

# Coopr Library API 3.1

COLLABORATORS			
	TITLE : Coopr Library API 3.1		
ACTION	NAME	DATE	SIGNATURE
WRITTEN BY		January 28, 2012	

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Declaring Coopr Functors</b>	<b>1</b>
<b>3</b>	<b>The Functor Registry</b>	<b>3</b>
<b>4</b>	<b>Functors and Workflow</b>	<b>4</b>
<b>5</b>	<b>Coopr Functor API</b>	<b>4</b>
5.1	pyomo.script Functors . . . . .	4

---

## 1 Introduction

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

A key aspect of Coopr's library are functors, which are function objects that can be executed like other functions. Coopr's functors provide a unifying interface for Coopr'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 Coopr API to be more than a simple library of function calls.

## 2 Declaring Coopr Functors

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

```
@coopr_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 CooprAPIData(a=2*data.z, b=x+y)
```

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

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

Coopr 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 `CooprAPIData` class. The `CooprAPIData` class generalizes the Python `dict` class in several ways. Most notably, an attribute added to an `CooprAPIData` object is also added to the underlying dictionary. In the previous example, the `CooprAPIData` object is passed in as the first argument. In fact, Coopr 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

```
@coopr_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 CooprAPIData(a=2*data.z, b=x+y)

```

and it is evaluated as follows

```

data = CooprAPIData(z=1)
val = f2(data=data, x=2)

```

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

```

data = CooprAPIData(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 `CooprAPIData` object before executing the function. Since `CooprAPIData` objects are subclasses of `dict`, this change may be transparent to the user. However, this is important in contexts where features of `CooprAPIData` are used.

The return value of a Coopr functor is an `CooprAPIData` object. However, the return value of the function used to declare the constructor may be either `None`, a `dict` object, or an `CooprAPIData` object. If the function returns `None` or the data object is returned, then the functor creates an `CooprAPIData` object with an element with key `data` whose value is the `CooprAPIData` object passed into the functor. Otherwise, if an `CooprAPIData` 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 `CooprAPIData` object and processed in the same manner. Consequently, the return value of a Coopr function is an `CooprAPIData` 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 Coopr functor. These docstrings are needed to properly execute Coopr functors. The `Required`, `CooprAPIData`, 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:

```

@coopr_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 CooprAPIData(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 Coopr 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 `CooprAPIData` 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 `CooprAPIData` is motivated by the use of functors in formal computational workflows.

### 3 The Functor Registry

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

```
g = CooprAPIFactory('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 `CooprAPIFactory`, and thus the user does not need to know the specific Coopr 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:

```
@coopr_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 CooprAPIData(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 = CooprAPIFactory('utility.f1')
```

Finally, the functor registry allows for the automatic generation of documentation for the Coopr API. The `coopr` 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:

```
=====
Coopr 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 Coopr API, which is used to generate this document (see below).

## 4 Functors and Workflow

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

## 5 Coopr Functor API

### 5.1 pyomo.script Functors

#### 5.1.1 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.

#### 5.1.2 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.

### 5.1.3 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.

### 5.1.4 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.

### 5.1.5 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.

### 5.1.6 print\_components

Print information about modeling components supported by Pyomo.

- Required Keyword Arguments:

**data**

A container of labeled data.

- Return Values:

**data**

A container of labeled data.

### 5.1.7 print\_solver\_help

Print information about the solvers that are available.

- Required Keyword Arguments:

**data**

A container of labeled data.

- Return Values:

**data**

A container of labeled data.

---

### 5.1.8 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.

### 5.1.9 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).

- Return Values:

**data**

A container of labeled data.

**retval**

Return values from the command-line execution.

#### 5.1.10 `setup_environment`

Setup Pyomo execution environment

- Required Keyword Arguments:

**data**

A container of labeled data.

- Return Values:

**data**

A container of labeled data.

---