# Purpose of This Document

The purpose behind the OpenMDAO Conversion Guide is to help users of previous versions of OpenMDAO (versions up to and including 0.13.0) to change their models over to the new OpenMDAO 1.0.0 design. This will require some re-thinking and re-structuring. If you are new to OpenMDAO, you should be able to start writing new models, and this guide should not pertain to you. You should check out our OpenMDAO User Guide.

If you do not have OpenMDAO v 1.0 installed, you should first view our Getting Started Guide. Then we would recommend becoming familiar with the new building blocks of OpenMDAO in the User Guide's 'Basics' section

Conceptually, the core building blocks of OpenMDAO 1.0 are similar to those found in previous versions, but the syntax you use to define those building blocks is different. This guide will start by describing the differences you'll see when defining a Component. Then we'll move on to the process of connecting your Components and building your model.

# Declaring a Simple Component

We'll start off by defining a very simple component, one that has an input *x* and an output *y*, both having a value of type *float*. When the component runs, it will assign the value of `x*2.0` to *y*.

## Imports

To define our new class, we need to import some class definitions. In old OpenMDAO, we had to import the definition of Component and the trait class Float.

```
from openmdao.main.api import Component
from openmdao.main.datatypes.api import Float
```

In new OpenMDAO, we just need the definition of Component, but it now lives in a different location.

```
from openmdao.api import Component
```

## Declaring Variables

Our component needs 2 variables, *x* and *y*. In old OpenMDAO, variables were typically declared at the class level.

```
class Times2(Component):
    x = Float(1.0, iotype='in', desc='my var x')
    y = Float(2.0, iotype='out', desc='my var y')
```

In new OpenMDAO, we add variables in our component's *__init__* method, using component methods *add_param* to add an input, *add_output* to add an output, and *add_state* to add a state variable. For our component, it would look like this:

```
class Times2(Component):
    def __init__(self):
        self.add_param('x', 1.0, desc='my var x')
        self.add_output('y', 2.0, desc='my var y')
```

The various *add_** methods in new OpenMDAO allow arbitrary metadata to be specified as keyword arguments in the same manner that they were specified in *Float* and the other trait constructors in older versions, so you could do the following, for example:

```
def __init__(self):
    self.add_param('z', 1.0, units='ft', weird_meta='foo')
```

The example above also specifies *units*. New OpenMDAO uses the same unit names that work in the same way as in old OpenMDAO.

v: 1.6.3

## Specifying how Data is Passed

In both old and new versions of OpenMDAO, data can be passed between components in two different ways. By default, variables that are differentiable are passed as part of a flattened numpy *float* array. Other variables are just passed by reference. To force a variable to be passed by value in old OpenMDAO, you would set the *noflat* metadata to *True* when creating the variable, for example:

```
x = Float(1.0, iotype='in', desc='my var x', noflat=True)
```

In new OpenMDAO, you would set the *pass_by_obj* metadata to *True*, e.g.,

```
self.add_output('y', 2.0, pass_by_obj=True)
```

**Caution**

When you force a variable to be *pass_by_obj*, you are excluding it from all derivative calculations, which could result in incorrect answers, so use *pass_by_obj* with caution.

## Defining Calculations

In old OpenMDAO, we would specify how our component updates its outputs based on the values of its inputs by defining an *execute* method.

```
def execute(self):
    self.y = self.x * 2.0
```

In new OpenMDAO, we do the same thing by defining a *solve_nonlinear* method.

```
def solve_nonlinear(self, params, unknowns, resids):
    unknowns['y'] = params['x'] * 2.0
```

Aside from the name change, the other big difference here is that the variables are no longer attributes of our component. Our inputs now live in the *params* dict-like-object and our outputs are found in the *unknowns* dict-like-object.

One really nice feature of this new syntax is a very clear separation between framework variables and regular python attributes. Anything that OpenMDAO knows about, and should be passed around by its data passing system, will live in *params*, *unknowns*, or *resids*.

## Defining Derivatives

In old OpenMDAO, we specified a Jacobian as a monolithic dense ndarray in the *linearize* method.

```
def linearize(self):
    J = numpy.array([2.0])
    return J
```

In new OpenMDAO, the jacobian is instead stored in a dictionary whose keys are (output, input) tuples. This is a much more convenient manner for specifying the derivatives, especially for large numbers of variables. Also, there is no longer any need for the *list_deriv_vars* function from older versions.

```
def linearize(self, params, unknowns, resids):
    J = {}
    J[('y', 'x')] = numpy.array([2.0])
    return J
```

If your component does not have derivatives, you *must* set it up to be finite-differenced. Old OpenMDAO handled this automatically, but you now need to manually force the finite difference by:

```
self.fd_options['force_fd'] = True
```

Here, self is the component instance. You can set any *Component* or *Group* to be finite differenced by setting this option to True. If you force finite difference around a group, then you are taking the FD across that group as a single block.

## Variable Trees

Vartrees are supported in new OpenMDAO, but they are much simpler now. We will show you the differences in how they are declared.

> **Note**
>
> Vartrees don't have full functionality in 1.0 yet. The most notable missing feature is a convenience method to connect whole vartrees between two components. Currently, you have to connect it one variable at a time, or write your own convenience method. We'll be working to get better support for vartrees in the near future.

For Old OpenMDAO:

```
from openmdao.main.api import Component, VariableTree
from openmdao.lib.datatypes.api import Float, VarTree

class FlightCondition(VariableTree):
    """Container of variables"""

    airspeed = Float(120.0, units='nmi/h')
    angle_of_attack = Float(0.0, units='deg')
    sideslip_angle = Float(0.0, units='deg')


class AircraftSim(Component):
    """This component contains variables in a VariableTree"""

    # create VarTrees to handle updates to our FlightCondition attributes
    fcc1 = VarTree(FlightCondition(), iotype='in')
    fcc2 = VarTree(FlightCondition(), iotype='out')
```

And for new OpenMDAO:

```
from openmdao.api import Component

class AircraftSim(Component):
    def __init__(self):

        self.add_param('fcc1:airspeed', 120.0, units='nmi/h')
        self.add_param('fcc1:angle_of_attack', 0.0, units='deg')
        self.add_param('fcc1:sideslip_angle', 0.0, units='deg')

        self.add_output('fcc2:airspeed', 120.0, units='nmi/h')
        self.add_output('fcc2:angle_of_attack', 0.0, units='deg')
        self.add_output('fcc2:sideslip_angle', 0.0, units='deg')
```

The main difference here is that you don't actually create a new class to hold the hierarchical data structure. Instead, you just create variables with a hierarchical naming pattern; a ":" is used to delineate different levels of the data structure. Each individual leaf behaves like any other variable.

## Full Component Definition

Putting together the code from the previous sections, we get the following component definition for old OpenMDAO:

```
from openmdao.main.api import Component
from openmdao.main.datatypes.api import Float

class Times2(Component):
    x = Float(1.0, iotype='in', desc='my var x')
    y = Float(2.0, iotype='out', desc='my var y')

    def execute(self):
        self.y = self.x * 2.0
```

```
    def list_deriv_vars(self):
        return ('x', ), ('y',
        )
    def linearize(self):
        J = numpy.array([2.0])
        return J
```

And for new OpenMDAO:

```
from openmdao.api import Component

class Times2(Component):
    def __init__(self):
        self.add_param('x', 1.0, desc='my var x')
        self.add_output('y', 2.0, desc='my var y')

    def solve_nonlinear(self, params, unknowns, resids):
        unknowns['y'] = params['x'] * 2.0

    def linearize(self, params, unknowns, resids):
        J = {}
        J[('y', 'x')] = numpy.array([2.0])
        return J
```

To summarize the differences in Component definition:

- The *execute* method is now called *solve_nonlinear*.
- Variables are declared in *__init__* instead of at class level.
- Variables are no longer attributes of the Component but instead are accessed via the *params* and *unknowns* objects that are passed into *solve_nonlinear*.
- In Variable metadata, *noflat* is now *pass_by_obj*.
- The `Component` class definition is imported from a different place.
- OpenMDAO no longer uses the strong typing of Traits, so the associated imports (e.g. *Float*) are no longer needed.

# Building a Model

## Grouping Components

In old OpenMDAO, Components can be grouped together in an Assembly, e.g.,

```
asm = Assembly()
asm.add('comp1', Times2())
asm.add('comp2', Times2())
```

In new OpenMDAO, grouping of Components is done using a Group object, e.g.,

```
group = Group()
group.add('comp1', Times2())
group.add('comp2', Times2())
```

## Promoting Variables

In old OpenMDAO, Assemblies are Components and can have their own variables, and these variables can be either explicitly linked to variables on the Assembly's internal Components using *connect*, or can be automatically created and linked using the *create_passthrough* convenience function. For example:

```
asm = Assembly()
asm.add('comp1', Times2())
asm.create_passthrough('comp1.x')
```

In new OpenMDAO, Groups are NOT Components and do not have their own variables. Variables can be promoted to the Group level by passing the *promotes* arg to the *add* call, e.g.,

```
group = Group()
group.add('comp1', Times2(), promotes=['x'])
```

This will allow the variable *x* that belongs to *comp1* to be accessed via *group.params['x']*.

## Linking Variables

In old OpenMDAO, linking two variables within an Assembly is done by calling the *connect* method on the Assembly.

```
asm.connect('comp1.y', 'comp2.x')
```

In new OpenMDAO, *explicitly* linking two variables within a Group is done by calling the *connect* method on the Group.

```
group.connect('comp1.y', 'comp2.x')
```

Linking in new OpenMDAO can also be done *implicitly*, by using the *promotes* arg in the *add* call that we saw earlier. See Basics, Group for details of linking using promotion.

## Connecting Parts of Array variables

In old OpenMDAO, you can put array entry references in your *connect* statement. For example, to connect a slice of an output variable to an input variable, you can do the following:

```
asm.connect('mycomp1.y[2:10]', 'mycomp2.x')
```

In new OpenMDAO, you would do it like this:

```
group.connect('mycomp1.y', 'mycomp2.x', src_indices=range(2,10))
```

**Note**

Support for setting *src_indices* to a slice object or tuple is likely in the future, but for now, you must specify *all* of the indices.

**Caution**

Old OpenMDAO also supported specifying array entries on the destination variable, e.g.,

```
asm.connect('mycomp1.y', 'mycomp2.x[5]')
```

New OpenMDAO does not support that functionality.

## Model Tree

In both old and new OpenMDAO, the model has a tree structure. In old OpenMDAO, the tree has an Assembly at the top, and that Assembly contains Components and/or other Assemblies. In new OpenMDAO, the top of the tree is a Problem object, and that Problem contains a single Group called *root* that contains the rest of the model. A Group cannot be executed unless it is contained within a Problem object and that Problem's *setup* method has been called.

## Drivers and Solvers

In old OpenMDAO, every Assembly has a Driver, and a Driver can be an optimizer **or** a Solver, as well as some other iterative executive like a DOEDriver, etc.

In new OpenMDAO, a Solver is **not** a Driver, and only the Problem object can have a Driver. Every Group has a nonlinear solver and a linear solver. The default nonlinear solver is RunOnce, which just runs solve_nonlinear once on each of its children. The default linear solver is ScipyGMRES, just as it was in old OpenMDAO.

## Execution Order

In old OpenMDAO, execution order of the components within an Assembly is determined by a combination of the order of the names in the Driver's *workflow* attribute and the order of the data flow, which is determined automatically based on connections between components.

In new OpenMDAO, subsystems within a Group are executed in an automatically determined order based on the direction of data flow between them. You can override the automatic ordering by calling the *set_order* method on the Group, giving it a list of names of subsystems in the order that you want. The *setup* method of Problem will report any out-of-order systems that it finds.

## Running the Model

The full code for defining and running our old OpenMDAO model, leaving out the necessary imports, is the following:

```
asm = Assembly()
asm.add('comp1', Times2())
asm.add('comp2', Times2())
asm.connect('comp1.y', 'comp2.x')
asm.run()
```

The corresponding model in new OpenMDAO looks like this:

```
prob = Problem(root=Group())
prob.root.add('comp1', Times2())
prob.root.add('comp2', Times2())
prob.root.connect('comp1.y', 'comp2.x')
prob.setup()
prob.run()
```

## Support

Moving your previous models to OpenMDAO 1.0 may be a bit of work, but one that we feel will be worth the effort. If things get confusing or difficult, we're here to help. Ask conversion questions at the old forum, or email us at support@openmdao.org .