Coopr User Manual: Customizing Coopr with Plugins

William E. Hart[1] John Siirola[2] Jean-Paul Watson[3]

February 15, 2013

[1]Sandia National Laboratories, Discrete Math and Complex Systems Department, PO Box 5800, Albuquerque, NM 87185; wehart@sandia.gov

[2]TBD

[3]Sandia National Laboratories, Discrete Math and Complex Systems Department, PO Box 5800, Albuquerque, NM 87185; jwatson@sandia.gov

Contents

Chapter 1

Introduction

Coopr leverages the PyUtilib Component Architecture (PCA) to support plugins that extend Coopr's built-in functionality. PCA supports an object oriented approach to software design, which is an accepted strategy for managing software complexity in large systems. Object oriented design is traditionally done using classes and class inheritance; classes define interfaces, which are extended and customized in subclasses using class inheritance.

The main idea behind PCA is to separate the declaration of component interfaces from their implementation. This allows for a more flexible design of software components that further encourages modularity of components. Additionally, PCA includes a global component registry, as well as a framework for automating the execution of components that match a given interface. This capability facilitates the dynamic registration and application of components within large software systems.

The plugin components supported by Coopr have a variety of signficant impacts on the development and deployment of Coopr applications:

- Plugins facilitate extensions of Coopr without risk of destabilizing core functionality.
- Plugins allow third-party developers to add value without requiring direct involvement of the core developers. For example, Pyomo extensions can be developed and distributed without requiring developer access to Coopr's subversion repository. Similarly, third-party developers can develop plugins that are specific to their working environment or business needs.
- The PCA can automate the activation of external software interfaces, based on the user's environment. For example, optimization solvers can be automatically registered if they are found with the user's `PATH` environment.
- New capabilities can be dynamically loaded at run-time. Python EGG files provide a standard, modular mechanism for distributing plugins. The PCA can dynamically load EGG files, which allows users to dynamically activate plugins.

Chapter 2 provides a detailed description of the PyUtilib Component Architecture. Chapter 3 descripts the plugin interfaces that are supported in Coopr. Further, this chapter provides examples for how these plugins can be used to augment the Coopr's core functionality.

Chapter 2

PyUtilib Component Architecture

The PyUtilib Component Architecture (PCA) is an extension of the Trac plugin framework [8] that is included in the PyUtilib software package [5]. There does not appear to be a standard Python plugin framework, though there are some mature packages that support plugins including Zope [10], Envisage [3], Trac [8], yapsy [9] and SprinklesPy [6]. Although we discuss the design requirements for PCA later, PCA was initially motivated by the need to have a Trac-like plugin framework that was self-contained. The core of PCA is provided by PyUtilib's `pyutilib.plugin.core` module.

2.1 Overview

The PCA is comprised of a small set of Python classes. A *plugin* is a class that implements a set of related methods the context of the application, and a *service* is an instance of a plugin class. Two different types of plugins are available: singleton and non-singleton plugins. There is at most one service for a singleton plugin, whereas there can be multiple services of non-singleton plugins.

A software application can declare *extension points* that other components can *plug in* to. This mechanisms supports a flexible, modular programming paradigm that enables software applications to be extended in a dynamic manner. Extension points and the extensions contributed to them are stored in a global registry, and execution of these extensions is handled in a standardized manner. Thus, an application developer can define extension points without knowing how they will be implemented, and extension developers can register extensions without needing to know the details of how they are employed.

Extension points are defined with respect to an *interface* class, which defines the type that plugins use to register their capabilies. A plugin class includes declarations that denote that it implements one-or-more interfaces. An interface is defined by the methods and data that are used. However, the PCA does not enforce this interface or support interface conversions (see Zope [10] and Envisage [3] for examples of plugin frameworks that support this functionality).

2.2 Core Plugin Classes

The PyUtilib plugin framework consists of the following core classes:

pyutilib.plugin.core.Interface Subclasses of this class declare plugin interfaces that are registered in the framework

pyutilib.plugin.core.ExtensionPoint A class used to declare extension points, which can access services with a particular interface

pyutilib.plugin.core.Plugin Subclasses of this class declare plugins, which can be used to implement interfaces within this plugin framework.

pyutilib.plugin.core.SingletonPlugin Subclasses of this class declare singleton plugins, for which a single service is constructed.

pyutilib.plugin.core.PluginEnvironment A class that maintains the registries for interfaces, extension points, plugins and services.

pyutilib.plugin.core.PluginGlobals A class that maintains global data concerning the set of environments that are currently being used.

pyutilib.plugin.core.PluginError The exception class that is raised when errors arise in this framework.

2.2.1 Interfaces and Extension Points

A subclass of the `Interface` class is used to declare extension points in an application. The `ExtensionPoint` class is used to declare an extension point and to retrieve information about the plugins that implement the specified interface. For example, the following is a minimal declaration of an interface and extension point:

```
class MyInterface(Interface):
    """An interface subclass"""

ep = ExtensionPoint(MyInterface)
```

Note that the `MyInterface` class is not required to define the API of the interface. The PCA does not enforce checking of API conformance for plugins, and hence any declaration in the `MyInterface` class would be ignored. Additionally, note that an instance of `MyInterface` is not created; the `MyInterface` class is simply used to declare a type that is used to index related plugins.

An instance of `ExtensionPoint` can be used to iterate through all extensions, or to search for an extension that matches a particular keyword. For example, the following code iterates through all extensions and applies the `pprint` method:

```
for service in ep:
  service.pprint()
```

If you wish to know the number of services that are registered with an extension point, you can call the standard `len` function:

```
print len(ep)
```

Several other methods can be used to more carefully select services from an extension point. The `extensions` method returns a Python `set` object that contains the services:

```
#
# This loop iterates over all services, just the same as when an
# the iterator method is used (see above).
#
for service in ep.extensions():
  service.pprint()

#
# This loop iterates over all services, including the 'disabled'
# services.
#
for service in ep.extensions(all=True):
  service.pprint()
```

This example illustrates the optional argument that indicates whether the set returned by `extensions` includes all disabled services. It is convenient to explicitly support enabling and disabling services in many applications, though services are enabled by default. Disabled services remain in the registry, but by default they are not included in the set returned by an extension point.

Finally, the PCA can support *named services*, which requires that the services have a `name` attribute. Service names are not required to be unique. For example, when multiple instances of a non-singleton plugin are created, then these services can be accessed as follows:

```
#
# A simple plugin that implements the MyInterface interface
#
class MyPlugin(Plugin):
  implements(MyInterface)

  def __init__(self):
      self.name="myname"

#
```

```
# Another simple plugin that implements the MyInterface interface
#
class MyOtherPlugin(Plugin):
  implements(MyInterface)

  def __init__(self):
      self.name="myothername"

#
# Constructing services
#
service1 = MyPlugin()
service2 = MyPlugin()
service3 = MyOtherPlugin()

#
# A function that iterates over all MyInterface services, and
# returns the MyPlugin instances (which are service1 and service2).
#
def get_services():
    ep = ExtensionPoint(MyInterface)
    return ep("myname")
```

In some applications, there is a one-to-one correspondence between service names and their instances. In this context, a simpler syntax is to use the `service` method:

```
class MySingletonPlugin(SingletonPlugin):
  implements(MyInterface)

  def __init__(self):
      self.name="mysingletonname"

ep = ExtensionPoint(MyInterface)
ep.service("mysingletonname").pprint()
```

The `service` method raises a `PluginError` if there is more than one service with a given name. Note, however, that this method returns `None` if no service has been registered with the specified name.

Note that an integer cannot be used to select a service from an extension point. Services are not registered in an indexable array, so this option does not make sense.

2.2.2 Plugins and Environments

PCA plugins are subclasses of either the `Plugin` or `SingletonPlugin` classes. Subclasses of `Plugin` need to be explicitly constructed, but otherwise they do not need to be registered; simply constructing a subclass of `Plugin` invokes the registration of that instance. Similarly, simply declaring a subclass of `SingletonPlugin` invokes both the construction and registration of this plugin service.

PCA plugins are registered with different interfaces using the `implements` function (which is a static method of `Plugin`). Note that a plugin can be registered with more than one interface. Plugins are applied to different extension points independently, but they can maintain state information that impacts their use across different extension points.

The plugin and interfaces are organized within namespaces using the `PluginEnvironment` class. A global registry of plugin environments is maintained by the `PluginGlobals` class. This registry is a stack of environments, and the top of this stack defines the current environment. When an interface is declared, its namespace is the name of the current environment. For example:

```
#
# Declare an interface in the current environment
#
class Interface1(Interface):
    pass

#
# Set the current environment to 'new_environ'
#
PluginGlobals.push_env( "new_environ" )

#
# Declare an interface in the 'new_environ' environment
#

class Interface2(Interface):
    pass

#
# Go back to the original environment
#
PluginGlobals.pop_env()
```

The namespace that an `Interface` subclass is declared in defines the namespace where plugin services will be registered. Additionally, a plugin service will be registered in the namespace where it is declared. For example:

```
#
# Declare Interface1 in namespace env1
#
PluginGlobals.push_env("env1")

class Interface1(Interface):
    pass

#
# Declare Interface2 in namespace env2
#
PluginGlobals.push_env("env2")

class Interface2(Interface):
    pass

PluginGlobals.pop_env()

#
# Declare Plugin1 in namespace env3
#
PluginGlobals.push_env("env3")

class Plugin1(Plugin):

    implements(Interface1)
    implements(Interface2)
    implements(Interface1,"env4")

PluginGlobals.pop_env()
```

When `Plugin1` is instantiated, it's services are registered in the following environments:

```
    env1 for Interface1
    env2 for Interface2
    env4 for Interface1
    env3
```

The last registration is the default. A plugin service is always registered in the environment in which it was declared.

Namespaces provide a mechanism for organizing plugin services in an extensible manner. Applications can define new namespaces that contain their services without worrying about conflicts with services defined in other Python libraries.

2.2.3 Global Plugin Data

Global plugin data in PCA is managed in the `PluginGlobals` class. This class contains a variety of static methods that are used to access this data:

default_env This method returns the default environment, which is constructed when the plugins framework is loaded.

env This method returns the current environment if no argument is specified. Otherwise, it returns the specified environment.

push_env,pop_env These methods respectively push a new environment onto the environment stack and pop the current environment from the stack.

services This method returns the plugin services in the current environment (or the named environment if one is specified).

load_services Load services using IPluginLoader extension points.

pprint This method provides a text summary of the registered interfaces, plugins and services.

clear This method empties the environment stack and defines a new default environment. This setup then bootstraps the configuration of the `pyutilib.plugin.core` environment. Note that this does not clear the plugin registry; in practice that may not make sense since it is not easy to reload modules in Python.

2.3 A Simple Example

Figure 2.1 provides a simple example that is adapted from the description of the Trac component architecture [7]. This example illustrates the three main steps to setting up a plugin:

1. Defining an interface

2. Declaring extension points

3. Defining classes that implement the interface.

In this example, a singleton plugin is declared, which automatically registers the plugin service. Non-singleton plugin services need to explicitly created, but they are also automatically registered.

If the script in Figure 2.1 is in the `todo.py` file, then the following Python script illustrates how this plugin is used:

```
from todo import *

# Construct a TodoList object and then add several items.
todo_list = TodoList()
todo_list.add('Make coffee', 'Really need to make some coffee')
todo_list.add('Bug triage',
        'Double-check that all known issues were addressed')
```

This script generates the following output:

```
Task: Make coffee
      Really need to make some coffee
Task: Bug triage
      Double-check that all known issues were addressed
```

2.4 Plugin Implementations

In addition to the core plugin framework, PCA includes implementations for a variety of plugins that support commonly used functionality. The following sections briefly describe these plugins.

2.4.1 Options and Configuration Files

The `pyutilib.plugin.config` package defines interfaces and plugins for managing service options. The `Configuration` service is used to manage the global configuration of all services. This class coordinates with `Option` services. Plugins can declare options with the `declare_option` method, which registers these options with the `Configuration` service. This service reads and writes options to configuration files (using Python's `ConfigParser` package).

This package also declares the `ManagedPlugin` and `ManagedSingletonPlugin` classes, which are plugin base classes that include options that can be used to enable or disable services using the `Configuration` service. In practice, most plugins will be derived from these plugin base classes.

2.4.2 Plugin Loaders

PCA plugins can be loaded from either Python modules or Python eggs. This capability supports the runtime extension of the plugins framework, which has proven very powerful in frameworks like Trac. The core plugin framework defines extension points that use these loaders, which can be applied as follows:

```
import sys
```

```
# A simple example that manages a TODO list.  An observer
# interface is used to add actions that occur when a TODO
# item is added.
from pyutilib.plugin.core import *

# An interface class that defines the API for plugins that
# observe when a TODO item is added.
class ITodoObserver(Interface):

        def todo_added(name, description):
            """Called when a to-do item is added."""

# The TODO application, which declares an extension point
# for observers.  Observers are notified when a new TODO
# item is added to the TODO list.
class TodoList(object):
        observers = ExtensionPoint(ITodoObserver)

        def __init__(self):
            """
            The TodoList constructor, which initializes the list
            """
            self.todos = {}

        def add(self, name, description):
            """Add a TODO, and notify the observers"""
            assert not name in self.todos, 'To-do already in list'
            self.todos[name] = description
            for observer in self.observers:
                observer.todo_added(name, description)

# A plugin that prints information about TODO items when they
# are added.
class TodoPrinter(SingletonPlugin):
        implements(ITodoObserver)

        def todo_added(self, name, description):
            print 'Task:', name
            print '     ', description
```

Figure 2.1: A simple example of the Python Component Architecture

Trac Class Name	PyUtilib Class Name
Interface	Interface
ExtensionPoint	ExtensionPoint
Component	SingletonPlugin
ComponentManager	PluginEnvironment

```
import os
env = sys.environ["PATH"]
PluginGlobals.load_services(path=env.split(os.sep))
```

In this example, the user's `PATH` environment is used to define the list of directories that are searched for Python modules and eggs.

2.4.3 Registering Executables

The `ExternalExecutable` plugin is used to define services that provide information about external executables. Services declare the executable name and user documentation, and then service methods indicate whether the executable is enabled (i.e. whether it is found, and the path of the executable:

```
service = ExternalExecutable(name='ls',
            doc='A utility to list file in Unix operating systems')

service.enabled()
# Returns True if the executable is found on the user path.

service.get_path()
# Returns a string that defines the path to this executable,
# or None if service is disabled.
```

2.5 Related Frameworks

The general design of PCA is adapted from Trac [8]. The PCA generalizes the Trac component architecture by supporting namespace management of plugins, as well as non-singleton plugins. For those familiar with Trac, the following classes roughly correspond with each other: The `PluginEnvironment` class is used to manage plugins, but unlike Trac this class does not need to be explicitly constructed.

As we noted earlier, there are a variety of mature component architectures that support plugins. The following requirements were motivated by our plugin use cases, which ultimately led to the development of PCA:

- *Well-defined framework core*: Many component architectures are embedded in larger software frameworks, which makes it difficult to extract and use just the software packages related to the component architecture.

- *Non-Singleton plugins*: The computational science applications that motivate PCA require both singleton and non-singleton plugins.

- *Namespaces*: Using plugins in large software projects requires management across multiple libraries. Namespaces are needed to effectively manage plugins in these complex software projects.

- *Caching*: PCA plugins need to be used in applications where plugin services are called many times. Thus, caching of extension point setup is needed to minimize the overhead of the PCA infrastructure.

- *Loading from EGGs*: Support for loading EGG files is invaluable in dynamic applications. Further, loading plugins from EGG files provides another level of modularity to the management of software applications.

Chapter 3

Coopr Plugins

3.1 Overview

A common object oriented approach for mathematical programming software is to use classes and class inheritance. For example, the OPT++ [4] optimization software library defines base classes with different characteristics (e.g. differentiability), and a concrete optimization solver is instantiated as a subclass of an appropriate base class. In this context, the base class can be viewed as defining the interface for the solvers that inherit from it.

Coopr plugins leverage the PyUtilib Component Architecture (PCA) to separate the declaration of component interfaces from their implementation. For example, the interface to optimization solvers are again declared with a class. However, solver plugins are not required to be subclasses of the interface class. Instead, they are simply required to provide the same interface methods and data.

The following sections detail the component interfaces that are supported in Coopr, and provide examples for how new plugins can be used with Coopr's Pyomo modeling language.

3.2 Coopr Interfaces

Coopr defines a variety of solver interfaces that can be used to customized and extend core Coopr functionality. The `coopr.opt` interfaces define capabilities that are needed perform optimization. The main interface is `IOptSolver`, which defines how optimizers behave:

```
class IOptSolver(Interface):
    """Interface class for creating optimization solvers"""

    def available(self, exception_flag=True):
        """Determine if this optimizer is available."""

    def solve(self, *args, **kwds):
        """
```

```
        Perform optimization and return an SolverResults object.
        """

    def reset(self):
        """Reset the state of an optimizer"""

    def set_options(self, istr):
        """
        Set the options in the optimizer from a string.
        """
```

The remaining interfaces in `coopr.opt` support capabilities that are used to execute external optimizers that are executed from a shell command. Executing external optimizers requires the following steps:

- write files that define the optimization problem
- convert files into a format that can be read by the external optimizer
- execute the optimizer
- read files that define the optimizer results and execution status

The `IProblemWriter`, `IProblemConverter` and `IProblemReader` interfaces are used to define components to execute these steps. This allows for a very flexible infrastructure, since the user no longer needs to worry about the compatibility between the optimization software and the modeling tool.

The `coopr.pyomo` interfaces define capabilities that are needed to model integer programs in Pyomo. The `IPyomoExpression` interface is used within Pyomo to define expression types. The `IPyomoSet` interface can be used to define new types of set objects (e.g. `Integers`).

The `IModelComponent` interface can be used to define new modeling components that are recognized by Pyomo. The standard modeling components include `Var`, `Objective` and `Param` objects. However, Pyomo generates models by sequentially constructing each component. This construction process can be customized to perform general operations on the model. For example, the `BuildCheck` plugin applies a function to a set of indices and validates that the function is true for all indices:

```
#
# Verify that all A[i] values are non-negative.
#
def rule(i, instance):
    return instance.A[i] >= 0.0
#
model.check = BuildCheck(model.I, rule)
```

Finally, the `IPyomoPresolver` interface can be used to define actions that are applied to Pyomo models before they are handed to solvers. Pyomo current defines presolve plugins that standardize variable names, identify active variables, and collect linear terms.

3.3 Example: A p-Median Solver

A natural extension of Coopr is the integration of domain-specific heuristics. Pyomo models provide an intuitive interface for accessing model data and variables. Consequently, we expect that users will develop heuristics directly in Python. As we shall see, setting up and applying this type of solver can be done quite naturally with the `pyomo` command-line interface.

Figure 3.1 describes a plugin that implements the `IOptSolver` interface. This plugin implements a `solve` method, which performs a greedy search for p-median optimization problems.[1] The only other step that is needed to use Coopr's `SolverRegistration` function, which associates the solver name, *greedy*, with the plugin class, `MySolver`.

Activating this plugin simply requires importing the Python module that contains it. All other registration is automated within Coopr. For example, if this plugin is in the file `solver.py`, then the following Python script can be used to allocate and apply this solver to Coopr's p-median example:

```
import coopr.opt
import pmedian
import solver

instance=pmedian.model.create('pmedian.dat')
opt = coopr.opt.SolverFactory('greedy')
results = opt.solve(instance)
print results
```

The `pyomo` command-line interface can also be used to apply a custom optimizer in a natural manner. The following command-line is used to solve the Coopr's p-median example with the `cbc` integer programming solver:

```
pyomo --solver=cbc pmedian.py pmedian.dat
```

Applying the custom solver requires the specification of the solver name, `greedy`, and indicating that the `solver.py` file should be imported before optimization:

```
pyomo --solver=greedy --preprocess=solver.py pmedian.py pmedian.dat
```

[1] The details of the greedy search are omitted here due to space constraints. This example is provided in with the Coopr examples: `coopr/examples/pyomo/p-median/solver.py`.

```
# Imports from Coopr and PyUtilib
from coopr.pyomo import *
from pyutilib.plugin.core import *
from coopr.opt import *

class MySolver(object):

    # Declare that this is an IOptSolver plugin
    implements(IOptSolver)

    # Solve the specified problem and return
    # a SolverResults object
    def solve(self, instance, **kwds):
        print "Starting greedy heuristic"
        val, instance = self._greedy(instance)
        n = value(instance.N)
        # Setup results
        results = SolverResults()
        results.problem.name = instance.name
        results.problem.sense = ProblemSense.minimize
        results.problem.num_constraints = 1
        results.problem.num_variables = n
        results.problem.num_objectives = 1
        results.solver.status = SolverStatus.ok
        soln = results.solution.create()
        soln.value = val
        soln.status = SolutionStatus.feasible
        for j in range(1,n+1):
            if instance.y[j].value is 1:
                soln.variable[instance.y[j].name] = {``value'': 1}
        return results

    # Perform a greedy search
    def _greedy(self, instance):
        # Details omited here...
        return [best, instance]

# Register the solver with the name 'greedy'
SolverRegistration("greedy", MySolver)
```

Figure 3.1: A simple customized solver for p-Median problems.

Thus, the user can develop custom solvers in Python modules, which are tested directlying using the `pyomo` command-line interface.

3.4 Deploying Plugin Packages

3.4.1 Background

The Python `setuptools` package is the *de facto* standard for deploying Python software. This package extends Python's `distutils` functionality. A key element of this extension is the `easy_install` command, which allows the installation of Python software from remote repositories. In particular, the Python Package Index (PyPI) provides a convenient repository for hosting Python packages. The `easy_install` command can easily upload and download packages from PyPI, thereby simplifying the distribution of Packages like Coopr, which depends on a variety of freely available packages.

Coopr's Python software architecture leverages features of the `setuptools` package to facilitate the integration of Python packages that contain plugin components. Here are the details:

- The `coopr.plugins` subpackage is a *namespace package*. Namespace packages are a mechanism for splitting a single Python package across multiple directories on disk. This allows different Python packages to provide plugin components in separate plugin subpackages (e.g. `coopr.plugins.mine` and `coopr.plugins.yours`).

- Subpackages in Coopr have been setup to dynamically load plugins that are registered in `coopr.plugins` packages. This leverages the `pkg_resources` package that included with `setuptools` by defining *entry points* for the Python packages that are loaded under the `coopr.plugins` namespace.

Although configuring Coopr to leverage these capabilities requires some black magic, we hope that Coopr developers will not need to delve into the details of this mechanism. The following section provides some guidelines for configuring a package such that its plugin components are automatically loaded when Coopr is imported.

3.4.2 Example: coopr.plugins.neos

The `coopr.plugins.neos` package is hosted by the Coopr Forum repository [1], which facilitates community involvement in Coopr. Coopr Forum allows people to contribute code extensions and plugins without going directly through the Coopr software repository. The `coopr.plugins.neos` package provides a simple example of how Coopr can be extended with plugins to enable optimization on the NEOS optimization server [2].

This package illustrates the basic organization that is needed to seamlessly integrate plugins from external software packages with Coopr. There are a variety of important details, which we enumerate in the following sections.

Directory Structure

The trunk version of the `coopr.plugins.neos` package is available at `http://coopr-forum.googlecode.com/svn/neos/trunk/`. This package has the following directory structure:

```
setup.py
coopr/
coopr/__init__.py
coopr/plugins/
coopr/plugins/__init__.py
coopr/plugins/neos/
coopr/plugins/neos/__init__.py
coopr/plugins/neos/NEOS.py
coopr/plugins/neos/NEOS_CBC.py
coopr/plugins/neos/kestrel.py
coopr/plugins/neos/kestrel_plugin.py
```

A key aspect of this directory structure is that it mimics the structure in Coopr. Further, the files `coopr/__init__.py` and `coopr/plugins/__init__.py` must have the following definitions to ensure that `coopr.plugins` is a namespace package:

```
# this is a namespace package
try:
    import pkg_resources
    pkg_resources.declare_namespace(__name__)
except ImportError:
    import pkgutil
    __path__ = pkgutil.extend_path(__path__, __name__)
```

Plugin Modules

There are few restrictions on the content of the modules in `coopr/plugins/neos`. However, automatic loading of plugin components requires that their associated Python modules are imported by `coopr/plugins/neos/__init__.py`. In this package, plugin components are defined in the `kestrel_plugin.py` and `NEOS_CBC.py` modules, which are imported by the `__init__.py` module.

Package Configuration

The `setuptools` package uses the `setup.py` module to configure the installation of `coopr.plugins.neos`. Figure 3.2 contains the listing of this file. Only a handful of these arguments are specific to an installation with `setuptools`:

- The `packages` options lists all package directories that are included in this package.
- The `namespace_packages` options lists all package directories that are namespace packages. There are two namespace packges in Coopr that need to be specified: `coopr` and `coopr.plugins`.
- The `entry_points` option specifies how components of this package are registered with `setuptools`. The `entry_points` option specifies a dictionary. The keys of this dictionary are *group names* that specify a set of related entry points. Coopr uses this dictionary to load packages that have been installed with `setuptools`. By convention, Coopr packages load plugins with the same group name; for example, the `coopr.opt` package imports entry points with the 'coopr.opt' group name.

As this example illustrates, an entry point relates an *entry name* with a package in the software repository. Although these entry names must be unique, Coopr does not rely on them having any particular syntax or semantics. Instead, Coopr simply loads each entry point within a given group. This triggers the registration of plugin components within Coopr, which is the desired result.

Uploading Package Releases

Once your Python package is ready for a release to other users, you can upload it to the PyPI repository using the following command:

```
python setup.py sdist upload -s
```

Note that before you upload the first time you will need to register your package with the following command:

```
python setup.py register
```

```
from setuptools import setup, find_packages

classifiers = """\
Development Status :: 3 - Alpha
Intended Audience :: End Users/Desktop
Intended Audience :: Science/Research
License :: OSI Approved :: BSD License
Natural Language :: English
Operating System :: Microsoft :: Windows
Operating System :: Unix
Programming Language :: Python
Topic :: Scientific/Engineering :: Mathematics
Topic :: Software Development :: Libraries :: Python Modules
"""

import coopr.plugins.neos
doclines = coopr.plugins.neos.__doc__.split("\n")

setup(name = "coopr.plugins.neos",
      version = coopr.plugins.neos.__version__,
      maintainer = coopr.plugins.neos.__maintainer__,
      maintainer_email = coopr.plugins.neos.__maintainer_email__,
      url = coopr.plugins.neos.__url__,
      license = coopr.plugins.neos.__license__,
      platforms = ["any"],
      description = doclines[0],
      classifiers = filter(None, classifiers.split("\n")),
      long_description = "\n".join(doclines[2:]),
      packages = ['coopr', 'coopr.plugins', 'coopr.plugins.neos'],
      keywords = ['optimization'],
      namespace_packages=['coopr', 'coopr.plugins'],
      entry_points = {
        'coopr.opt': [
            'solvermanager.neos = coopr.plugins.neos.kestrel_plugin',
            'solver.neos_cplex = coopr.plugins.neos.NEOS_CPLEX',
        ]
      },
      install_requires=['Coopr>=1.2']
      )
```

Figure 3.2: The `setup.py` file used by `coopr.plugins.neos`.

Acknowledgements

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Bibliography

[1] *Coopr forum*. http://code.google.com/p/coopr-forum/, 2009.

[2] E. D. Dolan, R. Fourer, J.-P. Goux, T. S. Munson, and J. Sarich, *Kestrel: An interface from optimization modeling systems to the NEOS server*, INFORMS Journal on Computing, 20 (2008), pp. 525–538.

[3] *Envisagecore*. https://svn.enthought.com/enthought/wiki/EnvisageThree/core.html, 2009.

[4] J. C. Meza, *OPT++: An object-oriented class library for nonlinear optimization*, Tech. Rep. SAND94-8225, Sandia National Laboratories, 1994.

[5] *PyUtilib optimization framework*. http://software.sandia.gov/pyutilib, 2009.

[6] *Sprinklespy*. http://termie.pbworks.com/SprinklesPy, 2009.

[7] *Trac component architecture*. http://trac.edgewall.org/wiki/TracDev/ComponentArchitecture, 2009.

[8] *Trac*. http://trac.edgewall.org/, 2009.

[9] *yapsy*. http://yapsy.sourceforge.net/, 2009.

[10] *Zope*. http://www.zope.org/, 2009.