CodeTools Documentation

Release 3.0.1

Enthought

December 12, 2008

CONTENTS

The CodeTools project provide a series of modules for analysis and control of code execution. The key components are Blocks, sections of code to execute, and Contexts, dictionary-like objects which are used as namespaces for Block execution. CodeTools is designed to perform such tasks as:

- executing code in a controlled environment
- performing dependency analysis on code so that expensive recalculations can be avoided
- having Traits-aware code listen for changes inside the namespace of executing code
- automatically converting or adapting variables on access from, or assignment to, a namespace

CodeTools depends upon Traits, with an optional dependency on SciMath and Numpy for unit support, but has no major dependencies beyond that.

CodeTools is under active development, so the implementation and API are somewhat fluid.

CHAPTER

ONE

CONTENTS

1.1 CodeTools Tutorial

> Namespaces are one honking great idea — let's do more of those!
>
> —Tim Peters, *The Zen of Python* (**PEP 20**)

This tutorial introduces the key concepts behind the CodeTools modules, as well as highlighting some of their potential applications. This tutorial assumes some familiarity with Traits, and uses Chaco to illustrate some potential uses. Some familiarity with Numpy is also potentially useful for the more involved examples.

The two building-blocks of the CodeTools system are the Block class and the DataContext class (and its various subclasses).

A Block object simply holds a set of executable content, such as might be run by a Python exec command. However, unlike a simple code string, the Block object performs analysis of the code it contains so that it can identify which of its variables depend on which other variables.

A DataContext can be thought of as a Traits-aware dictionary object. The base DataContext simply emits Traits events whenever an item is added, modified or deleted. Subclasses allow more sophisticated actions to take place on access or modification.

This system of code and data objects evolved out of Enthought's scientific applications as a common pattern where a series of computationally expensive transformations and calculations needed to be repeatedly performed on different sets of data. These calculations often involved with simple permutations of inputs into the calculation blocks, and often were applied in interactive situations where responsiveness is important. The Block object, by being aware of how data flows through the code that it contains, can restrict the code that it actually needs to execute based on a permutation of an input. The DataContext, by being able to trigger Traits events whenever it changes, can call upon a Block to recalculate dependent variables, and user-interface objects can listen to the events it generates to update based upon changes in its namespace.

Subclasses of the DataContext class extend its functionality to provide functionality such as unit conversion, data masking, name translation and datatype separation.

Put together, these concepts allow the creation of applications where the science code and the underlying application code are kept almost completely separate.

Tutorial Sections

1.1.1 Blocks

Creating a `Block` object is as simple as invoking it on a string containing some Python code:

```
>>> from enthought.blocks.api import Block
>>>
>>> b = Block("""# my calculations
... velocity = distance/time
... momentum = mass*velocity
... """)
```

The code in the Block can be executed by using its `execute()` method in much the same way that the `exec()` statement works:

```
>>> global_namespace = {}
>>> local_namespace = {'distance': 10.0, 'time': 2.5, 'mass': 3.0}
>>> b.execute(local_namespace, global_namespace)
```

After this code, the variables *local_namespace* and *global_namespace* hold the same contents as if the Block's code had been executed by the `exec()` statement. In particular:

```
>>> local_namespace
{'distance': 10.0, 'mass': 3.0, 'time': 2.5, 'velocity': 4.0, 'momentum': 12.0}
```

Whenever you create a Block, it performs an analysis of the code, so the block can tell you which variables are its inputs and outputs:

```
>>> b.inputs
set(['distance', 'mass', 'time'])
>>> b.outputs
set(['velocity', 'momentum'])
```

In more complex situations, the Block object can give further useful information about the code, such as imported names and variables which may conditionally be output.

Restricting Execution

Where the Block object is unique is that it is aware of which variables are dependent on which other variables within its code. This allows you to restrict the code that is executed by specifying which input and output variables you are concerned with. This is achieved through the `restrict()` method of the Block object, which expects one or both of the following arguments:

inputs a list or tuple of input variables

outputs a list or tuple of output variables

For example:

```
>>> restricted_block = b.restrict(inputs=('mass',))
```

This restricted block consists of every line that depends on the variable *mass* in the original code block. In this case, this is the single line:

```
momentum = mass*velocity
```

Internally, the Block object maintains a representation of the code block as an abstract syntax tree from the Python standard library compiler package. This representation is not particularly human-friendly, but the `unparse()` function allows us to reconstruct the Python source:

```
>>> restricted_block.ast
Assign([AssName('momentum', 'OP_ASSIGN')], Mul((Name('mass'), Name('velocity'))))
>>> from enthought.blocks.api import unparse
>>> unparse(restricted_block.ast)
'momentum = mass*velocity'
```

This allows us to perform the minimum amount of re-calculation in response to changes in the inputs. For example, if we change *mass* in the local name space, then we only need to execute the restricted block which depends upon *mass* as input:

```
>>> local_namespace['mass'] = 4.0
>>> restricted_block.execute(local_namespace, global_namespace)
>>> local_namespace
{'distance': 10.0, 'mass': 3.0, 'time': 2.5, 'velocity': 4.0, 'momentum': 16.0}
```

On the other hand, if we are interested in calculating only a particular output, we can restrict on the outputs:

```
>>> velocity_comp = b.restrict(outpts=('velocity',))
>>> unparse(velocity_comp.ast)
'velocity = distance/time\n'
>>> velocity_comp.inputs
set(['distance', 'time'])
```

Note: Block restriction is designed to answer the questions "What do I need to compute when this changes?" or "What do I need to compute to calculate this output?" It doesn't (yet) answer the question "If I have these inputs, wha outputs can I calculate?"

Example: Rocket Science

At this point, an extended example is probably worthwhile. Consider the following code which calculates quantities involved in the motion of a rocket as it loses reaction mass:

```
from helper import simple_integral

thrust = fuel_density*fuel_burn_rate*exhaust_velocity + nozzle_pressure*nozzle_area

mass = mass_rocket + fuel_density*(fuel_volume - simple_integral(fuel_burn_rate,t))

acceleration = thrust/mass

velocity = simple_integral(acceleration, t)

momentum = mass*velocity

displacement = simple_integral(velocity, t)

kinetic_energy = 0.5*mass*velocity**2

work = simple_integral(thrust, displacement)
```

The `simple_integral()` function in the helper module looks something like this:

```
from numpy import array, ones
```

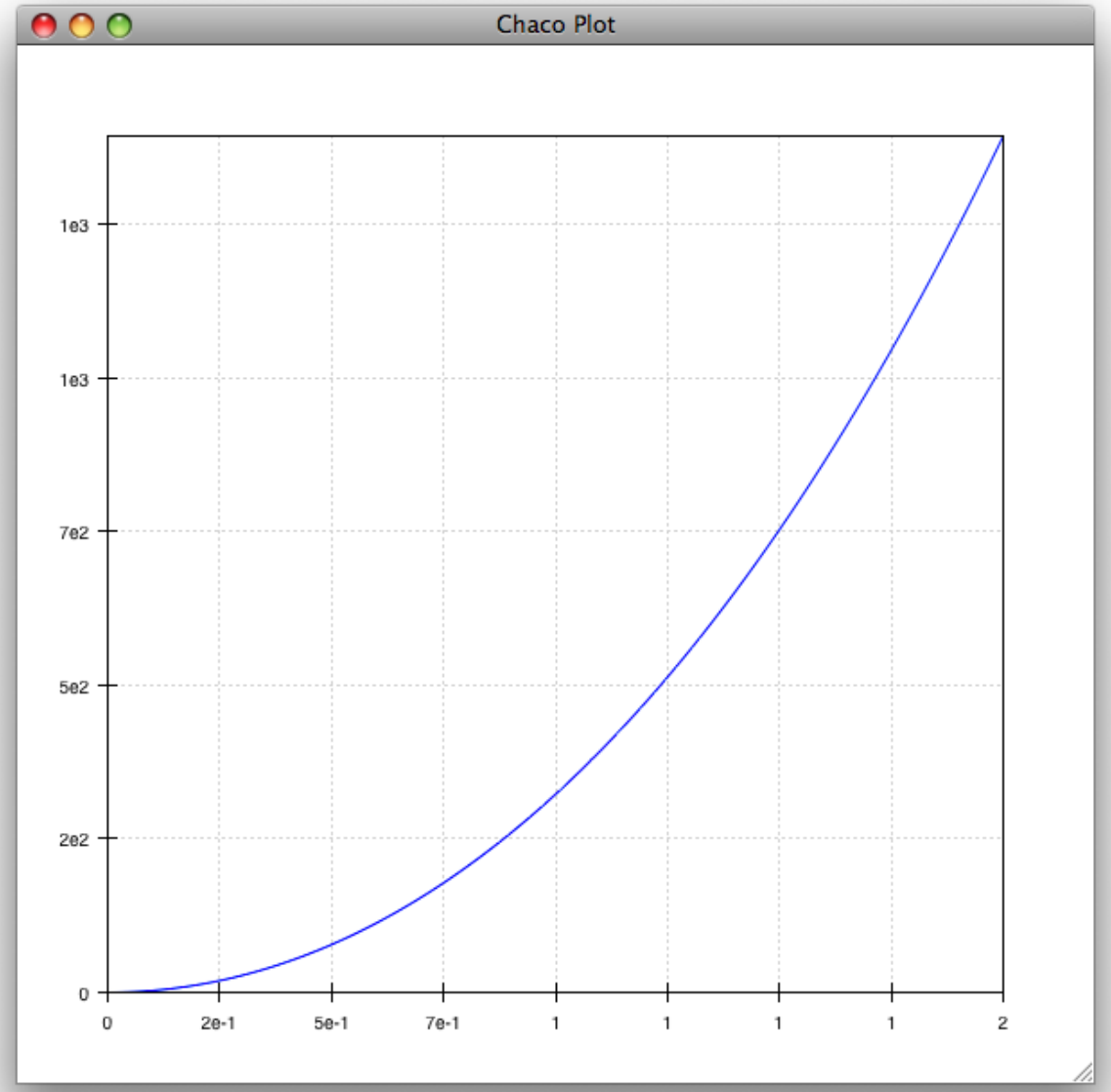
```
def simple_integral(y, x):
    """Return an array of trapezoid sums of y"""
    dx = x[1:] - x[:-1]
    if array(y).shape == ():
        y_avg = y*ones(len(dx))
    else:
        y_avg = (y[1:]+y[:-1])/2.0
    integral = [0]
    for i in xrange(len(dx)):
      integral.append(integral[-1] + y_avg[i]*dx[i])
    return array(integral)
```

Inputs to these computations are expected to be either scalars or 1-D Numpy arrays that hold the values of quantities as they vary over time. Some of these computations, particularly the `simple_integral()` computations, are potentially expensive. We can set up a Block to hold this computation:

```
>>> rocket_science = """
...     ...
... """
>>> rocket_block =     Block(rocket_science)
>>> rocket_block.inputs
set(['fuel_volume', 'nozzle_area', 'fuel_density', 'nozzle_pressure', 'mass_rocket',
'exhaust_velocity', 'fuel_burn_rate', 't'])
>>>
>>> rocket_block.outputs
set(['acceleration', 'work', 'mass', 'displacement', 'thrust', 'velocity',
'kinetic_energy', 'momentum'])
```

We can use this code by setting up a dictionary of local values for the inputs and then inspecting it:

```
>>> from numpy import linspace
>>> local_namespace = {
...     mass_rocket = 100.0,          # kg
...     fuel_density = 1000.0,        # kg/m**3
...     fuel_volume = 0.060,          # m**3
...     fuel_burn_rate = 0.030,       # m**3/s
...     exhaust_velocity = 3100.0,    # m/s
...     nozzle_pressure = 5000.0,     # Pa
...     nozzle_area = 0.7,            # m**2
...     t = linspace(0.0, 2.0, 2000) # calculate every millisecond
... }
>>> rocket_block.execute(local_namespace)
>>> print local_namespace["velocity"][::100]  # values every 0.1 seconds
[    0.            60.91584683   123.00759628   186.32154205   250.90676661
   316.81536979   384.10272129   452.82774018   523.05320489   594.84609779
   668.27798918   743.42546606   820.37061225   899.2015473    980.01303322
  1062.90715923  1147.9941173   1235.39308291  1325.23321898  1417.65482395]
>>>
>>> from enthought.chaco.shell import *
>>> plot(local_namespace[t], local_namespace["displacement"], "b-")
>>> show()
```



Restricting on Inputs

If we want to change the inputs into this calculation, say to increase the nozzle area of the rocket to 0.8 m**2 and decrease the nozzle pressure to 4800 Pa, then we don't want to have to recalculate everything. We want to calculate only the quantities which depend upon *nozzle_pressure* and *nozzle_area*. We can do this as follows:

```
>>> restricted_block = rocket_block.restrict(inputs=("nozzle_area", "nozzle_pressure"))
>>> local_namespace["nozzle_area"] = 0.7
>>> local_namespace["nozzle_pressure"] = 4800
>>> restricted_block.execute(local_namespace)
>>> print local_namespace["velocity"][::100]
[    0.           61.13047262   123.44099092   186.97801173   251.79079045
   317.93161047   385.45603658   454.42319544   524.89608665   596.9419286
```

```
   670.63254375   746.04478895   823.2610372    902.36971856   983.46592888
  1066.65211709  1152.03886341  1239.7457632   1329.90243447  1422.64966996]
>>> print local_namespace["displacement"][::100]
[    0.             3.04840167    12.27156425    27.78985293    49.72841906
    78.21749145   113.39269184   155.39537691   204.37300999   260.47956554
   323.87597012   394.73058422   473.21972951   559.52826736   653.8502346
   756.38954417   867.36075888   986.98994825  1115.51563971  1253.18987771]
```

Other values from the namespace can be extracted similarly.

The structure of the new block can be observed from its traits:

```
>>> restricted_block.outputs
set(['acceleration', 'work', 'displacement', 'thrust', 'velocity', 'kinetic_energy',
'momentum'])
>>> print unparse(restricted_block.ast)
from numpy import array, sum, ones, linspace
thrust = fuel_density*fuel_burn_rate*exhaust_velocity+nozzle_pressure*nozzle_area
acceleration = thrust/mass
velocity = simple_integral(acceleration, t)
kinetic_energy = 0.5*mass*velocity**2
displacement = simple_integral(velocity, t)
momentum = mass*velocity
work = simple_integral(thrust, displacement)
```

Restricting on Outputs

In the plot above, we only really needed to know the value of *displacement* — so to simplify the calculation of that value for the plot, we could have restricted on the output:

```
>>> restricted_block = rocket_block.restrict(outputs=("displacement",))
>>> local_namespace["mass_rocket"] = 110
>>> restricted_block.execute(local_namespace)
```

Once again, we can introspect the code block and have a look at what is actually going on:

```
>>> restricted_block.inputs
set(['fuel_volume', 'nozzle_area', 'fuel_density', 'nozzle_pressure',
'exhaust_velocity', 'mass_rocket', 't', 'fuel_burn_rate'])
>>> unparse(restricted_block.ast)
thrust = fuel_density*fuel_burn_rate*exhaust_velocity + nozzle_pressure*nozzle_area
mass = mass_rocket + fuel_density*(fuel_volume - simple_integral(fuel_burn_rate,t))
acceleration = thrust/mass
velocity = simple_integral(acceleration, t)
displacement = simple_integral(velocity, t)
```

Restricting on Both

If we wanted to go even further, and just update the plot depending on changes to just one of the inputs (say, *mass_rocket*), we could do the following:

```
>>> restricted_block = rocket_block.restrict(inputs=("mass_rocket",),
...     outputs=("displacement",))
>>> unparse(restricted_block.ast)
mass = mass_rocket + fuel_density*(fuel_volume - simple_integral(fuel_burn_rate,t))
acceleration = thrust/mass
velocity = simple_integral(acceleration, t)
displacement = simple_integral(velocity, t)
```

To really see the full power of the Block class, and to incorporate it into programs, we really need the other half of the system: the DataContext class.

1.1.2 DataContexts

The DataContext class is a HasTraits subclass that provides a dictionary-like interface, and wraps another dictionary-like object (including other DataContexts, if desired). When the DataContext is modified, the wrapper layer generates *items_modified* events that other Traits objects can listen for and react to. In addition, there is a suite of subclasses of DataContext which perform different sorts of manipulations to items in the wrapped object.

At its most basic level, a DataContext object looks like a dictionary:

```
>>> from enthought.contexts.api import DataContext
>>> d = DataContext()
>>> d['a'] = 1
>>> d['b'] = 2
>>> d.items()
[('a', 1), ('b', 2)]
```

Internally, the DataContext has a `subcontext` trait attribute which holds the wrapped dictionary-like object:

```
>>> d.subcontext
{'a': 1, 'b': 2}
```

In the above case, the subcontext is a regular dictionary, but we can pass in any dictionary-like object into the constructor, including another DataContext object:

```
>>> data = {'c': 3, 'd': 4}
>>> d1 = DataContext(subcontext=data)
>>> d1.subcontext is data
True
>>> d2 = DataContext(subcontext=d)
>>> d2.subcontext.subcontext
{'a': 1, 'b': 2}
```

Whenever a DataContext object is modified, it generates a Traits event named `items_modified`. The object returned to listeners for this event is an `ItemsModifiedEvent` object, which has three trait attributes:

`added` a list of keys which have been added to the DataContext

`modified` a list of keys which have been modified in the DataContext

`removed` a list of keys which have been deleted from the DataContext

To listen for the Traits events generated by the DataContext, you need to do something like the following:

```
from enthought.traits.api import HasTraits, Instance, on_trait_change
from enthought.contexts.api import DataContext

class DataContextListener(HasTraits):
    # the data context we are listening to
    data = Instance(DataContext)

    @on_trait_change('data.items_modified')
    def data_items_modified(self, event):
        print "Event: items_modified"
        for added in event.added:
            print "  Added:", added, "=", repr(self.data[added])
        for modified in event.modified:
            print "  Modified:", modified, "=", repr(self.data[modified])
        for removed in event.removed:
            print "  Removed:", removed
```

This class keeps a reference to a DataContext object, and listens for any `items_modified` events that it generates. When one occurs, the `data_items_modified()` method gets the event and prints the details. The following code shows the DataContextListener in action:

```
>>> d = DataContext()
>>> l = DataContextListener(data=d)
>>> d['a'] = 1
Event: items_modified
  Added: a = 1
>>> d['a'] = 'red'
Event: items_modified
  Modified: a = 'red'
>>> del d['a']
Event: items_modified
  Removed: a
```

Where this event generation becomes powerful is when a DataContext object is used as a namespace of a Block. By listening to events, we can have code which reacts to changes in a Block's namespace as they occur. Consider the simple example from the *Blocks* section used in conjunction with a DataContext which is being listened to:

```
>>> block = Block("""# my calculations
... velocity = distance/time
... momentum = mass*velocity
... """)
>>> namespace = DataContext(subcontext={'distance': 10.0, 'time': 2.5, 'mass': 3.0})
>>> listener = DataContextListener(namespace)
>>> block.execute(namespace)
Event: items_modified
  Added: velocity = 4.0
Event: items_modified
  Added: momentum = 12.0
>>> namespace['mass'] = 4.0
Event: items_modified
  Modified: mass = 4.0
>>> block.restrict(inputs=('mass',)).execute(namespace)
Event: items_modified
  Modified: momentum = 16.0
```

The final piece in the pattern is to automate the execution of the block in the listener. When the listener detects a

change in the input values for a block, it can restrict the block to the changed inputs and then execute the restricted block in the context, automatically closing the loop between changes in inputs and the resulting changes in outputs. Because the code is being restricted, only the absolute minimum of calculation is performed. The following example shows how to implement such an execution manager:

```
from enthought.traits.api import HasTraits, Instance
from enthought.blocks.api import Block
from enthought.contexts.api import DataContext

class ExecutionManager(HasTraits):
    # the data context we are listening to
    data = Instance(DataContext)

    # the block we are executing
    block = Instance(Block)

    @on_trait_change('data.items_modified')
    def data_items_modified(self, event):
        changed = set(event.added + event.modified + event.deleted)
        inputs = changed & self.block.inputs
        outputs = changed & self.block.outputs
        for output in outputs:
            print "%s: %s" % (repr(output), repr(self.data[output]))
        self.execute(inputs)

    def execute(self, inputs):
        # only execute if we have all inputs
        if self.block.inputs.issubset(set(self.data.keys())):
            self.block.restrict(inputs=inputs).execute(self.data)
```

1.1.3 The Block-Context-Execution Manager Pattern

The last example of the previous section is an instance of a pattern that has been found to work very well at Enthought for the rapid development of scientific applications. In principle it consists of 3 components:

- **Block:** an object that executes code in a namespace and knows its inputs and outputs.
- **Context:** a namespace that generates events when modified.
- **Execution Manager:** an object that listens for changes in the Context, and when a Block's inputs change, tells the Block to execute.

This is not an unfamiliar model: it's how most spreadsheet applications work. You have the spreadsheet application itself (the Execution Manager), the values contained in the cells (the Context), and the the functions and scripts which perform the computations (the Block).

The example from the previous section might seem like an awful lot of work to replace what, as far as basic computation is concerned, could be implemented with a fairly simple HasTraits class like this:

```
class DoItAll(HasTraits):
    # inputs
    distance = Float
    time = Float
    mass = Float

    # outputs
```

```
    velocity = Float
    momentum = Float

    def calculate(self):
        self.velocity = self.distance/self.time
        self.momentum = self.mass*self.velocity
```

However, the Block-Context-Execution Manager pattern gives us the following advantages:

- ability to restrict computations to the bare minimum required
- immediate feedback on computations while they occur
- automated recalculation in response to changes
- (perhaps most importantly) almost complete separation between the application code (the execution manager and UI components), the computation code, and the data

Actually, by using Traits properties you can get the first 3 of these — although the code will become more and more complex as the computations get longer.

The last point, however, is very powerful and is something that you cannot get with an all-in-one approach. It means that computational code can be written almost completely independently of application and UI code. With appropriate `try ... except ...` blocks, the inevitable errors in the calculation blocks (particularly if user-supplied) or problems with the data can be contained. The Context is well-suited to being the Model in a Model-View-Controller UI pattern, particularly since it uses Traits.

And it encourages code re-use. In fact, a well-written execution manager has the potential to be a complete application framework which can be repurposed to different domains by simply replacing the blocks that it executes. Similarly, well-written Blocks will most likely have clean libraries associated with them and can be re-used with different types of variables. (At Enthought it is common to use the same Block code with scalars in the UI and with arrays to produce plots.)

It is worth noting that the roles do not have to be kept completely separate. There are situations where bundling together the Execution Manager with the Block (to make a smart block that re-executes whenever it needs to) or a DataContext (to make a smart data set) makes sense. The ExecutingContext class in `enthought.execution.api` is precisely such an example: it combines a DataContext and a listener to automatically execute.

1.1.4 TraitslikeContextWrapper

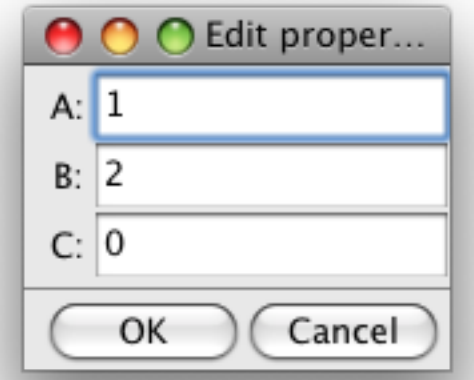
As noted, DataContexts are often put into the role of Model in a MVC UI. However, the DataContext namespace doesn't have Traits information associated with it, which can be an obstacle to its use in a Traits UI. For fairly homogeneous namespaces, or those where it is hard to know what variables will be present, one approach is to extract and wrap the individual items in the DataContext namespace and use them directly in the UI (often in a TableEditor).

However, sometimes we want the DataContext itself to appear like a regular HasTraits object. This approach involves listening to the events generated by the DataContext and using them to keep local copies of the DataContext's items synchronised with it. This pattern is sufficiently common and useful that the TraitslikeContextWrapper class is available to simplify this procedure.

To use the TraitslikeContextWrapper, you need to use the `add_traits()` method to tell it which names in the Context should appear as traits:

```
>>> from enthought.traits.api import Int
>>> from enthought.traits.ui.api import View, Item
>>> from enthought.traits.ui.menu import OKButton, CancelButton
```

```
>>> from enthought.contexts.api import DataContext, TraitslikeContextWrapper
>>> d = DataContext(subcontext={'a': 1, 'b': 2, 'z': 20})
>>> tcw = TraitslikeContextWrapper(_context=d)
>>> tcw.add_traits('a', 'b', c=Int)
>>> d.items()
[('a', 1), ('c', 0), ('b', 2), ('z', 20)]
>>> view = View(Item(name='a'), Item(name='b'), Item(name='c'),
...     buttons = [OKButton, CancelButton])
>>> tcw.configure_traits(view=view)
```



As can be seen from the window, the TraitslikeContextWrapper makes the wrapped object act just like a regular HasTraits object.

The example also demonstrates that you can add items into the DataContext object via the `add_traits()` call, and that you can specify trait types in the call (i.e., `c=Int`).

Example: Simple Block Context Application

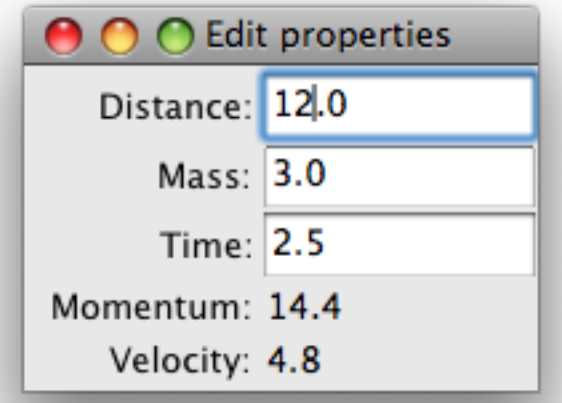
Putting TraitslikeContextWrapper together with the Block-Context-Execution Manager pattern, we can easily create simple Traits UI applications around a code block. The following is a simple but general application that can be found in the CodeTools examples:

```
"""Simple Block Context Application

This application demonstrates the use of the Block-Context-Execution Manager
pattern, together with using a TraitslikeContextWrapper to make items inside a
data context appear like traits so that they can be used in a TraitsUI app.
"""
from enthought.traits.api import HasTraits, Instance, Property, Float, \
    on_trait_change, cached_property
from enthought.traits.ui.api import View, Group, Item

from enthought.contexts.api import DataContext, TraitslikeContextWrapper
from enthought.contexts.items_modified_event import ItemsModified
from enthought.blocks.api import Block

code = """# my calculations
```

```
velocity = distance/time
momentum = mass*velocity
"""

class SimpleBlockContextApp(HasTraits):
    # the data context we are listening to
    data = Instance(DataContext)

    # the block we are executing
    block = Instance(Block)

    # a wrapper around the data to interface with the UI
    tcw = Property(Instance(TraitslikeContextWrapper), depends_on=["block", "data"])

    # a view for the wrapper
    tcw_view = Property(Instance(View), depends_on="block")

    @on_trait_change('data.items_modified')
    def data_items_modified(self, event):
        """Execute the block if the inputs in the data change"""
        if isinstance(event, ItemsModified):
            changed = set(event.added + event.modified + event.removed)
            inputs = changed & self.block.inputs
            if inputs:
                self.execute(inputs)

    @cached_property
    def _get_tcw_view(self):
        """Getter for tcw_view: returns View of block inputs and outputs"""
        inputs = tuple(Item(name=input)
                       for input in sorted(self.block.inputs))
        outputs = tuple(Item(name=output, style="readonly")
                        for output in sorted(self.block.outputs))
        return View(Group(*(inputs+outputs)),
                    kind="live")

    @cached_property
    def _get_tcw(self):
        """Getter for tcw: returns traits-like wrapper for data context"""
        in_vars = dict((input, Float) for input in self.block.inputs)
        out_vars = tuple(self.block.outputs)
        tcw = TraitslikeContextWrapper(_context=self.data)
        tcw.add_traits(*out_vars, **in_vars)
        return tcw

    def execute(self, inputs):
        """Restrict the code block to inputs and execute"""
        # only execute if we have all inputs
        if self.block.inputs.issubset(set(self.data.keys())):
            try:
                self.block.restrict(inputs=inputs).execute(self.data)
            except:
                # ignore exceptions in the block
                pass

if __name__ == "__main__":
    block = Block(code)
    data = DataContext(subcontext=dict(distance=10.0, time=2.5, mass=3.0))
```

```
    execution_manager = SimpleBlockContextApp(block=block, data=data)
    execution_manager.tcw.configure_traits(view=execution_manager.tcw_view)
```

The interface looks like this:



Notice that the SimpleBlockContextApp has no explicit knowledge of the contents either of the Block or of the DataContext other than expecting floats for the input variable values. If the code variable were replaced with any other code block, the code would work just as well.

1.1.5 MultiContexts

There is one notable issue with the application shown in *TraitslikeContextWrapper*: the UI assumes that every input is a float, and that every output should be displayed.

Suppose we try to use a slightly modified version of the code block from *Example: Rocket Science* section with that application:

```
from numpy import array, ones

def simple_integral(y, x):
    """Return an array of trapezoid sums of y"""
    dx = x[1:] - x[:-1]
    if array(y).shape == ():
        y_avg = y*ones(len(dx))
    else:
        y_avg = (y[1:]+y[:-1])/2.0
    integral = [0]
    for i in xrange(len(dx)):
      integral.append(integral[-1] + y_avg[i]*dx[i])
    return array(integral)

thrust = fuel_density*fuel_burn_rate*exhaust_velocity + nozzle_pressure*nozzle_area
mass = mass_rocket + fuel_density*(fuel_volume - simple_integral(fuel_burn_rate,t))
acceleration = thrust/mass
velocity = simple_integral(acceleration, t)
momentum = mass*velocity
```

```
displacement = simple_integral(velocity, t)
kinetic_energy = 0.5*mass*velocity**2
work = simple_integral(thrust, displacement)
```

We would discover that the function `simple_integral()` appears in the list of outputs. The reason that this function appears as an output is that, as far as a namespace is concerned, defining a function is the same as assigning to a variable. Also note that the imports *don't* appear — imported names are available in the `fromimports` trait attribute of a Block and don't appear as outputs.

So one solution to this problem is to always import functions. However there is a second problem: the variable *t* needs to be an array, not a float, and we probably shouldn't have the user interacting with it directly anyway. So we need to solve the more general problem of which outputs should be displayed.

There are several approaches to solving this problem, but perhaps the most elegant is to have the DataContext itself keep track. One way to achieve this is through the use of a MultiContext, which is a context that contains a number of subcontexts, together with rules to decide which of these it should use for a particular variable. To an external viewer, the MultiContext appears just like a DataContext, but objects can keep references to particular subcontexts that supply the information that they require.

The subcontexts need to be able to tell the MultiContext which items they can accept, and which they do not wish to store. To do this they implement the IRestrictedContext interface, which simply means that they have to provide an `allows()` method which should take a key and value as input and return True if the Context accepts the item. Regular DataContext objects implement the IRestrictedContext interface, deferring to their subcontext if it is a DataContext, but allowing any variable to be set otherwise.

Let's say that we want to have a context available which contains only variables whose values are floats. That would be done like this:

```
>>> from enthought.contexts.api import MultiContext
>>> class FloatContext(DataContext):
...     def allows(key, value):
...         return isinstance(value, float)
>>> class BContext(DataContext):
...     def allows(key, value):
...         return key[0] == "b"
>>> float_context = FloatContext()
>>> b_context = BContext()
>>> default_context = DataContext() # subcontext is a dict, so allows() is always True
>>> multi_context = MultiContext(float_context, b_context, default_context)
>>> multi_context['a'] = 34.0
>>> multi_context['b'] = 34
>>> multi_context['c'] = "Hello"
>>> multi_context.items()
[('a', 34.0), ('b', 34), ('c', 'Hello')]
>>> float_context.items()
[('a', 34.0)]
>>> b_context.items()
[('b', 34)]
>>> default_context.items()
[('c', 'Hello')]
```

Note: There are some wrinkles to the way that the MultiContext handles setting an item when multiple subcontexts will accept it:

```
>>> multi_context['c'] = 10.0
>>> multi_context['c']
10.0
```

```
>>> float_context['c']
10.0
>>> default_context['c']
'Hello'
```

There are also some wrinkles in how it handles matching keys in contexts that won't accept an item:

```
>>> multi_context['a'] = "Goodbye"
>>> multi_context['a']
"Goodbye"
>>> default_context['a']
"Goodbye"
>>> "a" in float_context
False
>>> default_context['b'] = "foo"
>>> multi_context['b'] = "bar"
>>> multi_context['b']
'bar'
>>> 'b' in default_context
True
>>> default_context['b']
'foo'
```

Note that if a context rejects an item, the MultiContext removes the key for that item from the rejecting context. If a context accepts an item, and the same key exists in later contexts (in the context list), the items with that key in the later contexts are untouched.

If this sort of behavior is not what you want, then you can easily subclass MultiContext to provide the semantics that your application requires.

Using a MultiContext in the Block-Context-Execution Manager pattern allows us to have the Execution Manager looking only at the inputs, and allows us to separate out the UI from the Execution Manager.

1.1.6 Adapted Data Contexts

Often the data inside a context needs to be transformed in some way before it is used:

- possibly in different ways by different parts on an application
- with different transformations on different variables
- with multiple different transformations applied sequentially

And the same sequence of transformations may need to be reversed when setting a value in to the context. These transformations may even need to be changed depending on the situation or a user request.

The AdaptedDataContext class provides a framework for applying transformations to data as it comes in and out of a context. AdaptedDataContext provides a list of adapters in its `adapters` trait attribute, as well as `push()` and `pop()` methods for manipulating them. On get, set, or delete operations, the AdapterManagerMixin goes through each of the adapters and first calls the `adapt_name()` method to perform any manipulations of the name, and then calls `adapt_getitem()` or `adapt_setitem()` for get and set operations respectively. Once all the adapters have had a chance to perform transformations, the operation is applied to the underlying context.

To create an adapter all you need to do is to implement the IAdapter interface by providing at least one of the `adapt_name()`, `adapt_getitem()`, or `adapt_setitem()` methods.

```
from numpy import ndarray
from numpy.fft import fft, ifft
from enthought.traits.api import HasTraits
from enthought.contexts.api import IAdapter

class FFTAdapter(HasTraits):
    implements(IAdapter)

    def adapt_getitem(self, context, key, value):
        """Take Fourier transform of 1-D Numpy arrays in the context."""
        if isinstance(value, ndarray) and len(value.shape) == 1:
            return key, fft(value)

    def adapt_setitem(self, context, key, value):
        """Take inverse Fourier transform of 1D numpy arrays when setting into
        the context."""
        if isinstance(value, ndarray) and len(value.shape) == 1:
            return key, ifft(value)
```

The `enthought.contexts` package contains a number of adapters to perform operations like masking, unit conversion, and name translation.

1.1.7 Context Functions

It should be clear by this point that the ability to use a general context as the namespace when executing code permits some complex behaviors to be implemented simply. There are circumstances where it would be useful to replace the namespace of a function with a context. The `context_function` module provides some tools for doing this.

This example shows how to set up a simple listening data context that displays changes within a function's local namespace as it executes:

```
from enthought.traits.api import HasTraits, on_trait_change
from enthought.contexts.api import DataContext, context_function

class ListeningDataContext(DataContext):
    """ A simple subclass of DataContext which listens for items_modified
    events and pretty-prints them."""

    @on_trait_change('items_modified')
    def print_event(self, event):
        print "Event: items_modified"
        for added in event.added:
            print "  Added:", added, "=", repr(self[added])
        for modified in event.modified:
            print "  Modified:", modified, "=", repr(self[modified])
        for removed in event.removed:
            print "  Removed:", removed

def f(x, t=3):
    """ A function which fires add, modify and delete events. """
    y = x+2
    y += 1
    z = '12'
    del z
    return y
```

```
f = context_function(f, ListeningDataContext)
```

The functionality is also available as a function decorator:

```
@local_context(ListeningDataContext)
def f(x, t=3):
    """ A function which will fire add, modify and delete events. """
    y = x+2
    y += 1
    z = '12'
    del z
    return y
```

The `context_function()` and `local_context()` functions both take an argument which is a context factory: each invocation of the function results in a call to the context factory. In most cases it returns a freshly created context.

1.2 CodeTools Roadmap

This document exists as a place to keep track of proposed enhancements to CodeTools and indicate where development effort could profitably be spent.

1.2.1 Things That Won't Change

- There will be a Block class:
  - that accepts code and ASTs and other Blocks for initialization.
  - that has an `execute()` method with similar semantics to an `exec()` statement
  - that has `inputs`, `outputs` and `fromimport` attributes, as well as some sort of conditional output attribute
- There will be a DataContext class very similar to the current one. I don't think that there will be much change needed in this other than changes/improvements based on changes to the traits API
- The MultiContext, AdaptedDataContext, TraitslikeContextWrapper objects should be fairly stable.

1.2.2 Blocks

Language Support

The biggest deficiency with blocks right now is that they don't fully support all Python language features, such as:

- list comprehensions
- generator expressions

A near term goal should be that code should be able to make a round-trip through a block without changing the outcome of executing it:

```
exec code
Block(code).execute()
exec unparse(Block(code).ast)
```

should (in the absence of other manipulations) do the same thing, and `unparse(Block(code).ast)` should produce code that is as close as practical to the original code block.

Speed

Currently whenever a Block is created it immediately parses the code to an AST. This is not a particularly fast operation, and can cause slowdowns if a large number of Blocks are created in quick succession. Deferring this AST generation using Traits properties would alleviate this somewhat, but would require a refactor of the Block object.

It may be worthwhile doing some serious profiling of the parser and compiler to see if there are places where speed can be improved.

New Functionality

There are almost certainly new pieces of functionality that may be worth adding:

- improved branching analysis tools — what variables depend on what other variables
- common AST manipulation routines in the spirit of enthought.blocks.rename

1.2.3 Contexts

The main area that has room for development in the context packages is adding to the provided collections of filters and adapters.

1.2.4 Execution

There should be a robust and general ExecutionManager class that listens to a Context and then executes a Block in that (or a related) Context. This would essentially split this functionality out from the ExecutingContext class.

1.2.5 New Modules

ByteCodeTools

There may be a place for a library that provides a rich and consistent interface for bytecode manipulation. Currently the only place where we do this sort of manipulation is in the `context_function` module, and there the bytecode substitution is fairly simple.

For example, there may be a more efficient way to provide the functionality of `enthought.blocks.rename` by manipulating code objects and bytecode rather than AST.

INDEX

P