

Traits Tutorial

SciPy 2010

Corran Webster

Enthought, Inc

1

Enthought Tool Suite

TRAITS

Initialization, Validation, Observation, and Visualization of Python class attributes

KIVA

2D primitives supporting path based rendering, affine transforms, alpha blending and more.

ENABLE

Object based 2D drawing canvas

CHACO

Plotting toolkit for building complex, interactive 2D plots

MAYAVI

3D Visualization of Scientific Data based on VTK

ENVISAGE

Application plugin framework for building scriptable and extensible applications

2

Enthought Tool Suite

| | | | |
|--------------------------------------|-----------------------------------|---|--|
| Chaco 2D Interactive Plots | | Mayavi 3D Interactive Plots | |
| Enable Interactive Drawing | | Envisage Plugin App Framework | |
| Kiva Traits + Agg | TraitsUI Traits + Wx/Qt | TVTK Traits + VTK | |
| Traits | | | |

All are projects are open source (BSD-style), reasonably mature and actively used and supported by Enthought and others. More information can be found at <http://code.enthought.com>

This tutorial is looking at Traits and TraitsUI.

What are traits?

Traits provide additional characteristics for Python object attributes:

- Visualization (TraitsUI)
- Notification
- Standardization
 - Initialization
 - Validation
 - Delegation

What are traits?

Traits provide additional characteristics for Python object attributes:

- Visualization (TraitsUI)
- Notification
- Standardization
 - Initialization
 - Validation
 - Delegation
- Documentation

5

A Note About Examples

- The code in the scripts do not exactly mimic the slides. It has usually been simplified for brevity.
- Most non-ui examples are simple scripts that can be run from the command line or IPython:

```
In[1] run rect_1.py
```

- Some examples have a `main()` method that must be run to get their output. This is typically done for examples that throw tracebacks as part of the demo.
- Most UI examples must be run from IPython started using the `'-wthread'` option, or from a wxPython based shell such as PyCrust.

6

Defining Simple Traits — rect_1.py

```

from enthought.traits.api import HasTraits, Float

class Rectangle(HasTraits): # <----- Derive from HasTraits
    """ Simple rectangle class with two traits.
    """

    # Width of the rectangle
    width = Float # <----- Declare Traits

    # Height of the rectangle
    height = Float # <----- Declare Traits

```

7

Defining Simple Traits — rect_1.py

```

from enthought.traits.api import HasTraits, Float

class Rectangle(HasTraits): # <----- Derive from HasTraits
    """ Simple rectangle class with two traits.
    """

    # Width of the rectangle
    width = Float # <----- Declare Traits

    # Height of the rectangle
    height = Float # <----- Declare Traits

```

Note: Run `main()` for this example to execute similar commands to those below.

```

In[1]: run rect_1.py
In[2]: main()

```

```

# Demo Code
>>> rect = Rectangle()
>>> rect.width
0.0

# Set rect width to 1.0
>>> rect.width = 1.0
>>> rect.width
1.0

```

```

# Float traits convert integers
>>> rect.width = 2
>>> rect.width
2.0

```

```

# THIS WILL RAISE EXCEPTION
>>> rect.width = "1.0"
TraitError: The 'width' trait of a
Rectangle instance must be a float,
but a value of '1.0' <type 'str'>
was specified.

```

8

Default Values — rect_2.py

```
from enthought.traits.api import HasTraits, Float

class Rectangle(HasTraits):
    """ Simple rectangle class with two traits.
    """

    # Width of the rectangle
    width = Float(1.0) # <----- Set default to 1.0

    # Height of the rectangle
    height = Float(2.0) # <----- Set default to 2.0
```

Demo Code

```
>>> rect = Rectangle()
>>> rect.width
1.0
>>> rect.height
2.0
```

Initialization via # keyword arguments

```
>>> rect = Rectangle(width=2.0,
                      height=3.0)
>>> rect.width
2.0
>>> rect.height
3.0
```

9

Coercion and Casting — rect_3.py

```
from enthought.traits.api import HasTraits, Float, CFloat

class Rectangle(HasTraits):
    """ Simple rectangle class with two traits.
    """

    # Basic traits allow "widening" coercion (Int->Float).
    width = Float

    # CFloat traits apply float() to any assigned variable.
    height = CFloat # <----- CFloat is the casting version
                  #           of the basic Float trait
```

Demo Code

```
>>> rect = Rectangle()
>>> rect.height = "2.0" # <----- This Works!
>>> rect.width = "2.0"
TraitError: The 'width' trait of a Rectangle instance must be a float,
but a value of '2.0' <type 'str'> was specified.
```

10

Traits for Basic Python Types

| Coercing Trait | Casting Trait | Python Type | Default Value |
|----------------|-----------------|---|---------------|
| Bool | CBool | bool | False |
| Complex | CComplex | complex | 0+0j |
| Float | CFloat | float | 0.0 |
| Int | CInt | int | 0 |
| Long | CLong | long | 0L |
| Str | CStr | str or unicode (whichever assigned) | ' ' |
| Unicode | CUnicode | unicode | u' ' |

Strings can take the following optional arguments beyond a default value:

- minlen:** The minimum length allowed for the string
- maxlen:** The maximum length allowed for the string
- regex:** A Python regular expression that the string must match

Traits for Basic Python Types

| Trait | Python Type | Default Value | Options |
|-----------------|----------------------|------------------|--------------------------------------|
| List | list | [] | trait, value, maxlen, minlen |
| Dict | dict | {} | key_trait, value_trait, value |
| Tuple | tuple | () | value or *types |
| Set | set | set() | trait, value |
| Instance | Class instance | None | klass, factory, args, kw, allow_none |
| Array | numpy.ndarray | array([]) | dtype, shape, value |
| Date | A datetime date | None | value |
| Time | A datetime time | None | value |

There are also traits for functions, methods, classes, modules, types, weak references, but these are much less frequently used

Other Useful Trait Types

| Trait | Description | Default Value | Options |
|-----------------|------------------------|---------------|---|
| Any | Any value | None | value |
| Enum | One of a set of values | First value | *values |
| Range | Number in a range | low | low, high, value, exclude_low, exclude_high |
| String | A validated string | ' ' | value, minlen, maxlen, regex |
| Constant | Constant value | Given value | value |
| Event | Write-only, no value | - | trait |
| HTML | HTML text | ' ' | As String |
| Code | Python code string | ' ' | As String |
| File | A filename | ' ' | value, filter, exists |
| Director | A directory name | ' ' | value, exists |
| UUID | A unique identifier | Random UUID | - |

13

Traits Speed

| Attribute Access Method | Get Attribute | | Set Attribute | |
|------------------------------|---------------|-------------|---------------|-------------|
| | Time (µs) | Speed-up | Time (µs) | Speed-up |
| Global Module Variable | 0.033 | 2.29 | 0.058 | 2.22 |
| Old Style Instance Attribute | 0.061 | 1.23 | 0.095 | 1.35 |
| New Style Instance Attribute | 0.075 | - | 0.129 | - |
| Standard Python Property | 0.499 | 0.15 | 0.593 | 0.22 |
| “Any” Traits Attribute | 0.048 | 1.56 | 0.160 | 0.80 |
| “Int” Traits Attribute | 0.054 | 1.39 | 0.174 | 0.74 |
| “Range” Traits Attribute | 0.048 | 1.56 | 0.185 | 0.70 |
| Statically Observed Trait | 0.048 | 1.56 | 1.499* | 0.09 |
| Dynamically Observed Trait | 0.048 | 1.56 | 3.118* | 0.04 |

* When value actually changes, otherwise similar to other trait's set times.

Comparison of setting various types of traits to setting standard Python class attributes and properties. (from enthought/traits/tests/check_timing.py on 2.33 GHz MacBook Pro)

14

List Traits

```
class Foo(HasTraits):  
  
    # same as List(Any)  
    a = List  
  
    # List of Strings  
    b = List(Str)  
  
    # List of Person objects  
    c = List(Instance(Person))  
  
    # List of Ints with 3-5 elements. The default  
    # value is [1,2,3]  
    d = List([1,2,3], Int, minlen=3, maxlen=5)
```

15

Dict Traits

```
class Foo(HasTraits):  
  
    # Signature: Dict(key_type, value_type)  
  
    # Basic dictionary with unchecked key/value types  
    # Same as Dict(Any, Any)  
    a = Dict  
  
    # Dictionary with checked Str key type  
    # Same as Dict(Str, Any)  
    b = Dict(Str)  
  
    # Dictionary with string for keys and floats for values  
    c = Dict(Str, Float)  
  
    # Default value specified for the dictionary  
    d = Dict(Str, Float, value={"hello": 1.0})
```

16

Array Traits

```

from numpy import float32, int32
from enthought.traits.api import Array, HasTraits

class TriangleMesh(HasTraits):

    # An Nx3 floating point array of points (vertices) within the mesh.
    points = Array(dtype=float32, shape = (None,3))

    # An Mx3 integer array of indices into the points array.
    # Each row defines a triangle in the mesh.
    triangles = Array(dtype=int32, shape=(None,3))

# Demo Code
points = numpy.array([[0,0,0], [1,0,0], [0,1,0], [0,0,1]], dtype=float32)
triangles = numpy.array([[0,1,3], [0,3,2], [1,2,3], [0,2,1]], dtype=int32)

# Demo Code
>>> tetra = TriangleMesh()
# Set the data points and connectivity
>>> tetra.points = points
>>> tetra.triangles = triangles
# THIS WILL RAISE AN EXCEPTION
>>> tetra.points = points[:, :2]
TraitError: The 'points' trait of a TriangleMesh instance must be an array of float32
values with shape ('*', 3), but a value of array([[ 0.,  0.]])

```

17

Instance Traits – instance_1.py

```

class Person(HasTraits):
    first_name = Str("John")
    last_name = Str("Doe")

    def __repr__(self): return 'Person("%s %s")' % (self.first_name, self.last_name)

class Family(HasTraits):
    # Instantiate the default Person
    dad = Instance(Person, args=())

    # Instantiate a Person object with a different first name
    mom = Instance(Person, args=(), kw={'first_name': 'Jane'})

    # Son is a Person object, but it defaults to 'None'
    son = Instance(Person)

    # In case you need "forward" declarations, you can use
    # the name as a string. Default is None
    daughter = Instance('Person')

# Demo Code
>>> family = Family()
>>> family.dad
Person("John Doe")
>>> family.mom
Person("Jane Doe")
>>> family.son
None

>>> family.daughter
None
>>> family.son = Person(first_name="Bubba")
>>> family.daughter = Person(first_name='Sissy')
>>> family.son
Person("Bubba Doe")
>>> family.daughter
Person("Sissy Doe")

```

18

Dynamic defaults — task.py

```
from datetime import date, timedelta
from enthought.traits.api import HasTraits, Str, Date

class Task(HasTraits):
    """ A task to be performed. """
    # a description of the task
    description = Str

    # the list of tasks
    due_date = Date

    def _due_date_default(self):
        """By default due date is a week from today"""
        return date.today() + timedelta(days=7)
```

Demo Code

```
>>> todo = Task(description="Attend SciPy")
>>> todo.due_date
datetime.date(2010, 7, 6)
>>> todo.due_date = date(2010, 6, 30)
>>> todo.due_date
datetime.date(2010, 6, 30)
```

19

Traits Properties

20

Properties — rect_4.py

```

from enthought.traits.api import \
    HasTraits, Float, Property

class Rectangle(HasTraits):
    """ Rectangle class with
        read-only area property.
    """
    # Width of the rectangle
    width = Float(1.0)

    # Height of the rectangle
    height = Float(2.0)

    # The area of the rectangle
    # Defined as a property.
    area = Property

    # specially named method
    # automatically associated
    # with area.
    def _get_area(self):
        return self.width * self.height
    
```

Demo Code

```

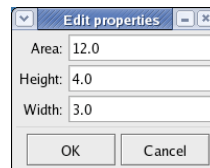
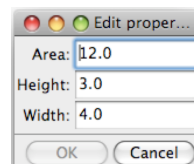
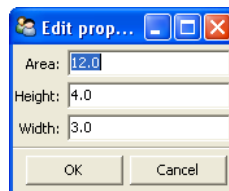
>>> rect = Rectangle(width=2.0,
                      height=3.0)
>>> rect.area
6.0
>>> rect.width = 4.0
>>> rect.area
8.0
>>> rect.area = 16.0
TraitError: The 'area' trait of a
Rectangle instance is 'read
only'.
    
```

21

Traits UI – Default Views

```

>>> rect = Rectangle(width=3.0, height = 4.0)
# Create a UI to edit the traits of the object.
>>> rect.edit_traits()
    
```



22

Properties — rect_5.py

```

from enthought.traits.api import \
    HasTraits, Float, Property

class Rectangle(HasTraits):
    """ Rectangle class with
        read-write area property.
    """
    width = Float(1.0)
    height = Float(2.0)
    area = Property

    def _get_area(self):
        return self.width * self.height

    def _set_area(self, value):
        """Adjust width to match new area"""
        self.width = value/self.height

```

Demo Code

```

>>> rect = Rectangle(width=3.0,
                      height=2.0)
>>> rect.area
6.0
>>> rect.area = 8.0
>>> rect.width
4.0

```

23

Properties Dependencies — rect_6.py

```

from enthought.traits.api import HasTraits, Float, Property
from enthought.traits.ui.api import View, Item

class Rectangle(HasTraits):

    width = Float(1.0)
    height = Float(2.0)

    # Specify dependencies with 'depends_on' meta-data.
    # This updates the area whenever width or height change.
    area = Property(depends_on=['width', 'height'])

    def _get_area(self):
        return self.width * self.height

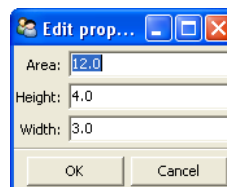
```

Demo Code

```

>>> rect = Rectangle(width=3.0,
                    height=4.0)
>>> rect.configure_traits()

```



24

Cached Properties — rect_7.py

```

from enthought.traits.api import HasTraits, Float, Property, \
    cached_property
from enthought.traits.ui.api import View, Item

class Rectangle(HasTraits):
    width = Float(1.0)
    height = Float(2.0)
    area = Property(depends_on=['width', 'height'])

    # only re-compute when depends_on traits change
    @cached_property
    def _get_area(self):
        print 'recalculating...'
        return self.width * self.height

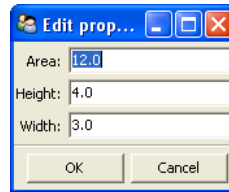
```

Demo Code

```

>>> rect = Rectangle(width=3.0,
                    height=4.0)
>>> rect.configure_traits()

```



25

Traits Validation

26

Validation: Range — rect_8.py

```
from enthought.traits.api import HasTraits, BaseFloat

class Rectangle(HasTraits):
    # the width of the rectangle (in the interval (0,1e6] )
    width = Range(0.0, 1e6, value=1.0, exclude_low=True)

    # the height of the rectangle (in the interval (0,1e6] )
    height = Range(0.0, 1e6, value=1.0, exclude_low=True)
```

Demo Code

```
>>> rect = Rectangle()
>>> rect.height = -2.0
TraitError: The 'height' trait of a Rectangle instance must be a
positive float, but a value of type 'float' (i.e. -2.0) was supplied.
>>> rect.height = 0.0
TraitError: The 'height' ...
>>> rect.height = 1e6
>>> rect.height = 2e6
TraitError: The 'height' ...
```

27

Validation: Enum — rect_9.py

```
from enthought.traits.api import HasTraits, Enum, List

class Rectangle(HasTraits):
    # the width of the rectangle
    width = Enum(1.0, 1.5, 2.0, 5.0, 10.0)

    # the allowable heights
    allowable_heights = List([0.5, 1.0, 2.0, 3.0, 4.0])

    # the height of the rectangle
    height = Enum(values='allowable_heights')
```

Demo Code

```
>>> rect = Rectangle(width=2.0)
>>> rect.height
0.5
>>> rect.height = 5.0
TraitError: The 'height' trait of a Rectangle instance must be 0.5 or
1.0 or 2.0 or 3.0 or 4.0, ...
>>> rect.allowable_heights.append(5.0)
>>> rect.height = 5.0
>>> rect.height
5.0
```

28

Validation: String — hexcolor.py

```

from enthought.traits.api import HasTraits, String, Property, \
    Tuple, Int
class ColorConverter(HasTraits):
    # a hexadecimal string
    hexcolor = String('#FFFFFF', minlen=7, maxlen=7,
        regex='#[0-9A-Fa-f]+')
    # the corresponding decimal value
    rgbcolor = Property(Tuple(Int, Int, Int), depends_on='hexcolor')

    def _get_rgbcolor(self):
        return tuple(int(self.hexcolor[i:i+2], 16)
            for i in range(0,6,2))

```

Demo Code

```

>>> color = ColorConverter()
>>> color.hexcolor = '#FFFF'
TraitError: ...
>>> color.hexcolor = '#1234567'
TraitError: ...
>>> rect.hexcolor = 'purple'
TraitError: ...

```

29

General Validation — rect_10.py

```

from enthought.traits.api import HasTraits, BaseFloat

class PositiveFloat(BaseFloat):
    default = 1.0
    info_text = "a positive float"

    def validate(self, obj, name, value):
        value = super(PositiveFloat, self).validate(obj, name, value)
        if value > 0:
            return value
        self.error(obj, name, value)

class Rectangle(HasTraits):
    width = PositiveFloat
    height = PositiveFloat(2.0)

```

Demo Code

```

>>> rect = Rectangle()
>>> rect.height = -2.0
TraitError: The 'height' trait of a Rectangle instance must be a
positive float, but a value of type 'float' (i.e. -2.0) was supplied.

```

30

Traits UI

31

TraitsUI

- Traits UI is a cross-platform (Windows, OS X, Linux, Solaris), cross-toolkit (Wx, Qt) system for writing user interfaces.
- It is based around the Model-View-Controller paradigm, where HasTraits objects are the model.
- Very easy to make basic UIs
- Powerful enough to handle full-blown applications (we write these all the time at Enthought)

32

Default UI Views — rect_11.py

```

from enthought.traits.api import HasTraits, Float, Property
from enthought.traits.ui.api import View, Item

class Rectangle(HasTraits):

    width = Float(1.0)
    height = Float(2.0)
    area = Property(depends_on=['width', 'height'])

    # Define a default view with the area as a readonly editor.
    view = View('width', 'height', Item('area', style='readonly'))

    def _get_area(self):
        return width * height

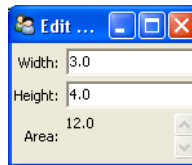
```

Demo Code

```

>>> rect = Rectangle(width=3.0,
                      height=4.0)
>>> rect.configure_traits()

```



33

Simple UI Layout — rect_11.py

```

from enthought.traits.ui.api import View, HGroup, VGroup, Item
from enthought.traits.ui.menu import OKCancelButtons

view1 = View(
    VGroup( # Create a vertical layout group.
        HGroup(# And a horizontal group within that.
            # Change the labels on each item.
            Item('width', label='w'),
            Item('height', label='h')
        ),
        Item('area', style='readonly'),
    ),
    # Add OK, Cancel buttons to the UI
    buttons=OKCancelButtons
)

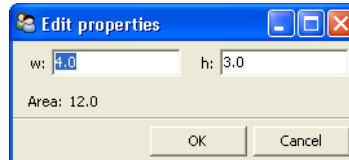
```

Demo Code

```

>>> rect = Rectangle(width=3.0,
                      height=4.0)
>>> rect.configure_traits(view=view1)

```



34

Editors

```

from enthought.traits.ui.api import View, Item, CheckListEditor

class Burger(HasTraits):
    patties = Enum(1, 2)
    cheese = Bool
    condiments = List

    view = View(
        Item('patties', style='custom'),
        Item('cheese'),
        # use the CheckListEditor instead of default
        Item('condiments', editor=CheckListEditor(cols=3,
            values=['ketchup', 'relish', 'mustard']), style='custom'
        )
    )

```

Demo Code

```

>>> burger = Burger()
>>> burger.configure_traits()

```

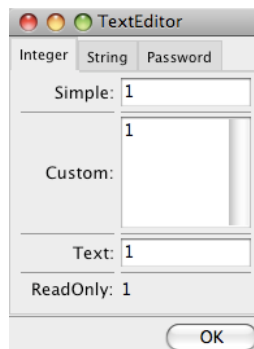
35

TextEditor

This is the default editor for most trait types. It displays a text box in which the user can enter text and the entered text is evaluated to find the trait value.

OPTIONS

- auto_set:** set the trait value on every keystroke
- enter_set:** set the value when the enter key is pressed
- password:** obscure entered text
- multi_line:** is multi-line text allowed?
- mapping:** a dictionary that maps strings to trait values
- evaluate:** a callable which takes the text and returns an appropriate trait value
- evaluate_name:** the name of a trait which is an evaluation callable



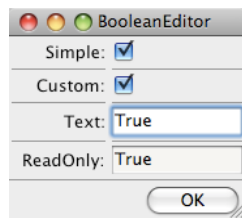
36

Basic Editors

BooleanEditor

This is the default editor for Bool and CBool.

mapping: a dictionary that maps strings to True/False values



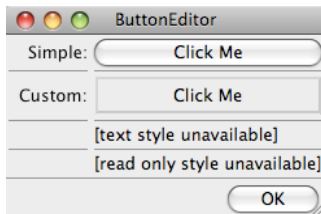
ButtonEditor

This is the default editor for Buttons.

label: the label to display on the button

value: the value to set the trait to when the button is pressed

style: the type of button



HTMLEditor

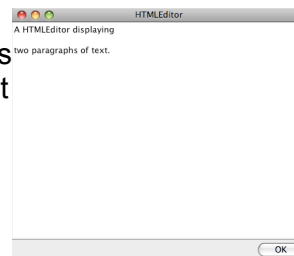
This is a wrapper around the toolkit's HTML widget, so it is either weak (Wx) or powerful (Qt).

format_text: do simple plain text to HTML formatting.

base_url: base URL to use for external links

base_url_name: name of the trait holding the base URL

open_externally: should links open in an external browser (eg. IE, Firefox, Safari, Opera, Chrome...)



Basic Editors

SetEditor

Suitable for list traits.

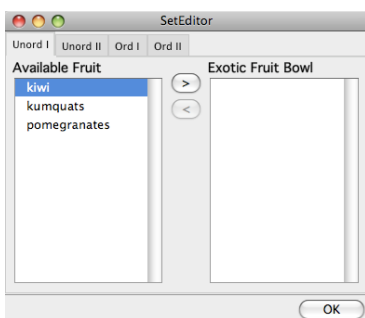
values: list of available values

name: name of the trait containing available values

ordered: are values ordered?

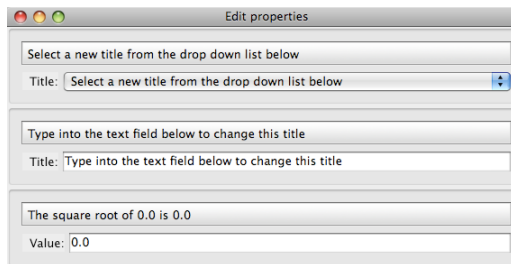
can_move_all: move all items at once?

left_column_title, right_column_title: title columns



TitleEditor

Display a string as a read-only header.



ShellEditor

A Python shell.

shared: use the object's value dictionary?

command_executed: the most recent command executed

Basic Editors

SearchEditor

This is a text editor in the style of a search control.

search_button, cancel_button: should these buttons be shown?

search_event_trait: event to fire whenever a search should be performed.

auto_set, enter_set: as TextEditor.

ValueEditor

This displays a tree of Python values.

auto_open: how many levels to show in the tree initially

ProgressEditor

Display a progress bar.

title: title to display.

message: message alongside progress bar.

min, max: the min and max values of the progress bar.

can_cancel: display a cancel button?

show_time: display estimated time?.

show_percent: display percent completion?

RangeEditor

OPTIONS

low, high: the low and high ends of the editable range (static)

low_name, high_name: names of traits holding the low and high ends of the editable range (dynamic)

low_label, high_label: labels for the low and high ends of the range

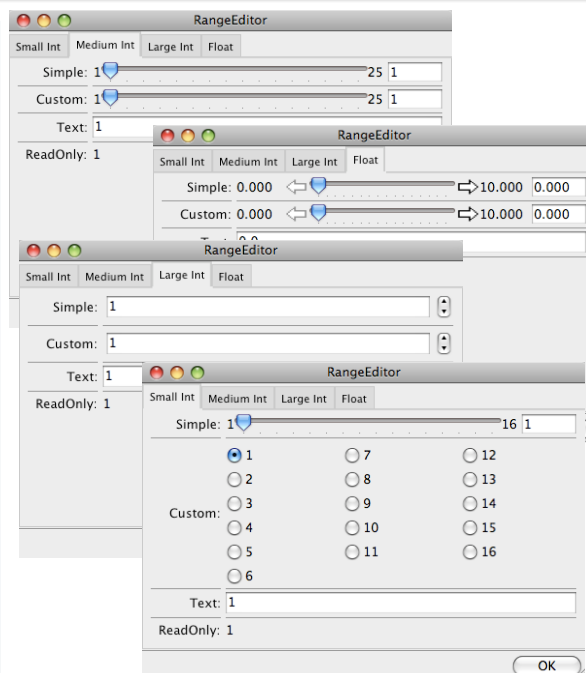
format: % format string for text values

is_float: floating point or integer values (autodetect if unset)

mode: the type of editor: auto, slider, xslider, spinner, enum, text, logslider

auto_set, enter_set: when should textual editors set the trait value

evaluate, evaluate_name: callable (or trait containing one) to evaluate values



EnumEditor

This is the default editor for an Enum, but can be used for any trait. The user can select the trait value from a pre-defined list. In the simple style, the user selects from a menu, in the custom style, the user selects the item using radio buttons or a list box.

OPTIONS

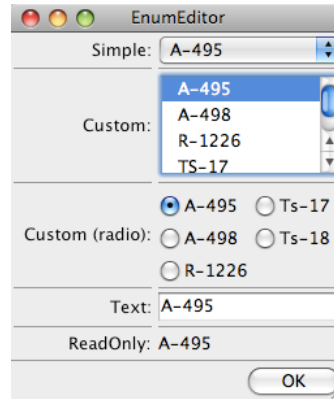
values: either a list, tuple or dictionary of values define the values that can be chosen from.

name: the name of a trait that holds a list, tuple or dictionary of names

evaluate: a function to convert textual inputs to values

cols: the number of columns to layout the radio buttons in when using the custom style (default is 1)

mode: the type of custom editor to use, one of 'radio' or 'list'



41

InstanceEditor

This is the default editor for an Instance, but can be used for any trait. This editor either displays a button which opens a new editor window for an instance, or embeds the editor's view as a panel in the current view. Don't use the text or read-only styles.

OPTIONS

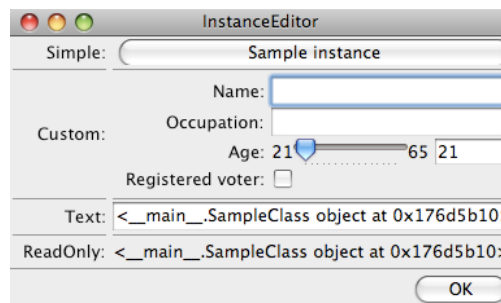
kind: the window kind to use (eg. modal, livemodal) when the simple editor opens a new window.

view: the view to use in the editor

evaluate: a function to convert textual inputs to values

cols: the number of columns to layout the radio buttons in when using the custom style (default is 1)

mode: the type of custom editor to use, one of 'radio' or 'list'



42

CodeEditor

This is a wrapper around the Scintilla (or QScintilla) text editor.

OPTIONS

auto_set: set the trait value on every keystroke

show_line_numbers: display line numbers in the margin?

lexer: the lexer to use for highlighting

selected_line: name of the trait to hold the selected line

selected_text: name of the trait to hold the selected text

line: the name of a trait which holds the line of the cursor

column: the name of a trait which holds the column of the cursor

auto_scroll: scroll to the selected line when it changes?

mark_lines: name of a trait containing a list of lines to mark

dim_lines: name of a trait containing a list of lines to dim

squiggle_lines: name of a trait containing a list of lines to put squiggles under

selected_color, mark_color, dim_color, squiggle_color: the color for each type of feature

calltip_clicked: name of an event to fire if user clicks a calltip

43

ListEditor

This is the default editor for Lists. It is designed to support all of the changes that can be made to a Python list. There is a variant style that presents items in the list as a series of tabs. For simple lists, look at CheckListEditor and ListStrEditor. For complex lists consider TableEditor or TabularEditor.

OPTIONS

editor: the editor to use for the items

style: the editor style to use for the items

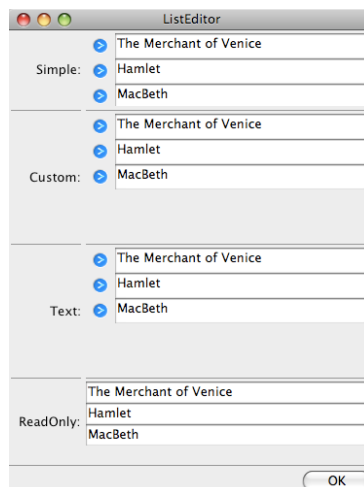
trait_handler: the handler to use for the items (see Handlers later)

mutable: can the list be re-ordered or have items added or deleted

cols: the number of columns to create

rows: the number of rows to display

use_notebook: use the notebook style of editor



44

ListEditor

OPTIONS (NOTEBOOK)

deletable: are notebook items deletable?

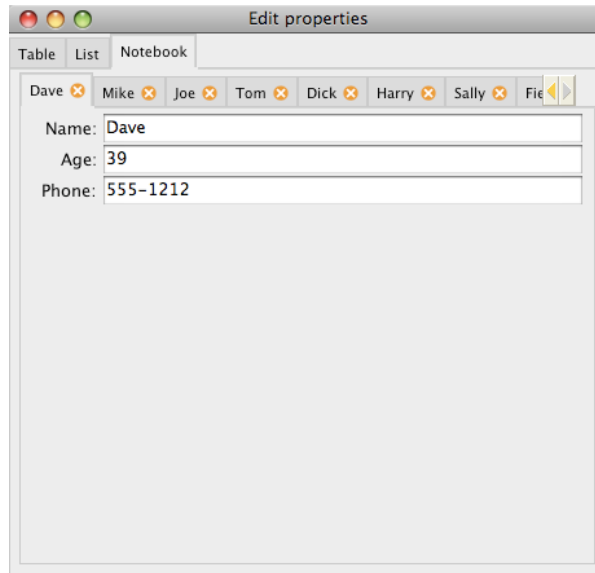
view: the view to use for the items

ui_kind: the type of UI to construct ('panel' or 'subpanel')

factory: a factory to define the actual object to be edited in the view

page_name: the extended trait name of the trait that holds the title of the notebook page

selected: the name of a trait to synchronize with the active notebook page



45

CheckListEditor

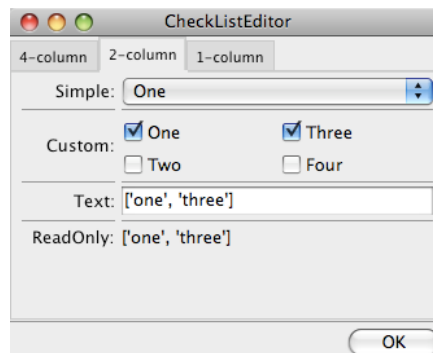
This is an alternative editor for a List. The user can select which of a pre-defined list of values should be items in the list. In the simple style, the user selects only one element from a menu, in the custom style, the user selects items using a series of checkboxes.

OPTIONS

values: either a list of values, (value, label) pairs which define the values that can be items in the list

names: a list of strings which correspond to the values (used instead of (value, label) pairs

cols: the number of columns to layout the checkboxes in when using the custom style (default is 1)



46

ListStrEditor

This is an alternative editor for a List. The items in the list are displayed as a list of strings, so this is well suited to ListStr traits.

OPTIONS

editable: are the values editable?

horizontal_lines: display horizontal lines?

title: the title to display above the list

title_name: the name of a trait to use for the title instead of a fixed value

multi_select: can multiple items be selected?

auto_add: automatically add a new item at the end of the list?

operations: a list of permitted operations chosen from 'delete', 'insert', 'append', 'edit', 'move'.

drag_move: allow re-ordering by dragging items

selected: the name of a trait to synchronize with the selection

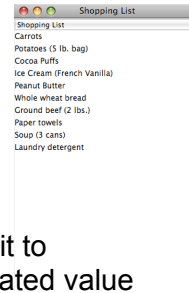
selected_index: the name of a trait to synchronize with the selected indices

activated: the name of a trait to synchronize with the activated value

activated_index: the name of a trait to synchronize with the activated index

right_click: the name of a trait to synchronize with the clicked value

right_click_index: the name of a trait to synchronize with the clicked index



47

Images and Icons

TraitsUI has a fairly sophisticated image resource management system, most of which you don't need most of the time. The basic class which represents an image is ImageResource(), which takes an image name and an optional search path.

Traits handles simple static images loaded from a file best. For more powerful and dynamic image handling, use Kiva and Enable.

```
import os
from enthought.pyface.api import ImageResource

# looks in images subdirectory
# will match 'weather-clear.bmp', 'weather-clear.png', etc.
image1 = ImageResource('weather-clear')
# specify a preferred size (optional)
image1.create_image(size=(32,32))

# looks in search_path/images
image2 = ImageResource('weather-clear', search_path=[
    os.path.join(os.path.dirname(__file__), '..', 'resources'),
    os.path.expanduser(os.path.join('~', 'images')),
])

# a standard TraitsUI image
image3 = ImageResource('@icons:red_ball')
```

48

Images and Icons

You can display an image using an ImageEditor. The ImageEditor expects to be the editor for an Image trait, or have an image argument supplied to it.

```

from enthought.traits.api import HasTraits
from enthought.traits.ui.api import Image, ImageEditor, View, Item

class ImageViewer(HasTraits):
    my_image = Image

    view = View(
        Item('my_image', ImageEditor(), style='readonly')
        Item('static image', ImageEditor(image='weather-clear'),
            style='readonly')
    )

weather_view = ImageViewer(image='weather-clear')
weather_view.configure_traits()

```

49

Other Editors

ComponentEditor

Used for Enable canvases and Chaco interactive plotting.

```

class LinePlot(HasTraits):
    plot = Instance(Plot)

    traits_view = View(
        Item('plot', editor=ComponentEditor(),
            show_label=False,
            width=500, height=500,
            resizable=True,
            title="Chaco Plot")

    def __init__(self):
        x = linspace(-14, 14, 100)
        y = sin(x) * x**3
        plotdata = ArrayPlotData(x = x, y = y)
        plot = Plot(plotdata)
        plot.plot(("x", "y"), type="line",
            color="blue")
        plot.title = "sin(x) * x^3"
        self.plot = plot

if __name__ == "__main__":
    LinePlot().configure_traits()

```

SceneEditor

This is the editor for showing a Mayavi or TVTK scene in a view.

```

from enthought.traits.api import \
    HasTraits, Instance
from enthought.traits.ui.api import \
    View, Item
from enthought.tvtk.pyface.scene_model \
    import SceneModel
from enthought.tvtk.pyface.scene_editor \
    import SceneEditor
from enthought.mayavi.core.ui.mayavi_scene \
    import MayaviScene

class MyModel(HasTraits):
    scene = Instance(SceneModel, ())

    view = View(
        Item('scene', height=400,
            show_label=False,
            editor=SceneEditor(
                scene_class=MayaviScene)
        ))

MyModel().configure_traits()

```

50

Other Editors

CustomEditor

Custom editor embeds a native toolkit control as an editor.

HistoryEditor

A text entry with a drop-down menu that holds a history of values entered by the user.

ScrubberEditor

A numeric text entry field where dragging the mouse in the editor changes the value.

PopupEditor

A text field where clicking pops up an interactive editor.

TimeEditor

A simple editor for Time traits.

DateEditor

A quite powerful editor for dates. The custom style editor can show a calendar and allow selection of days by clicking.

ColorEditor and RGBColorEditor

Two editors for colors (one is for Color traits, the other for RGBColor traits).

FontEditor

An editor for selecting a font.

FileEditor and DirectoryEditor

Editors for choosing files and directories.

ArrayEditor and ArrayViewEditor

Two editors for numpy arrays. ArrayViewEditor is built on top of TabularEditor and is much better for large arrays (particularly many rows, few columns).

51

Advanced Editors

TableEditor

The table editor provides a grid-based editor for a list of instances. You must supply either the columns or columns_name options.

OPTIONS

columns: a list of column objects

columns_name: the name of a trait which contains a list of columns

other_columns: other available columns not initially displayed

rows: the desired number of visible rows

filters: the set of available filters

filter: the currently active filter

filter_name: the name of a trait containing the currently active filter

search: filter object for searching

menu: a context menu for the table

editable: can the table be modified?

deletable: can rows be deleted?

reorderable: can rows be reordered?

configurable: can columns be configured by the user?

sortable: does clicking a header sort by that column?

sort_model: does sorting change the underlying list?

auto_add: automatically add a new row

row_factory: callable that creates a new row

53

TableEditor

OPTIONS (CONT.)

edit_view: a view for the current row's object displayed in a pane

edit_handler: a handler for the edit_view

edit_view_width, edit_view_height: the width and height of the edit_view

orientation: where the edit_view_pane is placed relative to the table

selection_mode: how should items in the table be selected?

selected: the name of a trait to keep synchronized with the selection

selected_indices: the name of a trait to keep synchronized with the selection indices

filtered_indices: the name of a trait to keep synchronized with the currently filtered items

click, dclick: the name of an Event trait to be assigned (object, column) tuples when a cell is clicked (or double-clicked)

on_select: a callable called when an item is selected

on_dclick: a callable called when an item is double-clicked

auto_size: automatically size columns?

Plus a good few more (mainly handling appearance of cells).

54

TableColumn

TableColumn is the base class for all column objects. Typically you will use one of its subclasses (most often ObjectColumn).

OPTIONS

label: the label of the column

type: the type of data in the column

visible: is the column visible?

editable: is the column editable?

droppable: does the column allow objects to be dropped on it

menu: a context menu for the column

width: the width of the column, either the absolute width in pixels if > 1, or the relative width if between 0 and 1

edit_width: the width of the column when a cell is being edited

text_font: the font for the column

text_color: the color of the text

cell_color: the background color of the cell

read_only_cell_color: the cell color when not editable

horizontal_alignment,
vertical_alignment: alignment of text in the column

renderer: an optional toolkit-specific renderer to render the contents of the cell

55

ObjectColumn

ObjectColumn is what you want to use most of the time when specifying a column. Object column displays the value of a specified trait on each element of the table's list.

OPTIONS

name: the name of the trait to use

format: a Python % format string to apply to the values

format_func: a callable that takes trait values and returns a string

editor: the editor to use when editing a cell

style: the editor style to use when editing a cell

Note that the editor is only displayed when editing, otherwise you get a textual representation of the contents. This can be surprising.

```
class Student(HasTraits):
    name = Str
    age = Int
    grade = Enum('A', 'B', 'C', 'D', 'F')

    editor = TableEditor(
        columns = [
            ObjectColumn(name='name'),
            ObjectColumn(name='age',
                          format='%d years'),
            ObjectColumn(name='grade',
                          editor=EnumEditor())
        ]
    )

class School(HasTraits):
    students = List(Instance(Student))
    view = View(
        Item('students', editor=editor)
    )
```

56

Other Column Types

ExpressionColumn

ExpressionColumn evaluates a Python expression in the namespace of an object

expression: the expression to evaluate

globals: a dictionary of additional values to use in all expressions

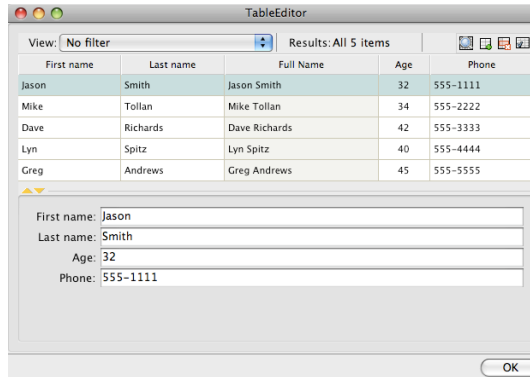
ListColumn

ListColumn selects a given index from a list.

index: the list index to use

CheckboxColumn

Checkbox column shows a column of checkboxes for a Bool trait on an object.



57

TabularEditor

The tabular editor provides an alternative grid-based editor. It is more efficient for large lists, but is not as fully featured as the TableEditor.

OPTIONS

adapter: the adapter that takes trait values to editor values

editable: can the values be edited?

show_titles: show column titles or not?

horizontal_lines, vertical_lines: show horizontal/vertical grid lines

operations: editing operations allowed, a list taken from 'delete', 'insert', 'append', 'edit', 'move'

drag_move: rows can be rearranged by dragging?

multi-select: select multiple rows at once?

selected, selected_row, activated, activated_row: names of traits to be synchronized with the appropriate values.

clicked, dclicked, right_clicked, right_dclicked, column_clicked: names of Event traits fired upon appropriate user actions. Values are instances of TabularEditorEvent.

update: name of a trait that editor listens to for when it needs an update

auto_update: if value is list of HasTraits objects, setting this to true will handle updating automatically

58

TabularAdapter

The TabularEditor requires a TabularAdapter to produce values to display in the editor. In most cases you will need to subclass TabularAdapter for your particular needs.

OPTIONS

columns: a list of (label, columnId) pairs, or label strings (in which case the columnId is the label). If columnId is a trait name, it will be the default value for the column.

Other options occur as groups of the forms: *classname_columnid_option*, *classname_option*, *columnid_option*, *option* (with that order of precedence)

text: the text value to display in the cell

content: the contents of the cell

format: a % format string for the cell

image: the image for the cell

font: the font for the cell

alignment: the alignment of the text

width: the width of the column.

text_color, odd_text_color, even_text_color, default_text_color, bg_color, odd_bg_color, even_bg_color, default_bg_color: the appropriate color of the cell.

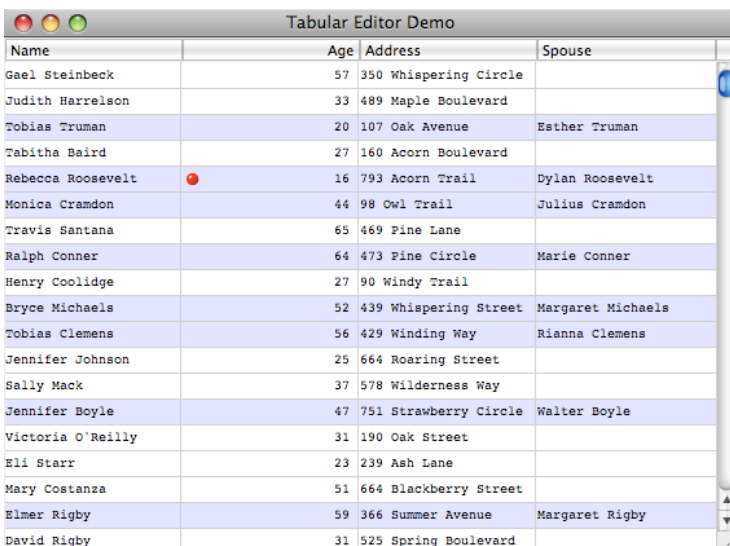
can_edit: can this cell be edited?

drag, can_drop, dropped: drag and drop properties of the cell

Frequently you will want to implement these as Property traits of a subclass.

59

TabularEditor



| Name | Age | Address | Spouse |
|-------------------|-----|-----------------------|-------------------|
| Gael Steinbeck | 57 | 350 Whispering Circle | |
| Judith Harrelson | 33 | 489 Maple Boulevard | |
| Tobias Truman | 20 | 107 Oak Avenue | Esther Truman |
| Tabitha Baird | 27 | 160 Acorn Boulevard | |
| Rebecca Roosevelt | 16 | 793 Acorn Trail | Dylan Roosevelt |
| Monica Cramdon | 44 | 98 Owl Trail | Julius Cramdon |
| Travis Santana | 65 | 469 Pine Lane | |
| Ralph Conner | 64 | 473 Pine Circle | Marie Conner |
| Henry Coolidge | 27 | 90 Windy Trail | |
| Bryce Michaels | 52 | 439 Whispering Street | Margaret Michaels |
| Tobias Clemens | 56 | 429 Winding Way | Rianna Clemens |
| Jennifer Johnson | 25 | 664 Roaring Street | |
| Sally Mack | 37 | 578 Wilderness Way | |
| Jennifer Boyle | 47 | 751 Strawberry Circle | Walter Boyle |
| Victoria O'Reilly | 31 | 190 Oak Street | |
| Eli Starr | 23 | 239 Ash Lane | |
| Mary Costanza | 51 | 664 Blackberry Street | |
| Elmer Rigby | 59 | 366 Summer Avenue | Margaret Rigby |
| David Rigby | 31 | 525 Spring Boulevard | |

TreeEditor

The tree editor provides a collapsable tree widget to display hierarchical structures. It displays 2 panes by default: the tree widget and a traits view of the selected object.

OPTIONS

nodes: a list of node objects that the tree editor can use (required).

editable: can the tree be modified?

orientation: position of the editor pane relative to the tree widget

hide_root: whether or not to display the root node in the tree widget

auto_open: the number of levels deep in the tree to open automatically

auto_close: only allow one expanded node at a time

show_icons: should nodes display icons?

selected: the trait name of a trait that is synchronized with the current selection

on_select: a callable that is invoked when the user selects an object

on_click: a callable that is invoked when the user clicks an object

on_dclick: a callable that is invoked when the user double-clicks an object

click: Event trait name of fired on clicks

dclick: Event trait name of fired on double-clicks

veto: an Event trait name that the TreeEditor listens to so you can veto default behaviour

61

TreeNode

Much of the functionality of the tree editor is supplied by the TreeNode instances in the tree's node trait.

OPTIONS

node_for: a list of classes which this node will be used for (the first matching TreeNode in the TreeEditor's node trait is used).

children: the name of the trait that holds the list of children of the node (or " for a leaf node)

label: the name of the trait that holds the text label to use (if it starts with '=' then the label is constant)

formatter: a callable that formats the label into a string

view: the View to use for this node's editor (use View() for no editor)

name: the name for a new node in menu

rename: can children be renamed?

copy: can children be copied?

delete: can children be deleted?

insert: can children be inserted?

rename_me: can these nodes be renamed?

delete_me: can these nodes be deleted?

add: list of classes that can be added to this node's children

move: list of classes that can be drag & dropped from elsewhere to the children (defaults to same as add),

62

TreeNode

OPTIONS

- menu:** a context menu for this type of node.
- icon_path:** search path for icon resources
- icon_leaf:** the icon to use when the node has no children
- icon_group:** the icon to use when the node has children
- icon_open:** the icon to use when the node is being viewed/edited
- on_select:** callable called when node is selected
- on_click:** callable called when node is clicked
- on_dclick:** callable called when node is double-clicked

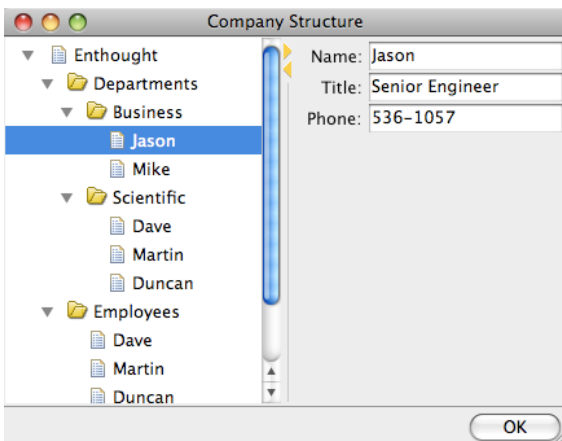
ALTERNATIVES

When you need more sophisticated interactions with nodes in a TreeEditor, you have a number of options:

- Subclass **TreeNode** or implement the **ITreeNode** interface.
- **ObjectTreeNode:** for classes which are subclasses of **TreeNodeObject** and implement its API, you can use **ObjectTreeNode** instead of **TreeNode**
- **ITreeNodeAdapter:** create a subclass of **ITreeNodeAdapter** which adapts the class you are interested to **ITreeNode**.

More information can be found in the Traits API docs and examples.

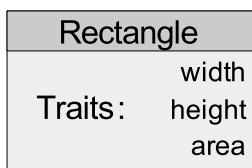
TreeEditor



Notification

Trait Listeners

HasTraits Object
 All traits automatically support the Listener Pattern



Listener Functions and Methods
 Listeners are called in order whenever the 'width' trait changes



Static Trait Notification—amplifier_1.py

```
class Amplifier(HasTraits):
    """ Guitar Amplifier Model
    """

    # Volume setting for the amplifier.
    volume = Range(0.0, 11.0, default=5.0)

    # Static observer method called whenever volume is set.
    def _volume_changed(self, old, new):
        if new == 11.0:
            print "This one goes to eleven"

# Demo Code
>>> spinal_tap = Amplifier()
>>> spinal_tap.volume = 11.0
This one goes to eleven
>>> spinal_tap.volume = 11.0 # nothing is printed because
                             # the value didn't change.
```

67

Valid Static Trait Notification Signatures

```
def _volume_changed(self):
    # no arguments...

def _volume_changed(self, new):
    # new - the just-set value of volume

def _volume_changed(self, old, new):
    # old - the previous value of volume
    # new - the just-set value of volume

def _volume_changed(self, name, old, new):
    # name - the name of the trait ('volume')
    # old - the previous value of volume
    # new - the just-set value of volume

# This signature is usually used for the
# _anytrait_changed() observer that is called
# whenever any trait is changed on the object.
```

68

Dynamic Trait Notification – amplifier_2.py

```
class Amplifier(HasTraits):
    """ Guitar Amplifier Model
    """

    # Volume setting for the amplifier.
    volume = Range(0.0, 11.0, default=5.0)

def printer(value):
    print "new value:", value

# Demo Code
>>> spinal_tap = Amplifier()
# In the following, name can also be a list of trait names
>>> spinal_tap.on_trait_change(printer, name='volume')
>>> spinal_tap.volume = 11.0
new value: 11.0
```

69

Valid Dynamic Trait Notification Signatures

```
def observer():
    # no arguments...

def observer(new):
    # new – the new value of the changed trait

def observer(name, new):
    # name – the name of the changed trait
    # new – the new value of the changed trait

def observer(object, name, new):
    # object – the object containing the changed trait
    # name – the name of the changed trait
    # new – the new value of the changed trait

def observer(object, name, old, new):
    # object – the object containing the changed trait
    # name – the name of the changed trait
    # old – the previous value of the changed trait
    # new – the new value of the changed trait
```

70

Dynamic Trait Notification — amplifier_3.py

```
class Amplifier(HasTraits):
    """ Guitar Amplifier Model
    """
    # Volume setting for the amplifier.
    volume = Range(0.0, 11.0, default=5.0)

class Listener:
    """ Class that will listen to the Amplifier volume
    """
    def printer(self, value):
        print "new value:", value

# Demo Code
>>> spinal_tap = Amplifier()
>>> listener = Listener()
>>> spinal_tap.on_trait_change(listener.printer, name='volume')
>>> spinal_tap.volume = 11.0
new value: 11.0

# on_trait_change has a weak reference to the class method.
# When the class goes away, the method no longer fires.
>>> del listener
>>> spinal_tap.volume = 10.0
```

71

@on_trait_change decorator — amplifier_4.py

```
from enthought.traits.api import HasTraits, Range, on_trait_change

class Amplifier(HasTraits):
    """ Guitar Amplifier Model
    """

    volume = Range(0.0, 11.0, value=5.0)
    reverb = Range(0, 10.0, value=5.0)

    # The on_trait_change decorator can listen to multiple traits
    # Note the "list" of traits is specified as a string.
    @on_trait_change('reverb, volume')
    def update(self, name, value):
        print 'trait %s updated to %s' % (name, value)

# Demo Code
>>> spinal_tap = Amplifier()
>>> spinal_tap.volume = 11.0
trait volume updated to 11.0
>>> spinal_tap.reverb = 2.0
trait reverb updated to 2.0
```

72

Listeners on a different thread — amplifier_5.py

```
class Amplifier(HasTraits):
    volume = Range(0.0, 11.0, value=5.0)

    def __init__(self, *args, **kw):
        super(Amplifier, self).__init__(*args, **kw)

        # Use the "dispatch" keyword to run the listener on a
        # different thread.
        self.on_trait_change(self.update, 'volume', dispatch='new')

    def update(self, name, value):
        print 'thread %s sleeping for 1 second' % thread.get_ident()
        sleep(1.0)
        print 'trait %s updated to %s' % (name, value)
```

Demo Code

```
>>> spinal_tap = Amplifier()
>>> spinal_tap.volume = 11.0
main thread: -1601355872
thread identity -1340948480 sleeping for 1 second
trait volume updated to 11.0
```

73

Event Traits – event_1.py

```
class Rectangle(HasTraits):

    # Width of the rectangle
    width = Float(1.0)

    # Height of the rectangle
    height = Float(2.0)

    # Set to notify others that you have made changes to a Rectangle
    # Listen to this if you want to react to any changes to a Rectangle
    updated = Event

    def rect_printer(rect, name, value):
        print 'rectangle (width, height): rect.width, rect.height'
```

Demo Code

```
>>> rect = Rectangle()
# Hook up a dynamic listener to respond whenever rect is updated.
>>> rect.on_trait_change(rect_printer, name='updated')
# update multiple items
>>> rect.width = 10
>>> rect.height = 20
# now explicitly tell the item that it is updated
>>> rect.updated = True
rectangle (width, height): 10.0 20.0
```

74

List Trait – school_class_1.py

```
class SchoolClass(HasTraits):

    # List of the students in the class
    students = List(Str) # <-- List of strings

    def _students_changed(self, old, new): # <-- called when list replaced
        print "The entire class has changed:", new

    def _students_items_changed(self, event): # <-- called when list items changed
        """ event.added -- A list of the items added to students
            event.removed -- A list of the items removed from students
            event.index -- Start index of the items that were added/removed
        """
        if event.added: print "added (index,name):", event.index, event.added
        else: print "removed (index,name):", event.index, event.removed

# Demo Code
>>> school_class = SchoolClass()
>>> school_class.students = ["John", "Jane", "Jill"] # initial set of students.
The entire class has changed: ['John', 'Jane', 'Jill']
>>> school_class.students.append("Bill") # add a student
students added (index,name): 3 ['Bill']
>>> del school_class.students[1:3] # remove some students
students removed (index,name): 1 ['Jane', 'Jill']
```

75

Dynamic List Trait – school_class_2.py

```
class SchoolClass(HasTraits):

    # List of the students in the class
    students = List(Str)

    @on_trait_change('students[]')
    def _students_changed(self, obj, name, old, new):
        if name == 'students':
            print "The entire class has changed:", new
        else:
            if event.added:
                print "added (index,name):", event.index, event.added
            else:
                print "removed (index,name):", event.index, event.removed

# Demo Code
>>> school_class = SchoolClass()
>>> school_class.students = ["John", "Jane", "Jill"] # initial set of students.
The entire class has changed: ['John', 'Jane', 'Jill']
>>> school_class.students.append("Bill") # add a student
students added (index,name): 3 ['Bill']
>>> del school_class.students[1:3] # remove some students
students removed (index,name): 1 ['Jane', 'Jill']
```

76

Instance List — instance_observer_1.py

```
class Person(HasTraits):
    name = Str
    age = Int

class SchoolClass(HasTraits):
    teacher = Instance(Person)
    students = List(Person)

    def _age_changed_for_teacher(self, object, name, old, new):
        print 'The teacher is now ', new, ' years old.'

    def _age_changed_for_students(self, object, name, old, new):
        print object.name, ' is now ', new, ' years old.'
```

Demo Code

```
the_class = SchoolClass()
teacher_ben = Person(name="Ben",
                    age=35)
the_class.teacher = teacher_ben
bob = Person(name="Bob", age=10)
the_class.students.append(bob)
jane = Person(name="Jane", age=11)
the_class.students.append(jane)
```

```
# This calls the SchoolClass observer
teacher_ben.age = 36
The teacher is now 36 years old.
# This calls the SchoolClass observer
>>> bob.age = 11
Bob is now 11 years old.
# Remove Bob from the Class and the
# observer is no longer called.
>>> the_class.students.remove(bob)
>>> bob.age = 12
```

77

Dynamic Instance List — instance_observer_2.py

```
class Person(HasTraits):
    name = Str
    age = Int

class SchoolClass(HasTraits):
    teacher = Instance(Person)
    students = List(Person)

    @on_trait_change('teacher.age')
    def teacher_age_change(self, object, name, old, new):
        print 'The teacher is now ', new, ' years old.'

    @on_trait_change('students:age')
    def students_age_change(self, object, name, old, new):
        print object.name, ' is now ', new, ' years old.'
```

Demo Code

```
the_class = SchoolClass()
teacher_ben = Person(name="Ben",
                    age=35)
the_class.teacher = teacher_ben
bob = Person(name="Bob", age=10)
the_class.students.append(bob)
jane = Person(name="Jane", age=11)
the_class.students.append(jane)
```

```
# This calls the SchoolClass observer
teacher_ben.age = 36
The teacher is now 36 years old.
# This calls the SchoolClass observer
>>> bob.age = 11
Bob is now 11 years old.
# Remove Bob from the Class and the
# observer is no longer called.
>>> the_class.students.remove(bob)
>>> bob.age = 12
```

78

Extended Trait Names

You can refer to more than just a trait on a current object. The following modifiers allow you to specify more general things to listen to with `on_trait_change`.

TRAIT NAMES

item1.item2: a trait called `item1` contains an object or objects with a trait called `item2`. Changes to either will fire the notification.

item1:item2: a trait called `item1` contains an object or objects with a trait called `item2`. Only changes to `item2` will fire the notification.

[item1, item2, ..., itemN]: Changes to any trait in the list will fire the notification.

item1[]: a trait called `item1` is a list. Changes to `item1` or the list will fire the notification.

anytrait or **+**: matches any trait name.

item1?: don't raise an error if the object being listened to does not have a trait called `item1`.

prefix+: fire a notification on a change to any trait whose name starts with `prefix`.

+metadata_name: fire a notification on a change to any trait with metadata item `metadata_name`.

-metadata_name: fire a notification on a change to any trait without metadata item `metadata_name`.

(item1).item2: `item1` is optional, will fire on changes to `item1`, `item1.item2` or `item2`.

*pattern**: match object graphs where `pattern` occurs one or more times.

79

Delegation – instance_delegate.py

```
class Person(HasTraits):
    first_name = Str("John")
    last_name = Str("Doe")

    def __repr__(self):
        return 'Person("%s %s")' % (self.first_name, self.last_name)

class Child(Person):
    parent = Instance(Person, args=())

    # Define last_name to "delegate" to the parent's last name
    last_name = DelegatesTo('parent', 'last_name')

# Demo Code
>>> dad = Person(first_name="Sam", last_name="Barns")
>>> child = Child(first_name="Jane", parent=dad)
>>> dad
Person("Sam Barns")
>>> child
Person("Jane Barns")
# Changing the child's last name changes the parent's as well
>>> child.last_name = "Smith"
>>> print dad, child
Person("Sam Smith") Person("Jane Smith")
```

80

Prototyping – instance_prototype.py

```
class Person(HasTraits):
    first_name = Str("John")
    last_name = Str("Doe")

    def __repr__(self):
        return 'Person("%s %s")' % (self.first_name, self.last_name)

class Child(Person):
    parent = Instance(Person, args=())

    # Define last_name to "prototype" using the parent's last name
    last_name = PrototypeFrom('parent', 'last_name')

# Demo Code
>>> dad = Person(first_name="Sam", last_name="Barns")
>>> child = Child(first_name="Jane", parent=dad)
>>> dad
Person("Sam Barns")
>>> child
Person("Jane Barns")
# Copy on write, but linked before that point
>>> child.name = "Smith"
>>> print dad, child
Person("Sam Barns") Person("Jane Smith")
```

81

More Traits UI

82

edit_traits()

The `edit_traits()` and `configure_traits()` methods take the following arguments.

ARGUMENTS

- view:** the view to use (if not the default)
- kind:** the kind of the view
- handler:** the handler for the view
- parent:** the parent ui to embed a panel into.
- context:** the context for the ui.

UI OBJECTS

The `edit_traits()` and `configure_traits()` methods return a UI object. For modal dialogs the UI object has a result trait that is True if the dialog was not cancelled.

The UI also has traits keeping track of the main moving parts of a user interaction: the context, view, handler and UIInfo, as well as the toolkit's control.

KINDS

- modal:** the view is in a modal dialog, object updated when dialog closes (or apply button pressed)
- livemodal:** the view is a modal dialog, object is updated live
- nonmodal:** the view is a nonmodal dialog, object updated on close or apply.
- live:** nonmodal, live updates.
- panel:** a view embedded in another window, has command buttons.
- subpanel:** a view embedded in another window, no command buttons
- wizard:** a wizard dialog. Wizards are live and modal, but with their own standard button set. Each group in the view is a page in the wizard.

83

View

There are a quite a number of traits available to control the appearance and behaviour of the View.

TRAITS

- height, width:** the requested height and width as pixels or proportion of screen
- x, y:** the requested x and y coordinates for the window (positive for top/left, negative for bottom/right, either pixels or proportions)
- resizable:** can the view be resized?
- scrollable:** should scroll bars be added if the size of the window is too small
- title:** the title to display on the window
- icon:** the icon to display on the window
- dock:** how to arrange subgroups: 'fixed', 'horizontal', 'vertical' or 'tabbed'.
- image:** the image to display in tabs
- close_result:** what to return if the window is closed via the window's close button
- handler:** a handler for the view (see later)
- menubar:** the menubar for the view
- toolbar:** a toolbar for the view
- statusbar:** a statusbar for the view
- buttons:** the buttons in the view
- key_bindings:** key bindings for the view
- style:** the default style for editors
- id:** a globally unique id for preferences
- object:** the object being edited

84

Group

Items in view can be linked together in Groups, and layed out in various ways.

TRAITS

height, width: the requested height and width as pixels or proportion of screen

padding: the amount of padding about the group

show_border: should a border be shown or not

label: the label to display on the group

layout: the layout style of the group, one of 'normal', 'flow', 'split' or 'tabbed'

orientation: the orientation of the group

columns: the number of columns in the group.

dock: dock style of sub-groups

show_labels: show labels of items?

show_left: show labels on the left or the right

selected: in a tabbed layout, should this be the visible tab?

image: image to show on tabs

springy: use extra space in the parent layout?

defined_when: expression that determines inclusion of group in parent

visible_when: expression that determines visibility of group

enabled_when: expression that determines whether of group can be edited

Item

Items are the basic units of layout, and usually contain trait editors.

TRAITS

label: the label to display on the item

name: the name of the trait being edited

editor: the editor to use

style: the editor style

height, width: the requested height and width as pixels or proportion of screen

padding: the amount of padding about the group

resizable: can the item be resized to use extra space

springy: use extra space in the parent layout?

full_size: expand to use all available space in non-layout direction?

has_focus: should this item get initial focus?

emphasized: should label text be emphasized?

tooltip: tooltip to display on mouse-over

image: image to show on tabs

format_str: % format string for text

format_func: format function for text

invalid: trait name for holding editor state

defined_when: expression that determines inclusion of group in parent

visible_when: expression that determines visibility of group

enabled_when: expression that determines whether of group can be edited

Subclasses

Several subclasses of Group and Item are defined for convenience

GROUP SUBCLASSES

- HGroup:** a horizontally arranged group
- HFlow:** a horizontally arranged group that flows vertically once horizontal space is used up
- HSplit:** a horizontally arranged group with a splitter bar to separate it from other groups
- Tabbed:** a group displayed as a notebook tab
- VGroup:** a vertically arranged group
- VFlow:** as HFlow, but vertical
- VFold:** a vertically arranged group where items can be collapsed by clicking their titles
- VGrid:** a grid, default 2 columns

ITEM SUBCLASSES

- Label:** an item which is a label
- Heading:** an item which is a heading
- Spring:** an item that injects springy space into a layout
- Custom:** an item with a custom editor style
- ReadOnly:** an item with readonly editor style
- UItem, UCustom, UReadOnly:** as Item, Custom or ReadOnly, but with no label.

A Word On Using Traits Themes

Don't!

Traits UI Handlers

89

Handlers

Handlers react to UI events like changes to text fields, button-clicks and menu selections.

ADDING A HANDLER TO A VIEW

```
# Handlers are a subclass of
# Handler
class MyHandler(Handler):
    ...

# add them to a view via the
# handler argument (shared by
# all views)
view = View(
    ...
    handler=MyHandler(),
)

# or supply it as part of the
# edit_traits call
obj.edit_traits(view=view,
                handler=MyHandler())
```

REACTING TO TRAIT CHANGES

Handlers can react to trait changes by overriding the global setattr method or using specially named methods for each trait.

```
class MyHandler(Handler):
    def setattr(self, info,
                obj, name, value):
        print name, value

    def object_temp_changed(self,
                             info):
        print info.object.temp
```

The info argument is a UIInfo instance, which has 'object' and 'ui' attributes corresponding to the edited object and view's ui.

90

Handlers

HANDLER LIFE-CYCLE

```
class MyHandler(Handler):
    # called when UIInfo created
    def init_info(self, info):
        ...

    # called when UI created but
    # before it is displayed
    def init(self, info):
        ...
        return True # UI created

    def position(self, info):
        ...

    # called after the window is
    # closed for clean-up
    def closed(self, info, is_ok):
        ...
```

STANDARD BUTTONS

```
class MyHandler(Handler):
    # called when 'cancel' or 'OK'
    # clicked or closed by window
    # manager
    def close(self, info, is_ok):
        ...

    # called when 'revert' clicked
    def revert(self, info):
        ...

    # called when 'apply' clicked
    def apply(self, info):
        ...

    # called when 'help' clicked
    def show_help(self, info,
                  control):
        ...
```

91

Controller

Instead of having a Handler attached to a View, handlers can have default Views (just like HasTraits classes). In these cases two subclasses make life easier

CONTROLLER

Controller objects are designed to assist with the traditional MVC pattern.

Controller handlers have the object being edited as their 'model' trait and the UIInfo available as their 'info' trait.

They inject their model into their view as the object being edited, which means that names in the view refer to the model object. The controller is available as 'controller' or 'handler'.

```
rect = Rect()
controller = RectController(
    model=rect)
controller.configure_traits()
```

EXAMPLE

```
class Rect(HasTraits):
    h = Float
    w = Float

class RectControl(Controller):
    area = Float

    def setattr(self, info, obj,
                name, value):
        if name in ['h', 'w']:
            self.area = \
                self.model.h * \
                self.model.w

view = View(Item('h'),
            Item('w'),
            Item('controller.area'))
```

92

ModelView

At Enthought we've found that sometimes it's better to have a handler which wraps and simplifies the model for the view. This pattern has been dubbed 'ModelView'.

MODELVIEW

ModelView handlers have the object being edited as their 'model' trait and the UIInfo available as their 'info' trait.

They inject themselves into their view as the object being edited, which means that names in the view refer to the ModelView. The model is available as 'model'.

```
reactor = Reactor(
    core_temperature=200.0)
view = ReactorModelView(
    model=reactor)
view.edit_traits(view=my_view)
```



EXAMPLE

```
class Reactor(HasTraits):
    core_temperature = Range(-273.0,100000.0)

class ReactorModelView(ModelView):
    # The "dummy" view of the reactor should
    # be a warning string.
    core_temperature = Property(depends_on=
        'model.core_temperature')

    def _get_core_temperature(self):
        temp = self.model.core_temperature
        if temp <= 500.0:
            return 'Normal'
        if temp < 2000.0:
            return 'Warning'
        return 'Meltdown'

my_view = View(Item('core_temperature',
    style = 'readonly'))
```

93

Actions

Buttons, menu items and toolbar buttons are all implemented by Actions which call methods on the handler.

OPTIONS

action: the name of the handler method to call when the action is invoked

visible_when: an boolean expression that will hide the UI component when False

enabled_when: an boolean expression that will disable the UI component when False

checked_when: an boolean expression that will put a checkmark next to the menu item if True

defined_when: an boolean expression that will create the UI item if True (evaluated once at creation time).

name: the name, displayed on buttons and in menus

image: an image to display where appropriate

accelerator: a keyboard short-cut to invoke the action

tooltip: a tooltip to display when the mouse hovers over the UI component

style: the button style ('push', 'toggle' or 'radio').

on_perform: a callable to use instead of the named method on the handler.

94

Buttons

Views have a button area at the bottom of the view window. The contents are a list of Actions

USAGE

Traits UI provides the following buttons for use with the default handler: OKButton, CancelButton, ApplyButton, RevertButton and UndoButton which are. Any other Action can be used as a button, provided that something will handle it.

Buttons are added to a view via the buttons argument, which expects a list of Actions.

EXAMPLE

```
class Rect(HasTraits):
    h = Float
    w = Float

    calculate = Action(name='Calculate',
                      action='do_calculate')

class RectControl(Controller):
    area = Float

    def do_calculate(self, info):
        self.area = self.model.h * \
            self.model.w

    view = View(Item('h'),
                Item('w'),
                Item('controller.area'),
                buttons=[OKButton, CancelButton,
                       calculate])

rect = Rect()
controller = RectControl(model=rect)
controller.configure_traits()    95
```

ToolBars

Views have a toolbar area at the top of the view window. This should be a Toolbar object whose contents are a list of Actions or ActionGroups.

OPTIONS

image_size: the size of the toolbar button icons (default is 16x16).

show_tool_names: whether to show the names of actions under their icons

show_divider: whether to show the divider that separates the toolbar from the rest of the view

orientation: whether the toolbar should be 'horizontal' or 'vertical'.

enabled: is the toolbar active?

visible: is the toolbar visible?

The toolbar can be accessed in a controller via `self.info.ui.view.toolbar` to change its state.

EXAMPLE

```
class Rect(HasTraits):
    h = Float
    w = Float

    calculate = Action(name='Calculate',
                      action='do_calculate',
                      image=ImageResource('calculator'),
                      tooltip='Calculate the area')

class RectControl(Controller):
    area = Float

    def do_calculate(self, info):
        self.area = self.model.h * \
            self.model.w

    view = View(
        ...
        toolbar=ToolBar(calculate))

rect = Rect()
controller = RectControl(model=rect)
controller.configure_traits()    96
```

Menus

Views also have a menubar for displaying pull-down menus. This should be a MenuBar object whose contents are a list of Menu which contain Actions or ActionGroups.

MENU OPTIONS

name: the name to display in the menubar

enabled: is the menu active?

visible: is the menu visible?

The menubar is accessible from a controller as `self.info.ui.view.menubar`.

Traits provides the following actions for use in Menus: CloseAction, UndoAction, RedoAction, RevertAction, HelpAction, Separator. There is a StandardMenuBar object which provides these arranged in menus.

EXAMPLE

```
class Rect(HasTraits):
    h = Float
    w = Float

    calculate = Action(name='Calculate',
                      action='do_calculate')

    menubar = MenuBar(
        Menu(CloseAction, name='File'),
        Menu(UndoAction, RedoAction, Separator,
             calculate, name='Edit'),
        Menu(HelpAction, name='Help'))

class RectControl(Controller):
    ...
    view = View(
        ...
        menubar=menubar)

rect = Rect()
controller = RectControl(model=rect)
controller.configure_traits()
```

97

KeyBindings

Views can have key bindings associated with them. These are set as a keyword option on the View. These are independent of menu item accelerators.

KEYBINDING OPTIONS

binding1: the first key sequence

binding2: an optional alternative key sequence

method_name: the method name on the handler to call when the binding fires

description: a user-friendly description

KEYBINDINGS OPTIONS

bindings: a list of KeyBinding instances

prefix: an optional prefix to apply to each method name

suffix: an optional suffix to apply to each method name

EXAMPLE

```
class Rect(HasTraits):
    h = Float
    w = Float

    key_bindings = KeyBinding(
        KeyBinding(binding1='Ctrl-=',
                   method_name='do_calculate',
                   description='Calculate area'))

class RectControl(Controller):
    ...
    def do_calculate(self, info):
        self.area = self.model.h * \
                    self.model.w

    view = View(
        ...
        keybindings=keybindings)

rect = Rect()
controller = RectControl(model=rect)
controller.configure_traits()
```

98

Drag And Drop

Traits relies on the underlying toolkit for a lot of its drag and drop support. There is a special editor DNDEditor for Wx which allows you to drag and drop class instances.

OPTIONS

image: an image to display in the editor

disabled_image: an image to display when the target is disabled

The behaviour of the DNDEditor depends on its style. Simple editors can be both dragged onto and dragged from. Custom editors can be dragged onto but not dragged from, and readonly editors can only be dragged from.

EXAMPLE

```
class DNDExample(HasTraits):
    drag_source = Any('drag me!')
    drop_target = Any

    def _anytrait_changed(self,
                          name, old, new):
        print name, old, new

    view = View(
        Item('drop_target',
            editor=DNDEditor(),
            style='custom')
        Item('drag_source',
            editor=DNDEditor(),
            style='readonly')
    )
```

99

Drag And Drop

There is also a editor DropEditor for both Wx and Qt which allows you to set its value by dropping a value onto it.

OPTIONS

klass: the allowable classes for drop instances

readonly: should the text field be read only?

A droppable class can supply a drop_editor_value() method to give an appropriate textual representation. Alternatively it can supply a drop_editor_update method which takes the underlying UI control and performs appropriate actions.

EXAMPLE

```
class DropExample(HasTraits):
    drop_target = Any

    view = View(
        Item('drop_target',
            editor=DropEditor(),
        )
```

100

Helpful resources

- Docs
<http://code.entthought.com/projects/traits/documentation.php>
- Mailing List Help
entthought-dev@entthought.com
- Wiki
<https://svn.entthought.com/entthought/wiki/Traits>