

# Pyomo Developer Guide 3.1

**COLLABORATORS**

	<i>TITLE :</i> Pyomo Developer Guide 3.1		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	William E. Hart	July 15, 2016	

# Contents

<b>1</b>	<b>Overview</b>	<b>1</b>
<b>2</b>	<b>Porting Pyomo to Python 3.x</b>	<b>2</b>
2.1	Different Types	2
2.2	Printing	2
2.3	Exception Management	3
2.4	Importing	3
2.5	Managing API Changes with <code>six</code>	4
<b>3</b>	<b>Pyomo Library API</b>	<b>5</b>
3.1	Introduction	5
3.2	Declaring Pyomo Functors	5
3.3	The Functor Registry	7
3.4	Functors and Workflow	8
3.5	API Notes	8
<b>4</b>	<b>Pyomo Functor API</b>	<b>10</b>
4.1	<code>pyomo.model</code> Functors	10
4.1.1	<code>simple_preprocessor</code>	10
4.2	<code>pyomo.repn</code> Functors	10
4.2.1	<code>compute_ampl_repn</code>	10
4.2.2	<code>compute_canonical_repn</code>	11
4.3	<code>pyomo.script</code> Functors	11
4.3.1	<code>apply_optimizer</code>	11
4.3.2	<code>apply_postprocessing</code>	11
4.3.3	<code>apply_preprocessing</code>	12
4.3.4	<code>create_model</code>	12
4.3.5	<code>finalize</code>	12
4.3.6	<code>process_results</code>	13
4.3.7	<code>run_command</code>	13
4.3.8	<code>setup_environment</code>	14

---

- 5 How To Run Tests** **15**
- 5.1 Running Smoke Tests . . . . . 15
- 5.2 Interpreting Output . . . . . 15

# Chapter 1

## Overview

This overview chapter outlines the book.

---

## Chapter 2

# Porting Pyomo to Python 3.x

We have begun the process of porting Pyomo to support Python 3.x. Specifically, we are working on support for Python 3.2 and 3.3; earlier version of Python 3.x were a bit buggy.

We are currently trying to adapt Pyomo to simultaneously support both Python 2.x and Python 3.x without requiring the use of conversion scripts (e.g. 2to3). This simplifies the management of the code base, though at the price of some additional complexity when developing the code.

The following subsections describe various issues that developers need to consider while writing code to ensure portability. Further details are available in the following online references:

- [Porting to Python 3](#)
- [Python 3.4 Porting Guide](#)
- [Porting Mock to Python 3](#)
- [The `six` Python Package](#)

## 2.1 Different Types

Python 3 changes some of the fundamental types in Python. The following code provides cross-compatibility for these changes:

```
try:
    unicode
except NameError:
    # Python 3
    basestring = unicode = str
    long = int
```

## 2.2 Printing

The `print` statement is a function in Python 3. However, Python 2.6 supports this function, so simple print commands can be ported easily:

```
# print "Python 2 syntax"
print("Python 3 syntax")
```

However, more advanced features of the `print` function are not supported in Python 3. A common issue with printing is the redirection syntax:

```
OUTPUT = open('filename', 'w')
print >>OUTPUT, "This works in Python 2", "More text"
OUTPUT.close()
```

In Python 3, this code needs to be changed to use `sys.stdout.write()`:

```
OUTPUT = open('filename', 'w')
sys.stdout.write("This works in Python 2"+" "+"More text"+"\\n")
OUTPUT.close()
```

Note that (a) the comma used in Python 2 needs to be replaced with an explicit space in Python 3, and (b) the `write()` function needs to have an explicit end-of-line character provided. (Note that the `six.print_` function can also be used for Python 2/3 portability, though this function does work exactly like the Python 3 `print` function.)

## 2.3 Exception Management

Python 3 introduces a new syntax of exceptions:

```
try:
    raise Exception()
except Exception as exc:
    # Current exception is 'exc'
    pass
```

The Python documentation claims that this syntax should work in Python 2.6 or newer. However, it is unclear that early versions of Python 2.6 supported this syntax (e.g. 2.6.1). The following syntax is guaranteed to work with both Python 2.x and 3.x:

```
try:
    raise Exception()
except Exception:
    exc = sys.exc_info()[1]
    # Current exception is 'exc'
    pass
```

## 2.4 Importing

The following import is invalid syntax in Python 3:

```
def f():
    from module import *
```

Additionally, Python 3 treats all forms of imports not starting with `.` as absolute imports. Thus, Pyomo packages use absolute imports to ensure portability between Python 2/3. This requires explicit importing of all levels of Pyomo. For example, in `pyomo.core/base`, the `numvalue.py` file uses the import:

```
from pyomo.core.base.set_types import Reals, Any
```

instead of the following, which works only in Python 2:

```
from set_types import Reals, Any
```

## 2.5 Managing API Changes with `six`

The `six` package provides a lot of API changes that make it very easy to translate between Python 2 and 3. One key utility of `six` is the translation of package names. For example, the `StringIO` class is now in the `io` module rather than the `StringIO` module. The following syntax is portable between Python 2/3:

```
from six.moves import StringIO

stringio = StringIO()
```

Similarly, the `xrange` function is no longer supported in Python 3 (since the `range` function now returns a generator). The following import ensures that `xrange` can be used portably in Python 2 and 3:

```
from six.moves import xrange
```

Another key feature of `six` is the inclusion of functions that portably manage iteration between Python 2 and 3. Specifically, the API of dictionaries changed substantially between Python 2 and 3. The `dict.iteritems()` function is not available in Python 3, and instead `dict.items()` returns a generator. This makes it difficult to have efficient code supported in all versions of Python, so `six.iteritems()` is provided to provide that portability:

```
d = {1:'one', 2:'two'}
for k, v in six.iteritems(d):
    print("%d %s" % (k,v))
```

Similar functions are provide for `iterkeys` and for advancing iterators (`next`).

## Chapter 3

# Pyomo Library API

### 3.1 Introduction

This chapter describes the API of Pyomo's library and functions and classes. This API is focused on the functions and classes that support the transformation and analysis of Pyomo models.

A key aspect of Pyomo's library are functors, which are function objects that can be executed like other functions. Pyomo's functors provide a unifying interface for Pyomo's API, and they support the following functionality:

- Functor registration standardizes function definition, which facilitates plug-and-play of the functions in the API.
- The functor docstrings are parsed to support error checking for functor inputs and outputs.
- Functors are globally registered, which allows functors to be created on the fly and enables documentation/enumeration of all functors in the API.
- Functors can be integrated into formal workflows (using components from `pyutilib.workflow`)

Thus, functor declarations allow the Pyomo API to be more than a simple library of function calls.

### 3.2 Declaring Pyomo Functors

Functors in the Pyomo API are declared using the `pyomo_api` decorator with a Python function. The following example illustrates the use of this decorator:

```
@pyomo_api
def f1(data, x=0, y=1):
    """A simple example.

    Required:
        x: A required keyword argument

    Optional:
        y: An optional keyword argument

    Return:
        a: A return value
        b: Another return value
    """
    return PyomoAPIData(a=2*data.z, b=x+y)
```

The `f1` function is a normal Python declaration, and the `pyomo_api` decorator transforms this function into a functor. This functor can be executed as if it was a function. For example:

```
data = PyomoAPIData(z=1)
val = f1(data, x=2)
```

Pyomo functors are required to have an argument that is a container of labeled data, which is treated specially. The container is required to be a `dict` or `PyomoAPIData` class. The `PyomoAPIData` class generalizes the Python `dict` class in several ways. Most notably, an attribute added to an `PyomoAPIData` object is also added to the underlying dictionary. In the previous example, the `PyomoAPIData` object is passed in as the first argument. In fact, Pyomo functors are only allowed to have one non-keyword argument. Alternatively, a functor can be declared with a `data` keyword argument, in which case it has no non-keyword arguments. For example, the functor can be declared as follows

```
@pyomo_api
def f2(x=0, y=1, data=None):
    """A simple example.

    Required:
        data: The required data argument
        x: A required keyword argument

    Optional:
        y: An optional keyword argument

    Return:
        a: A return value
        b: Another return value
    """
    return PyomoAPIData(a=2*data.z, b=x+y)
```

and it is evaluated as follows

```
data = PyomoAPIData(z=1)
val = f2(data=data, x=2)
```

In fact, this functor can also be evaluated as before:

```
data = PyomoAPIData(z=1)
val = f1(data, x=2)
```

Although this creates a syntactic difference between the declaration and formulation of functors, this allows functors to be used with a common API.

If a `dict` is passed into a functor to provide the container of labeled data, then the functor converts it to a `PyomoAPIData` object before executing the function. Since `PyomoAPIData` objects are subclasses of `dict`, this change may be transparent to the user. However, this is important in contexts where features of `PyomoAPIData` are used.

The return value of a Pyomo functor is an `PyomoAPIData` object. However, the return value of the function used to declare the constructor may be either `None`, a `dict` object, or an `PyomoAPIData` object. If the function returns `None` or the `data` object is returned, then the functor creates an `PyomoAPIData` object with an element with key `data` whose value is the `PyomoAPIData` object passed into the functor. Otherwise, if an `PyomoAPIData` object is returned then the functor adds an element with key `data` if it does not already exist. If a `dict` object is returned, then it is converted to an `PyomoAPIData` object and processed in the same manner. Consequently, the return value of a Pyomo function is an `PyomoAPIData` object that is guaranteed to contain a container of labeled data with key `data`.

The docstring comments used in these examples are needed to fully specify the API of the Pyomo functor. These docstrings are needed to properly execute Pyomo functors. The `Required`, `PyomoAPIData`, and `Return` keywords declare blocks where keyword arguments and return values are described. Although all keywords are declared with a default value, these values are only used in a functor for the optional arguments. An exception is generated if a required argument to a functor is omitted. The labeled return values specify the possible outputs of a functor. An exception is generated if a unexpected label for a return value is specified. When a functor returns without defining a defined return label, then its value is `None`.

The `Required` block can also be used to validate the existence of data that is nested inside of the functor arguments. Consider the following example, which validates the existence of values in the `data` and `x` arguments:

```
@pyomo_api
def f3(data, x=None):
    """A simple example.

    Required:
        data.z: A nested required data value
        x: A required keyword argument
        x.y: A nested required data value

    Return:
        a: A return value
        b: Another return value
    """
    return PyomoAPIData(a=2*data.z, b=x.y)
```

Note that the nested values are assumed to be simple nested attributes of the form `a.b.c.d`. General purpose tests are not supported for checking the validity of data, and the test for a nested value simply verifies that it exists and that it is not equal to `None`.

These requirements on functors enforce a uniform API for the input and output values. Input values consist of a container of labeled data along with keyword arguments, and output values have the same form. This consistency facilitates the use of functors in a larger computational workflows. The incorporation of the container object into the Pyomo functor API allows keyword arguments to be added in an extensible manner. For example, this feature enables the incorporation of data that is used by subsequent functors in the computation without requiring an extension of a the APIs of the preceding functors.

Note: The `PyomoAPIData` class supports a container for labeled data that generalizes the dictionary used for Python nonformal keyword arguments, which are specified with the syntax `**kwd`. Nonformal keyword arguments are used in the APIs of other Python packages (e.g. `Matplotlib`). The use of `PyomoAPIData` is motivated by the use of functors in formal computational workflows.

### 3.3 The Functor Registry

Declarations of Pyomo functors automatically populate a global registry of the Pyomo API. This registry allows functor objects to be *created* on the fly. For example:

```
g = PyomoAPIFactory('f1')
```

This example illustrates how the functor object `g` can be created from the registered functor `f1`. The functor `g` acts exactly like `f1`, and in practice there is little difference between using `g` and `f1`. However, functors can be created by name using the factory `PyomoAPIFactory`, and thus the user does not need to know the specific Pyomo library in which a functor is defined.

To help organize functors, a `namespace` option can be specified when declaring the functor. This allows functors to be defined with the same name in different packages, while distinguishing how they are registered. For example:

```
@pyomo_api(namespace='utility')
def f1(data, x=0, y=1):
    """A simple example.

    Required:
        x: A required keyword argument

    Optional:
        y: An optional keyword argument

    Return:
        a: A return value
        b: Another return value
    """
```

```
"""
return PyomoAPIData(a=2*data.z, b=x+y)
```

The functor `f1` is declared within the `utility` namespace. The `f1` object can be used within the python module containing this declaration. However, this functor is registered as `utility.f1` in the registry, so the functor is created as follows:

```
g = PyomoAPIFactory('utility.f1')
```

Finally, the functor registry allows for the automatic generation of documentation for the Pyomo API. The `pyomo` command supports the `api` subcommand, which generates a simple summary of all functor namespaces and their corresponding functors. This output looks something like the following:

```
=====
Pyomo Functor API
=====

-----
pyomo.script Functors
-----

apply_optimizer:
    Perform optimization with a concrete instance

apply_postprocessing:
    Apply post-processing steps.

apply_preprocessing:
    Execute preprocessing files

create_model:
    Create instance of Pyomo model.

finalize:
    Perform final actions to finish the execution of the pyomo script.
```

Additionally, the `--asciidoc` option can be specified to generate a detailed description of the Pyomo API, which is used to generate this document (see below).

## 3.4 Functors and Workflow

TODO: Pyomo functors can be tied together into formal workflows that can be executed in an arbitrary manner

## 3.5 API Notes

The following notes describe further action items that we've outlined for the Pyomo API:

- Tasks/WF objects
  - Parallelization
  - Reuse?
  - Push/Pull input/output ports
  - Pre/Post events
    - \* Global
    - \* Task-specific (extension point or events?)
  - Tasks support extension points

\* Use *implements()* to define the EP that is supported

- API code review
  - Document code review along-side API functions?
- Case studies (and tests)
- Pyomo model transformations
  - In-place vs. copies
  - Need to clarify semantics
  - Active/Frozen models
  - Flattening structured models
    - \* Utilities for walking a non-flattened model (iterator?)
  - EP's supported by new components (?)
- Script generation
  - Record sequence of calls
  - Tracing logic OR auto-scripting

("// Generated with *pyomo api* on ", datetime.date(2016, 7, 15))

## Chapter 4

# Pyomo Functor API

### 4.1 `pyomo.model` Functors

#### 4.1.1 `simple_preprocessor`

This plugin simply applies preprocess actions in a fixed order.

- Required Keyword Arguments:

**data**

A container of labeled data.

**model**

A concrete model instance.

- Return Values:

**data**

A container of labeled data.

### 4.2 `pyomo.repn` Functors

#### 4.2.1 `compute_ampl_repn`

This plugin computes the ampl representation for all objectives and constraints. All results are stored in a ComponentMap named "`_ampl_repn`" at the block level.

We break out preprocessing of the objectives and constraints in order to avoid redundant and unnecessary work, specifically in contexts where a model is iteratively solved and modified. we don't have finer-grained resolution, but we could easily pass in a Constraint and an Objective if warranted.

- Required Keyword Arguments:

**data**

A container of labeled data.

**model**

A concrete model instance.

- Return Values:

**data**

A container of labeled data.

---

## 4.2.2 compute\_canonical\_repn

This plugin computes the canonical representation for all objectives and constraints linear terms. All results are stored in a ComponentMap named "\_canonical\_repn" at the block level.

We break out preprocessing of the objectives and constraints in order to avoid redundant and unnecessary work, specifically in contexts where a model is iteratively solved and modified. we don't have finer-grained resolution, but we could easily pass in a Constraint and an Objective if warranted.

- Required Keyword Arguments:

**data**

A container of labeled data.

**model**

A concrete model instance.

- Return Values:

**data**

A container of labeled data.

## 4.3 pyomo.script Functors

### 4.3.1 apply\_optimizer

Perform optimization with a concrete instance

- Required Keyword Arguments:

**data**

A container of labeled data.

**instance**

Problem instance.

- Return Values:

**data**

A container of labeled data.

**opt**

Optimizer object.

**results**

Optimization results.

### 4.3.2 apply\_postprocessing

Apply post-processing steps.

- Required Keyword Arguments:

**data**

A container of labeled data.

**instance**

Problem instance.

**results**

Optimization results object.

- Return Values:

**data**

A container of labeled data.

---

### 4.3.3 `apply_preprocessing`

Execute preprocessing files

- Required Keyword Arguments:

**data**

A container of labeled data.

**parser**

Command line parser object

- Return Values:

**data**

A container of labeled data.

**error**

This is true if an error has occurred.

### 4.3.4 `create_model`

Create instance of Pyomo model.

- Required Keyword Arguments:

**data**

A container of labeled data.

- Return Values:

**data**

A container of labeled data.

**filename**

Filename that a model instance was written to.

**instance**

Problem instance.

**model**

Model object.

**symbol\_map**

Symbol map created when writing model to a file.

### 4.3.5 `finalize`

Perform final actions to finish the execution of the pyomo script.

This function prints statistics related to the execution of the pyomo script. Additionally, this function will drop into the python interpreter if the `interactive` option is `True`.

- Required Keyword Arguments:

**data**

A container of labeled data.

**model**

A pyomo model object.

- Optional Keyword Arguments:
-

**instance**

A problem instance derived from the model object.

**results**

Optimization results object.

- Return Values:

**data**

A container of labeled data.

**4.3.6 process\_results**

Process optimization results.

- Required Keyword Arguments:

**data**

A container of labeled data.

**instance**

Problem instance.

**opt**

Optimizer object.

**results**

Optimization results object.

- Return Values:

**data**

A container of labeled data.

**4.3.7 run\_command**

Execute a function that processes command-line arguments and then calls a command-line driver.

This function provides a generic facility for executing a command function is rather generic. This function is segregated from the driver to enable profiling of the command-line execution.

- Required Keyword Arguments:

**command**

The name of a function that will be executed to perform process the command-line options with a parser object.

**parser**

The parser object that is used by the command-line function.

- Optional Keyword Arguments:

**args**

Command-line arguments that are parsed. If this value is `None`, then the arguments in `sys.argv` are used to parse the command-line.

**data**

A container of labeled data.

**name**

Specifying the name of the command-line (for error messages).

**options**

If this is not `None`, then ignore the `args` option and use this to specify command options.

- Return Values:

**data**

A container of labeled data.

**errorcode**

0 if Pyomo ran successfully

**retval**

Return values from the command-line execution.

### 4.3.8 setup\_environment

Setup Pyomo execution environment

- Required Keyword Arguments:

**data**

A container of labeled data.

- Return Values:

**data**

A container of labeled data.

---

## Chapter 5

# How To Run Tests

### 5.1 Running Smoke Tests

These commands run the Python Nose utility (`nosetests`) for each package within the named argument.

To test all of pyomo, use the `test.pyomo` script. From the `pyomo` directory, use the command:

```
bin/test.pyomo
```

Or from any sub-directory of `pyomo`, use

```
lbin test.pyomo
```

To test `pyutilib`, use

```
bin/test.pyutilib
```

To test just `pyomo`, from any `pyomo` sub-directory, use

```
lbin test.pyomo pyomo.core
```

There are options for tests (e.g., `--cat=all`), to see a list of options use the `-h` or `--help` option.

### 5.2 Interpreting Output

There are two modes of "not-pass:" `FAIL` and `ERROR`. `FAIL` is an assertion failure. `ERROR` is a crash.