Managing Scientific Workflows in Python with `pyutilib.workflow`

William E. Hart[*]

August 3, 2010

Abstract

We describe the capabilities of the `pyutilib.workflow` software package. This package provides Python classes that provide an intuitive interface for defining and executing scientific workflows. Further, `pyutilib.workflow` is a native Python package, so it can be used to define workflows within complex Python software libraries.

[*]Sandia National Laboratories, Informatics and Computer Science Department, PO Box 5800, Albuquerque, NM 87185; `wehart@sandia.gov`

1 Introduction

Scientific workflow is an increasingly popular strategy for managing complex scientific computation processes. Workflows allow scientists to automate data transformations, describe complex computational procedures, and parallelize these analyses. Scientific workflow systems are closely related to workflow models used in business process management systems. The key difference is that scientific workflows focus on the transformation of data through algorithms, whereas business workflows focus on scheduling and execution of tasks.

Many of the workflow packages developed in Python are best described as business workflow systems. For example, packages like `django-workflows` and `zope.app.workflow` provide workflows for web content management. The following native Python workflow packages appear to be suitable for scientific workflows:

- *Pyphant*: This is a framework for scientific data analysis. A computational analysis is defined by a graph of processing steps, which is managed with a workflow engine.
- *Python Workflow Engine*: This is a simple workflow engine that was initially based on the workflow engine used in the ACE project.
- *Spiff Workflow*: This package is designed around the workflow patterns defined at `http://www.workflowpatterns.com`.

Other packages like VisTrails [3] and Weaver [1] also support the management of scientific workflows in Python, though they rely on external software packages to execute these workflows.

The `pyutilib.workflow` (PW) package supports the definition and execution of scientific workflows. PW is provides a self-contained kernel for defining and executing workflows in Python; no external software packages are required to execute PW workflows. Further, PW defines a workflow through the interaction of Python objects, rather than an explicit definition of a workflow graph. PW provides an intuitive interface for describing the relationship between tasks in a workflow, and a workflow can be simply treated as a functor that is executes and returns a dictionary of computed data.

The remainder of this manuscript provides a detailed description of the capabilities in PW. We include many examples that illustrate how PW objects interact to define and execute workflows.

2 Managing Workflows

2.1 Overview

Figure 1 provides a graphical illustration of the components of a workflow. A *workflow* is comprised of one or more computational steps, which we call a *task* or *component*. A task maps a set of input data into a set of output data. Input and output data are managed with *port* objects, and tasks are linked together with *connectors* that define a connection from an output port in one task to an input port for another. These connections form a directed acyclic graph (DAG), which defines how task executions need to be coordinated to correctly execute the entire workflow.

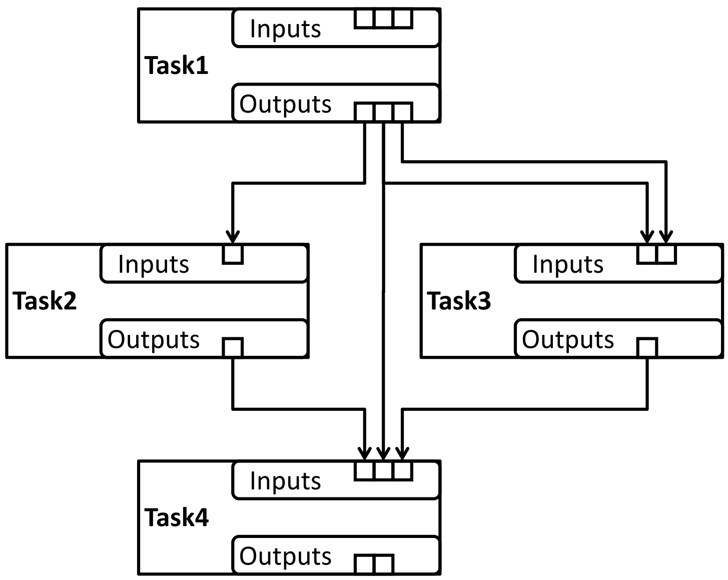

Figure 1: A graphical illustration of a workflow with four tasks. Black lines between tasks represent connectors, and square boxes in the tasks represent the input and output ports.

2.2 A Simple Example

The main goal of PW is to support the definition of workflows in an intuitive manner using Python objects. There are two fundamental classes defined by PW that are used to define a workflow: `Task` and `Workflow`. A user defines tasks by creating subclasses of the `Task` class. For example, the following task computes the sum of its two inputs:

```
class TaskA(pyutilib.workflow.Task):
```

```python
    def __init__(self, *args, **kwds):
        """Constructor."""
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('x')
        self.inputs.declare('y')
        self.outputs.declare('z')

    def execute(self):
        """Compute the sum of the inputs."""
        self.z = self.x + self.y
```

The `Task` class defines the `inputs` and `outputs` attributes that are used to respectively declare input and output ports. These declarations must be included in the task constructor, since the inputs and outputs are treated as static task properties by PW.

The task computation is performed by the `execute` method, which must be defined by the user. Note that the input and output values are attributes of the task object. This simplifies the syntax for users developing task computations by allowing them to treat task data as they would in any other Python object. PW initializes the value of these attributes before calling `execute`, and it interrogates the task afterwards to set the value of the output ports.

The following Python code creates the `TaskA` object, creates a `Workflow` object, initializes the workflow with this task, and then executes the workflow with input values:

```python
A = TaskA()
w = pyutilib.workflow.Workflow()
w.add(A)
print w(x=1, y=3)
```

Note that the workflow defines a functor, which is executed with keyword arguments that are mapped to the task inputs. This functor returns an `Options` object, which is a glorified Python dict class. The output of printing the workflow results is:

```
Options:
    z = 4
```

2.3 Defining Connections

The previous example was a trivial illustration of the setup and execution of a workflow. In practice, workflows will be defined by constructing two or more tasks that are linked together. Suppose we wish to compute the expression:

$$z = 2 * x + y.$$

We can employ `TaskA` to perform the sum, and the following task to double the value of x:

```
class TaskB(pyutilib.workflow.Task):

    def __init__(self, *args, **kwds):
        """Constructor."""
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('X')
        self.outputs.declare('Z')

    def execute(self):
        """Compute the sum of the inputs."""
        self.Z = 2*self.X
```

The following Python code creates the `TaskA` and `TaskB` objects, links the output of B to the input of A, and then creates and executes a workflow:

```
A = TaskA()
B = TaskB()
A.inputs.x = B.outputs.Z

w = pyutilib.workflow.Workflow()
w.add(A)
print w(X=1, y=3)
```

The connection between `TaskA` and `TaskB` is defined with the command

```
A.inputs.x = B.outputs.Z
```

The syntax transparently creates a `Connector` object that connects the `Z` output of `TaskB` to the `x` input of `TaskA`. This greatly simplifies the declaration of connections when compared with other Python workflow packages. Note that this mechanism allows an output port to be connected to one or more input ports. The default setup of ports allows an input port to only connect to a single output port. (See Section 2.4 for further discussion.)

As in our earlier example, the workflow is created by constructing a `Workflow` object and then adding tasks to it. Note, however, that in this example only `TaskA` was added. The `Workflow` object traverses the connections between tasks to identify all tasks connected to the task that is added. Consequently, only a single task in a workflow needs to be added to the `Workflow` object.

Note that the functor defined by the workflow has a slightly different API in this example; it uses inputs `X` and `y`. To understand why, consider Figure 2, which shows the workflow in this example. Tasks `TaskA` and `TaskB` are connected to each other, but also to a start and end task. The start and end tasks are constructed when a `Workflow` object loads the workflow. The start task contains outputs that correspond to every input port that is not connected to an output port. Similarly, the end task contains inputs that correspond to every output port that is not connect to an input port. In this way, the inputs and outputs of the workflow are automatically defined.

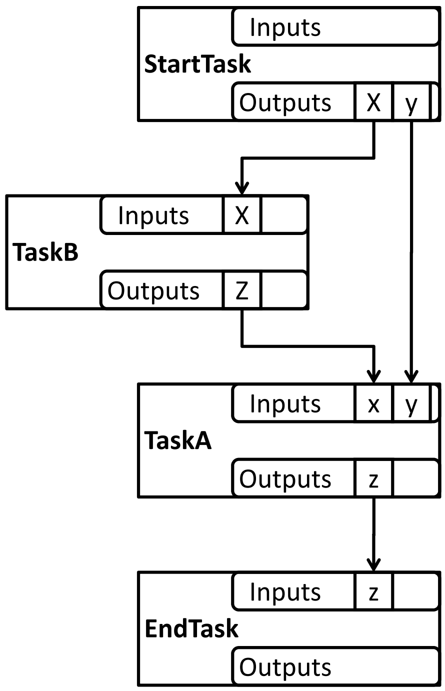

Figure 2: An illustration of the workflow defined with tasks `TaskA` and `TaskB`.

To see further implications of this logic, suppose that `TaskC` is used instead of `TaskB`:

```
class TaskC(pyutilib.workflow.Task):

    def __init__(self, *args, **kwds):
        """Constructor."""
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('X')
        self.inputs.declare('y')
        self.outputs.declare('W')
        self.outputs.declare('Z')

    def execute(self):
        """Compute the sum of the inputs."""
```

```
        self.W =  self.X+self.y
        self.Z = 2*self.W
```

Figure 3 shows the workflow for this example. The setup and execution of this task does not change. However, the input `y` is now used by both tasks `TaskA` and `TaskC`. Further, the output `W` is now included in the final results. The output of printing the workflow results is:

```
Options:
   W = 4
   z = 11
```

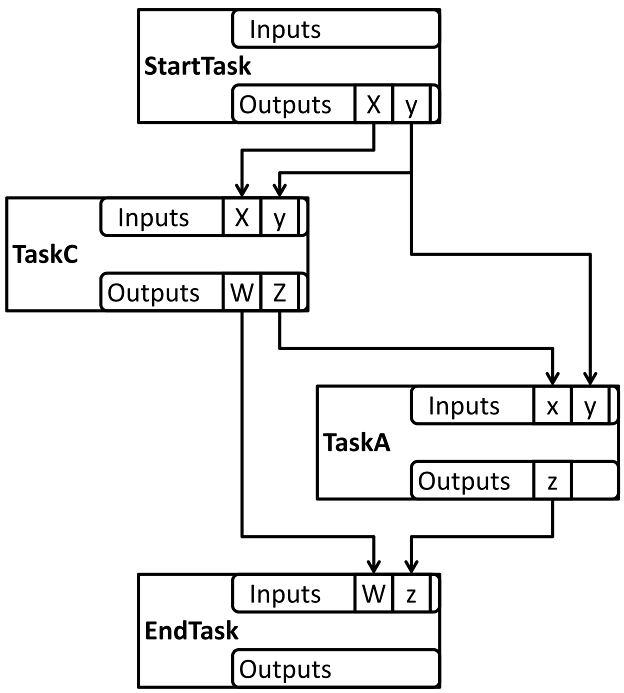

Figure 3: An illustration of the workflow defined with tasks `TaskA` and `TaskC`.

2.4 Input Ports with Multiple Connections

The `action` constructor option for the `Port` class defines how input connections are used to compute the input value. The default action is `store`, which indicates that the connector value is stored in the port. This behavior reflects the previous examples, and it is well-suited for workflows where there is a direct correspondence between output ports and input ports.

However, contexts often arise in practice where a suite of tasks needs to be computed and their results are analyzed together. For example, consider `TaskD` which generalizes `TaskA` to sum an arbitrary number of inputs:

```
class TaskD(pyutilib.workflow.Task):

    def __init__(self, *args, **kwds):
        """Constructor."""
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('x', action='append')
        self.outputs.declare('z')

    def execute(self):
        """Compute the sum of the inputs."""
        self.z = sum(self.x)
```

Note that the input port `x` is defined with the `append` action, which configures it to create a list of input values.

The following example use `TaskD` to define a workflow with inputs from `TaskE`, which generates a random integer value:

```
class TaskE(pyutilib.workflow.Task):

    def __init__(self, *args, **kwds):
        """Constructor."""
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('Y')
        self.outputs.declare('Z')

    def execute(self):
        """Compute the sum of the inputs."""
        self.Z = random.randint(1,self.Y)

random.seed(123809870128)
D = TaskD()
for i in range(100):
    E = TaskE()
    D.inputs.x = E.outputs.Z

w = pyutilib.workflow.Workflow()
w.add(D)
```

```
print w(Y=100)
```

In this example, `TaskE` objects are created and connected to the `TaskD` object with the command:

```
D.inputs.x = E.outputs.Z
```

The input `x` port is configured to append inputs to a list, and no special syntax is needed to indicate how the connections are configured between the `x` port and the `Z` ports.

The `map` action can also be specified to define an input as a dictionary with keys that are the task ids from the connection that generated the values. For example, this can be used to associate data generated in different branches of a workflow. The following example uses this associate to define a dictionary, which is the final result:

```
class TaskF1(pyutilib.workflow.Task):

    def __init__(self, *args, **kwds):
        """Constructor."""
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('a',)
        self.inputs.declare('aval')
        self.outputs.declare('a', self.inputs.a)
        self.outputs.declare('aval', self.inputs.aval)

    def execute(self):
        pass

class TaskF2(pyutilib.workflow.Task):

    def __init__(self, *args, **kwds):
        """Constructor."""
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('A',)
        self.inputs.declare('Aval')
        self.outputs.declare('A', self.inputs.A)
        self.outputs.declare('Aval', self.inputs.Aval)

    def execute(self):
        pass

class TaskG(pyutilib.workflow.Task):

    def __init__(self, *args, **kwds):
        """Constructor."""
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('x', action='map')
```

```
        self.inputs.declare('y', action='map')
        self.outputs.declare('z')

    def execute(self):
        """Compute the sum of the inputs."""
        self.z = {}
        for key in self.x:
            self.z[ self.x[key] ] = self.y[key]

F1 = TaskF1()
F2 = TaskF2()
G = TaskG()
G.inputs.x = F1.outputs.a
G.inputs.y = F1.outputs.aval
G.inputs.x = F2.outputs.A
G.inputs.y = F2.outputs.Aval

w = pyutilib.workflow.Workflow()
w.add(G)
print w(a='a', aval=1, A='A', Aval=2)
```

Tasks `TaskF1` and `TaskF2` simply map their inputs to outputs. Their outputs are connected to two inputs in `TaskG`, and these inputs are used to create a dictionary. The output of this computation is:

```
Options:
    z = {'a': 1, 'A': 2}
```

2.5 Using Workflows as Tasks

A key feature of PW is the ability to use workflows as components of other workflows. This is possible because `Workflow` is a subclass of `Task`.

For example, consider the following workflows that are defined with `TaskA` and `TaskC`:

```
A = TaskA()
C = TaskC()
A.inputs.x = C.outputs.Z

w1 = pyutilib.workflow.Workflow()
w1.add(A)

AA = TaskA()
AA.inputs.x = w1.outputs.W
AA.inputs.y = w1.outputs.z

w2 = pyutilib.workflow.Workflow()
```

```
w2.add(AA)

print w2(X=1, y=3)
```

Workflow `w1` is the workflow defined in the previous example. This object is used to define workflow `w2`, which uses `TaskA` to sum the outputs of `w1`: `W` and `z`. The output of executing `w2` is

```
Options:
    z = 15
```

2.6 Initializing Port Values

Task ports are initialized through the execution of a workflow, and through the explicit specification of port values. The simplest way to specify port values is to define them directly. For example, consider the following variation of the example in Section 2.2:

```
A = TaskA()
w = pyutilib.workflow.Workflow()
w.add(A)
A.inputs.x = 1
A.inputs.y = 3
print w()
```

The workflow is constructed as before, but the values of ports `x` and `y` are defined explicitly, and the execution of the workflow does not specify these values.

PW also supports the initialization of port values with command-line options. The goal of this capability is to facilitate the use of PyUtilib in command-line applications, by allowing command-line options to be used to directly initialize a workflow. The following example is a simple extension of the example in Section 2.2.

```
class TaskAA(pyutilib.workflow.Task):

    def __init__(self, *args, **kwds):
        """Constructor."""
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('x')
        self.inputs.declare('y')
        self.parser.add_option('--x', dest='x', type='int')
        self.parser.add_option('--y', dest='y', type='int')
        self.outputs.declare('z')

    def execute(self):
```

```
        """Compute the sum of the inputs."""
        self.z = self.x + self.y

AA = TaskAA()
w = pyutilib.workflow.Workflow()
w.add(AA)
w.set_options(['--x=1', '--y=3', '--bad=4'])
print w()
```

Some additional logic is added to the `TaskAA` class to specify the command-line options that are recognized by the `parser` object. Note that this parser mimics the standard `options.OptionParser` facility. The main difference is that this class can be configured to not enforce strict parse semantics. That is, it allows for parsing of 'invalid' options, which are ignored. Unfortunately, this leads to ambiguities that require that all option with values use the `--option=value` syntax.

In this example, the `set_options` method is used to initialize a workflow with a list of option strings. This syntax mimics the format of data provided by `sys.argv`. Again, the execution of the workflow does not specify these values.

Note that port values specified in these ways are viewed as default values for the port. When a port value is computed from input connections, then the port value will be overriden if the input connections provide a non-trivial value. For example, if the port action is `store`, then the value will be overriden if the input connection has a value other than `None`. Similarly, if the port action is `append` or `map`, then the value will be overriden if one or more of the input connections are not None.

Additionally, port values are redefined by the workflow keyword options. For example, in the following script we initialize input ports for `TaskAA`, which are then redefined when the workflow is executed:

```
AA = TaskAA()
w = pyutilib.workflow.Workflow()
w.add(AA)
w.set_options(['--x=1', '--y=3'])
print w(y=4)
```

The output of this script is

```
Options:
    z = 5
```

which reflects the fact that the value of `y` was redefined by the workflow keyword option.

3 Defining Task Resources

There are many contexts in which task execution is dependent on the availability of external resources. For example, data files may need to be available, a database may need to be unlocked, or a software license may need to be free. PW allows these constraints on workflow execution to be represented with `Resource` objects that represent the state of a dependent resource. A resource may or may not be available, and the workflow can lock and unlock a resource as it employs it for execution.

PW defines the `ExecutableResource`, which allows a user to specify an executable that is automatically found by searching the `PATH` environment. If the specified executable is not found, then it is unavailable for execution in a workflow. This resource also includes a utility method for applying this executable with command-line arguments.

The following example illustrates the use of this resource to define a task that lists all of the files in a specified directory:

```
class TaskH(pyutilib.workflow.Task):

    def __init__(self, *args, **kwds):
        """Constructor."""
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('dir')
        self.outputs.declare('list')
        self.add_resource(pyutilib.workflow.ExecutableResource('ls'))

    def execute(self):
        self.resource('ls').run(self.dir, logfile=currdir+'logfile')
        self.list = []
        for line in open(currdir+'logfile','r'):
            self.list.append( line.strip() )
        self.list.sort()

H = TaskH()
w = pyutilib.workflow.Workflow()
w.add(H)
print w(dir=currdir+'dummy')
```

4 The Task Factory

PW leverages the PyUtilib Component Architecture [2] to support the definition of a task factory. The PW task factory allows users to create plugin tasks on the fly without requiring knowledge of where these tasks are defined. This capability exposes a variety of standard tasks that are defined in PW , and it can be used to create tasks that are defined by third-party libraries in a standard manner.

The `TaskFactory` object defined in PW is a functor. This functor can be used to create a task that has been registered as a plugin. For example, the `Selection_Task` class is registered with the string 'workflow.selection', and it can be instantiated as follows:

task = pyutilib.workflow.TaskFactory('workflow.selection')

The following sections describe the task plugins that are defined by PW, and we provide an example of how a task plugin is defined.

4.1 Selection Task

The `workflow.selection` task has the following inputs:

- `data`: a dictionary
- `index`: an index key in the dictionary

This task returns the value of the dictionary with the specified index key.

Note that this task does not fail gracefully if the index key is not defined in the dictionary. An exception will occur that will terminate the execution of the workflow.

4.2 Defining Plugin Tasks

A plugin task is created as a subclass of the `TaskPlugin` class. This registers this task as a plugin with the PyUtilib Component Architecture. The only additional step required for a plugin task is to use the `alias` declaration to define the string that is used to create this task in the task factory.

For example, the following code defines the task `PluginTaskA` that is registered with the string 'TaskA':

```
class PluginTaskA(pyutilib.workflow.TaskPlugin):

    pyutilib.component.core.alias('TaskA')

    def __init__(self, *args, **kwds):
        """Constructor."""
        pyutilib.workflow.Task.__init__(self, *args, **kwds)
        self.inputs.declare('x')
        self.inputs.declare('y')
        self.outputs.declare('z')

    def execute(self):
        """Compute the sum of the inputs."""
        self.z = self.x + self.y
```

Note that the only difference with the definition of `TaskA` is the specification of the base class and the `alias` declaration.

The following Python code creates the `PluginTaskA` object, creates a `Workflow` object, initializes the workflow with this task, and then executes the workflow with input values:

```
A = pyutilib.workflow.TaskFactory('TaskA')
w = pyutilib.workflow.Workflow()
w.add(A)
print w(x=1, y=3)
```

This has the same logical steps as the example in Section 2.2. The only difference is that the task is created by the task factory.

Acknowledgements

Sandia National Laboratories is a multi-program laboratory operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin company, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[1] P. Bui, L. Yu, and D. Thain, *Weaver: Integrating distributed computing abstractions into scientific workflows using python*, in Challenges of Large Applications in Distributed Environments at ACM HPDC 2010, June 2010.

[2] W. E. Hart and J. D. Siirola, *The pyutilib component architecture*, Tech. Rep. SAND2010-2516, Sandia National Laboratories, May 2010.

[3] *Vistrails*. `http://www.vistrails.org`, 2010.