tr>
<tr>
<td>I4</td>
<td>1.6</td>
<td>2.6</td>
<td>3.6</td>
</tr>
</tbody>
</table>
```

Then the following load command loads this value into parameter $U$ with a 2-dimensional index using the `array` format value:

```
load U.tab format=array: A=[X] U;
```

The `transpose_array` format value also interprets the table as a matrix, but it loads the data in a transposed format:

```
load U.tab format=transpose_array: A=[X] U;
```

Note that these format values do not support the initialization of the index data.

### Loading from Spreadsheets and Relational Databases

Many of the options for the `load` command are specific to spreadsheets and relational databases. The `range` option is used to specify the range of cells that are loaded from a spreadsheet. The range of cells represents a table in which the first row of cells defines the column names for the table.

Suppose that file `ABCD.xls` contains the range `ABCD` that is shown in the following figure:
The following command loads this data to initialize parameter $D$ and index $Z$:

```
load ABCD.xls range=ABCD : Z=[A,B,C] Y=D ;
```

Thus, the syntax for loading data from spreadsheets only differs from CSV and ASCII text files by the use of the `range` option.

When loading from a relational database, the data source specification is a filename or data connection string. Access to a database may be restricted, and thus the specification of `username` and `password` options may be required. Alternatively, these options can be specified within a data connection string.

A variety of database interface packages are available within Python. The `using` option is used to specify the database interface package that will be used to access a database. For example, the `pyodbc` interface can be used to connect to Excel spreadsheets. The following command loads data from the Excel spreadsheet `ABCD.xls` using the `pyodbc` interface. The command loads this data to initialize parameter $D$ and index $Z$:

```
load ABCD.xls using=pyodbc table=ABCD : Z=[A,B,C] Y=D ;
```

The `using` option specifies that the `pyodbc` package will be used to connect with the Excel spreadsheet. The `table` option specifies that the table `ABCD` is loaded from this spreadsheet. Similarly, the following command specifies a data connection string to specify the ODBC driver explicitly:

```
load "Driver={Microsoft Excel Driver (*.xls)}; Dbq=ABCD.xls;" using=pyodbc
table=ABCD : Z=[A,B,C] Y=D ;
```

ODBC drivers are generally tailored to the type of data source that they work with; this syntax illustrates how the `load` command can be tailored to the details of the database that a user is working with.

The previous examples specified the `table` option, which declares the name of a relational table in a database. Many databases support the Structured Query Language (SQL), which can be used to dynamically compose a relational table from other tables in a database. The classic diet problem will be used to illustrate the use of SQL queries to initialize a Pyomo model. In this problem, a customer is faced with the task of minimizing the cost for a meal at a fast food restaurant – they must purchase a sandwich, side, and a drink for the lowest cost. The following is a Pyomo model for this problem:

```python
# diet1.py
from pyomo.environ import *

infinity = float('inf')
MAX_FOOD_SUPPLY = 20.0 # There is a finite food supply

model = AbstractModel()
```

4.2. Data Command Files
Suppose that the file `diet1.sqlite` be a SQLite database file that contains the following data in the `Food` table:

<table>
<thead>
<tr>
<th>FOOD</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cheeseburger</td>
<td>1.84</td>
</tr>
<tr>
<td>Ham Sandwich</td>
<td>2.19</td>
</tr>
<tr>
<td>Hamburger</td>
<td>1.84</td>
</tr>
<tr>
<td>Fish Sandwich</td>
<td>1.44</td>
</tr>
<tr>
<td>Chicken Sandwich</td>
<td>2.29</td>
</tr>
<tr>
<td>Fries</td>
<td>0.77</td>
</tr>
<tr>
<td>Sausage Biscuit</td>
<td>1.29</td>
</tr>
<tr>
<td>Lowfat Milk</td>
<td>0.60</td>
</tr>
<tr>
<td>Orange Juice</td>
<td>0.72</td>
</tr>
</tbody>
</table>
In addition, the Food table has two additional columns, f_min and f_max, with no data for any row. These columns exist to match the structure for the parameters used in the model.

We can solve the diet1 model using the Python definition in diet1.py and the data from this database. The file diet.sqlite.dat specifies a load command that uses the sqlite3 data manager and embeds a SQL query to retrieve the data:

```plaintext
# File diet.sqlite.dat
load "diet.sqlite"
  using=sqlite3
  query="SELECT FOOD,cost,f_min,f_max FROM Food"
  : FOOD=[FOOD] cost f_min f_max ;
```

The PyODBC driver module will pass the SQL query through an Access ODBC connector, extract the data from the diet1.mdb file, and return it to Pyomo. The Pyomo ODBC handler can then convert the data received into the proper format for solving the model internally. More complex SQL queries are possible, depending on the underlying database and ODBC driver in use. However, the name and ordering of the columns queried are specified in the Pyomo data file; using SQL wildcards (e.g., SELECT *) or column aliasing (e.g., SELECT f AS FOOD) may cause errors in Pyomo’s mapping of relational data to parameters.

### 4.2.6 The include Command

The include command allows a data command file to execute data commands from another file. For example, the following command file executes data commands from ex1.dat and then ex2.dat:

```plaintext
include ex1.dat;
include ex2.dat;
```

Pyomo is sensitive to the order of execution of data commands, since data commands can redefine set and parameter values. The include command respects this data ordering; all data commands in the included file are executed before the remaining data commands in the current file are executed.

### 4.2.7 The namespace Keyword

The namespace keyword is not a data command, but instead it is used to structure the specification of Pyomo’s data commands. Specifically, a namespace declaration is used to group data commands and to provide a group label. Consider the following data command file:

```plaintext
set C := 1 2 3 ;

namespace ns1
  { set C := 4 5 6 ; }

namespace ns2
  { set C := 7 8 9 ; }
```

This data file defines two namespaces: ns1 and ns2 that initialize a set C. By default, data commands contained within a namespace are ignored during model construction; when no namespaces are specified, the set C has values 1,2,3. When namespace ns1 is specified, then set C values are overridden with the set 4,5,6.
4.3 Data Portals

Pyomo’s DataPortal class standardizes the process of constructing model instances by managing the process of loading data from different data sources in a uniform manner. A DataPortal object can load data from the following data sources:

- **TAB File**: A text file format that uses whitespace to separate columns of values in each row of a table.
- **CSV File**: A text file format that uses comma or other delimiters to separate columns of values in each row of a table.
- **JSON File**: A popular lightweight data-interchange format that is easily parsed.
- **YAML File**: A human friendly data serialization standard.
- **XML File**: An extensible markup language for documents and data structures. XML files can represent tabular data.
- **Excel File**: A spreadsheet data format that is primarily used by the Microsoft Excel application.
- **Database**: A relational database.
- **DAT File**: A Pyomo data command file.

Note that most of these data formats can express tabular data.

**Warning**: The DataPortal class requires the installation of Python packages to support some of these data formats:

- **YAML File**: pyyaml
- **Excel File**: win32com, openpyxl or xlrd
  
  These packages support different data Excel data formats: the win32com package supports .xls, .xlsm and .xlsx, the openpyxl package supports .xlsx and the xlrd package supports .xls.
- **Database**: pyodbc, pypyodbc, sqlite3 or pymysql
  
  These packages support different database interface APIs: the pyodbc and pypyodbc packages support the ODBC database API, the sqlite3 package uses the SQLite C library to directly interface with databases using the DB-API 2.0 specification, and pymysql is a pure-Python MySQL client.

DataPortal objects can be used to initialize both concrete and abstract Pyomo models. Consider the file A.tab, which defines a simple set with a tabular format:

```
A
A1
A2
A3
```

The load method is used to load data into a DataPortal object. Components in a concrete model can be explicitly initialized with data loaded by a DataPortal object:

```python
data = DataPortal()
data.load(filename='A.tab', set="A", format="set")
model = ConcreteModel()
model.A = Set(initialize=data['A'])
```
All data needed to initialize an abstract model must be provided by a DataPortal object, and the use of the DataPortal object to initialize components is automated for the user:

```python
model = AbstractModel()
model.A = Set()
data = DataPortal()
data.load(filename='A.tab', set=model.A)
instance = model.create_instance(data)
```

Note the difference in the execution of the load method in these two examples: for concrete models data is loaded by name and the format must be specified, and for abstract models the data is loaded by component, from which the data format can often be inferred.

The load method opens the data file, processes it, and loads the data in a format that can be used to construct a model instance. The load method can be called multiple times to load data for different sets or parameters, or to override data processed earlier. The load method takes a variety of arguments that define how data is loaded:

- **filename**: This option specifies the source data file.
- **format**: This option specifies the how to interpret data within a table. Valid formats are: set, set_array, param, table, array, and transposed_array.
- **set**: This option is either a string or model compent that defines a set that will be initialized with this data.
- **param**: This option is either a string or model compent that defines a parameter that will be initialized with this data. A list or tuple of strings or model components can be used to define multiple parameters that are initialized.
- **index**: This option is either a string or model compent that defines an index set that will be initialized with this data.
- **using**: This option specifies the Python package used to load this data source. This option is used when loading data from databases.
- **select**: This option defines the columns that are selected from the data source. The column order may be changed from the data source, which allows the DataPortal object to define
- **namespace**: This option defines the data namespace that will contain this data.

The use of these options is illustrated below.

The DataPortal class also provides a simple API for accessing set and parameter data that are loaded from different data sources. The [ ] operator is used to access set and parameter values. Consider the following example, which loads data and prints the value of the [ ] operator:

```python
data = DataPortal()
data.load(filename='A.tab', set="A", format="set")
print(data['A'])  # ['A1', 'A2', 'A3']
data.load(filename='Z.tab', param="z", format="param")
print(data['z'])  # 1.1
data.load(filename='Y.tab', param="y", format="table")
for key in sorted(data['y']):
    print("%s %s" % (key, data['y'][key]))
```

The DataPortal class also has several methods for iterating over the data that has been loaded:

- **keys()**: Returns an iterator of the data keys.
- **values()**: Returns an iterator of the data values.
- **items()**: Returns an iterator of (name, value) tuples from the data.
Finally, the `data()` method provides a generic mechanism for accessing the underlying data representation used by `DataPortal` objects.

### 4.3.1 Loading Tabular Data

Many data sources supported by Pyomo are tabular data formats. Tabular data is numerical or textual data that is organized into one or more simple tables, where data is arranged in a matrix. Each table consists of a matrix of numeric string values, simple strings, and quoted strings. All rows have the same length, all columns have the same length, and the first row typically represents labels for the column data.

The following section describes the tabular data sources supported by Pyomo, and the subsequent sections illustrate ways that data can be loaded from tabular data using TAB files. Subsequent sections describe options for loading data from Excel spreadsheets and relational databases.

#### Tabular Data

TAB files represent tabular data in an ascii file using whitespace as a delimiter. A TAB file consists of rows of values, where each row has the same length. For example, the file `PP.tab` has the format:

```
A  B  PP
A1 B1 4.3
A2 B2 4.4
A3 B3 4.5
```

CSV files represent tabular data in a format that is very similar to TAB files. Pyomo assumes that a CSV file consists of rows of values, where each row has the same length. For example, the file `PP.csv` has the format:

```
A,B,PP
A1,B1,4.3
A2,B2,4.4
A3,B3,4.5
```

Excel spreadsheets can express complex data relationships. A `range` is a contiguous, rectangular block of cells in an Excel spreadsheet. Thus, a range in a spreadsheet has the same tabular structure as a TAB file or a CSV file. For example, consider the file `excel.xls` that has the range `PPtable`:

```
<table>
<thead>
<tr>
<th>PPtable</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>A1</td>
</tr>
<tr>
<td>A2</td>
</tr>
<tr>
<td>A3</td>
</tr>
</tbody>
</table>
```

A relational database is an application that organizes data into one or more tables (or `relations`) with a unique key in each row. Tables both reflect the data in a database as well as the result of queries within a database.

XML files represent tabular using `table` and `row` elements. Each sub-element of a `row` element represents a different column, where each row has the same length. For example, the file `PP.xml` has the format:

```
<table>
  <row>
    <A value="A1"/> <B value="B1"/> <PP value="4.3"/>
  </row>
  <row>
    <A value="A2"/> <B value="B2"/> <PP value="4.4"/>
  </row>
</table>
```
Loading Set Data

The `set` option is used specify a `Set` component that is loaded with data.

### Loading a Simple Set

Consider the file `A.tab`, which defines a simple set:

```
A
A1
A2
A3
```

In the following example, a `DataPortal` object loads data for a simple set `A`:

```python
model = AbstractModel()
model.A = Set()
data = DataPortal()
data.load(filename='A.tab', set=model.A)
instance = model.create_instance(data)
```

### Loading a Set of Tuples

Consider the file `C.tab`:

```
A B
A1 1
A1 2
A1 3
A2 1
A2 2
A2 3
A3 1
A3 2
A3 3
```

In the following example, a `DataPortal` object loads data for a two-dimensional set `C`:

```python
model = AbstractModel()
model.C = Set(dimen=2)
data = DataPortal()
data.load(filename='C.tab', set=model.C)
instance = model.create_instance(data)
```

In this example, the column titles do not directly impact the process of loading data. Column titles can be used to select a subset of columns from a table that is loaded (see below).
Loading a Set Array

Consider the file `D.tab`, which defines an array representation of a two-dimensional set:

```
B  A1  A2  A3
1  +  -  -
2  -  +  -
3  -  -  +
```

In the following example, a `DataPortal` object loads data for a two-dimensional set `D`:

```python
model = AbstractModel()
model.D = Set(dimen=2)
data = DataPortal()
data.load(filename='D.tab', set=model.D, format='set_array')
instance = model.create_instance(data)
```

The `format` option indicates that the set data is declared in an array format.

Loading Parameter Data

The `param` option is used to specify a `Param` component that is loaded with data.

Loading a Simple Parameter

The simplest parameter is simply a singleton value. Consider the file `Z.tab`:

```
1.1
```

In the following example, a `DataPortal` object loads data for a simple parameter `z`:

```python
model = AbstractModel()
data = DataPortal()
model.z = Param()
data.load(filename='Z.tab', param=model.z)
instance = model.create_instance(data)
```

Loading an Indexed Parameter

An indexed parameter can be defined by a single column in a table. For example, consider the file `Y.tab`:

```
A  Y
A1 3.3
A2 3.4
A3 3.5
```

In the following example, a `DataPortal` object loads data for an indexed parameter `y`:

```python
model = AbstractModel()
data = DataPortal()
model.A = Set(initialize=['A1','A2','A3'])
model.y = Param(model.A)
data.load(filename='Y.tab', param=model.y)
instance = model.create_instance(data)
```
When column names are not used to specify the index and parameter data, then the `DataPortal` object assumes that the rightmost column defines parameter values. In this file, the `A` column contains the index values, and the `Y` column contains the parameter values.

### Loading Set and Parameter Values

Note that the data for set `A` is predefined in the previous example. The index set can be loaded with the parameter data using the `index` option. In the following example, a `DataPortal` object loads data for set `A` and the indexed parameter `y`:

```python
model = AbstractModel()
data = DataPortal()
model.A = Set()
model.y = Param(model.A)
data.load(filename='Y.tab', param=model.y, index=model.A)
instance = model.create_instance(data)
```

An index set with multiple dimensions can also be loaded with an indexed parameter. Consider the file `PP.tab`:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>B1</td>
<td>4.3</td>
</tr>
<tr>
<td>A2</td>
<td>B2</td>
<td>4.4</td>
</tr>
<tr>
<td>A3</td>
<td>B3</td>
<td>4.5</td>
</tr>
</tbody>
</table>

In the following example, a `DataPortal` object loads data for a tuple set and an indexed parameter:

```python
model = AbstractModel()
data = DataPortal()
model.A = Set(dimen=2)
model.p = Param(model.A)
data.load(filename='PP.tab', param=model.p, index=model.A)
instance = model.create_instance(data)
```

### Loading a Parameter with Missing Values

Missing parameter data can be expressed in two ways. First, parameter data can be defined with indices that are a subset of valid indices in the model. The following example loads the indexed parameter `y`:

```python
model = AbstractModel()
data = DataPortal()
model.A = Set(initialize=['A1','A2','A3','A4'])
model.y = Param(model.A)
data.load(filename='Y.tab', param=model.y)
instance = model.create_instance(data)
```

The model defines an index set with four values, but only three parameter values are declared in the data file `Y.tab`. Parameter data can also be declared with missing values using the period (.) symbol. For example, consider the file `S.tab`:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>B1</td>
<td>4.3</td>
</tr>
<tr>
<td>A2</td>
<td>B2</td>
<td>4.4</td>
</tr>
<tr>
<td>A3</td>
<td>B3</td>
<td>4.5</td>
</tr>
</tbody>
</table>

4.3. Data Portals
In the following example, a `DataPortal` object loads data for the index set `A` and indexed parameter `y`:

```python
model = AbstractModel()
data = DataPortal()
model.A = Set()
model.s = Param(model.A)
data.load(filename='S.tab', param=model.s, index=model.A)
instance = model.create_instance(data)
```

The period (.) symbol indicates a missing parameter value, but the index set `A` contains the index value for the missing parameter.

**Loading Multiple Parameters**

Multiple parameters can be initialized at once by specifying a list (or tuple) of component parameters. Consider the file `XW.tab`:

```
A   X   W
A1  3.3 4.3
A2  3.4 4.4
A3  3.5 4.5
```

In the following example, a `DataPortal` object loads data for parameters `x` and `w`:

```python
model = AbstractModel()
data = DataPortal()
model.A = Set(initialize=['A1','A2','A3'])
model.x = Param(model.A)
model.w = Param(model.A)
data.load(filename='XW.tab', param=(model.x,model.w))
instance = model.create_instance(data)
```

**Selecting Parameter Columns**

We have previously noted that the column names do not need to be specified to load set and parameter data. However, the `select` option can be to identify the columns in the table that are used to load parameter data. This option specifies a list (or tuple) of column names that are used, in that order, to form the table that defines the component data.

For example, consider the following load declaration:

```python
model = AbstractModel()
data = DataPortal()
model.A = Set()
model.w = Param(model.A)
data.load(filename='XW.tab', select=('A','W'),
        param=model.w, index=model.A)
instance = model.create_instance(data)
```

The columns `A` and `W` are selected from the file `XW.tab`, and a single parameter is defined.

**Loading a Parameter Array**

Consider the file `U.tab`, which defines an array representation of a multiply-indexed parameter:

```python
36 Chapter 4. Managing Data in Pyomo Models
```
In the following example, a `DataPortal` object loads data for a two-dimensional parameter `u`:

```python
model = AbstractModel()
data = DataPortal()
model.A = Set(initialize=['A1', 'A2', 'A3'])
model.I = Set(initialize=['I1', 'I2', 'I3', 'I4'])
model.u = Param(model.I, model.A)
data.load(filename='U.tab', param=model.u,
          format='array')
instance = model.create_instance(data)
```

The `format` option indicates that the parameter data is declared in an array format. The `format` option can also indicate that the parameter data should be transposed.

```python
model = AbstractModel()
data = DataPortal()
model.A = Set(initialize=['A1', 'A2', 'A3'])
model.I = Set(initialize=['I1', 'I2', 'I3', 'I4'])
model.t = Param(model.A, model.I)
data.load(filename='U.tab', param=model.t,
          format='transposed_array')
instance = model.create_instance(data)
```

Note that the transposed parameter data changes the index set for the parameter.

### Loading from Spreadsheets and Databases

Tabular data can be loaded from spreadsheets and databases using auxiliary Python packages that provide an interface to these data formats. Data can be loaded from Excel spreadsheets using the `win32com`, `xlrd`, and `openpyxl` packages. For example, consider the following range of cells, which is named `PPtable`:

<table>
<thead>
<tr>
<th></th>
<th>A1</th>
<th>B2</th>
<th>PP</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>4.1</td>
<td></td>
<td>4.3</td>
</tr>
<tr>
<td>A2</td>
<td>4.2</td>
<td></td>
<td>4.4</td>
</tr>
<tr>
<td>A3</td>
<td>4.3</td>
<td></td>
<td>4.5</td>
</tr>
</tbody>
</table>

In the following example, a `DataPortal` object loads the named range `PPtable` from the file `excel.xls`:

```python
model = AbstractModel()
data = DataPortal()
model.A = Set(dimen=2)
model.p = Param(model.A)
data.load(filename='excel.xls', range='PPtable',
         param=model.p, index=model.A)
instance = model.create_instance(data)
```

Note that the `range` option is required to specify the table of cell data that is loaded from the spreadsheet.
There are a variety of ways that data can be loaded from a relational database. In the simplest case, a table can be specified within a database:

```python
model = AbstractModel()
data = DataPortal()
model.A = Set(dimen=2)
model.p = Param(model.A)
data.load(filename='PP.sqlite', using='sqlite3',
          table='PPtable',
          param=model.p, index=model.A)
instance = model.create_instance(data)
```

In this example, the interface `sqlite3` is used to load data from an SQLite database in the file `PP.sqlite`. More generally, an SQL query can be specified to dynamically generate a table. For example:

```python
model = AbstractModel()
data = DataPortal()
model.A = Set()
model.p = Param(model.A)
data.load(filename='PP.sqlite', using='sqlite3',
          query="SELECT A,PP FROM PPtable",
          param=model.p, index=model.A)
instance = model.create_instance(data)
```

### 4.3.2 Data Namespaces

The `DataPortal` class supports the concept of a namespace to organize data into named groups that can be enabled or disabled during model construction. Various `DataPortal` methods have an optional namespace argument that defaults to `None`:

- `data(name=None, namespace=None)`: Returns the data associated with data in the specified namespace
- `[]`: For a `DataPortal` object `data`, the function `data['A']` returns data corresponding to `A` in the default namespace, and `data['ns1','A']` returns data corresponding to `A` in namespace `ns1`.
- `namespaces()`: Returns an iterator for the data namespaces.
- `keys(namespace=None)`: Returns an iterator of the data keys in the specified namespace.
- `values(namespace=None)`: Returns an iterator of the data values in the specified namespace.
- `items(namespace=None)`: Returns an iterator of (name, value) tuples in the specified namespace.

By default, data within a namespace are ignored during model construction. However, concrete models can be initialized with data from a specific namespace. Further, abstract models can be initialized with a list of namespaces that define the data used to initialized model components. For example, the following script generates two model instances from an abstract model using data loaded into different namespaces:

```python
model = AbstractModel()
model.C = Set(dimen=2)
data = DataPortal()
data.load(filename='C.tab', set=model.C, namespace='ns1')
data.load(filename='D.tab', set=model.C, namespace='ns2',
          format='set_array')
instance1 = model.create_instance(data, namespaces=['ns1'])
instance2 = model.create_instance(data, namespaces=['ns2'])
```
4.4 Storing Data from Pyomo Models

Currently, Pyomo has rather limited capabilities for storing model data into standard Python data types and serialized data formats. However, this capability is under active development.

4.4.1 Serialized Data Formats

JSON and YAML files are structured data formats that are well-suited for data serialization. These data formats do not represent data in tabular format, but instead they directly represent set and parameter values with lists and dictionaries:

- **Simple Set**: a list of string or numeric value
- **Indexed Set**: a dictionary that maps an index to a list of string or numeric value
- **Simple Parameter**: a string or numeric value
- **Indexed Parameter**: a dictionary that maps an index to a numeric value

For example, consider the following JSON file:

```json
{
    "A": ["A1", "A2", "A3"],
    "B": [[1, "B1"], [2, "B2"], [3, "B3"],
    "C": {"A1": [1, 2, 3], "A3": [10, 20, 30]},
    "p": 0.1,
    "q": {"A1": 3.3, "A2": 3.4, "A3": 3.5},
    "r": [
        {"index": [1, "B1"], "value": 3.3},
        {"index": [2, "B2"], "value": 3.4},
        {"index": [3, "B3"], "value": 3.5}]
}
```

The data in this file can be used to load the following model:

```python
model = AbstractModel()
data = DataPortal()
model.A = Set()
model.B = Set(dimen=2)
model.C = Set(model.A)
model.p = Param()
model.q = Param(model.A)
model.r = Param(model.B)
data.load(filename='T.json')
```

Note that no set or param option needs to be specified when loading a JSON or YAML file. All of the set and parameter data in the file are loaded by the DataPortal object, and only the data needed for model construction is used.

The following YAML file has a similar structure:

```yaml
A: [A1, A2, A3]
- [1, B1]
- [2, B2]
- [3, B3]
C:
  'A1': [1, 2, 3]
  'A3': [10, 20, 30]
p: 0.1
q: {A1: 3.3, A2: 3.4, A3: 3.5}
r:
```
The data in this file can be used to load a Pyomo model with the same syntax as a JSON file:

```python
model = AbstractModel()
data = DataPortal()
model.A = Set()
model.B = Set(dimen=2)
model.C = Set(model.A)
model.p = Param()
model.q = Param(model.A)
model.r = Param(model.B)
data.load(filename='T.yaml')
```

### 4.4.2 Storing Model Data in Excel

**TODO**

More here.
Scripting examples...

```python
>>> print('Hello World')
Hello World
```
6.1 Bilevel Programming

TODO

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

6.2 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.

6.2.1 Modeling Components

Pyomo.DAE introduces three new modeling components to Pyomo:

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pyomo.dae.ContinuousSet</code></td>
<td>Represents a bounded continuous domain</td>
</tr>
<tr>
<td><code>pyomo.dae.DerivativeVar</code></td>
<td>Represents derivatives in a model and defines how a</td>
</tr>
</tbody>
</table>
As will be shown later, differential equations can be declared 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.

```python
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

        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 discretized 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.

6.2. Dynamic Optimization with pyomo.DAE
```python
set t := 0 0.5 2.25 3.75 5;
```

Required imports

```python
>>> 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.

```python
>>> 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**

```python
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`

```python
def 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.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))
```

6.2. Dynamic Optimization with pyomo.DAE 47
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.

### 6.2.2 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.

```python
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.

```
Deactivate the differential equations at certain boundary points

```python
>>> 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.

### 6.2.3 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.

```python
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:

```python
>>> 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]
```

6.2. Dynamic Optimization with pyomo.DAE
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:

\[
\sum_s \int_{t_2} \int_{t_1} X(t_1, t_2, s) \, dt_1 \, dt_2
\]

6.2.4 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:

\[ \frac{dx}{dt} = f(t, x), \quad x(t_0) = x_0 \]

discretize \( t \) and \( x \) such that

\[
\begin{align*}
x(t_0 + kh) & = x_k \\
x_{k+1} & = x_k + h \cdot f(t_{k+1}, x_{k+1}) \\
t_{k+1} & = t_k + h
\end{align*}
\]

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 `ContinuousSet` 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.

```python
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):
```

6.2. Dynamic Optimization with pyomo.DAE
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.

```python
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.

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

reduce_collocation_points(self, 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:

```python
>>> 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. 6.1 shows the solution profile before and after applying 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:

```python
>>> 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:

```python
>>> 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:

```python
>>> 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. 6.1: (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 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:

```python
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.

### 6.2.5 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.

```python
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'`: CV odes 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:

\[
\frac{d\theta}{dt} = \omega \\
\frac{d\omega}{dt} = -b \times \omega - c \times \sin(\theta)
\]

We begin by formulating the model using Pyomo.DAE

```python
>>> 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.

```python
>>> 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.

```python
>>> 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-Varying 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.

```python
>>> 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 \( m.t \))

### 6.2.6 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**
```python
>>> sim = Simulator(m, package='scipy')
>>> tsim, profiles = sim.simulate(numpoints=100, integrator='vode',
...                               varying_inputs=m.var_input)
```
Discretize the model using Orthogonal Collocation

```python
>>> discretizer = TransformationFactory('dae.collocation')
>>> discretizer.apply_to(m, nfe=10, ncp=3)
```

Initialize the discretized model using the simulator profiles

```python
>>> sim.initialize_model()
```

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

### 6.3 Stochastic Programming

TODO

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

### 6.4 Generalized Disjunctive Programming

TODO

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

### 6.5 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.
This section provides documentation about fundamental capabilities in Pyomo. This documentation serves as a reference for both (1) Pyomo developers and (2) advanced users who are developing Python scripts using Pyomo.

### 7.1 Pyomo Expressions

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

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

```plaintext
M = ConcreteModel()
M.v = Var()
e = M.v*2
```

Python determines that the magic method `__mul__` is called on the `M.v` object, with the argument 2. This method returns a Pyomo expression object `ProductExpression` that has arguments `M.v` and 2. This represents the following symbolic expression tree:
Note: End-users will not likely need to know details related to how symbolic expressions are generated and managed in Pyomo. Thus, most of the following documentation of expressions in Pyomo is most useful for Pyomo developers. However, the discussion of runtime performance in the first section will help end-users write large-scale models.

7.1.1 Building Expressions Faster

Expression Generation

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

```python
M = ConcreteModel()
M.x = Var(range(5))
s = 0
for i in range(5):
    s = s + M.x[i]
```

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

```python
s = sum(M.x[i] for i in range(5))
```

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

Linear, Quadratic and General Nonlinear Expressions

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

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

For example, consider the following quadratic polynomial:

```python
s = sum(M.x[i] for i in range(5))**2
```

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

**Pyomo Utility Functions**

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

- **prod** A function to compute a product of Pyomo expressions.
- **quicksum** A function to efficiently compute a sum of Pyomo expressions.
- **sum_product** A function that computes a generalized dot product.

**prod**

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

```python
M = ConcreteModel()
M.x = Var(range(5))
M.z = Var()

# The product M.x[0] * M.x[1] * ... * M.x[4]
e1 = prod(M.x[i] for i in M.x)

# The product M.x[0]*M.z
e2 = prod([M.x[0], M.z])

# The product M.z*(M.x[0] + ... + M.x[4])
e3 = prod([sum(M.x[i] for i in M.x), M.z])
```

**quicksum**

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

```python
M = ConcreteModel()
M.x = Var(range(5))

# Summation using the Python sum() function
e1 = sum(M.x[i]**2 for i in M.x)

# Summation using the Pyomo quicksum function
e2 = quicksum(M.x[i]**2 for i in M.x)
```
The summation is customized based on the `start` and `linear` arguments. The `start` defines the initial value for summation, which defaults to zero. If `start` is a numeric value, then the `linear` argument determines how the sum is processed:

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

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

Consider the following example:

```python
M = ConcreteModel()
M.A = RangeSet(100000)
M.p = Param(M.A, mutable=True, initialize=1)
M.x = Var(M.A)

start = time.time()
e = quicksum((M.x[i] - 1)**M.p[i] for i in M.A)
print("quicksum: %f" % (time.time() - start))

start = time.time()
generate_standard_repn(e)
print("repn: %f" % (time.time() - start))

start = time.time()
e = quicksum((M.x[i] - 1)**M.p[i] for i in M.A)
print("quicksum: %f" % (time.time() - start))

start = time.time()
generate_standard_repn(e)
print("repn: %f" % (time.time() - start))
```

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

```
sum: 1.447861
repn: 0.870225
quicksum: 1.388344
repn: 0.864316
```

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

**Warning:** By default, `linear` is `None`. While this allows for efficient expression generation in normal cases, there are circumstances where the inspection of the first term in `args` is misleading. Consider the following example:

```python
M = ConcreteModel()
M.x = Var(range(5))
e = quicksum(M.x[i]**2 if i > 0 else M.x[i] for i in range(5))
```

64 Chapter 7. Developer Reference
The first term created by the generator is linear, but the subsequent terms are nonlinear. Pyomo gracefully transitions to a nonlinear sum, but in this case quicksum is doing additional work that is not useful.

**sum_product**

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

```python
M = ConcreteModel()
M.z = RangeSet(5)
M.x = Var(range(10))
M.y = Var(range(10))

# Sum the elements of x
e1 = sum_product(M.x)

# Sum the product of elements in x and y
e2 = sum_product(M.x, M.y)

# Sum the product of elements in x and y, over the index set z
e3 = sum_product(M.x, M.y, index=M.z)
```

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

```python
# Sum the product of x_i/y_i
e1 = sum_product(M.x, denom=M.y)

# Sum the product of 1/(x_i*y_i)
e2 = sum_product(denom=(M.x, M.y))
```

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

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

### 7.1.2 Design Overview

#### Historical Comparison

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

- Pyomo5 supports both CPython and PyPy implementations of Python, while Coopr3 only supports CPython.
  
  The key difference in these implementations is that Coopr3 relies on CPython reference counting, which is not part of the Python language standard. Hence, this implementation is not guaranteed to run on other implementations of Python.

  Pyomo5 does not rely on reference counting, and it has been tested with PyPy. In the future, this should allow Pyomo to support other Python implementations (e.g. Jython).
• Pyomo5 expression objects are immutable, while Coopr3 expression objects are mutable.

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

• Pyomo5 provides more consistent runtime performance than Coopr3.

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

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

```python
M = ConcreteModel()
M.x = Var(range(100))

# This loop is fast.
e = 0
for i in range(100):
    e = e + M.x[i]

# This loop is slow.
e = 0
for i in range(100):
    e = M.x[i] + e
```

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

```python
M = ConcreteModel()
M.p = Param(initialize=3)
M.q = 1/M.p
M.x = Var(range(100))

# The value M.q is cloned every time it is used.
e = 0
for i in range(100):
    e = e + M.x[i]*M.q
```

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

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

**Expression Entanglement and Mutability**

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

• Python variables can point to the same expression tree
M = ConcreteModel()
M.v = Var()
e = f = 2*M.v

This is illustrated as follows:

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

M = ConcreteModel()
M.v = Var()
e = 2*M.v
f = e + 3

This is illustrated as follows:
Two expression trees can point to the same sub-tree

```python
M = ConcreteModel()
M.v = Var()

e = 2*M.v
f = e + 3
g = e + 4
```

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

### Entangled Sub-Expressions

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

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

```python
M = ConcreteModel()
M.v = Var()
M.w = Var()

e = 2*M.v
f = e + 3
e += M.w
```

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

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

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

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

\[
\begin{align*}
    e + f + \cdot w & \cdot 2 \cdot v \\
    & + \\
    w & \cdot 3 \\
    & + \\
    2 & \cdot v \\
\end{align*}
\]

With Pyomo5 expressions, each sub-expression is immutable. Thus, the summation operation generates a new expression $e$ without changing existing expression objects referenced in the expression tree for $f$. By contrast, Coopr3 imposes the same property by cloning the expression $e$ before added $M \cdot w$, resulting in the following:
This example also illustrates that leaves may be shared between expressions.

**Mutable Expression Components**

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

```python
M = ConcreteModel()
M.v = Var()
M.w = Var()

M.e = Expression(expr=2*M.v)
f = M.e + 3
M.e += M.w
```

Here, the expression `M.e` is a so-called *named expression* that the user has declared. Named expressions are explicitly intended for re-use within models, and they provide a convenient mechanism for changing sub-expressions in complex applications. In this example, the expression tree is as follows before `M.w` is added:
And the expression tree is as follows after $M.w$ is added.
When considering named expressions, Pyomo5 and Coopr3 enforce the following semantics:

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

### 7.1.3 Design Details

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

1. Interior nodes are objects that inherit from the `ExpressionBase` class. These objects typically have one or more child nodes. Linear expression nodes do not have child nodes, but they are treated as interior nodes in the expression tree because they references other leaf nodes.

2. Leaf nodes are numeric values, parameter components and variable components, which represent the inputs to the expression.

**Expression Classes**

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Python Syntax</th>
<th>Pyomo Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>sum</td>
<td>x + y</td>
<td>SumExpression</td>
</tr>
<tr>
<td>product</td>
<td>x * y</td>
<td>ProductExpression</td>
</tr>
<tr>
<td>negation</td>
<td>- x</td>
<td>NegationExpression</td>
</tr>
<tr>
<td>reciprocal</td>
<td>1 / x</td>
<td>ReciprocalExpression</td>
</tr>
<tr>
<td>power</td>
<td>x ** y</td>
<td>PowExpression</td>
</tr>
<tr>
<td>inequality</td>
<td>x &lt;= y</td>
<td>InequalityExpression</td>
</tr>
<tr>
<td>equality</td>
<td>x == y</td>
<td>EqualityExpression</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Pyomo Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>external function</td>
<td>myfunc(x,y,z)</td>
<td>ExternalFunctionExpression</td>
</tr>
<tr>
<td>logical if-then-else</td>
<td>Expr_if(IF=x, THEN=y, ELSE=z)</td>
<td>Expr_ifExpression</td>
</tr>
<tr>
<td>intrinsic function</td>
<td>sin(x)</td>
<td>UnaryFunctionExpression</td>
</tr>
<tr>
<td>absolute function</td>
<td>abs(x)</td>
<td>AbsExpression</td>
</tr>
</tbody>
</table>

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

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

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

Expression trees can be categorized in four different ways:

- constant expressions - expressions that do not contain numeric constants and immutable parameters.
- mutable expressions - expressions that contain mutable parameters but no variables.
- potentially variable expressions - expressions that contain variables, which may be fixed.
- fixed expressions - expressions that contain variables, all of which are fixed.
These three categories are illustrated with the following example:

```python
m = ConcreteModel()
m.p = Param(default=10, mutable=False)
m.q = Param(default=10, mutable=True)
m.x = Var()
m.y = Var(initialize=1)
m.y.fixed = True
```

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

<table>
<thead>
<tr>
<th>Category</th>
<th>m.p</th>
<th>m.q</th>
<th>m.x</th>
<th>m.y</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>True</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>not potentially variable</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>potentially_variable</td>
<td>False</td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>fixed</td>
<td>True</td>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>

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

**Special Expression Classes**

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

**Named Expressions**

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

```python
M = ConcreteModel()
M.v = Var()
M.w = Var()
M.e = Expression(expr=2*M.v)

f = M.e + 3  # f == 2*v + 3
M.e += M.w   # f == 2*v + 3 + w
```

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

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

**Linear Expressions**

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

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

**Sum Expressions**

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

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

**Mutable Expressions**

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

- `_MutableSumExpression` This class is used in the `nonlinear_expression` context manager to efficiently combine sums of nonlinear terms.
- `_MutableLinearExpression` This class is used in the `linear_expression` context manager to efficiently combine sums of linear terms.

**Expression Semantics**

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

The following classes are valid interior nodes:

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

The following classes are valid leaf nodes:

- Members of `nonpyomo_leaf_types`, which includes standard numeric data types like `int`, `float`, and `long`, as well as numeric data types defined by `numpy` and other commonly used packages. This set also includes `NonNumericValue`, which is used to wrap non-numeric arguments to the `ExternalFunctionExpression` class.
- Parameter component classes like `SimpleParam` and `_ParamData`, which arise in expression trees when the parameters are declared as mutable. (Immutable parameters are identified when generating expressions, and they are replaced with their associated numeric value.)
- Variable component classes like `SimpleVar` and `_GeneralVarData`, which often arise in expression trees.
Note: In some contexts the LinearExpression class can be treated as an interior node, and sometimes it can be treated as a leaf. This expression object does not have any child arguments, so nargs() is zero. But this expression references variables and parameters in a linear expression, so in that sense it does not represent a leaf node in the tree.

Context Managers

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

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

```python
M = ConcreteModel()
M.x = Var(range(5))
s = 0
for i in range(5):
    s += M.x[i]

with linear_expression() as e:
    for i in range(5):
        e += M.x[i]
```

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

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

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

```python
M = ConcreteModel()
M.x = Var(range(5))
M.y = Var(range(5))

with linear_expression() as e:
    quicksum((M.x[i] for i in M.x), start=e)
    quicksum((M.y[i] for i in M.y), start=e)
```

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

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

Creating a String Representation of an Expression

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

Algebraic vs. Nested Functional Form

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

```python
from pyomo.core.expr import current as EXPR

M = ConcreteModel()
M.x = Var()
e = sin(M.x) + 2*M.x

# sin(x) + 2*x
print(EXPR.expression_to_string(e))

# sum(sin(x), prod(2, x))
print(EXPR.expression_to_string(e, verbose=True))
```

Labeler and Symbol Map

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

```python
from pyomo.core.expr import current as EXPR

M = ConcreteModel()
M.x = Var()
M.y = Var()
e = sin(M.x) + 2*M.y

# sin(x1) + 2*x2
print(EXPR.expression_to_string(e, labeler=NumericLabeler('x')))  
```

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

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

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

```python
from pyomo.core.expr import current as EXPR

M = ConcreteModel()
M.x = Var()
M.y = Var()

e = sin(M.x) + 2*M.y + M.x*M.y - 3
# -3 + 2*y + sin(x) + x*y
print(EXPR.expression_to_string(e, standardize=True))
```

Other Ways to Generate String Representations

There are two other standard ways to generate string representations:

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

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

Cloning Expressions

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

For example:

```python
from pyomo.core.expr import current as EXPR

M = ConcreteModel()
M.x = Var()

with EXPR.clone_counter() as counter:
    start = counter.count
    e1 = sin(M.x)
    e2 = e1.clone()
    total = counter.count - start
    assert(total == 1)
```
Evaluating Expressions

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

```python
M = ConcreteModel()
M.x = Var()
M.x.value = math.pi/2.0
val = value(M.x)
assert (math.isclose(val, math.pi/2.0))
```

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

```python
val = M.x()
assert (math.isclose(val, math.pi/2.0))
```

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

```python
M = ConcreteModel()
M.x = Var()
val = value(M.x, exception=False)
assert (val is None)
```

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

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

Identifying Components and Variables

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

```python
from pyomo.core.expr import current as EXPR
M = ConcreteModel()
M.x = Var()
M.p = Param(mutable=True)
e = M.p+M.x
s = set([type(M.p)])
assert (list(EXPR.identify_components(e, s)) == [M.p])
```

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

```python
from pyomo.core.expr import current as EXPR
```
M = ConcreteModel()
M.x = Var()
M.y = Var()

e = M.x + M.y
M.y.value = 1
M.y.fixed = True

assert(set(id(v) for v in EXPR.identify_variables(e)) == set([id(M.x), id(M.y)]))
assert(set(id(v) for v in EXPR.identify_variables(e, include_fixed=False)) == set([id(M.x)]))

Walking an Expression Tree with a Visitor Class

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

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

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

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

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

These classes define a variety of suitable tree search methods:

- **SimpleExpressionVisitor**
  - `xbfs`: breadth-first search where leaf nodes are immediately visited
  - `xbfs_yield_leaves`: breadth-first search where leaf nodes are immediately visited, and the visit method yields a value

- **ExpressionValueVisitor**
  - `dfs_postorder_stack`: postorder depth-first search using a stack

- **ExpressionReplacementVisitor**
  - `dfs_postorder_stack`: postorder depth-first search using a stack

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

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

`visitor()` Defines the operation that is performed when a node is visited. In the `ExpressionValueVisitor` and `ExpressionReplacementVisitor` visitor classes, this method returns a value that is used by its parent node.
visiting_potential_leaf() Checks if the search should terminate with this node. If no, then this method returns the tuple \((False, None)\). If yes, then this method returns \((False, value)\), where \(value\) is computed by this method. This method is not used in the \texttt{SimpleExpressionVisitor} visitor class.

finalize() This method defines the final value that is returned from the visitor. This is not normally redefined. Detailed documentation of the APIs for these methods is provided with the class documentation for these visitors.

**SimpleExpressionVisitor Example**

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

```python
from pyomo.core.expr import current as EXPR
class SizeofVisitor(EXPR.SimpleExpressionVisitor):
    def __init__(self):
        self.counter = 0
    def visit(self, node):
        self.counter += 1
    def finalize(self):
        return self.counter
```

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

```python
def sizeof_expression(expr):
    # Create the visitor object
    visitor = SizeofVisitor()
    # Compute the value using the :func:`xbfs` search method.
    return visitor.xbfs(expr)
```

**ExpressionValueVisitor Example**

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

```python
from pyomo.core.expr import current as EXPR
class CloneVisitor(EXPR.ExpressionValueVisitor):
    def __init__(self):
        self.memo = {'__block_scope__': { id(None): False }}
    def visit(self, node, values):
        # Clone the interior node
        return visitor.xbfs(expr)
```

```python
def sizeof_expression(expr):
    # Create the visitor object
    visitor = SizeofVisitor()
    # Compute the value using the :func:`xbfs` search method.
    return visitor.xbfs(expr)
```
def visiting_potential_leaf(self, node):
    # Clone leaf nodes in the expression tree
    if node.__class__ in native_numeric_types or
       node.__class__ not in pyomo5_expression_types:
        return False, copy.deepcopy(node, self.memo)
    return True, node

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

```python
from pyomo.core.expr import current as EXPR
class ScalingVisitor(EXPR.ExpressionReplacementVisitor):

    def __init__(self, scale):
        super(ScalingVisitor, self).__init__()
        self.scale = scale

    def visiting_potential_leaf(self, node):
        # Clone leaf nodes in the expression tree
        if node.is_variable_type():
            return True, self.scale[id(node)]*node
        if isinstance(node, EXPR.LinearExpression):
            node_ = copy.deepcopy(node)
            node_.constant = node.constant
            node_.linear_vars = copy.copy(node.linear_vars)
            node_.linear_coefs = []
            for i, v in enumerate(node.linear_vars):
                node_.linear_coefs.append( node.linear_coefs[i]*self.scale[id(v)] )
            return True, node_
```
No visit() method needs to be defined. The visiting_potential_leaf() function identifies variable nodes and returns a product expression that contains a mutable parameter. The _LinearExpression class has a different representation that embeds variables. Hence, this class must be handled in a separate condition that explicitly transforms this sub-expression.

```python
def scale_expression(expr, scale):
    # Create the visitor object
    visitor = ScalingVisitor(scale)
    # Scale the expression using the :func:`dfs_postorder_stack`
    # search method.
    return visitor.dfs_postorder_stack(expr)
```

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

```python
M = ConcreteModel()
M.x = Var(range(5))
M.p = Param(range(5), mutable=True)

scale={}
for i in M.x:
    scale[id(M.x[i])] = M.p[i]

e = quicksum(M.x[i] for i in M.x)
f = scale_expression(e, scale)

print(f)
```

### 7.2 Optimization Interfaces

**Warning:** This is draft documentation for Pyomo’s optimization interface classes. This includes both the core interface classes as well as the solver interfaces supported by Pyomo.
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 maintains 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!

### 8.1 Using Persistent Solvers

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

```python
>>> 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.

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

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

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

This will create a gurobipy Model object and include the appropriate variables and constraints. We can now solve the model.
We can also add or remove variables, constraints, blocks, and objectives. For example,

```py
>>> 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.

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

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

```py
>>> 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 \( y \geq -2x + 5 \) and \( y \leq x \), which is not what was intended!

```py
>>> 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:

```py
>>> 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.
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:

```python
>>> 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)
```

### 8.2 Persistent Solver Performance

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

```python
>>> 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=True)
```

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

```python
>>> 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:

```python
>>> 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:

```python
>>> results = opt.solve(save_results=False, load_solutions=False)
>>> if results.solver.termination_condition == TerminationCondition.optimal:
...   opt.load_vars(m.x)
```
Pyomo is being increasingly used as a library to support Python scripts. This section describes library APIs for key elements of Pyomo’s core library. This documentation serves as a reference for both (1) Pyomo developers and (2) advanced users who are developing Python scripts using Pyomo.

9.1 Expression Reference

9.1.1 Utilities to Build Expressions

`pyomo.core.util.prod(terms)`
A utility function to compute the product of a list of terms.

**Parameters**
- `terms (list)` – A list of terms that are multiplied together.

**Returns**
The value of the product, which may be a Pyomo expression object.

`pyomo.core.util.quicksum(args, start=0, linear=None)`
A utility function to compute a sum of Pyomo expressions.

The behavior of `quicksum()` is similar to the built-in `sum()` function, but this function generates a more compact Pyomo expression.

**Parameters**
- `args` – A generator for terms in the sum.
- `start` – A value that initializes the sum. If this value is not a numeric constant, then the `+=` operator is used to add terms to this object. Defaults to zero.
- `linear` – If `start` is not a numeric constant, then this option is ignored. Otherwise, this value indicates whether the terms in the sum are linear. If the value is `False`, then the terms are treated as nonlinear, and if `True`, then the terms are treated as linear. Default is `None`, which indicates that the first term in the `args` is used to determine this value.

**Returns**
The value of the sum, which may be a Pyomo expression object.
pyomo.core.util.sum_product(*args, **kwds)
A utility function to compute a generalized dot product.

This function accepts one or more components that provide terms that are multiplied together. These products are added together to form a sum.

Parameters
- **args** – Variable length argument list of generators that create terms in the summation.
- **kwds** – Arbitrary keyword arguments.

Keyword Arguments
- **index** – A set that is used to index the components used to create the terms.
- **denom** – A component or tuple of components that are used to create the denominator of the terms.
- **start** – The initial value used in the sum.

Returns The value of the sum.

pyomo.core.util.summation = <function sum_product>
An alias for sum_product

pyomo.core.util.dot_product = <function sum_product>
An alias for sum_product

### 9.1.2 Utilities to Manage and Analyze Expressions

#### Functions

pyomo.core.expr.current.expression_to_string(expr, verbose=None, labeler=None, smap=None, compute_values=False, standardize=False)

Return a string representation of an expression.

Parameters
- **expr** – The root node of an expression tree.
- **verbose** (bool) – If True, then the output is a nested functional form. Otherwise, the output is an algebraic expression. Default is False.
- **labeler** – If specified, this labeler is used to label variables in the expression.
- **smap** – If specified, this SymbolMap is used to cache labels.
- **compute_values** (bool) – If True, then parameters and fixed variables are evaluated before the expression string is generated. Default is False.
- **standardize** (bool) – If True and verbose is False, then the expression form is standardized to pull out constant and linear terms. Default is False.

Returns A string representation for the expression.

pyomo.core.expr.current.decompose_term(expr)
A function that returns a tuple consisting of (1) a flag indicated whether the expression is linear, and (2) a list of tuples that represents the terms in the linear expression.

Parameters **expr** (expression) – The root node of an expression tree
Returns A tuple with the form \((\text{flag}, \text{list})\). If \text{flag} is False, then a nonlinear term has been found, and \text{list} is None. Otherwise, \text{list} is a list of tuples: \((\text{coef}, \text{value})\). If \text{value} is None, then this represents a constant term with value \text{coef}. Otherwise, \text{value} is a variable object, and \text{coef} is the numeric coefficient.

\text{pyomo.core.expr.current.clone_expression}(\text{expr, memo=None, clone_leaves=True})

A function that is used to clone an expression.

Cloning is roughly equivalent to calling \text{copy.deepcopy}. However, the \text{clone_leaves} argument can be used to clone only interior (i.e. non-leaf) nodes in the expression tree. Note that named expression objects are treated as leaves when \text{clone_leaves} is True, and hence those subexpressions are not cloned.

This function uses a non-recursive logic, which makes it more scalable than the logic in \text{copy.deepcopy}.

Parameters

- \text{expr} – The expression that will be cloned.
- \text{memo (dict)} – A dictionary mapping object ids to objects. This dictionary has the same semantics as the memo object used with \text{copy.deepcopy}. Defaults to None, which indicates that no user-defined dictionary is used.
- \text{clone_leaves (bool)} – If True, then leaves are cloned along with the rest of the expression. Defaults to True.

Returns The cloned expression.

\text{pyomo.core.expr.current.evaluate_expression}(\text{exp, exception=True})

Evaluate the value of the expression.

Parameters

- \text{expr} – The root node of an expression tree.
- \text{exception (bool)} – A flag that indicates whether exceptions are raised. If this flag is False, then an exception that occurs while evaluating the expression is caught and the return value is None. Default is True.

Returns A floating point value if the expression evaluates normally, or None if an exception occurs and is caught.

\text{pyomo.core.expr.current.identify_components}(\text{expr, component_types})

A generator that yields a sequence of nodes in an expression tree that belong to a specified set.

Parameters

- \text{expr} – The root node of an expression tree.
- \text{component_types (set or list)} – A set of class types that will be matched during the search.

Yields Each node that is found.

\text{pyomo.core.expr.current.identify_variables}(\text{expr, include_fixed=True})

A generator that yields a sequence of variables in an expression tree.

Parameters

- \text{expr} – The root node of an expression tree.
- \text{include_fixed (bool)} – If True, then this generator will yield variables whose value is fixed. Defaults to True.

Yields Each variable that is found.
Classes

class pyomo.core.expr.symbol_map.SymbolMap(labeler=None)
    A class for tracking assigned labels for modeling components.
    
    Symbol maps are used, for example, when writing problem files for input to an optimizer.

    Warning: A symbol map should never be pickled. This class is typically constructed by solvers and writers, and it may be owned by models.

    Note: We should change the API to not use camelcase.

    byObject
dict – maps (object id) to (string label)

    bySymbol
dict – maps (string label) to (object weakref)

    alias
    dict – maps (string label) to (object weakref)

    default_labeler
    used to compute a string label from an object

9.1.3 Context Managers

class pyomo.core.expr.current.nonlinear_expression
    Context manager for mutable sums.
    
    This context manager is used to compute a sum while treating the summation as a mutable object.

class pyomo.core.expr.current.linear_expression
    Context manager for mutable linear sums.
    
    This context manager is used to compute a linear sum while treating the summation as a mutable object.

class pyomo.core.expr.current.clone_counter
    Context manager for counting cloning events.
    
    This context manager counts the number of times that the clone_expression function is executed.

    count
    A property that returns the clone count value.

9.1.4 Core Classes

The following are the two core classes documented here:

    • NumericValue
    • ExpressionBase

The remaining classes are the public classes for expressions, which developers may need to know about. The methods for these classes are not documented because they are described in the ExpressionBase class.
Sets with Expression Types

The following sets can be used to develop visitor patterns for Pyomo expressions.

```python
pyomo.core.expr.numvalue.native_numeric_types = set([<type 'bool'>, <type 'float'>, <type 'int'>, <type 'long'>])
```

- Python set used to identify numeric constants. This set includes native Python types as well as numeric types from Python packages like numpy, which may be registered by users.

```python
pyomo.core.expr.numvalue.native_types = set([<type 'NoneType'>, <type 'float'>, <type 'str'>, <type 'long'>, <type 'bool'>, <type 'int'>, <type 'unicode'>])
```

- Python set used to identify numeric constants and related native types. This set includes native Python types as well as numeric types from Python packages like numpy.

```python
native_types = :data:`native_numeric_types <pyomo.core.expr.numvalue.native_numeric_types> + { str }
```

```python
pyomo.core.expr.numvalue.nonpyomo_leaf_types = set([<type 'NoneType'>, <type 'float'>, <class 'pyomo.core.expr.numvalue.NonNumericValue'>, <type 'str'>, <type 'long'>, <type 'bool'>, <type 'int'>, <type 'unicode'>])
```

- Python set used to identify numeric constants, boolean values, strings and instances of NonNumericValue, which is commonly used in code that walks Pyomo expression trees.

```python
nonpyomo_leaf_types = native_types }
```

NumericValue and ExpressionBase

```python
class pyomo.core.expr.numvalue.NumericValue
```

This is the base class for numeric values used in Pyomo.

- `__abs__()`
  - Absolute value
  - This method is called when Python processes the statement:
    ```python
    abs(self)
    ```

- `__add__(other)`
  - Binary addition
  - This method is called when Python processes the statement:
    ```python
    self + other
    ```

- `__div__(other)`
  - Binary division
  - This method is called when Python processes the statement:
    ```python
    self / other
    ```

- `__eq__(other)`
  - Equal to operator
  - This method is called when Python processes the statement:
    ```python
    self == other
    ```

- `__float__()`
  - Coerce the value to a floating point
  - Raises TypeError
__ge__(other)
Greater than or equal operator
This method is called when Python processes statements of the form:

\[
\begin{align*}
self & \geq other \\
other & \leq self
\end{align*}
\]

__ge__

__getstate__()
Prepare a picklable state of this instance for pickling.
Nominally, __getstate__() should execute the following:

\[
\begin{align*}
\text{state} & = \text{super(Class, self).__getstate__()} \\
\text{for } i \text{ in Class.__slots__:} \\
& \quad \text{state}[i] = \text{getattr(self, } i) \\
\text{return} & \text{state}
\end{align*}
\]

However, in this case, the (nominal) parent class is ‘object’, and object does not implement __getstate__.
So, we will check to make sure that there is a base __getstate__() to call. You might think that there is
nothing to check, but multiple inheritance could mean that another class got stuck between this class and
“object” in the MRO.

Further, since there are actually no slots defined here, the real question is to either return an empty dict or
the parent’s dict.

__gt__(other)
Greater than operator
This method is called when Python processes statements of the form:

\[
\begin{align*}
self & > other \\
other & < self
\end{align*}
\]

__gt__

__iadd__ (other)
Binary addition
This method is called when Python processes the statement:

\[
\begin{align*}
self & += other
\end{align*}
\]

__iadd__

__idiv__ (other)
Binary division
This method is called when Python processes the statement:

\[
\begin{align*}
self & /= other
\end{align*}
\]

__idiv__

__imul__ (other)
Binary multiplication
This method is called when Python processes the statement:

\[
\begin{align*}
self & *= other
\end{align*}
\]

__imul__

__int__()
Coerce the value to an integer

Raises TypeError
__ipow__(other)

Binary power

This method is called when Python processes the statement:

```
self **= other
```

__isub__(other)

Binary subtraction

This method is called when Python processes the statement:

```
self -= other
```

__itruediv__(other)

Binary division (when __future__.division is in effect)

This method is called when Python processes the statement:

```
self /= other
```

__le__(other)

Less than or equal operator

This method is called when Python processes statements of the form:

```
self <= other
other >= self
```

__lt__(other)

Less than operator

This method is called when Python processes statements of the form:

```
self < other
other > self
```

__mul__(other)

Binary multiplication

This method is called when Python processes the statement:

```
self * other
```

__neg__()

Negation

This method is called when Python processes the statement:

```
- self
```

__pos__()

Positive expression

This method is called when Python processes the statement:

```
+ self
```

__pow__(other)

Binary power
This method is called when Python processes the statement:

```
self ** other
```

__radd__(other)

Binary addition

This method is called when Python processes the statement:

```
other + self
```

__rdiv__(other)

Binary division

This method is called when Python processes the statement:

```
other / self
```

__rmul__(other)

Binary multiplication

This method is called when Python processes the statement:

```
other * self
```

when other is not a `NumericValue` object.

__rpow__(other)

Binary power

This method is called when Python processes the statement:

```
other ** self
```

__rsub__(other)

Binary subtraction

This method is called when Python processes the statement:

```
other - self
```

__rtruediv__(other)

Binary division (when __future__.division is in effect)

This method is called when Python processes the statement:

```
other / self
```

__setstate__(state)

Restore a pickled state into this instance

Our model for setstate is for derived classes to modify the state dictionary as control passes up the inheritance hierarchy (using super() calls). All assignment of state -> object attributes is handled at the last class before ‘object’, which may – or may not (thanks to MRO) – be here.

__sub__(other)

Binary subtraction

This method is called when Python processes the statement:
self - other

__truediv__(other)
Binary division (when __future__.division is in effect)
This method is called when Python processes the statement:
self / other

___compute_polynomial_degree(values)___
Compute the polynomial degree of this expression given the degree values of its children.

Parameters values (list) – A list of values that indicate the degree of the children expression.

Returns None

getname (fully_qualified=False, name_buffer=None)
If this is a component, return the component’s name on the owning block; otherwise return the value converted to a string

is_constant ()
Return True if this numeric value is a constant value

is_expression_type ()
Return True if this numeric value is an expression

is_fixed ()
Return True if this is a non-constant value that has been fixed

is_indexed ()
Return True if this numeric value is an indexed object

is_named_expression_type ()
Return True if this numeric value is a named expression

is_potentially_variable ()
Return True if variables can appear in this expression

is_relational ()
Return True if this numeric value represents a relational expression.

is_variable_type ()
Return False unless this class is a variable object

polynomial_degree ()
Return the polynomial degree of the expression.

Returns None

to_string (verbose=None, labeler=None, smap=None, compute_values=False)
Return a string representation of the expression tree.

Parameters

• verbose (bool) – If True, then the the string representation consists of nested functions. Otherwise, the string representation is an algebraic equation. Defaults to False.

• labeler – An object that generates string labels for variables in the expression tree. Defaults to None.

Returns A string representation for the expression tree.
class pyomo.core.expr.current.ExpressionBase(args)
    Bases: pyomo.core.expr.numvalue.NumericValue

The base class for Pyomo expressions.
This class is used to define nodes in an expression tree.

Parameters
    args (list or tuple) – Children of this node.

__deepcopy__(memo)
    Return a clone of the expression tree.

Note: This method clones the leaves of the tree.

Parameters
    memo (dict) – a dictionary that maps object ids to clone objects generated earlier during the cloning process.

Returns
    A new expression tree.

__bool__()
    Compute the value of the expression and convert it to a boolean.

Returns
    A boolean value.

__call__(exception=True)
    Evaluate the value of the expression tree.

Parameters
    exception (bool) – If False, then an exception raised while evaluating is captured, and the value returned is None. Default is True.

Returns
    The value of the expression or None.

__getstate__()
    Pickle the expression object

Returns
    The pickled state.

__nonzero__()
    Compute the value of the expression and convert it to a boolean.

Returns
    A boolean value.

__str__()
    Returns a string description of the expression.

Note: The value of pyomo.core.expr.expr_common.TO_STRING_VERBOSE is used to configure the execution of this method. If this value is True, then the string representation is a nested function description of the expression. The default is False, which is an algebraic description of the expression.

Returns
    A string.

_apply_operation(result)
    Compute the values of this node given the values of its children.

This method is called by the _EvaluationVisitor class. It must be over-written by expression classes to customize this logic.
Note: This method applies the logical operation of the operator to the arguments. It does not evaluate the arguments in the process, but assumes that they have been previously evaluated. But noted that if this class contains auxilliary data (e.g. like the numeric coefficients in the LinearExpression class, then those values must be evaluated as part of this function call. An uninitialized parameter value encountered during the execution of this method is considered an error.

Parameters values (list) – A list of values that indicate the value of the children expressions.

Returns A floating point value for this expression.

_compute_polynomial_degree (values)
Compute the polynomial degree of this expression given the degree values of its children.

This method is called by the _PolynomialDegreeVisitor class. It can be over-written by expression classes to customize this logic.

Parameters values (list) – A list of values that indicate the degree of the children expression.

Returns A nonnegative integer that is the polynomial degree of the expression, or None. Default is None.

_is_fixed (values)
Compute whether this expression is fixed given the fixed values of its children.

This method is called by the _IsFixedVisitor class. It can be over-written by expression classes to customize this logic.

Parameters values (list) – A list of boolean values that indicate whether the children of this expression are fixed.

Returns A boolean that is True if the fixed values of the children are all True.

_to_string (values, verbose, smap, compute_values)
Construct a string representation for this node, using the string representations of its children.

This method is called by the _ToStringVisitor class. It must must be defined in subclasses.

Parameters

• values (list) – The string representations of the children of this node.

• verbose (bool) – If True, then the string representation consists of nested functions. Otherwise, the string representation is an algebraic equation.

• smap – If specified, this SymbolMap is used to cache labels for variables.

• compute_values (bool) – If True, then parameters and fixed variables are evaluated before the expression string is generated.

Returns A string representation for this node.

arg (i)
Return the i-th child node.

Parameters i (int) – Nonnegative index of the child that is returned.

Returns The i-th child node.

args
A generator that yields the child nodes.
Yields  Each child node in order.

cloned (substitute=None)
   Return a clone of the expression tree.

Note: This method does not clone the leaves of the tree, which are numeric constants and variables. It only
   clones the interior nodes, and expression leaf nodes like _MutableLinearExpression. However,
   named expressions are treated like leaves, and they are not cloned.

Parameters substitute (dict) – a dictionary that maps object ids to clone objects gener-
   ated earlier during the cloning process.

Returns  A new expression tree.

create_node_with_local_data (args, memo)
   Construct a node using given arguments.

This method provides a consistent interface for constructing a node, which is used in tree visitor scripts.
In the simplest case, this simply returns:

   self.__class__(args)

But in general this creates an expression object using local data as well as arguments that represent the
child nodes.

Parameters
   • args (list) – A list of child nodes for the new expression object
   • memo (dict) – A dictionary that maps object ids to clone objects generated earlier during a cloning process. This argument is needed to clone objects that are owned by a model, and it can be safely ignored for most expression classes.

Returns  A new expression object with the same type as the current class.

create_potentially_variable_object ()
   Create a potentially variable version of this object.

This method returns an object that is a potentially variable version of the current object. In the simplest case, this simply sets the value of __class__:

   self.__class__ = self.__class__.__mro__[1]

Note that this method is allowed to modify the current object and return it. But in some cases it may create a new potentially variable object.

Returns  An object that is potentially variable.

gotename (*args, **kwds)
   Return the text name of a function associated with this expression object.

In general, no arguments are passed to this function.

Parameters
   • *arg – a variable length list of arguments
   • **kwds – keyword arguments

Returns  A string name for the function.
**is_constant()**
Return True if this expression is an atomic constant

This method contrasts with the is_fixed() method. This method returns True if the expression is an atomic constant, that is it is composed exclusively of constants and immutable parameters. NumericValue objects returning is_constant() == True may be simplified to their numeric value at any point without warning.

Note: This defaults to False, but gets redefined in sub-classes.

**is_expression_type()**
Return True if this object is an expression.

This method obviously returns True for this class, but it is included in other classes within Pyomo that are not expressions, which allows for a check for expressions without evaluating the class type.

Returns A boolean.

**is_fixed()**
Return True if this expression contains no free variables.

Returns A boolean.

**is_named_expression_type()**
Return True if this object is a named expression.

This method returns False for this class, and it is included in other classes within Pyomo that are not named expressions, which allows for a check for named expressions without evaluating the class type.

Returns A boolean.

**is_potentially_variable()**
Return True if this expression might represent a variable expression.

This method returns True when (a) the expression tree contains one or more variables, or (b) the expression tree contains a named expression. In both cases, the expression cannot be treated as constant since (a) the variables may not be fixed, or (b) the named expressions may be changed at a later time to include non-fixed variables.

Returns A boolean. Defaults to True for expressions.

**nargs()**
Returns the number of child nodes.

By default, Pyomo expressions represent binary operations with two arguments.

Note: This function does not simply compute the length of _args_ because some expression classes use a subset of the _args_ array. Thus, it is imperative that developers use this method!

Returns A nonnegative integer that is the number of child nodes.

**polynomial_degree()**
Return the polynomial degree of the expression.

Returns A non-negative integer that is the polynomial degree if the expression is polynomial, or None otherwise.

**size()**
Return the number of nodes in the expression tree.

Returns A nonnegative integer that is the number of interior and leaf nodes in the expression tree.
to_string (verbose=None, labeler=None, smap=None, compute_values=False)
   Return a string representation of the expression tree.

Parameters
   • verbose (bool) – If True, then the the string representation consists of nested func-
     tions. Otherwise, the string representation is an algebraic equation. Defaults to False.
   • labeler – An object that generates string labels for variables in the expression tree. 
     Defaults to None.
   • smap – If specified, this SymbolMap is used to cache labels for variables.
   • compute_values (bool) – If True, then parameters and fixed variables are evaluated 
     before the expression string is generated. Default is False.

Returns  A string representation for the expression tree.

Other Public Classes

class pyomo.core.expr.current.NegationExpression (args)
   Bases: pyomo.core.expr.expr_pyomo5.ExpressionBase
   Negation expressions:
   - x

   PRECEDENCE = 4
   _apply_operation (result)
   _compute_polynomial_degree (result)
   _precedence ()
   _to_string (values, verbose, smap, compute_values)
   getname (*args, **kwds)
   nargs ()

class pyomo.core.expr.current.ExternalFunctionExpression (args, fcn=None)
   Bases: pyomo.core.expr.expr_pyomo5.ExpressionBase
   External function expressions

Example:

model = ConcreteModel()
model.a = Var()
model.f = ExternalFunction(library='foo.so', function='bar')
expr = model.f(model.a)

Parameters
   • args (tuple) – children of this node
   • fcn – a class that defines this external function

   _apply_operation (result)
   _compute_polynomial_degree (result)
   _fcn
class pyomo.core.expr.current.ProductExpression(args)
    Bases: pyomo.core.expr.expr_pyomo5.ExpressionBase

    Product expressions:
    x*y

    PRECEDENCE = 4
    _apply_operation(result)
    _compute_polynomial_degree(result)
    _precedence()
    _to_string(values, verbose, smap, compute_values)
    getname(*args, **kwds)

class pyomo.core.expr.current.ReciprocalExpression(args)
    Bases: pyomo.core.expr.expr_pyomo5.ExpressionBase

    Reciprocal expressions:
    1/x

    PRECEDENCE = 3.5
    _apply_operation(result)
    _compute_polynomial_degree(result)
    _precedence()
    _to_string(values, verbose, smap, compute_values)
    getname(*args, **kwds)

class pyomo.core.expr.current.InequalityExpression(args, strict)
    Bases: pyomo.core.expr.expr_pyomo5._LinearOperatorExpression

    Inequality expressions, which define less-than or less-than-or-equal relations:
    x < y
    x <= y

Parameters

- **args** (tuple) – child nodes
- **strict** (bool) – a flag that indicates whether the inequality is strict

PRECEDENCE = 9
    _apply_operation(result)
class pyomo.core.expr.current.EqualityExpression(args)
Bases: pyomo.core.expr.expr_pyomo5._LinearOperatorExpression

Equality expression:

x == y

PRECEDENCE = 9

_apply_operation(result)

_precedence()

_to_string(values, verbose, smap, compute_values)

is_constant()

is_potentially_variable()

is_relational()

nargs()

class pyomo.core.expr.current.SumExpression(args)
Bases: pyomo.core.expr.expr_pyomo5.SumExpressionBase

Sum expression:

x + y

Parameters args(list) – Children nodes

PRECEDENCE = 6

_apply_operation(result)

_nargs

_precedence()

_shared_args

_to_string(values, verbose, smap, compute_values)

add(new_arg)

create_node_with_local_data(args, memo)

is_constant()

is_potentially_variable()
nargs()

class pyomo.core.expr.current.GetItemExpression(args, base=None):
    Bases: pyomo.core.expr.expr_pyomo5.ExpressionBase
    Expression to call __getitem__() on the base object.
    PRECEDENCE = 1
    _apply_operation(result)
    _base
    _compute_polynomial_degree(result)
    _is_fixed(values)
    _precedence()
    _to_string(values, verbose, smap, compute_values)
create_node_with_local_data(args, memo)
getname(*args, **kwds)
is_fixed()
is_potentially_variable()
nargs()
resolve_template()

class pyomo.core.expr.current.Expr_ifExpression(IF_=None, THEN_=None, ELSE_=None):
    Bases: pyomo.core.expr.expr_pyomo5.ExpressionBase
    A logical if-then-else expression:
    Expr_if(IF_=x, THEN_=y, ELSE_=z)

    Parameters
    • IF (expression) – A relational expression
    • THEN (expression) – An expression that is used if IF_ is true.
    • ELSE (expression) – An expression that is used if IF_ is false.

    _apply_operation(result)
    _compute_polynomial_degree(result)
    _else
    _if
    _is_fixed(args)
    _then
    _to_string(values, verbose, smap, compute_values)
getname(*args, **kwds)
is_constant()
is_potentially_variable()
nargs()

class pyomo.core.expr.current.UnaryFunctionExpression(args, name=None, fcn=None)

Bases: pyomo.core.expr.expr_pyomo5.ExpressionBase

An expression object used to define intrinsic functions (e.g. sin, cos, tan).

Parameters

• args (tuple) – Children nodes
• name (string) – The function name
• fcn – The function that is used to evaluate this expression

__apply_operation__(result)

__compute_polynomial_degree__(result)

__fcn

__name

__to_string__(values, verbose, smap, compute_values)

create_node_with_local_data(args, memo)

getname(*args, **kwds)

nargs()

class pyomo.core.expr.current.AbsExpression(arg)

Bases: pyomo.core.expr.expr_pyomo5.UnaryFunctionExpression

An expression object for the abs() function.

Parameters

• args (tuple) – Children nodes

create_node_with_local_data(args, memo)

9.1.5 Visitor Classes

class pyomo.core.expr.current.SimpleExpressionVisitor

Note: This class is a customization of the PyUtilib SimpleVisitor class that is tailored to efficiently walk Pyomo expression trees. However, this class is not a subclass of the PyUtilib SimpleVisitor class because all key methods are reimplemented.

finalize()

Return the “final value” of the search.

The default implementation returns None, because the traditional visitor pattern does not return a value.

Returns The final value after the search. Default is None.

visit(node)

Visit a node in an expression tree and perform some operation on it.

This method should be over-written by a user that is creating a sub-class.

Parameters node – a node in an expression tree
Returns nothing

xbfs(node)
Breadth-first search of an expression tree, except that leaf nodes are immediately visited.

Note: This method has the same functionality as the PyUtilib SimpleVisitor.xbfs method. The difference is that this method is tailored to efficiently walk Pyomo expression trees.

Parameters node – The root node of the expression tree that is searched.

Returns The return value is determined by the finalize() function, which may be defined by the user. Defaults to None.

xbfs_yield_leaves(node)
Breadth-first search of an expression tree, except that leaf nodes are immediately visited.

Note: This method has the same functionality as the PyUtilib SimpleVisitor.xbfs_yield_leaves method. The difference is that this method is tailored to efficiently walk Pyomo expression trees.

Parameters node – The root node of the expression tree that is searched.

Returns The return value is determined by the finalize() function, which may be defined by the user. Defaults to None.

class pyomo.core.expr.current.ExpressionValueVisitor

Note: This class is a customization of the PyUtilib ValueVisitor class that is tailored to efficiently walk Pyomo expression trees. However, this class is not a subclass of the PyUtilib ValueVisitor class because all key methods are reimplemented.

dfs_postorder_stack(node)
Perform a depth-first search in postorder using a stack implementation.

Note: This method has the same functionality as the PyUtilib ValueVisitor.dfs_postorder_stack method. The difference is that this method is tailored to efficiently walk Pyomo expression trees.

Parameters node – The root node of the expression tree that is searched.

Returns The return value is determined by the finalize() function, which may be defined by the user.

finalize(ans)
This method defines the return value for the search methods in this class.

The default implementation returns the value of the initial node (aka the root node), because this visitor pattern computes and returns value for each node to enable the computation of this value.

Parameters ans – The final value computed by the search method.
Returns
The final value after the search. Defaults to simply returning \texttt{ans}.

\texttt{visit (node, values)}
Visit a node in a tree and compute its value using the values of its children.
This method should be over-written by a user that is creating a sub-class.

\textbf{Parameters}
\begin{itemize}
  \item \texttt{node} – a node in a tree
  \item \texttt{values} – a list of values of this node’s children
\end{itemize}

\textbf{Returns}
The \texttt{value} for this node, which is computed using \texttt{values}

\texttt{visiting_potential_leaf (node)}
Visit a node and return its value if it is a leaf.

\textbf{Note:}
This method needs to be over-written for a specific visitor application.

\textbf{Parameters} \texttt{node} – a node in a tree

\textbf{Returns} \texttt{(flag, value)}. If \texttt{flag} is \texttt{False}, then the node is not a leaf and \texttt{value} is \texttt{None}. Otherwise, \texttt{value} is the computed value for this node.

\textbf{Return type} A tuple

\texttt{class pyomo.core.expr.current.ExpressionReplacementVisitor (memo=None)}

\textbf{Note:}
This class is a customization of the PyUtilib \texttt{ValueVisitor} class that is tailored to support replacement of sub-trees in a Pyomo expression tree. However, this class is not a subclass of the PyUtilib \texttt{ValueVisitor} class because all key methods are reimplemented.

\texttt{construct_node (node, values)}
Call the expression create_node_with_local_data() method.

\texttt{dfs_postorder_stack (node)}
Perform a depth-first search in postorder using a stack implementation.
This method replaces subtrees. This method detects if the \texttt{visit()} method returns a different object. If so, then the node has been replaced and search process is adapted to replace all subsequent parent nodes in the tree.

\textbf{Note:} This method has the same functionality as the PyUtilib \texttt{ValueVisitor}. \texttt{dfs_postorder_stack} method that is tailored to support the replacement of sub-trees in a Pyomo expression tree.

\textbf{Parameters} \texttt{node} – The root node of the expression tree that is searched.

\textbf{Returns} The return value is determined by the \texttt{finalize()} function, which may be defined by the user.

\texttt{finalize (ans)}
This method defines the return value for the search methods in this class.
The default implementation returns the value of the initial node (aka the root node), because this visitor pattern computes and returns value for each node to enable the computation of this value.

**Parameters**
- **ans** – The final value computed by the search method.

**Returns**
The final value after the search. Defaults to simply returning `ans`.

**visit** *(node, values)*
Visit and clone nodes that have been expanded.

**Note:** This method normally does not need to be re-defined by a user.

**Parameters**
- **node** – The node that will be cloned.
- **values** *(list)* – The list of child nodes that have been cloned. These values are used to define the cloned node.

**Returns**
The cloned node. Default is to simply return the node.

**visiting_potential_leaf** *(node)*
Visit a node and return a cloned node if it is a leaf.

**Note:** This method needs to be over-written for a specific visitor application.

**Parameters**
- **node** – a node in a tree

**Returns** *(flag, value)*
If flag is False, then the node is not a leaf and value is `None`. Otherwise, value is a cloned node.

**Return type**
A tuple

### 9.2 AML Library Reference

Modeling Components:

- **`pyomo.core.base.sets.Set(*args, **kwds)`**
  A set object that is used to index other Pyomo objects.

- **`pyomo.core.base.var.Var(*args, **kwds)`**
  A numeric variable, which may be defined over an index.

- **`pyomo.core.base.constraint.Constraint(*args, **kwds)`**
  This modeling component defines a constraint expression using a rule function.

- **`pyomo.core.base.objective.Objective(*args, **kwds)`**
  This modeling component defines an objective expression.

- **`pyomo.core.base.block.Block(*args, **kwds)`**
  Blocks are indexed components that contain other components (including blocks).

### 9.2.1 Member Documentation

**class** `pyomo.core.base.sets.Set(*args, **kwds)`

Bases: `pyomo.core.base.indexed_component.IndexedComponent`

A set object that is used to index other Pyomo objects.
This class has a similar look-and-feel as a Python set class. However, the set operations defined in this class return another abstract Set object. This class contains a concrete set, which can be initialized by the load() method.

**Constructor Arguments:**
- name: The name of the set
- doc: A text string describing this component within A set that defines the type of values that can be contained in this set
- domain: A set that defines the type of values that can be contained in this set
- initialize: A dictionary or rule for setting up this set with existing model data
- validate: A rule for validating membership in this set. This has the functional form:
  \[ f: \text{data} \rightarrow \text{bool} \]
  and returns true if the data belongs in the set
- dimen: Specify the set’s arity, or None if no arity is enforced. virtual: If true, then this is a virtual set that does not store data using the class dictionary
- bounds: A 2-tuple that specifies the range of possible set values. ordered: Specifies whether the set is ordered. Possible values are:
  - False: Unordered
  - True: Ordered by insertion order
  - InsertionOrder: Ordered by insertion order
  - SortedOrder: Ordered by sort order
  - <function>: Ordered with this comparison function
- filter: A function that is used to filter set entries.

**Public class attributes:**
- concrete: If True, then this set contains elements.
- dimen: The dimension of the data in this set.
- doc: A text string describing this component.
- domain: A set that defines the type of values that can be contained in this set
- filter: A function that is used to filter set entries.
- initialize: A dictionary or rule for setting up this set with existing model data
- ordered: Specifies whether the set is ordered.
- validate: A rule for validating membership in this set. virtual: If True, then this set does not store data using the class dictionary

**API:**
- active
  Return the active attribute
- clear()
  Clear the data in this component
- clear_suffix_value(suffix_or_name, expand=True)
  Clear the suffix value for this component data
- construct(data=None)
  API definition for constructing components
- dim()
  Return the dimension of the index
- get_suffix_value(suffix_or_name, default=None)
  Get the suffix value for this component data
getname *(fully_qualified=False, name_buffer=None)*

Returns the component name associated with this object.

    Parameters
    
    • Generate full name from nested block names *(fully_qualified)*
    • Can be used to optimize iterative name *(name_buffer)*

    (using a dictionary)

id_index_map ()

Return an dictionary id->index for all ComponentData instances.

index_set ()

Return the index set

is_constructed ()

Return True if this class has been constructed

is_indexed ()

Return true if this component is indexed

items ()

Return a list (index, data) tuples from the dictionary

iteritems ()

Return an iterator of (index, data) tuples from the dictionary

iterkeys ()

Return an iterator of the keys in the dictionary

itervalues ()

Return an iterator of the component data objects in the dictionary

keys ()

Return a list of keys in the dictionary

local_name

Get the component name only within the context of the immediate parent container.

model ()

Returns the model associated with this object.

name

Get the fully qualified component name.

parent_block ()

Returns the parent of this object.

parent_component ()

Returns the component associated with this object.

pprint *(ostream=None, verbose=False, prefix= ”)*

Print component information

reconstruct *(data=None)*

Re-construct model expressions

root_block ()

Return self.model()

set_suffix_value *(suffix_or_name, value, expand=True)*

Set the suffix value for this component data
set_value(value)
Set the value of a scalar component.

to_dense_data()
TODO
to_string(verbos=None, labeler=None, smap=None, compute_values=False)
Return the component name
type()
Return the class type for this component
valid_model_component()
Return True if this can be used as a model component.
values()
Return a list of the component data objects in the dictionary

class pyomo.core.base.var.Var(*args, **kwd)
Bases: pyomo.core.base.indexed_component.IndexedComponent
A numeric variable, which may be defined over an index.

Parameters

- domain (Set or function, optional) – A Set that defines valid values for the variable (e.g., Reals, NonNegativeReals, Binary), or a rule that returns Sets. Defaults to Reals.
- bounds (tuple or function, optional) – A tuple of (lower, upper) bounds for the variable, or a rule that returns tuples. Defaults to (None, None).
- initialize (float or function, optional) – The initial value for the variable, or a rule that returns initial values.
- rule (function, optional) – An alias for rule
- dense (bool, optional) – Instantiate all elements from index_set() when constructing the Var (True) or just the variables returned by initialize/rule (False). Defaults to True.

active
Return the active attribute
add(index)
Add a variable with a particular index.
clear()
Clear the data in this component
clear_suffix_value(suffix_or_name, expand=True)
Clear the suffix value for this component data
construct(data=None)
Construct this component.
dim()
Return the dimension of the index
extract_values(include_fixed_values=True)
Return a dictionary of index-value pairs.
flag_as_stale()
Set the ‘stale’ attribute of every variable data object to True.
get_suffix_value(suffix_or_name, default=None)
Get the suffix value for this component data

get_values(include_fixed_values=True)
Return a dictionary of index-value pairs.

getname(fully_qualified=False, name_buffer=None)
Returns the component name associated with this object.

Parameters

• Generate full name from nested block names(fully_qualified) –
• Can be used to optimize iterative name (name_buffer) – generation
  (using a dictionary)

id_index_map()
Return an dictionary id->index for all ComponentData instances.

index_set()
Return the index set

is_constructed()
Return True if this class has been constructed

is_expression_type()
Returns False because this is not an expression

is_indexed()
Return true if this component is indexed

items()
Return a list (index,data) tuples from the dictionary

iteritems()
Return an iterator of (index,data) tuples from the dictionary

iterkeys()
Return an iterator of the keys in the dictionary

itervalues()
Return an iterator of the component data objects in the dictionary

keys()
Return a list of keys in the dictionary

local_name
Get the component name only within the context of the immediate parent container.

model()
Returns the model associated with this object.

name
Get the fully qualified component name.

parent_block()
Returns the parent of this object.

parent_component()
Returns the component associated with this object.

pprint(ostream=None, verbose=False, prefix="")
Print component information
```
reconstruct (data=None)
    Re-construct model expressions

root_block ()
    Return self.model()

set_suffix_value (suffix_or_name, value, expand=True)
    Set the suffix value for this component data

set_value (value)
    Set the value of a scalar component.

set_values (new_values, valid=False)
    Set the values of a dictionary.
    The default behavior is to validate the values in the dictionary.

to_dense_data ()
    TODO

to_string (verbose=None, labeler=None, smap=None, compute_values=False)
    Return the component name

type ()
    Return the class type for this component

valid_model_component ()
    Return True if this can be used as a model component.

values ()
    Return a list of the component data objects in the dictionary

class pyomo.core.base.constraint.Constraint (*args, **kwargs)
    Bases: pyomo.core.base.indexed_component.ActiveIndexedComponent

This modeling component defines a constraint expression using a rule function.

Constructor arguments: expr A Pyomo expression for this constraint rule A function that is used to construct constraint expressions
doc A text string describing this component name A name for this component

Public class attributes: doc A text string describing this component name A name for this component active A boolean that is true if this component will be used to construct a model instance rule The rule used to initialize the constraint(s)

Private class attributes:
    _constructed A boolean that is true if this component has been constructed
    _data A dictionary from the index set to component data objects
    _index The set of valid indices _implicit_subsets A tuple of set objects that represents the index set _model A weakref to the model that owns this component _parent A weakref to the parent block that owns this component _type The class type for the derived subclass

activate ()
    Set the active attribute to True

active
    Return the active attribute
```
clear()
    Clear the data in this component

clear_suffix_value(suffix_or_name, expand=True)
    Clear the suffix value for this component data

construct(data=None)
    Construct the expression(s) for this constraint.

disable()
    Set the active attribute to False

dim()
    Return the dimension of the index

display(prefix='', ostream=None)
    Print component state information
    This duplicates logic in Component.pprint()

get_suffix_value(suffix_or_name, default=None)
    Get the suffix value for this component data

getname(fully_qualified=False, name_buffer=None)
    Returns the component name associated with this object.

    Parameters
    - Generate full name from nested block names(fully_qualified) -
      - Can be used to optimize iterative name(name_buffer) -
        generation (using a dictionary)

id_index_map()
    Return an dictionary id->index for all ComponentData instances.

index_set()
    Return the index set

is_constructed()
    Return True if this class has been constructed

is_indexed()
    Return true if this component is indexed

items()
    Return a list (index, data) tuples from the dictionary

iteritems()
    Return an iterator of (index, data) tuples from the dictionary

iterkeys()
    Return an iterator of the keys in the dictionary

itervalues()
    Return an iterator of the component data objects in the dictionary

keys()
    Return a list of keys in the dictionary

local_name
    Get the component name only within the context of the immediate parent container.

model()
    Returns the model associated with this object.
name
   Get the fully qualified component name.

parent_block()
   Returns the parent of this object.

parent_component()
   Returns the component associated with this object.

pprint(ostream=None, verbose=False, prefix="")
   Print component information

reconstruct(data=None)
   Re-construct model expressions

root_block()
   Return self.model()

set_suffix_value(suffix_or_name, value, expand=True)
   Set the suffix value for this component data

set_value(value)
   Set the value of a scalar component.

to_dense_data()
   TODO

to_string(verbose=None, labeler=None, smap=None, compute_values=False)
   Return the component name

type()
   Return the class type for this component

valid_model_component()
   Return True if this can be used as a model component.

values()
   Return a list of the component data objects in the dictionary

class pyomo.core.base.objective.Objective(*args, **kwargs)
   Bases: pyomo.core.base.indexed_component.ActiveIndexedComponent

This modeling component defines an objective expression.

Note that this is a subclass of NumericValue to allow objectives to be used as part of expressions.

Constructor arguments: expr A Pyomo expression for this objective rule A function that is used to construct objective expressions
   sense Indicate whether minimizing (the default) or maximizing doc A text string describing this component name A name for this component

Public class attributes: doc A text string describing this component name A name for this component active A boolean that is true if this component will be used to construct a model instance
   rule The rule used to initialize the objective(s) sense The objective sense

Private class attributes:
   _constructed A boolean that is true if this component has been constructed
   _data A dictionary from the index set to component data objects
activate()
Set the active attribute to True

active
Return the active attribute

clear()
Clear the data in this component

clear_suffix_value(suffix_or_name, expand=True)
Clear the suffix value for this component data

construct(data=None)
Construct the expression(s) for this objective.

deactivate()
Set the active attribute to False

dim()
Return the dimension of the index

display(prefix=", ostream=None)
Provide a verbose display of this object

get_suffix_value(suffix_or_name, default=None)
Get the suffix value for this component data

getname(fully_qualified=False, name_buffer=None)
Returns the component name associated with this object.

Parameters

- Generate full name from nested block names(fully_qualified) -
- Can be used to optimize iterative name (name_buffer) - generation (using a dictionary)

id_index_map()
Return an dictionary id->index for all ComponentData instances.

index_set()
Return the index set

is_constructed()
Return True if this class has been constructed

is_indexed()
Return true if this component is indexed

items()
Return a list (index,data) tuples from the dictionary

iteritems()
Return an iterator of (index,data) tuples from the dictionary

iterkeys()
Return an iterator of the keys in the dictionary

itervalues()
Return an iterator of the component data objects in the dictionary
keys()
    Return a list of keys in the dictionary

local_name
    Get the component name only within the context of the immediate parent container.

model()
    Returns the model associated with this object.

name
    Get the fully qualified component name.

parent_block()
    Returns the parent of this object.

parent_component()
    Returns the component associated with this object.

pprint(ostream=None, verbose=False, prefix="")
    Print component information

reconstruct(data=None)
    Re-construct model expressions

root_block()
    Return self.model()

set_suffix_value(suffix_or_name, value, expand=True)
    Set the suffix value for this component data

set_value(value)
    Set the value of a scalar component.

to_dense_data()
    TODO

to_string(verbos=None, labeler=None, smap=None, compute_values=False)
    Return the component name

type()
    Return the class type for this component

valid_model_component()
    Return True if this can be used as a model component.

values()
    Return a list of the component data objects in the dictionary

class pyomo.core.base.block.Block(*args, **kwargs)
    Bases: pyomo.core.base.indexed_component.ActiveIndexedComponent

    Blocks are indexed components that contain other components (including blocks). Blocks have a global attribute
    that defines whether construction is deferred. This applies to all components that they contain except blocks.
    Blocks contained by other blocks use their local attribute to determine whether construction is deferred.

activate()
    Set the active attribute to True

active
    Return the active attribute

clear()
    Clear the data in this component
clear_suffix_value(suffix_or_name, expand=True)
   Clear the suffix value for this component data

construct(data=None)
   Initialize the block

deactivate()
   Set the active attribute to False

dim()
   Return the dimension of the index

display(filename=None, ostream=None, prefix="")
   Display values in the block

find_component(label_or_component)
   Return a block component given a name.

get_suffix_value(suffix_or_name, default=None)
   Get the suffix value for this component data

getname(fully_qualified=False, name_buffer=None)
   Returns the component name associated with this object.
   Parameters
   
   - Generate full name from nested block names(fully_qualified) –
   - Can be used to optimize iterative name(name_buffer) – generation
      (using a dictionary)

id_index_map()
   Return an dictionary id->index for all ComponentData instances.

index_set()
   Return the index set

is_constructed()
   Return True if this class has been constructed

is_indexed()
   Return true if this component is indexed

items()
   Return a list (index,data) tuples from the dictionary

iteritems()
   Return an iterator of (index,data) tuples from the dictionary

iterkeys()
   Return an iterator of the keys in the dictionary

itervalues()
   Return an iterator of the component data objects in the dictionary

keys()
   Return a list of keys in the dictionary

local_name
   Get the component name only within the context of the immediate parent container.

model()
   Returns the model associated with this object.
name
Get the fully qualified component name.

parent_block()
Returns the parent of this object.

parent_component()
Returns the component associated with this object.

pprint (filename=None, ostream=None, verbose=False, prefix="")
Print block information

reconstruct (data=None)
Re-construct model expressions

root_block()
Return self.model()

set_suffix_value (suffix_or_name, value, expand=True)
Set the suffix value for this component data

set_value (value)
Set the value of a scalar component.

to_dense_data()
TODO

to_string (verbose=None, labeler=None, smap=None, compute_values=False)
Return the component name

type()
Return the class type for this component

valid_model_component()
Return True if this can be used as a model component.

values()
Return a list of the component data objects in the dictionary

9.3 Kernel Library Reference

Low-level Interfaces:

9.3.1 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 objects 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 isistance 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 isistance 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 isistance on classes that use abc.ABCMeta as a metaclass.

g�名 (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 ancestor 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 objects 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 objects 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 functionality 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 functionality 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 whether 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
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:

### 9.3.2 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:

```python
_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.

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

cchild_key(*args, **kwd)
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.

ggetname(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 ancestor 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 descendent of this block will be deepcopied, otherwise a reference to the original component is retained.

**collect_ctypes**  
(activity=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=*, *name_buffer=*, *convert=*

Dynamically generates a name for this object.

**Parameters**

- **fully_qualified** (bool) – Generate a full name by iterating through all ancestor 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=*, *comparison_tolerance_for_fixed_vars=*

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**

```python
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.

**preorder_traversal**

```python
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`  
`preorder_visit`  
\[\textit{visit, ctype=人都有object object>, active=None, include_all_parents=True, include_key=False, root_key=None}\]  
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 `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**  
  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**  
  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**  
  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**  
`write`  
\[\textit{filename, format=None, _solver_capability=None, _called_by_solver=False, **kwds}\]  
Write the model to a file, with a given format.

**Parameters**

- **filename**  
  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**  
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**  
`activate`  
\[\textit{from_parent_=False}\]  
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 ancestor 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

```python
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

```python
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 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

```python
setdefault(*k*, d) -> D.get(k,d), also set D[k]=d if k not in D
```

```python
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
```

```python
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

call (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

call_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

call_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

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

defall (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=str)*

Dynamically generates a name for this object.

**Parameters**

• **fully_qualified** *(bool)* – Generate a full name by iterating through all ancestor 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 \( \left[ \text{index} \right] \) → item – remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

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

**Parameters**

- **active** \((\text{True/None})\) – 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** \((\text{bool})\) – 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** \((\text{active}=\text{None}, \text{return_key}=\text{False}, \text{root_key}=\text{None})\)
Generates a preorder traversal of the storage tree.

**Parameters**

- **active** \((\text{True/None})\) – 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** \((\text{bool})\) – 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** \((\text{visit}, \text{active}=\text{None}, \text{include_key}=\text{False}, \text{root_key}=\text{None})\)
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 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** \((\text{True/None})\) – 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** \((\text{bool})\) – 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 \( (\text{value}) \)
\[ \text{S.remove(value)} \] – remove first occurrence of \text{value}. Raise ValueError if the value is not present.

reverse()
\[ \text{S.reverse()} \] – reverse \text{IN PLACE}

root_block
The root storage block above this object

class \text{pyomo.core.kernel.component_block.block_tuple(*)args, **kwds)}
\begin{center}
\text{Bases: pyomo.core.kernel.component_tuple.ComponentTuple, pyomo.core.kernel.}
\text{component_interface._ActiveComponentContainerMixin}
\end{center}
A tuple-style container for blocks.

activate \((_\text{from_parent_}=False)\)
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 \text{True}.

active
The active status of this container.

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

\text{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.

\text{Raises ValueError} – if the argument is not a child of this container

children \((\text{return_key}=False)\)
Iterate over the children of this container.

\text{Parameters return_key} (\text{bool}) – Set to \text{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.

\text{Returns} iterator of objects or (key,object) tuples

components \((active=\text{None}, return_key=False)\)
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).

\text{Parameters}
\begin{itemize}
\item \text{active} (\text{True/None}) – Set to \text{True} to indicate that only active objects should be in-
cluded. The default value of \text{None} indicates that all components (including those that
have been deactivated) should be included. \text{Note: This flag is ignored for any objects that
do not have an active flag.}
\item \text{return_key} (\text{bool}) – Set to \text{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.
\end{itemize}

\text{Returns} iterator of objects or (key,object) tuples

count \((value) \rightarrow \text{integer} \) return number of occurrences of \text{value}

deactivate \((_\text{from_parent_}=False)\)
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 ancestors 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 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

color()
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 ancestor 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 `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

### 9.3.3 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

9.3. Kernel Library Reference
__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

cchild_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

cchildren (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

ccomponents (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 in-
        cluded. 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='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='str')*

Dynamically generates a name for this object.

**Parameters**

- **fully_qualified** *(bool)* – Generate a full name by iterating through all ancestor 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)*.
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 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

### 9.3.4 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__**

**__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__**

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

cchild_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 ancestor 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 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
9.3.5 Dict-like Object Storage

class pyomo.core.kernel.component_dict.ComponentDict(*args, **kwds)

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

__getattr__
    x.__getattr__('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__
    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 ancestor 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 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** *(d)* → list of D’s values

Modeling Objects:

### 9.3.6 Variables

**Summary**
Member Documentation

class pyomo.core.kernel.component_variable.variable

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

```python
>>> import pyomo.kernel as pmo
>>> # 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.
```

copy.deepcopy(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`

### 9.3.7 Constraint

#### Summary

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`pyomo.core.kernel.component_constraint.constraint([])</td>
<td>A general algebraic constraint</td>
</tr>
<tr>
<td>`pyomo.core.kernel.component_constraint.linear_constraint([])</td>
<td>A linear constraint</td>
</tr>
<tr>
<td><code>pyomo.core.kernel.component_constraint.constraint_tuple(...)</code></td>
<td>A tuple-style container for constraints.</td>
</tr>
<tr>
<td><code>pyomo.core.kernel.component_constraint.constraint_list(...)</code></td>
<td>A list-style container for constraints.</td>
</tr>
<tr>
<td><code>pyomo.core.kernel.component_constraint.constraint_dict(...)</code></td>
<td>A dict-style container for constraints.</td>
</tr>
<tr>
<td><code>pyomo.core.kernel.component_matrix_constraint.matrix_constraint(A)</code></td>
<td>A container for constraints of the form $lb \leq Ax \leq ub$.</td>
</tr>
</tbody>
</table>

#### 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

```python
>>> import pyomo.kernel as pmo
>>> # 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

```python
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._MutableLinearConstraintBase`, `pyomo.core.kernel.component_constraint._MutableEqualityConstraintBase`
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**

```python
>>> import pyomo.kernel as pmo

>>> # Decision variables used to define constraints
>>> x = pmo.variable()
>>> y = pmo.variable()

>>> # An upper bound constraint
>>> c = pmo.linear_constraint(variables=[x, y], coefficients=[1, 2], ub=1)

>>> # (equivalent form)
>>> c = pmo.linear_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

**canonical_form** *(compute_values=True)*

Build a canonical representation of the body of this constraints

**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.

```python
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.

```python
class pyomo.core.kernel.component_constraint.constraint_dict(*args, **kwds)
component_interface._ActiveComponentContainerMixin
```

A dict-style container for constraints.

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

A container for constraints of the form lb <= Ax <= ub.

**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 defines the lower bound of the constraints
- **ub** – A scalar or array with the same number of rows as A that defines the upper bound of the constraints
- **rhs** – A scalar or array with the same number of rows as A that defines the right-hand side of the constraints (implies equality constraints)
- **x** – 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)

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

### 9.3.8 Parameters

**Summary**

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pyomo.core.kernel.component_parameter.parameter</code></td>
<td>A placeholder for a mutable, numeric value.</td>
</tr>
<tr>
<td><code>pyomo.core.kernel.component_parameter.parameter_tuple</code></td>
<td>A tuple-style container for parameters.</td>
</tr>
<tr>
<td><code>pyomo.core.kernel.component_parameter.parameter_list</code></td>
<td>A list-style container for parameters.</td>
</tr>
<tr>
<td><code>pyomo.core.kernel.component_parameter.parameter_dict</code></td>
<td>A dict-style container for parameters.</td>
</tr>
</tbody>
</table>

**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 parameter

**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.

### 9.3.9 Objectives

**Summary**

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pyomo.core.kernel.component_objective.objective</code></td>
<td>An optimization objective.</td>
</tr>
</tbody>
</table>

Continued on next page
### 9.3.10 Expressions

#### Summary

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pyomo.core.kernel.component_expression.expression</strong></td>
<td>A named, mutable expression.</td>
</tr>
<tr>
<td><strong>pyomo.core.kernel.component_expression.expression_tuple</strong></td>
<td>A tuple-style container for expressions.</td>
</tr>
<tr>
<td><strong>pyomo.core.kernel.component_expression.expression_list</strong></td>
<td>A list-style container for expressions.</td>
</tr>
<tr>
<td><strong>pyomo.core.kernel.component_expression.expression_dict</strong></td>
<td>A dict-style container for expressions.</td>
</tr>
</tbody>
</table>

#### 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`, `pyomo.core.kernel.component_interface._ActiveComponentContainerMixin`

A tuple-style container for expressions.

**Class** `pyomo.core.kernel.component_expression.expression_list` (%*args, **kwds*)

**Bases:** `pyomo.core.kernel.component_list.ComponentList`, `pyomo.core.kernel.component_interface._ActiveComponentContainerMixin`

A list-style container for expressions.

**Class** `pyomo.core.kernel.component_expression.expression_dict` (%*args, **kwds*)

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

A dict-style container for expressions.
A tuple-style container for expressions.

```python
class pyomo.core.kernel.component_expression.expression_list(*args, **kwds):
    Bases: pyomo.core.kernel.component_list.ComponentList
    A list-style container for expressions.
```

```python
class pyomo.core.kernel.component_expression.expression_dict(*args, **kwds):
    Bases: pyomo.core.kernel.component_dict.ComponentDict
    A dict-style container for expressions.
```

## 9.3.11 Special Ordered Sets

### Summary

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pyomo.core.kernel.component_sos.sos(variables)</code></td>
<td>A Special Ordered Set of type n.</td>
</tr>
<tr>
<td><code>pyomo.core.kernel.component_sos.sos1(variables)</code></td>
<td>A Special Ordered Set of type 1.</td>
</tr>
<tr>
<td><code>pyomo.core.kernel.component_sos.sos2(variables)</code></td>
<td>A Special Ordered Set of type 2.</td>
</tr>
<tr>
<td><code>pyomo.core.kernel.component_sos.sos_tuple(...)</code></td>
<td>A tuple-style container for Special Ordered Sets.</td>
</tr>
<tr>
<td><code>pyomo.core.kernel.component_sos.sos_list(...)</code></td>
<td>A list-style container for Special Ordered Sets.</td>
</tr>
<tr>
<td><code>pyomo.core.kernel.component_sos.sos_dict(...)</code></td>
<td>A dict-style container for Special Ordered Sets.</td>
</tr>
</tbody>
</table>

### Member Documentation

```python
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.
```

```python
pyomo.core.kernel.component_sos.sos1(variables, weights=None)
A Special Ordered Set of type 1.
This is an alias for `sos(..., level=1)`.
```

```python
pyomo.core.kernel.component_sos.sos2(variables, weights=None)
A Special Ordered Set of type 2.
This is an alias for `sos(..., level=2)`.
```

```python
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.
```

```python
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)

component_interface._ActiveComponentContainerMixin

A dict-style container for Special Ordered Sets.

### 9.3.12 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
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

```python
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.

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

### 9.3.13 Piecewise Function Library

**Modules**

**Single-variate Piecewise Functions**

**Summary**

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pyomo.core.kernel.component_piecewise.transforms.piecewise(...)</td>
<td>Models a single-variate piecewise linear function.</td>
</tr>
<tr>
<td>pyomo.core.kernel.component_piecewise.transforms.PiecewiseLinearFunction(...)</td>
<td>A piecewise linear function</td>
</tr>
<tr>
<td>pyomo.core.kernel.component_piecewise.transforms.PiecewiseLinearFunction(f)</td>
<td>Base class for transformed piecewise linear functions</td>
</tr>
<tr>
<td>pyomo.core.kernel.component_piecewise.transforms.piecewise_convex(...)</td>
<td>Simple convex piecewise representation</td>
</tr>
<tr>
<td>pyomo.core.kernel.component_piecewise.transforms.piecewise_sos2(...)</td>
<td>Discrete SOS2 piecewise representation</td>
</tr>
<tr>
<td>pyomo.core.kernel.component_piecewise.transforms.piecewise_dcc(...)</td>
<td>Discrete DCC piecewise representation</td>
</tr>
<tr>
<td>pyomo.core.kernel.component_piecewise.transforms.piecewise_cc(...)</td>
<td>Discrete CC piecewise representation</td>
</tr>
<tr>
<td>pyomo.core.kernel.component_piecewise.transforms.piecewise_mc(...)</td>
<td>Discrete MC piecewise representation</td>
</tr>
<tr>
<td>pyomo.core.kernel.component_piecewise.transforms.piecewise_inc(...)</td>
<td>Discrete INC piecewise representation</td>
</tr>
<tr>
<td>pyomo.core.kernel.component_piecewise.transforms.piecewise_dlog(...)</td>
<td>Discrete DLOG piecewise representation</td>
</tr>
<tr>
<td>pyomo.core.kernel.component_piecewise.transforms.piecewise_log(...)</td>
<td>Discrete LOG piecewise representation</td>
</tr>
</tbody>
</table>
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': $y \leq f(x)$
  - 'eq': $y = f(x)$
  - 'ub': $y \geq f(x)$

- **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*

```python
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

class pyomo.core.kernel.component_piecewise.transforms.TransformedPiecewiseLinearFunction(f,
input=None, output=None, bound='eq', validate=True, **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’).  

9.3. Kernel Library Reference
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

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

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pyomo.core.kernel.component_piecewise.transforms_nd.piecewise_nd(...)</td>
<td>Models a multi-variate piecewise linear function.</td>
</tr>
<tr>
<td>pyomo.core.kernel.component_piecewise.transforms_nd.PiecewiseLinearFunctionND(...)</td>
<td>A multi-variate piecewise linear function</td>
</tr>
<tr>
<td>pyomo.core.kernel.component_piecewise.transforms_nd.TransformedPiecewiseLinearFunctionND(f)</td>
<td>Base class for transformed multi-variate piecewise</td>
</tr>
<tr>
<td>pyomo.core.kernel.component_piecewise.transforms_nd.piecewise_nd_cc(...)</td>
<td>Discrete CC multi-variate piecewise representation</td>
</tr>
</tbody>
</table>

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’: $y \leq f(x)$
  - ’eq’: $y = f(x)$
  - ’ub’: $y \geq f(x)$
- **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-variate 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.TransformedPiecewiseLinearFunctionND

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’: $y \leq f(x)$
  - ’eq’: $y = f(x)$
  - ’ub’: $y \geq f(x)$

__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

class 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`. 
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.
dp = DataPortal()
dp[name]

If a two arguments are given, then the first is the namespace and the second is the symbol name:

dp = DataPortal()
dp[namespace, name]

**Parameters**  
*args (str) – A tuple of arguments.

**Returns**  
If a single argument is given, then the data associated with that symbol in the namespace None is returned. If two arguments are given, then the data associated with symbol in the given namespace is returned.

__init__(*args, **kwds)

Constructor

__setitem__(name, value)

Set the value of name with the given value.

**Parameters**

- name (str) – The name of the symbol that is set.
- value – The value of the symbol.

__weakref__

list of weak references to the object (if defined)

connect(**kwds)

Construct a data manager object that is associated with the input source. This data manager is used to process future data imports and exports.

**Parameters**

- filename (str) – A filename that specifies the data source. Default is None.
- server (str) – The name of the remote server that hosts the data. Default is None.
- using (str) – The name of the resource used to load the data. Default is None.

Other keyword arguments are passed to the data manager object.

data (name=None, namespace=None)

Return the data associated with a symbol and namespace

**Parameters**

- name (str) – The name of the symbol that is returned. Default is None, which indicates that the entire data in the namespace is returned.
- namespace (str) – The name of the namespace that is accessed. Default is None.

**Returns**  
If name is None, then the dictionary for the namespace is returned. Otherwise, the data associated with name in given namespace is returned. The return value is a constant if None if there is a single value in the symbol dictionary, and otherwise the symbol dictionary is returned.

disconnect()

Close the data manager object that is associated with the input source.
items (namespace=None)
    Return an iterator of (name, value) tuples from the data in the specified namespace.

    Yields The next (name, value) tuple in the namespace. If the symbol has a simple data value, then that is included in the tuple. Otherwise, the tuple includes a dictionary mapping symbol indices to values.

keys (namespace=None)
    Return an iterator of the data keys in the specified namespace.

    Yields A string name for the next symbol in the specified namespace.

load (**kwds)
    Import data from an external data source.

    Parameters model – The model object for which this data is associated. Default is None.

    Other keyword arguments are passed to the connect() method.

namespaces()
    Return an iterator for the namespaces in the data portal.

    Yields A string name for the next namespace.

store (**kwds)
    Export data to an external data source.

    Parameters model – The model object for which this data is associated. Default is None.

    Other keyword arguments are passed to the connect() method.

values (namespace=None)
    Return an iterator of the data values in the specified namespace.

    Yields The data value for the next symbol in the specified namespace. This may be a simple value, or a dictionary of values.

class pyomo.core.data.TableData.TableData
    A class used to read/write data from/to a table in an external data source.

    __init__()
        Constructor

    add_options (**kwds)
        Add the keyword options to the Options object in this object.

    available()
        Returns True if the data manager is available.

    clear()
        Clear the data that was extracted from this table

    close()
        Close the data manager.

    initialize (**kwds)
        Initialize the data manager with keyword arguments.

        The filename argument is recognized here, and other arguments are passed to the add_options() method.

    open()
        Open the data manager.
```
process(model, data, default)
    Process the data that was extracted from this data manager and return it.

read()
    Read data from the data manager.

write(data)
    Write data to the data manager.
```

## 9.5 Solver Interfaces

### 9.5.1 GurobiPersistent

#### Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GurobiPersistent.activate()</code></td>
<td>Register this plugin with all interfaces that it implements.</td>
</tr>
<tr>
<td><code>GurobiPersistent.add_block(block)</code></td>
<td>Add a single Pyomo Block to the solver’s model.</td>
</tr>
<tr>
<td><code>GurobiPersistent.add_constraint(con)</code></td>
<td>Add a constraint to the solver’s model.</td>
</tr>
<tr>
<td><code>GurobiPersistent.set_objective(obj)</code></td>
<td>Set the solver’s objective.</td>
</tr>
<tr>
<td><code>GurobiPersistent.add_sos_constraint(con)</code></td>
<td>Add an SOS constraint to the solver’s model (if supported).</td>
</tr>
<tr>
<td><code>GurobiPersistent.add_var(var)</code></td>
<td>Add a variable to the solver’s model.</td>
</tr>
<tr>
<td><code>GurobiPersistent.alias(name[, doc, subclass])</code></td>
<td>This function is used to declare aliases that can be used by a factory for constructing plugin instances.</td>
</tr>
<tr>
<td><code>GurobiPersistent.available([exception_flag])</code></td>
<td>True if the solver is available.</td>
</tr>
<tr>
<td><code>GurobiPersistent.deactivate()</code></td>
<td>Unregister this plugin with all interfaces that it implements.</td>
</tr>
<tr>
<td><code>GurobiPersistent.disable()</code></td>
<td>Disable this plugin.</td>
</tr>
<tr>
<td><code>GurobiPersistent.enable()</code></td>
<td>Enable this plugin.</td>
</tr>
<tr>
<td><code>GurobiPersistent.enabled()</code></td>
<td>Return value indicating if this plugin is enabled.</td>
</tr>
<tr>
<td><code>GurobiPersistent.has_capability(cap)</code></td>
<td>Returns a boolean value representing whether a solver supports a specific feature.</td>
</tr>
<tr>
<td><code>GurobiPersistent.has_instance()</code></td>
<td>True if set_instance has been called and this solver interface has a pyomo model and a solver model.</td>
</tr>
<tr>
<td><code>GurobiPersistent.implements(interface[, ...])</code></td>
<td>Can be used in the class definition of Plugin subclasses to declare the extension points that are implemented by this interface class.</td>
</tr>
<tr>
<td><code>GurobiPersistent.load_vars([vars_to_load])</code></td>
<td>Load the values from the solver’s variables into the corresponding pyomo variables.</td>
</tr>
<tr>
<td><code>GurobiPersistent.problem_format()</code></td>
<td>Returns the current problem format.</td>
</tr>
<tr>
<td><code>GurobiPersistent.remove_block(block)</code></td>
<td>Remove a single block from the solver’s model.</td>
</tr>
<tr>
<td><code>GurobiPersistent.remove_constraint(con)</code></td>
<td>Remove a single constraint from the solver’s model.</td>
</tr>
<tr>
<td><code>GurobiPersistent.remove_sos_constraint(con)</code></td>
<td>Remove a single SOS constraint from the solver’s model.</td>
</tr>
<tr>
<td><code>GurobiPersistent.remove_var(var)</code></td>
<td>Remove a single variable from the solver’s model.</td>
</tr>
<tr>
<td><code>GurobiPersistent.reset()</code></td>
<td>Reset the state of the solver.</td>
</tr>
<tr>
<td><code>GurobiPersistent.results_format()</code></td>
<td>Returns the current results format.</td>
</tr>
<tr>
<td><code>GurobiPersistent.set_callback(name[, ...])</code></td>
<td>Set the callback function for a named callback.</td>
</tr>
<tr>
<td><code>GurobiPersistent.set_instance(model, **kwds)</code></td>
<td>This method is used to translate the Pyomo model provided to an instance of the solver’s Python model.</td>
</tr>
<tr>
<td><code>GurobiPersistent.set_problem_format(format)</code></td>
<td>Set the current problem format (if it’s valid) and update the results format to something valid for this problem format.</td>
</tr>
</tbody>
</table>
### Table 9.10 – continued from previous page

<table>
<thead>
<tr>
<th>Method / Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GurobiPersistent.set_results_format(format)</code></td>
<td>Set the current results format (if it’s valid for the current problem format).</td>
</tr>
<tr>
<td><code>GurobiPersistent.solve(*args, **kwds)</code></td>
<td>Solve the model.</td>
</tr>
<tr>
<td><code>GurobiPersistent.update_var(var)</code></td>
<td>Update a single variable in the solver’s model.</td>
</tr>
<tr>
<td><code>GurobiPersistent.version()</code></td>
<td>Returns a 4-tuple describing the solver executable version.</td>
</tr>
<tr>
<td><code>GurobiPersistent.write(filename)</code></td>
<td>Write the model to a file (e.g., and lp file).</td>
</tr>
</tbody>
</table>

#### 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` method.
- **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 single Pyomo Block to the solver’s model.

This will keep any existing model components intact.

**Parameters**

- **block** *(Block (scalar Block or single _BlockData))* –

**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 single 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 (scalar Block or a single _BlockData)) –

remove_constraint (con)
Remove a single constraint from the solver’s model.
This will keep any other model components intact.

Parameters con (Constraint (scalar Constraint or single _ConstraintData)) –

remove_sos_constraint (con)
Remove a single SOS constraint from the solver’s model.
This will keep any other model components intact.

Parameters con (SOSConstraint) –

remove_var (var)
Remove a single variable from the solver’s model.
This will keep any other model components intact.

Parameters var (Var (scalar Var or single _VarData)) –

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:

```python
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 \( (\text{obj}) \)

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** \( \text{obj} \) (Objective) –

set_problem_format \( (\text{format}) \)

Set the current problem format (if it’s valid) and update the results format to something valid for this problem format.

set_results_format \( (\text{format}) \)

Set the current results format (if it’s valid for the current problem format).

solve \( (*\text{args}, **\text{kwds}) \)

Solve the model.

**Keyword Arguments**

- **suffixes** (list of str) – The strings should represent 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 \( (\text{var}) \)

Update a single variable in the solver’s model.

This will update bounds, fix/unfix the variable as needed, and update the variable type.

**Parameters** \( \text{var} \) (scalar Var or single _VarData) –

version()

Returns a 4-tuple describing the solver executable version.

write \( (\text{filename}) \)

Write the model to a file (e.g., and lp file).

**Parameters** \( \text{filename} \) (str) – Name of the file to which the model should be written.

### 9.5.2 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` method.

- **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 single Pyomo Block to the solver’s model.
This will keep any existing model components intact.

**Parameters**

- **block** (*Block (scalar Block or single _BlockData)*)

**add_constraint**(con)
Add a single constraint to the solver’s model.
This will keep any existing model components intact.

**Parameters**

- **con** (*Constraint (scalar Constraint or single _ConstraintData)*)

**add_sos_constraint**(con)
Add a single 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 single 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 single 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 (scalar Block or a single _BlockData)*) –

remove_constraint (**con**)  
Remove a single constraint from the solver’s model.

This will keep any other model components intact.

Parameters **con** (*Constraint (scalar Constraint or single _ConstraintData)*) –
remove_sos_constraint(con)
Remove a single SOS constraint from the solver’s model.
This will keep any other model components intact.

Parameters con (SOSConstraint) –

remove_var(var)
Remove a single variable from the solver’s model.
This will keep any other model components intact.

Parameters var (Var (scalar Var or single _VarData)) –

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 represent 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 single variable in the solver’s model.
This will update bounds, fix/unfix the variable as needed, and update the variable type.

**version** ()

Returns a 4-tuple describing the solver executable version.

**write** *(filename, filetype=“”)*

Write the model to a file (e.g., an lp file).

Parameters

- **filename** *(str)* – Name of the file to which the model should be written.
- **filetype** *(str)* – The file type (e.g., lp).

Pyomo is under active ongoing development. The following API documentation describes Beta functionality.

Finally, Pyomo includes a number of third-party extensions:

## 9.6 Third-Party Contributions

Pyomo includes a variety of additional features and functionality provided by third parties through the `pyomo.contrib` package. This package includes both contributions included with the main Pyomo distribution and wrappers for third-party packages that must be installed separately.

These packages are maintained by the original contributors and are managed as optional Pyomo packages.

Contributed packages distributed with Pyomo:

- ...

Contributed packages distributed independently of Pyomo, but accessible through `pyomo.contrib`:

- pyomo.contrib.simplemodel
Examples of Pyomo models for different types of problems . . .

```python
>>> print('Hello World')
Hello World
```
Pyomo uses the `pyomo.contrib` package to facilitate the inclusion of third-party contributions that enhance Pyomo’s core functionality. There are two ways that `pyomo.contrib` can be used to integrate third-party packages:

- `pyomo.contrib` can provide wrappers for separate Python packages, thereby allowing these packages to be imported as subpackages of `pyomo`.
- `pyomo.contrib` can include contributed packages that are developed and maintained outside of the Pyomo developer team.

Including contrib packages in the Pyomo source tree provides a convenient mechanism for defining new functionality that can be optionally deployed by users. We expect this mechanism to include Pyomo extensions and experimental modeling capabilities. However, contrib packages are treated as optional packages, which are not maintained by the Pyomo developer team. Thus, it is the responsibility of the code contributor to keep these packages up-to-date.

Contrib package contributions will be considered as pull-requests, which will be reviewed by the Pyomo developer team. Specifically, this review will consider the suitability of the proposed capability, whether tests are available to check the execution of the code, and whether documentation is available to describe the capability. Contrib packages will be tested along with Pyomo. If test failures arise, then these packages will be disabled and an issue will be created to resolve these test failures.

The following two examples illustrate the two ways that `pyomo.contrib` can be used to integrate third-party contributions.

### 11.1 Including External Packages

The `pyomocontrib_simplemodel` package is derived from Pyomo, and it defines the class `SimpleModel` that illustrates how Pyomo can be used in a simple, less object-oriented manner. Specifically, this class mimics the modeling style supported by PuLP.

While `pyomocontrib_simplemodel` can be installed and used separate from Pyomo, this package is included in `pyomo/contrib/simplemodel`. This allows this package to be referenced as if were defined as a subpackage of `pyomo.contrib`. For example:
```python
from pyomo.contrib.simplemodel import *
from math import pi

m = SimpleModel()

r = m.var('r', bounds=(0, None))
h = m.var('h', bounds=(0, None))

m += 2*pi*r*(r + h)
status = m.solve("ipopt")
```

This example illustrates that a package can be distributed separate from Pyomo while appearing to be included in the `pyomo.contrib` subpackage. Pyomo requires a separate directory be defined under `pyomo/contrib` for each such package, and the Pyomo developer team will approve the inclusion of third-party packages in this manner.

### 11.2 Contrib Packages within Pyomo

Third-party contributions can also be included directly within the `pyomo.contrib` package. The `pyomo/contrib/example` package provides an example of how this can be done, including a directory for plugins and package tests. For example, this package can be imported as a subpackage of `pyomo.contrib`:

```python
from pyomo.environ import *
from pyomo.contrib.example import a

# Print the value of 'a' defined by this package
print(a)
```

Although `pyomo.contrib.example` is included in the Pyomo source tree, it is treated as an optional package. Pyomo will attempt to import this package, but if an import failure occurs, Pyomo will silently ignore it. Otherwise, this pyomo package will be treated like any other. Specifically:

- Plugin classes defined in this package are loaded when `pyomo.environ` is loaded.
- Tests in this package are run with other Pyomo tests.
CHAPTER 12

Bibliography
CHAPTER 13

Indices and Tables

• genindex
• modindex
• search
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
pyomo.core.kernel.component_block, 125
pyomo.core.kernel.component_interface, 120
pyomo.core.kernel.component_piecewise.util, 177
pyomo.core.kernel.component_suffix, 166
Symbols

.getActiveComponentContainerMixin (class in pyomo.core.kernel.component_interface), 122
.getActiveComponentMixin (class in pyomo.core.kernel.component_interface), 123
.getSimpleContainerMixin (class in pyomo.core.kernel.component_interface), 123
.abs__() (pyomo.core.expr.numvalue.NumericValue method), 93
.add__() (pyomo.core.expr.numvalue.NumericValue method), 93
.bool__() (pyomo.core.current.ExpressionBase method), 98
.call__() (pyomo.core.current.ExpressionBase method), 98
.call__() (pyomo.core.kernel.component_piecewise.transforms.PiecewiseLinearFunction method), 170
.call__() (pyomo.core.kernel.component_piecewise.transforms.TransformedPiecewiseLinearFunction method), 171
.call__() (pyomo.core.kernel.component_piecewise.transforms.nd.PiecewiseLinearFunctionND method), 176
.call__() (pyomo.core.kernel.component_piecewise.transforms.nd.TransformedPiecewiseLinearFunctionND method), 176
.delattr__ (pyomo.core.kernel.component_dict_ComponentDict attribute), 153
.delattr__ (pyomo.core.kernel.component_list_ComponentList attribute), 149
.delattr__ (pyomo.core.kernel.component_tuple_ComponentTuple attribute), 145
.div__() (pyomo.core.expr.numvalue.NumericValue method), 93
.eq__() (pyomo.core.expr.numvalue.NumericValue method), 93
.float__() (pyomo.core.expr.numvalue.NumericValue method), 93
.format__() (pyomo.core.kernel.component_dict_ComponentDict method), 153
.format__() (pyomo.core.kernel.component_list_ComponentList method), 149
.format__() (pyomo.core.kernel.component_tuple_ComponentTuple method), 145
.ge__() (pyomo.core.expr.numvalue.NumericValue method), 93
.getattribute__ (pyomo.core.kernel.component_dict_ComponentDict attribute), 153
.getattribute__ (pyomo.core.kernel.component_list_ComponentList attribute), 149
.getattribute__ (pyomo.core.kernel.component_tuple_ComponentTuple attribute), 146
.getstate__ (pyomo.core.expr.current.ExpressionBase method), 98
.getstate__ (pyomo.core.expr.numvalue.NumericValue method), 94
.gt__() (pyomo.core.expr.numvalue.NumericValue method), 94
.hash__ (pyomo.core.kernel.component_dict_ComponentDict attribute), 146
.hash__ (pyomo.core.kernel.component_list_ComponentList attribute), 146
.hash__ (pyomo.core.kernel.component_tuple_ComponentTuple attribute), 146
.iadd__ (pyomo.core.expr.numvalue.NumericValue method), 94
.idiv__ (pyomo.core.expr.numvalue.NumericValue method), 94
.imul__ (pyomo.core.expr.numvalue.NumericValue method), 94
.init__() (pyomo.core.data.TableData.TableData method), 180
.int__() (pyomo.core.expr.numvalue.NumericValue method), 94
.ipow__ (pyomo.core.expr.numvalue.NumericValue method), 94
.isub__ (pyomo.core.expr.numvalue.NumericValue method), 95
.itruediv__ (pyomo.core.expr.numvalue.NumericValue method), 95
__le__() (pyomo.core.expr.numvalue.NumericValue method), 95
__lt__() (pyomo.core.expr.numvalue.NumericValue method), 95
__metaclass__ (pyomo.core.kernel.component_dict.ComponentDict attribute), 153
__metaclass__ (pyomo.core.kernel.component_list.ComponentList attribute), 149
__metaclass__ (pyomo.core.kernel.component_tuple.ComponentTuple attribute), 146
__mul__() (pyomo.core.expr.numvalue.NumericValue method), 95
__neg__() (pyomo.core.expr.numvalue.NumericValue method), 95
__new__() (pyomo.core.kernel.component_dict.ComponentDict method), 153
__new__() (pyomo.core.kernel.component_list.ComponentList method), 149
__new__() (pyomo.core.kernel.component_tuple.ComponentTuple method), 146
__nonzero__() (pyomo.core.expr.current.ExpressionBase method), 98
__pos__() (pyomo.core.expr.numvalue.NumericValue method), 95
__pow__() (pyomo.core.expr.numvalue.NumericValue method), 95
__radd__() (pyomo.core.expr.numvalue.NumericValue method), 96
__rdiv__() (pyomo.core.expr.numvalue.NumericValue method), 96
__reduce__() (pyomo.core.kernel.component_dict.ComponentDict method), 153
__reduce__() (pyomo.core.kernel.component_list.ComponentList method), 149
__reduce__() (pyomo.core.kernel.component_tuple.ComponentTuple method), 146
__reduce_ex__() (pyomo.core.kernel.component_dict.ComponentDict method), 153
__reduce_ex__() (pyomo.core.kernel.component_list.ComponentList method), 149
__reduce_ex__() (pyomo.core.kernel.component_tuple.ComponentTuple method), 146
__setattr__ (pyomo.core.kernel.component_dict.ComponentDict attribute), 153
__setattr__ (pyomo.core.kernel.component_list.ComponentList attribute), 149
__setattr__ (pyomo.core.kernel.component_tuple.ComponentTuple attribute), 146
__setitem__() (pyomo.core.base.DataPortal.DataPortal method), 179
__setstate__() (pyomo.core.expr.numvalue.NumericValue method), 96
__sizeof__() (pyomo.core.kernel.component_dict.ComponentDict method), 153
__sizeof__() (pyomo.core.kernel.component_list.ComponentList method), 149
__sizeof__() (pyomo.core.kernel.component_tuple.ComponentTuple method), 146
__str__() (pyomo.core.kernel.component_dict.ComponentDict method), 153
__str__() (pyomo.core.kernel.component_list.ComponentList method), 149
__str__() (pyomo.core.kernel.component_tuple.ComponentTuple method), 146
__sub__() (pyomo.core.expr.numvalue.NumericValue method), 96
__true_div__() (pyomo.core.expr.numvalue.NumericValue method), 97
__reduce__() (pyomo.core.kernel.component_dict.ComponentDict method), 153
__reduce__() (pyomo.core.kernel.component_list.ComponentList method), 149
__reduce__() (pyomo.core.kernel.component_tuple.ComponentTuple method), 146
__reduce_ex__() (pyomo.core.kernel.component_dict.ComponentDict method), 153
__reduce_ex__() (pyomo.core.kernel.component_list.ComponentList method), 149
__reduce_ex__() (pyomo.core.kernel.component_tuple.ComponentTuple method), 146
__repr__() (pyomo.core.kernel.component_interface._ActiveComponentMixin method), 123
apply_operation() (pyomo.core.expr.current.EqualityExpression method), 104
apply_operation() (pyomo.core.expr.current.Expr_ifExpression method), 105
apply_operation() (pyomo.core.expr.current.ExpressionBase method), 98
apply_operation() (pyomo.core.expr.current.ExternalFunctionExpression method), 102
apply_operation() (pyomo.core.expr.current.GetItemExpression method), 101
A

AbsExpression (class in pyomo.core.expr.current), 106
activate() (pyomo.core.base.block.Block method), 118
activate() (pyomo.core.base.constraint.Constraint method), 114
activate() (pyomo.core.base.objective.Objective method), 117
activate() (pyomo.core.kernel.component_block.block_tuple method), 127
activate() (pyomo.core.kernel.component_block.tiny_block method), 131
activate() (pyomo.core.kernel.component_interface._ActiveComponentContainerMixin method), 123
activate() (pyomo.core.kernel.component_interface._ActiveComponentMixin method), 123
activate() (pyomo.core.kernel.component_interface.IActiveObject method), 120
activate() (pyomo.core.kernel.component_interface.IActiveObject attribute), 120
activate() (pyomo.core.kernel.component_block.block attribute), 127
activate() (pyomo.core.kernel.component_block.block_dict attribute), 131
activate() (pyomo.core.kernel.component_block.block_list attribute), 135
activate() (pyomo.core.kernel.component_block.block_tuple attribute), 138
activate() (pyomo.core.kernel.component_block.IBlockStorage attribute), 126
activate() (pyomo.core.kernel.component_block.tiny_block attribute), 141
activate() (pyomo.core.kernel.component_interface._ActiveComponentContainerMixin method), 123
activate() (pyomo.core.kernel.component_interface._ActiveComponentMixin method), 123
activate() (pyomo.core.kernel.component_interface.IActiveObject attribute), 120
add() (pyomo.core.base.var.Var method), 112
add() (pyomo.core.expr.current.SumExpression method), 104
add_block() (pyomo.solvers.plugins.solvers.cplex_persistent.CPLEXPersistent method), 186
add_block() (pyomo.solvers.plugins.solvers.gurobi_persistent.GurobiPersistent method), 182
add_constraint() (pyomo.solvers.plugins.solvers.cplex_persistent.CPLEXPersistent method), 186
add_constraint() (pyomo.solvers.plugins.solvers.gurobi_persistent.GurobiPersistent method), 182
add_options() (pyomo.core.data.TableData.TableData method), 180
child_key() (pyomo.core.kernel.component_tuple.ComponentTuple method), 146
children() (pyomo.core.kernel.component_block.block method), 127
children() (pyomo.core.kernel.component_block.block_list collect_ctypes() (pyomo.core.kernel.component_block.block_tuple method), 138
children() (pyomo.core.kernel.component_block.block_tuple Collocation_Discretization_Transformation (class in pyomo.dae.plugins.colloc), 53
children() (pyomo.core.kernel.component_block.tiny_block ComponentDict (class in pyomo.core.kernel.component_dict), 153
children() (pyomo.core.kernel.component_dict.ComponentDict components() (pyomo.core.kernel.component_block.block_list method), 135
children() (pyomo.core.kernel.component_dict.ComponentList components() (pyomo.core.kernel.component_block.block_tuple method), 138
children() (pyomo.core.kernel.component_list.ComponentList components() (pyomo.core.kernel.component_block.block_list method), 135
children() (pyomo.core.kernel.component_tuple.ComponentTuple components() (pyomo.core.kernel.component_block.block_tuple method), 146
clear() (pyomo.core.base.block.Block method), 118
clear() (pyomo.core.base.constraint.Constraint method), 114
clear() (pyomo.core.base.objective.Objective method), 117
clear() (pyomo.core.base.sets.Set method), 110
clear() (pyomo.core.base.var.Var method), 112
clear() (pyomo.core.data.TableData.TableData method), 180
clear() (pyomo.core.kernel.component_block.block_dict method), 132
clear() (pyomo.core.kernel.component_dict.ComponentDict method), 154
clear() (pyomo.core.kernel.component_tuple.ComponentTuple method), 146
clear_suffix_value() (pyomo.core.base.block.Block method), 118
clear_suffix_value() (pyomo.core.base.constraint.Constraint method), 115
clear_suffix_value() (pyomo.core.base.objective.Objective method), 117
clear_suffix_value() (pyomo.core.base.sets.Set method), 110
clear_suffix_value() (pyomo.core.base.var.Var method), 112
clone() (pyomo.core.expr.current.ExpressionBase method), 100
clone() (pyomo.core.kernel.component_block.block method), 127
clone() (pyomo.core.kernel.component_block.IBlockStorage method), 126
clone() (pyomo.core.kernel.component_block.tiny_block method), 141
clone_counter (class in pyomo.core.expr.current), 92
close() (pyomo.core.data.TableData.TableData method), 180
Constraint (class in pyomo.core.kernel.component_constraint), 114
Constraint (class in pyomo.core.kernel.component_constraint), 159
Constraint (class in pyomo.core.kernel.component_constraint), 162
Constraint (class in pyomo.core.kernel.component_constraint), 162
Constraint_list (class in pyomo.core.kernel.component_constraint), 162
ComponentTuple (class in pyomo.core.kernel.component_tuple), 145
construct() (pyomo.core.base.block.Block method), 119
construct() (pyomo.core.base.constraint.Construct method), 115
construct() (pyomo.core.base.objective.Objective method), 117
construct() (pyomo.core.base.sets.Set method), 110
construct() (pyomo.core.base.var.Var method), 112
construct() (pyomo.dae.ContinuousSet method), 44
construct_node() (pyomo.core.expr.current.ExpressionReplacementVisitor method), 108
ContinuousSet (class in pyomo.dae), 44
count (pyomo.core.expr.current.clone_counter attribute), 92
count() (pyomo.core.kernel.component_block.block_list method), 135
count() (pyomo.core.kernel.component_block.block_tuple method), 138
count() (pyomo.core.kernel.component_list.ComponentList method), 150
count() (pyomo.core.kernel.component_tuple.ComponentTuple method), 147
CPLEXPersistent (class in pyomo.solvers.plugins.solvers.cplex_persistent), 185
create_node_with_local_data() (pyomo.core.expr.current.AbsExpression method), 106
create_node_with_local_data() (pyomo.core.expr.current.ExpressionBase method), 100
create_node_with_local_data() (pyomo.core.expr.current.ExternalFunctionExpression method), 103
create_node_with_local_data() (pyomo.core.expr.current.GetItemExpression method), 105
create_node_with_local_data() (pyomo.core.expr.current.InequalityExpression method), 104
create_node_with_local_data() (pyomo.core.expr.current.SumExpression method), 104
create_node_with_local_data() (pyomo.core.expr.current.UnaryFunctionExpression method), 106
create_potentially_variable_object() (pyomo.core.expr.current.ExpressionBase method), 100
create_variable_dict() (in module pyomo.core.kernel.component_variable), 159
create_variable_list() (in module pyomo.core.kernel.component_variable), 158
create_variable_tuple() (in module pyomo.core.kernel.component_variable), 158


DataPortal (class in pyomo.core.base.DataPortal), 178
datatype (pyomo.core.kernel.component_suffix.suffix attribute), 167
deactivate() (pyomo.core.base.block.Block method), 119
deactivate() (pyomo.core.base.constraint.Constraint method), 115
deactivate() (pyomo.core.base.objective.Objective method), 117
deactivate() (pyomo.core.kernel.component_block.block_list method), 128
deactivate() (pyomo.core.kernel.component_block.block_dict method), 132
deactivate() (pyomo.core.kernel.component_block.block_list method), 135
deactivate() (pyomo.core.kernel.component_block.block_tuple method), 138
deactivate() (pyomo.core.kernel.component_block.IBlockStorage method), 126
deactivate() (pyomo.core.kernel.component_block.tiny_block method), 142
deactivate() (pyomo.core.kernel.component_interface._ActiveComponentContainerMixin method), 123
deactivate() (pyomo.core.kernel.component_interface._ActiveComponentMixin method), 123
deactivate() (pyomo.core.kernel.component_interface.IActiveObject method), 120
deactivate() (pyomo.solvers.plugins.solvers.cplex_persistent.CPLEXPersistent method), 186
deactivate() (pyomo.solvers.plugins.solvers.gurobi_persistent.GurobiPersistent method), 183
decompose_term() (in module pyomo.core.expr.current), 90
default_labeler (pyomo.core.expr.symbol_map.SymbolMap attribute), 92
DerivativeVar (class in pyomo.dae), 46
dfs_postorder_stack() (pyomo.core.expr.current.ExpressionReplacementVisitor method), 108
dfs_postorder_stack() (pyomo.core.expr.current.ExpressionValueVisitor method), 107
dim() (pyomo.core.base.block.Block method), 119
dim() (pyomo.core.base.constraint.Constraint method), 115
dim() (pyomo.core.base.objective.Objective method), 117
dim() (pyomo.core.base.sets.Set method), 110
dim() (pyomo.core.base.var.Var method), 112
direction (pyomo.core.kernel.component_suffix.suffix attribute), 167
disable() (pyomo.solvers.plugins.solvers.cplex_persistent.CPLEXPersistent method), 186
disable() (pyomo.solvers.plugins.solvers.gurobi_persistent.GurobiPersistent method), 183

Index 211

display() (pyomo.core.base.block.Block method), 119
display() (pyomo.core.base.constraint.Constraint method), 115
display() (pyomo.core.base.objective.Objective method), 117
domain (pyomo.core.kernel.component_variable.variable attribute), 158
domain_type (pyomo.core.kernel.component_variable.variable attribute), 158
dot_product (in module pyomo.core.util), 90

temporary_value (class in pyomo.core.kernel.component_variable), 97
temporary_value_dict (class in pyomo.core.kernel.component_variable_dict), 107
ExternalFunctionExpression (class in pyomo.core.expr.current), 102

ExpressionValueVisitor (class in pyomo.core.expr.current), 107
extend() (pyomo.core.kernel.component_block.block_list method), 135
extend() (pyomo.core.kernel.component_list.ComponentList method), 150

generate_delaunay() (in module pyomo.core.kernel.component_piecewise.util), 177
generate_gray_code() (in module pyomo.core.kernel.component_piecewise.util), 178
generate_names() (pyomo.core.kernel.component_block.block method), 128
generate_names() (pyomo.core.kernel.component_block.block_dict method), 132
generate_names() (pyomo.core.kernel.component_block.block_list method), 135
generate_names() (pyomo.core.kernel.component_block.block_tuple method), 138
generate_names() (pyomo.core.kernel.component_block.tiny_block method), 142
generate_names() (pyomo.core.kernel.component_dict.ComponentDict method), 154
generate_names() (pyomo.core.kernel.component_interface._SimpleContainerMixin method), 124
generate_names() (pyomo.core.kernel.component_list.ComponentList method), 150
generate_names() (pyomo.core.kernel.component_tuple.ComponentTuple method), 147
generate_names() (pyomo.core.kernel.component_variable.attribute), 158

ExpressionBase (class in pyomo.core.expressions), 97
ExpressionReplacementVisitor (class in pyomo.core.expressions), 108

ExpressionValueVisitor (class in pyomo.core.expressions), 107
extend() (pyomo.core.kernel.component_block.block_list method), 135
extend() (pyomo.core.kernel.component_list.ComponentList method), 150
ExternalFunctionExpression (class in pyomo.core.expressions), 102
extract_values() (pyomo.core.base.var.Var method), 112

finalization (pyomo.core.expr.current.ExpressionReplacementVisitor method), 108
finalization (pyomo.core.expr.current.ExpressionValueVisitor method), 107

G

generate_delaunay() (in module pyomo.core.kernel.component_piecewise.util), 177
generate_gray_code() (in module pyomo.core.kernel.component_piecewise.util), 178
generate_names() (pyomo.core.kernel.component_block.block method), 128
generate_names() (pyomo.core.kernel.component_block.block_dict method), 132
generate_names() (pyomo.core.kernel.component_block.block_list method), 135
generate_names() (pyomo.core.kernel.component_block.block_tuple method), 138
generate_names() (pyomo.core.kernel.component_block.tiny_block method), 142
generate_names() (pyomo.core.kernel.component_dict.ComponentDict method), 154
generate_names() (pyomo.core.kernel.component_interface._SimpleContainer method), 124
generate_names() (pyomo.core.kernel.component_list.ComponentList method), 150
generate_names() (pyomo.core.kernel.component_tuple.ComponentTuple method), 147
generate_names() (pyomo.core.kernel.component_variable.attribute), 158

get() (pyomo.core.kernel.component_block.block_list method), 135
get() (pyomo.core.kernel.component_dict.ComponentDict method), 154
get_changed() (pyomo.dae.ContinuousSet method), 44
Index 213

H

has_capability() (pyomo.solvers.plugins.solvers.cplex_persistent.CPLEXPersistent method), 186
has_capability() (pyomo.solvers.plugins.solvers.gurobi_persistent.GurobiPersistent method), 183
has_instance() (pyomo.solvers.plugins.solvers.cplex_persistent.CPLEXPersistent method), 187
has_instance() (pyomo.solvers.plugins.solvers.gurobi_persistent.GurobiPersistent method), 183

I

IActiveObject (class in pyomo.core.kernel.component_interface), 120
IBlockStorage (class in pyomo.core.kernel.component_block), 125
ICategorizedObject (class in pyomo.core.kernel.component_interface), 120
ICOMPONENT (class in pyomo.core.kernel.component_interface), 122
ICOMPONENTContainer (class in pyomo.core.kernel.component_interface), 122
id_index_map() (pyomo.core.base.block.Block method), 119
id_index_map() (pyomo.core.base.constraint.Constraint method), 115
id_index_map() (pyomo.core.base.objective.Objective method), 117
id_index_map() (pyomo.core.base.sets.Set method), 111
id_index_map() (pyomo.core.base.var.Var method), 113
identify_components() (in module pyomo.core.expr.current), 91
identify_variables() (in module pyomo.core.expr.current), 91
implements() (pyomo.solvers.plugins.solvers.cplex_persistent.CPLEXPersistent
method), 187
implements() (pyomo.solvers.plugins.solvers.gurobi_persistent.GurobiPersistent
method), 183
import_enabled (pyomo.core.kernel.component_suffix.suffix
attribute), 167
import_suffix_generator() (in module pyomo.core.kernel.component_suffix), 166
index() (pyomo.core.kernel.component_block.block_list
method), 136
index() (pyomo.core.kernel.component_block.block_tuple
method), 139
index() (pyomo.core.kernel.component_list.ComponentList
method), 151
index() (pyomo.core.kernel.component_tuple.ComponentTuple
method), 147
index_set() (pyomo.core.base.block.Block method), 119
index_set() (pyomo.core.base.constraint.Constraint
method), 115
index_set() (pyomo.core.base.objective.Objective
method), 117
index_set() (pyomo.core.base.sets.Set method), 111
index_set() (pyomo.core.base.var.Var method), 113
InequalityExpression (class in pyomo.core.expr.current), 103
initialize() (pyomo.core.data.TableData.TableData
method), 180
initialize_model() (pyomo.dae.Simulator method), 56
input (pyomo.core.kernel.component_piecewise.transforms.TransformedPiecewiseLinearFunction
attribute), 172
input (pyomo.core.kernel.component_piecewise.transforms
attribute), 176
insert() (pyomo.core.kernel.component_block.block_list
method), 136
insert() (pyomo.core.kernel.component_list.ComponentList
method), 151
Integral (class in pyomo.dae), 49
is_constructed() (pyomo.core.base.block.Block method), 119
is_CPLEXPersistent() (pyomo.core.base.constraint.Constraint
method), 115
is_GurobiPersistent() (pyomo.core.base.objective.Objective
method), 117
is_expression_type() (pyomo.core.base.sets.Set method), 111
is_constructed() (pyomo.core.base.var.Var method), 113
is_expression_type() (pyomo.core.expr.current.ExpressionBase
method), 101
is_expression_type() (pyomo.core.expr.numvalue.NumericValue
method), 97
is_indexed() (pyomo.core.base.block.Block method), 119
is_indexed() (pyomo.core.base.constraint.Constraint
method), 115
is_indexed() (pyomo.core.base.objective.Objective
method), 117
is_indexed() (pyomo.core.base.sets.Set method), 111
is_indexed() (pyomo.core.base.var.Var method), 113
is_fully_discretized() (pyomo.dae.DerivativeVar
method), 47
is_indexed() (pyomo.core.base.block.Block method), 119
is_indexed() (pyomo.core.base.constraint.Constraint
method), 115
is_indexed() (pyomo.core.base.objective.Objective
method), 117
is_indexed() (pyomo.core.base.sets.Set method), 111
is_indexed() (pyomo.core.expr.numvalue.NumericValue
method), 97
is_nondecreasing() (pyomo.core.kernel.component_piecewise.util), 178
is_nonincreasing() (pyomo.core.kernel.component_piecewise.util), 178
is_positive_power_of_two() (pyomo.core.kernel.component_piecewise.util), 178
is_potentially_variable() (pyomo.core.expr.current.EqualityExpression
method), 104
is_potentially_variable() (pyomo.core.expr.current.Expr_ifExpression
method), 104
is_potentially_variable() (pyomo.core.expr.numvalue.NumericValue
method), 97
load_solution() (pyomo.core.kernel.component_block.block method), 129
load_solution() (pyomo.core.kernel.component_block.tiny_block method), 143
load_vars() (pyomo.solvers.plugins.solvers.cplex_persistent.CPLEXPersistent method), 187
load_vars() (pyomo.solvers.plugins.solvers.gurobi_persistent.GurobiPersistent method), 183
local_name (pyomo.core.base.block.Block attribute), 119
local_name (pyomo.core.base.constraint.Constraint attribute), 115
local_name (pyomo.core.base.objective.Objective attribute), 118
local_name (pyomo.core.base.sets.Set attribute), 111
local_name (pyomo.core.base.var.Var attribute), 113
local_name (pyomo.core.kernel.component_block.block attribute), 129
local_name (pyomo.core.kernel.component_block.block_dict attribute), 133
local_name (pyomo.core.kernel.component_block.block_list attribute), 136
local_name (pyomo.core.kernel.component_block.block_tuple attribute), 139
local_name (pyomo.core.kernel.component_dict.ComponentDict attribute), 155
local_name (pyomo.core.kernel.component_interface.ICategorizedObject attribute), 121
local_name (pyomo.core.kernel.component_list.ComponentList attribute), 151
local_name (pyomo.core.kernel.component_tuple.ComponentTuple attribute), 147
nargs() (pyomo.core.expr.current.EqualityExpression method), 104
nargs() (pyomo.core.expr.current.Expr_ifExpression method), 105
nargs() (pyomo.core.expr.current.ExpressionBase method), 101
nargs() (pyomo.core.expr.current.ExternalFunctionExpression method), 103
nargs() (pyomo.core.expr.current.GetItemExpression method), 105
nargs() (pyomo.core.expr.current.InequalityExpression method), 105
nargs() (pyomo.core.expr.current.NegationExpression method), 104
nargs() (pyomo.core.expr.current.ReciprocalExpression method), 102
nargs() (pyomo.core.expr.current.SumExpression method), 103
nargs() (pyomo.core.expr.current.UnaryFunctionExpression method), 106
native_numeric_types (in module pyomo.core.expr.numvalue), 93
native_types (in module pyomo.core.expr.numvalue), 93
NegationExpression (class in pyomo.core.expr.current),

M
matrix_constraint (class in pyomo.core.kernel.component_matrix_constraint),
model() (pyomo.core.base.block.Block method), 119
model() (pyomo.core.base.constraint.Constraint method), 115
model() (pyomo.core.base.objective.Objective method), 118
model() (pyomo.core.base.sets.Set method), 111
model() (pyomo.core.base.var.Var method), 113
name (pyomo.core.base.block.Block attribute), 119
name (pyomo.core.base.constraint.Constraint attribute), 115
name (pyomo.core.base.objective.Objective attribute), 118
name (pyomo.core.base.sets.Set attribute), 111
name (pyomo.core.base.var.Var attribute), 113
name (pyomo.core.kernel.component_block.block attribute), 129
name (pyomo.core.kernel.component_block.block_dict attribute), 133
name (pyomo.core.kernel.component_block.block_list attribute), 136
name (pyomo.core.kernel.component_block.block_tuple attribute), 139
name (pyomo.core.kernel.component_block.IBlockStorage attribute), 126
name (pyomo.core.kernel.component_block.tiny_block attribute), 143
name (pyomo.core.kernel.component_dict.ComponentDict attribute), 155
name (pyomo.core.kernel.component_interface.ICategorizedObject attribute), 121
name (pyomo.core.kernel.component_list.ComponentList attribute), 151
name (pyomo.core.kernel.component_tuple.ComponentTuple attribute), 147
nargs() (pyomo.core.expr.current.EqualityExpression method), 104
nargs() (pyomo.core.expr.current.Expr_ifExpression method), 105
nargs() (pyomo.core.expr.current.ExpressionBase method), 101
nargs() (pyomo.core.expr.current.ExternalFunctionExpression method), 103
nargs() (pyomo.core.expr.current.GetItemExpression method), 105
nargs() (pyomo.core.expr.current.InequalityExpression method), 105
nargs() (pyomo.core.expr.current.NegationExpression method), 104
nargs() (pyomo.core.expr.current.ReciprocalExpression method), 102
nargs() (pyomo.core.expr.current.SumExpression method), 103
nargs() (pyomo.core.expr.current.UnaryFunctionExpression method), 106
native_numeric_types (in module pyomo.core.expr.numvalue), 93
native_types (in module pyomo.core.expr.numvalue), 93
NegationExpression (class in pyomo.core.expr.current),
nonlinear_expression (class in pyomo.core.expr.current), 92
nonpyomo_leaf_types (in module pyomo.core.expr.numvalue), 93
NumericValue (class in pyomo.core.expr.numvalue), 93

Objective (class in pyomo.core.base.objective), 116
objective (class in pyomo.core.kernel.component_objective), 164
objective_dict (class in pyomo.core.kernel.component_objective), 164
objective_list (class in pyomo.core.kernel.component_objective), 164
objective_tuple (class in pyomo.core.kernel.component_objective), 164
open() (pyomo.core.data.TableData.TableData method), 180
output (pyomo.core.kernel.component_piecewise.Transforms.TransformedPiecewiseLinearFunction attribute), 172
output (pyomo.core.kernel.component_piecewise.Transforms.TransformedPiecewiseLinearFunctionND attribute), 177

Parameter (class in pyomo.core.kernel.component_parameter), 163
parameter (class in pyomo.core.kernel.component_parameter), 163
parameter_dict (class in pyomo.core.kernel.component_parameter), 163
parameter_list (class in pyomo.core.kernel.component_parameter), 163
parameter_tuple (class in pyomo.core.kernel.component_parameter), 163
parent (pyomo.core.kernel.component_block.block attribute), 130
parent (pyomo.core.kernel.component_block.block_dict attribute), 133
parent (pyomo.core.kernel.component_block.block_list attribute), 136
parent (pyomo.core.kernel.component_block.block_tuple attribute), 139
parent (pyomo.core.kernel.component_block.IBlockStorage attribute), 126
parent (pyomo.core.kernel.component_block.tiny_block attribute), 143
parent (pyomo.core.kernel.component_dict.ComponentDict attribute), 155
parent (pyomo.core.kernel.component_interface.ICategorizedObject attribute), 121
parent (pyomo.core.kernel.component_list.ComponentList attribute), 151
parent (pyomo.core.kernel.component_tuple.ComponentTuple attribute), 148
parent_block (pyomo.core.kernel.component_block.block attribute), 148
parent_block (pyomo.core.kernel.component_block.block_dict attribute), 133
parent_block (pyomo.core.kernel.component_block.block_list attribute), 136
parent_block (pyomo.core.kernel.component_block.block_tuple attribute), 139
parent_block (pyomo.core.kernel.component_block.IBlockStorage attribute), 126
parent_block (pyomo.core.kernel.component_block.tiny_block attribute), 143
parent_block (pyomo.core.kernel.component_dict.ComponentDict attribute), 155
parent_block (pyomo.core.kernel.component_interface(ICategorizedObject attribute), 122
parent_block (pyomo.core.kernel.component_list.ComponentList attribute), 151
parent_block (pyomo.core.kernel.component_tuple.ComponentTuple attribute), 148
parent_block() (pyomo.core.base.block.Block method), 120
parent_block() (pyomo.core.base.constraint.Constraint method), 116
parent_block() (pyomo.core.base.objective.Objective method), 118
parent_block() (pyomo.core.base.sets.Set method), 111
parent_component() (pyomo.core.base.var.Var method), 113
parent_component() (pyomo.core.base.block.Block method), 120
parent_component() (pyomo.core.base.constraint.Constraint method), 116
parent_component() (pyomo.core.base.objective.Objective method), 118
parent_component() (pyomo.core.base.sets.Set method), 111
parent_component() (pyomo.core.base.var.Var method), 113
piecewise() (in module pyomo.core.kernel.component_piecewise.transforms), 169
piecewise_cc (class in pyomo.core.kernel.component_piecewise.transforms), 173
piecewise_convex (class in pyomo.core.kernel.component_piecewise.transforms), 172
piecewise_dcc (class in py-
Index 221

sos2() (in module pyomo.core.kernel.component_sos), 165
sos_dict (class in pyomo.core.kernel.component_sos), 165
sos_list (class in pyomo.core.kernel.component_sos), 165
sos_tuple (class in pyomo.core.kernel.component_sos), 165
sparse (pyomo.core.kernel.component_matrix_constraint.matrix_constraint
attribute), 162
stale (pyomo.core.kernel.component_variable.variable attribute), 158
suffix (class in pyomo.core.kernel.component_suffix), 167
suffix_generator() (in module pyomo.core.kernel.component_suffix), 167
sum_product() (in module pyomo.core.util), 89
SumExpression (class in pyomo.core.expr.current), 104
summation (in module pyomo.core.util), 90
SymbolMap (class in pyomo.core.expr.symbol_map), 92

T
TableData (class in pyomo.core.data.TableData), 180
terms (pyomo.core.kernel.component_constraint.linear_constraint
attribute), 161
tiny_block (class in pyomo.core.kernel.component_block), 141
to_dense_data() (pyomo.core.base.block.Block method), 120
to_dense_data() (pyomo.core.base.constraint.Constraint method), 116
to_dense_data() (pyomo.core.base.objective.Objective method), 118
to_dense_data() (pyomo.core.base.sets.Set method), 112
to_dense_data() (pyomo.core.base.var.Var method), 114
to_string() (pyomo.core.base.block.Block method), 120
to_string() (pyomo.core.base.constraint.Constraint method), 116
to_string() (pyomo.core.base.objective.Objective method), 118
to_string() (pyomo.core.base.sets.Set method), 112
to_string() (pyomo.core.base.var.Var method), 114
to_string() (pyomo.core.expr.current.ExpressionBase method), 101
to_string() (pyomo.core.expr.numvalue.NumericValue method), 97
TransformedPiecewiseLinearFunction (class in pyomo.core.kernel.component_piecewise.transforms), 171
TransformedPiecewiseLinearFunctionND (class in pyomo.core.kernel.component_piecewise.transforms_nd), 176
triangulation (pyomo.core.kernel.component_piecewise.transforms_nd.PiecewiseLinearFunctionND
attribute), 176
triangulation (pyomo.core.kernel.component_piecewise.transforms_nd.TransformedPiecewiseLinearFunctionND
attribute), 176
type() (pyomo.core.base.block.Block method), 120
type() (pyomo.core.base.constraint.Constraint method), 116
type() (pyomo.core.base.objective.Objective method), 118
type() (pyomo.core.base.sets.Set method), 112
type() (pyomo.core.base.var.Var method), 114
U
ub (pyomo.core.kernel.component_matrix_constraint.matrix_constraint
attribute), 162
ub (pyomo.core.kernel.component_variable.variable attribute), 158
UnaryFunctionExpression (class in pyomo.core.expr.current), 106
update() (pyomo.core.kernel.component_block.block_dict
method), 134
update() (pyomo.core.kernel.component_dict.ComponentDict
method), 156
update_var() (pyomo.solvers.plugins.solvers.cplex_persistent.CPLEXPersistent
method), 189
update_var() (pyomo.solvers.plugins.solvers.gurobi_persistent.GurobiPersistent
method), 185
uslack (pyomo.core.kernel.component_matrix_constraint.matrix_constraint
attribute), 163
V
valid_model_component() (pyomo.core.base.block.Block
method), 120
valid_model_component() (pyomo.core.base.constraint.Constraint
method), 116
valid_model_component() (pyomo.core.base.objective.Objective
method), 118
valid_model_component() (pyomo.core.base.sets.Set method), 112
valid_model_component() (pyomo.core.base.var.Var method), 114
validate() (pyomo.core.kernel.component_piecewise.transforms.piecewise_cc
method), 173
validate() (pyomo.core.kernel.component_piecewise.transforms.piecewise_convex
method), 172
validate() (pyomo.core.kernel.component_piecewise.transforms.piecewise_dcc
method), 173
validate() (pyomo.core.kernel.component_piecewise.transforms.piecewise_dlog
method), 174
validate() (pyomo.core.kernel.component_piecewise.transforms.piecewise_inc
method), 173