Chapter 2
Pyomo Modeling Strategies

Abstract This chapter illustrates different strategies for formulating and optimizing algebraic optimization models using Pyomo. We provide a brief overview of the core modeling components supported by Pyomo. Then, we describe how to formulate both concrete and abstract models with Pyomo. Finally, we provide a brief tutorial on how these models can be analyzed with the `pyomo` command.

2.1 Modeling Components

Pyomo supports an object-oriented design for the definition of optimization models. The basic steps of a simple modeling process are as follows:

1. Create model and declare components
2. Instantiate the model
3. Apply solver
4. Interrogate solver results

In practice, these steps may be applied repeatedly with different data or with different constraints applied to the model. However, we focus on this simple modeling process to illustrate different strategies for modeling with Pyomo.

A Pyomo *model* consists of a collection of modeling *components* that define different aspects of the model. Pyomo includes the modeling components that are commonly supported by modern AMLs: index sets, symbolic parameters, decision variables, objectives, and constraints. These modeling components are defined in Pyomo through the following Python classes:

`Set`	set data that is used to define a model instance
`Param`	parameter data that is used to define a model instance
`Var`	decision variables in a model
`Objective`	expressions that are minimized or maximized in a model
`Constraint`	constraint expressions in a model

W.E. Hart et al., *Pyomo—Optimization Modeling in Python*,
Springer Optimization and Its Applications 67, DOI 10.1007/978-1-4614-3226-5_2,

Two model classes provide a context for defining and initializing these modeling components: `ConcreteModel` and `AbstractModel`.[1] For example, modeling components can be directly added to an `AbstractModel` object as an attribute of the object:

```
model = AbstractModel()
model.I = Set()
model.p = Param(model.I)
```

The `model` object is a class instance of the `AbstractModel` class, and `model.I` is a `Set` object that is contained by this model. Many modeling components in Pyomo can be optionally specified as *indexed components*: collections of components that are referenced using one or more values. In this example, the parameter `p` is indexed with set `I`.

NOTE: For simplicity, many short examples in this book omit the Python import statement for `coopr.pyomo`. These examples assume that this package has been previously imported with the following command:

```
from coopr.pyomo import *
```

This command imports all of the classes, functions and data from the `coopr.pyomo` package; see Appendix B for further details about importing Python packages.

The following sections describe different strategies for creating models with Pyomo. We begin with strategies for generating concrete models and progress to more general strategies for abstract models. To illustrate this flexibility, we consider a generalization of the simple LP that we introduced in Chapter 1:

$$\begin{array}{lll} \min & \sum_{i=1}^{n} c_i x_i & \\ \text{s.t.} & \sum_{i=1}^{n} a_{ij} x_i \geq b_j & \forall j = 1 \ldots m \\ & x_i \geq 0 & \forall i = 1 \ldots n \end{array} \tag{2.1}$$

The following LP is a model instance (i.e., a model object with fully specified data) of the abstract LP 2.1:

$$\begin{array}{ll} \min & x_1 + 2x_2 \\ \text{s.t.} & 3x_1 + 4x_2 \geq 1 \\ & 2x_1 + 5x_2 \geq 2 \\ & x_1, x_2 \geq 0 \end{array} \tag{2.2}$$

[1] In versions 1.x and 2.x of Coopr, the `Model` class serves as an alias for the `AbstractModel` class. Both the aliasing and the `Model` class will be deprecated in Coopr 3.1 and subsequent releases.

The examples in this chapter illustrate different ways of formulating both abstract and concrete models.

We conclude this chapter with a brief introduction to the `pyomo` command, which can be used to optimize Pyomo models within a Unix or Windows command shell. Additionally, we summarize how Pyomo model scripts can be setup to construct and solve model instances. Note that additional detail about Pyomo modeling components can be found in Chapter 3, Chapter 4, and Chapter 5. Additional detail about the `pyomo` command and scripting with Pyomo models can be found in Chapter 7.

2.2 Concrete Models: Specifying Components Via Expressions

The simplest way to express a concrete model is by directly creating the modeling components with expressions that specify numerical values; the alternative, two-step process involving an abstract model is described in Section 2.4. The `ConcreteModel` class is used to define these types of models:

```
model = ConcreteModel()
```

In contrast to `AbstractModel` objects, model components contained in `ConcreteModel` objects are immediately initialized as they are added to the model instance.

Decision variables are required to define non-trivial expressions in Pyomo. In the simplest case, we can define each decision variable separately:

```
model.x_1 = Var(within=NonNegativeReals)
model.x_2 = Var(within=NonNegativeReals)
```

The `within` argument used in these declarations constrains the values of the defined variables to the set of non-negative real numbers, in this case by having Pyomo to automatically generate the associated lower bound constraints.

An optimization objective is defined by passing an algebraic expression involving numeric constants and Pyomo variables as an argument – via the `expr` keyword – to the `Objective` class constructor:

```
model.obj = Objective(expr=model.x_1 + 2*model.x_2)
```

The objective expression is defined implicitly by Pyomo, using overloading of the Python multiplication and addition operators. This is done automatically so the modeller need only give the expression as shown in the example. Similarly, constraints of a concrete model are defined by passing an expression as an argument – again via the `expr` keyword – to the `Constraint` class constructor:

```
model.con1 = Constraint(expr=3*model.x_1 + 4*model.x_2 >= 1)
model.con2 = Constraint(expr=2*model.x_1 + 5*model.x_2 >= 2)
```

Note that this expression additionally specifies a constraint bound, via overloading of the Python >= operator. Example 2.A.1 includes the entire formulation for this concrete model.

For large concrete models, the use of explicit variables is cumbersome, because it requires separate declarations for each variable in the model. To address this issue, Pyomo supports the notion of variable *indexing*. For example, the variable `x` can be defined with an explicit index set:

```
model.x = Var([1,2], within=NonNegativeReals)
```

This indexed variable can be used to specify the objectives and constraints with a natural, extensible syntax:

```
model.obj = Objective(expr=model.x[1] + 2*model.x[2])
model.con1 = Constraint(expr=3*model.x[1] + 4*model.x[2]>=1)
model.con2 = Constraint(expr=2*model.x[1] + 5*model.x[2]>=2)
```

Indexed variables allow logically related sets of variables to be grouped under a common name, and naturally differentiated via a sub-script.

Example 2.A.2 includes the formulation for the indexed concrete model of Formulation (2.2).

A limitation of the formulation in Example 2.A.2 is that the data used is explicitly expressed in the specification of the index set for `x` and the numerical values in the constraints and objective. Any change in data values requires potentially extensive modifications to the concrete model. Alternatively, this data can be symbolically specified using standard Python data structures and subsequently used in the formulation of the expressions for objectives and constraints. For example, the following declarations define Python dictionary objects that represent the data used in the objectives and constraints for Formulation (2.2):

```
N = [1,2]
c = {1:1, 2:2}
a = {(1,1):3, (2,1):4, (1,2):2, (2,2):5}
b = {1:1, 2:2}
```

Using this symbolic data, the objective can be simply redefined as follows:

```
model.obj = Objective(expr=c[1]*model.x[1] + c[2]*model.x[2])
```

An even more extensible formulation of this objective uses Python iteration syntax to sum over a set of related (indexed) decision variables. The Python `sum` function and *generator* syntax can be used to provide a concise specification of a summation:

```
model.obj = Objective(expr=sum(c[i]*model.x[i] for i in N))
```

This syntax specifies that the terms `c[i]*model.x[i]` are generated by iterating over the set `N`. As these terms are generated, the function `sum` adds them together to form the overall expression. This type of inlining of summations is a powerful feature that provides a syntax that is similar to that used in AMLs with custom languages. Example 2.A.3 includes the complete formulation of the concrete model using these external data declarations.

2.3 Concrete Models: Specifying Components Via Rules

Although the abstract formulation in Formulation (2.1) specifies a set of constraints, each constraint declared in Example 2.A.3 is defined separately. As in the case of decision variables, this modeling approach will become cumbersome for large models.

Pyomo addresses this and other, related issues by allowing modeling components to be initialized with user-defined functions, which we call *rules*. The idea is that complex initialization of a collection of constraints or objectives (for example) can be managed by a function that generates each constraint or objective expression individually. Similarly, rules can be used to construct complex sets and parameters in a generic manner.

Example 2.A.4 illustrates the use of a rule to define the constraints for Formulation (2.2) in a generic manner. The index set `M` defines the indices of the constraint `con`, and the function `con_rule` is used to construct the individual constraint expressions. The arguments to this rule are the constraint indices, in addition to the model instance that is being constructed. As with all components added to a `ConcreteModel` object, initialization is immediately executed for all values of the specified index set. Note that the name of a rule used to define a constraint (or other modeling component) is not restricted. However, some advanced Pyomo features require the use of unique rule names (e.g., see Chapter 10).

Although the function arguments for component rules are similar for all component types, the expected type of the values returned are different:

`Set`	rule must return a Python set or list object
`Param`	rule must return an integer or float value
`Objective`	rule must return an expression
`Constraint`	rule must return a constraint expression.

The distinction between an objective and constraint expression is simply the imposition of a lower or upper bound or, the case of a constraint, both. The `Set` and `Param` classes also support direct initialization with the `initialize` argument. This argument allows the user to specify the data that initializes this class, thereby avoiding the need to specify a Python function for a `rule` argument.

Example 2.A.5 illustrates the use of `rule` and `initialize` arguments in the context of the full spectrum of modeling components. This example is more verbose than Example 2.A.4, and represents the model in a less flexible way. Thus, the modeling representation in Example 2.A.4 might be preferable. However, Example 2.A.5 avoids the specification of global data, which promotes encapsulation that may be important in complex applications.

2.4 Abstract Models

In many contexts a strong separation of model and data is either helpful or necessary. Model data may not be immediately available, or it may be stored in an external database or spreadsheet. Alternatively, it may simply be desirable to represent a model in an abstract form that is independent of the manner in which the data is managed.

A simple strategy for managing models abstractly is to create a Python function whose arguments reflect the data that is needed to create a concrete model. Example 2.A.6 illustrates this approach using the concrete model developed in Example 2.A.4. The function `create_model` has arguments `N`, `M`, `c`, `a`, and `b` that represent the data needed to create a concrete model for Formulation (2.1). Consequently, this function provides a strong separation of model and data, and it can be viewed as a constructor for an abstract model. The code in Examples 2.A.4 and 2.A.6 can be executed using the `pyomo` command (see Section 2.5), ultimately yielding identical model instances and therefore identical optimization solutions.

Pyomo also supports the definition of *abstract* models that are defined independent of any specific data. Example 2.A.7.1 illustrates the definition of an abstract model in Pyomo for Formulation (2.1). In contrast to concrete models, abstract models strictly define the existence of model components and the relationships between them (e.g., the fact that set `N` is used to index parameter `c`).

The abstract model in Example 2.A.7.1 differs from the concrete model in Example 2.A.5 in three main respects. The first difference is that the class `AbstractModel` is used instead of `ConcreteModel`. This informs Pyomo that this is an abstract model that will be constructed with auxiliary data. Because data is not immediately available, the construction of modeling components is deferred until a model instance is generated. The second difference is that declarations of data components do not contain references to the data that will be used to construct these components. The final difference is that all objective and constraint components must be defined via rules. This is a requirement in abstract models, since the construction of these components ultimately depends on the availability of specific data.

NOTE: Pyomo does allow for hybrid models where some components are initialized with data while others are defined abstractly. Further, Pyomo data components can be defined with default values that are used when data is not specified. These options are discussed in Chapters 3, 4, and 5. However, with few exceptions these hybrid models must be defined with the `AbstractModel` class, and thus Pyomo treats them as an abstract model.

Pyomo includes a rich set of options for initializing an abstract model to create a model instance that can then be optimized (see Chapter 6). A simple strategy is to supply a *data command file* that specifies values for set and parameter data.

The syntax of Pyomo's data command files is very similar to the data command syntax supported by AMPL [4]. A goal of Pyomo is to support the same syntax for `set` and `param` data commands that is used in AMPL. However, the syntax for other commands relating to file inclusion and table imports and exports is somewhat different (see Chapter 6). Example 2.A.7.2 shows a file of data commands that can be used to initialize the abstract model in Example 2.A.7.1. This process is described in Section 2.5.

2.5 Optimizing Models

The previous sections have outlined different strategies for creating and populating a Pyomo model object. After the modeler has specified how this will be done, the Python commands for doing it are generally invoked using an executable script. The most straightforward way is to invoke the `pyomo` command that calls a pre-written script to perform optimization in a standard manner. Alternatively, a script can be created to customize the process using solver components from Coopr as introduced in Section 2.5.2.

2.5.1 Optimization with the `pyomo` Command

The Pyomo software distribution includes the `pyomo` command that can be used to construct a Pyomo model, create a model instance from user-supplied data (in the case of abstract models), apply an optimizer, and summarize the results. For example, the following command line optimizes the `concrete1.py` model using the default LP solver `glpk`:

```
pyomo --solver=glpk concrete1.py
```

Similarly, the following command line optimizes the `abstract5.py` model using data in `abstract5.dat`, also using `glpk`:

```
pyomo --solver=glpk abstract5.py abstract5.dat
```

When the `pyomo` command loads a user-defined Pyomo model, by default it looks for a `ConcreteModel` or `AbstractModel` named `model` in the supplied Python file. We have used this name for all models introduced in this chapter. However, if a name other than `model` is used, this can be specified via the `pyomo` option `--model-name=MODEL_NAME` (e.g., `--model-name=mymodel`).

The `pyomo` command automatically executes the following steps:

1. Create a model
2. Read the instance data (if applicable)
3. Generate a model instance (if the model is abstract)
4. Apply simple preprocessors to the model instance

5. Apply a solver to the model instance
6. Load the results into the model instance
7. Display the solver results

A variety of optional command-line arguments are provided to further guide and provide additional information about the optimization process; documentation of the various available options is available by specifying the `--help` option. Options can control how much or even if debugging information is printed, including logging information generated by the optimizer and a summary of the model generated by Pyomo. Further, Pyomo can be configured to keep all intermediate files used during optimization, which facilitates debugging of the model construction process. See Chapter 7 for further details about this command.

2.5.2 Optimization Scripts

Scripts that control the optimization process provide users with powerful programmability. To introduce the topic, we describe a simple Python script to perform optimization of a Pyomo model instance. Suppose that the model in Example 2.A.1 is stored in the file `concrete1.py`. The script in Example 2.A.8 can then be used to import this model, display the created model, create a solver interface, perform optimization, and display the results. Note that this script does not rely on the `coopr.pyomo` package directly. Pyomo is only used to create an optimization model. Subsequent optimization and analysis of this model is handled in a generic manner with other Coopr packages.

Once a Pyomo model has been created, it can be printed using the `pprint` method:

```
model.pprint()
```

This command summarizes the information in the Pyomo model. For concrete models, this includes the constraint and objective expressions. In abstract models, this information is omitted unless the model object has been constructed with externally supplied data.

Before performing optimization, Pyomo needs to perform various preprocessing steps to collect variables, simplify expressions, and other pre-optimization tasks. Such preprocessing is automatically performed by the `create` method of both the concrete and abstract Pyomo models:

```
instance = model.create()
instance.pprint()
```

For concrete models this method simply returns the model instance, with various annotations performed by the preprocessor. For abstract models, additional arguments specifying the data used to construct the model instance must be supplied. We note that Pyomo's preprocessing capabilities do not currently involve the *presolve* oper-

ations that are commonly employed by industrial AMLs to simplify models before sending them to an optimizer.

Next, we apply an optimizer to find an optimal solution to our model instance. For example, the GLPK [29] linear programming solver can be used within Pyomo as follows:

```
opt = SolverFactory("glpk")
results = opt.solve(instance)
```

The first line in this example creates a Python object to interface to the GLPK solver. The Pyomo model instance is then optimized, with the solver returning an object that contains the solution(s) generated during optimization. This optimization process is executed using components of the `coopr.opt` package, which manage the setup and execution of optimizers. Additionally, Coopr optimization plugins are used to manage the execution of specific optimizers.

Finally, the results of the optimization can be displayed simply by executing the following command:

```
results.write()
```

This output is in the YAML format [62], which is a highly structured data format that is also human readable.

The process for optimizing abstract models (given a data source) is only slightly different. Suppose that the model in Example 2.A.7.1 is stored in the file `abstract5.py`, while the data in Example 2.A.7.2 is stored in the file `abstract5.dat`. The script in Example 2.A.9 imports this model, creates a model instance from the supplied data, creates a solver interface, performs optimization, and displays the results. The primary difference (other than the use of an abstract model) is the specification of a data file, which defines the sets and parameters that are used to construct the model instance.

2.6 Discussion

Perhaps the two Python modeling tools most similar to Pyomo are PuLP [47] and APLEpy [6]. These packages both support the construction of concrete models using Python objects. However, Pyomo is clearly distinguished by its ability to support the definition of abstract models. Although most of this chapter has focused on examples using concrete models, most of the development effort in Pyomo has focused on the mechanisms that support abstract models. In fact, the `ConcreteModel` class is just a specialization of that mechanism to immediately construct model components!

Another distinguishing aspect of Pyomo is the fact that models are complete and self-contained Python objects that a user creates. We believe that this is a feature, since it allows the user to create and managed multiple models simultaneously. However, we have heard users complain about the additional syntax that this re-

quires. In this way, PuLP and APLEpy may be a bit simpler for beginning users, perhaps at the expense of some object-orientation.

In practice, we have observed that users working on abstract models tend to work with the `pyomo` command, while users working on concrete models tend to work with Python scripts. The `pyomo` command is relatively mature, since its interface is well-defined. However, Pyomo's scripting capabilities are still being extended. Although the simple examples shown in this chapter are straightforward, more complex examples that we have developed often rely on features of Coopr and Pyomo that were not intended for general, public use. Consequently, we expect that the scripting capabilities of Coopr and Pyomo will significantly evolve in future releases of this software.

2.A Examples

2.A.1 Concrete Pyomo Model with Explicit Variables

A concrete Pyomo model for Formulation (2.2) using explicit variables.

```
from coopr.pyomo import *

model = ConcreteModel()
model.x_1 = Var(within=NonNegativeReals)
model.x_2 = Var(within=NonNegativeReals)
model.obj = Objective(expr=model.x_1 + 2*model.x_2)
model.con1 = Constraint(expr=3*model.x_1 + 4*model.x_2 >= 1)
model.con2 = Constraint(expr=2*model.x_1 + 5*model.x_2 >= 2)
```

2.A.2 Concrete Pyomo Model with Indexed Variables

A concrete Pyomo model for Formulation (2.2) using indexed variables.

```
from coopr.pyomo import *

model = ConcreteModel()
model.x = Var([1,2], within=NonNegativeReals)
model.obj = Objective(expr=model.x[1] + 2*model.x[2])
model.con1 = Constraint(expr=3*model.x[1] + 4*model.x[2]>=1)
model.con2 = Constraint(expr=2*model.x[1] + 5*model.x[2]>=2)
```

2.A.3 Concrete Pyomo Model with External Data

A concrete Pyomo model for Formulation (2.2) with (a) external data declarations and (b) expressions defined using Python's generator syntax.

```
from coopr.pyomo import *

N = [1,2]
c = {1:1, 2:2}
a = {(1,1):3, (2,1):4, (1,2):2, (2,2):5}
b = {1:1, 2:2}

model = ConcreteModel()
model.x = Var(N, within=NonNegativeReals)
model.obj = Objective(expr=
            sum(c[i]*model.x[i] for i in N))
model.con1 = Constraint(expr=
            sum(a[i,1]*model.x[i] for i in N) >= b[1])
model.con2 = Constraint(expr=
            sum(a[i,2]*model.x[i] for i in N) >= b[2])
```

2.A.4 Concrete Pyomo Model with Constraint Rules

A concrete Pyomo model for Formulation (2.2) using a constraint rule to create constraints `con`.

```
from coopr.pyomo import *

N = [1,2]
M = [1,2]
c = {1:1, 2:2}
a = {(1,1):3, (2,1):4, (1,2):2, (2,2):5}
b = {1:1, 2:2}

model = ConcreteModel()
model.x = Var(N, within=NonNegativeReals)
model.obj = Objective(expr=sum(c[i]*model.x[i] for i in N))

def con_rule(model, m):
   return sum(a[i,m]*model.x[i] for i in N) >= b[m]
model.con = Constraint(M, rule=con_rule)
```

2.A.5 Concrete Pyomo Model with Abstract Component Declarations

A concrete Pyomo model for Formulation (2.2) that uses `rule` and `initialize` arguments for all modeling components.

```
from coopr.pyomo import *


model = ConcreteModel()

def N_rule(model):
   return [1,2]
model.N = Set(rule=N_rule)

model.M = Set(initialize=[1,2])
model.c = Param(model.N, initialize={1:1, 2:2})
model.a = Param(model.N, model.M,
         initialize={(1,1):3, (2,1):4, (1,2):2, (2,2):5})
model.b = Param(model.M, initialize={1:1, 2:2})

model.x = Var(model.N, within=NonNegativeReals)

def obj_rule(model):
   return sum(model.c[i]*model.x[i] for i in model.N)
model.obj = Objective(rule=obj_rule)

def con_rule(model, m):
   return sum(model.a[i,m]*model.x[i] for i in model.N) \
              >= model.b[m]
model.con = Constraint(model.M, rule=con_rule)
```

2.A.6 Using a Function to Construct a Concrete Pyomo Model

A function that creates a concrete Pyomo model for Formulation (2.1) using only data provide in the argument list.

```
from coopr.pyomo import *


def create_model(N=[], M=[], c={}, a={}, b={}):
   model = ConcreteModel()
   model.x = Var(N, within=NonNegativeReals)
   model.obj = Objective(expr=
                  sum(c[i]*model.x[i] for i in N))

   def con_rule(model, m):
      return sum(a[i,m]*model.x[i] for i in N) >= b[m]
   model.con = Constraint(M, rule=con_rule)
   return model

model = create_model(N = [1,2], M = [1,2], c = {1:1, 2:2},
            a = {(1,1):3, (2,1):4, (1,2):2, (2,2):5},
            b = {1:1, 2:2})
```

2.A.7 Abstract Pyomo Model

2.A.7.1 Pyomo Model

An abstract Pyomo model for Formulation (2.1).

```
from coopr.pyomo import *

model = AbstractModel()

model.N = Set()
model.M = Set()
model.c = Param(model.N)
model.a = Param(model.N, model.M)
model.b = Param(model.M)

model.x = Var(model.N, within=NonNegativeReals)

def obj_rule(model):
   return sum(model.c[i]*model.x[i] for i in model.N)
model.obj = Objective(rule=obj_rule)

def con_rule(model, m):
   return sum(model.a[i,m]*model.x[i] for i in model.N) \
               >= model.b[m]
model.con = Constraint(model.M, rule=con_rule)
```

2.A.7.2 Data Commands

Data commands for the abstract Pyomo model in Example 2.A.7.1 that are used to generate the concrete model in Formulation (2.2).

```
set N := 1 2 ;

set M := 1 2 ;

param c :=
1 1
2 2 ;

param a :=
1 1 3
2 1 4
1 2 2
2 2 5 ;

param b :=
1 1
2 2 ;
```

2.A.8 A Python Script that Optimizes a Concrete Pyomo Model

A Python script that creates the concrete Pyomo model in Example 2.A.1 and performs optimization using the GLPK linear programming solver.

```
from coopr.opt import SolverFactory
from concrete1 import model

model.pprint()

instance = model.create()
instance.pprint()

opt = SolverFactory("glpk")
results = opt.solve(instance)

results.write()
```

2.A.9 A Python Script that Optimizes an Abstract Pyomo Model

A Python script that creates a model instance from the abstract Pyomo model in Example 2.A.7.1 and performs optimization using the GLPK linear programming solver.

```
from coopr.opt import SolverFactory
from abstract5 import model

model.pprint()

instance = model.create('abstract5.dat')
instance.pprint()

opt = SolverFactory("glpk")
results = opt.solve(instance)

results.write()
```

http://www.springer.com/978-1-4614-3225-8

Pyomo – Optimization Modeling in Python

Hart, W.E.; Laird, C.; Watson, J.-P.; Woodruff, D.L.

2012, XVIII, 238 p., Hardcover
ISBN: 978-1-4614-3225-8