Traits UI User Guide

Lyn Pierce
Janet Swisher

Document Version 3

12-Sep-2007

Enthought, Inc.
515 Congress Avenue
Suite 2100
Austin TX 78701
1.512.536.1057 (voice)
1.512.536.1059 (fax)
http://www.enthought.com
info@enthought.com

Table of Contents

1 Introduction

This guide is designed to act as both tutorial and reference for Traits UI, an open-source package built and maintained by Enthought, Inc. The Traits UI package is a set of GUI (Graphical User Interface) tools designed to complement Traits, another Enthought open-source package that provides manifest typing, validation, and change notification for Python. This guide is intended for readers who are already moderately familiar with Traits; those who are not may wish to refer to the *Traits User Manual* for an introduction.

1.1 The Model-View-Controller (MVC) Design Pattern

A common and well-tested approach to building end-user applications is the MVC ("Model-View-Controller") design pattern. In essence, the MVC pattern the idea that an application should consist of three separate entities: a *model*, which manages the data, state, and internal ("business") logic of the application; one or more *views*, which format the model data into a graphical display with which the end user can interact; and a *controller*, which manages the transfer of information between model and view so that neither needs to be directly linked to the other. In practice, particularly in simple applications, the view and controller are often so closely linked as to be almost indistinguishable, but it remains useful to think of them as distinct entities.

The three parts of the MVC pattern correspond roughly to three classes in the Traits and Traits UI packages.

- Model: HasTraits class (Traits package)
- View: View class (Traits UI package)
- Controller: Handler class (Traits UI package)

The remainder of this section gives an overview of these relationships.

1.1.1 The Model: HasTraits Subclasses and Objects

In the context of Traits, a model consists primarily of one or more subclasses or instances of the HasTraits class, whose trait attributes (typed attributes as defined in Traits) represent the model data. The specifics of building such a model are outside the scope of this manual; please see the *Traits User Manual* for further information.

1.1.2 The View: View Objects

A view for a Traits-based application is an instance of a class called, conveniently enough, View. A View object is essentially a display specification for a GUI window or panel. Its contents are defined in terms of instances of two other classes: Item and Group.[1] These three classes are described in detail in Chapters 3 through 5 of this manual; for the moment, it is important to note that they are all defined independently of the model they are used to display.

Note that the terms "view" and "View" are distinct for the purposes of this document. The former refers to the component of the MVC design pattern; the latter is a Traits UI construct.

1.1.3 The Controller: Handler Subclasses and Objects

The controller for a Traits-based application is defined in terms of the Handler class.[2] Specifically, the relationship between any given View instance and the underlying model is managed by an instance of the Handler class. For simple interfaces, the Handler can be implicit. For example, none of the examples in Chapters 2 through 5 includes or requires any specific Handler code; they are managed by a default Handler that performs the basic operations of window initialization, transfer of data between GUI and model, and window closing. Thus, a programmer new to Traits UI need not be

[1] A third type of content object, Include, is discussed briefly in Chapter 5, but presently is not commonly used.

[2] Not to be confused with the TraitHandler class of the Traits package, which enforces type validation.

concerned with Handlers at all. Nonetheless, custom handlers can be a powerful tool for building sophisticated application interfaces, as discussed in Chapter 6.

1.2 Structure of this Guide

The intent of this guide is to present the capabilities of the Traits UI package in usable increments, so that you can create and display gradually more sophisticated interfaces from one chapter to the next. Thus, Chapters 2 through 5 show how to construct and display views from the simple to the elaborate, while leaving such details as GUI logic and widget selection to system defaults. Chapter 6 explains how to use the Handler class to implement custom GUI behaviors, as well as menus and toolbars. Chapters 7 through 9 show how to control GUI widget selection by means of *trait editors*. Chapters 10 and 11 cover miscellaneous additional topics. Further reference materials, including a glossary of terms and an API summary for the Traits UI classes covered in this Guide, are located in the Appendices.

2 Introduction to the View Object

A simple way to edit (or simply observe) the attribute values of a HasTraits object in a GUI window is to call the object's configure_traits()[3] method. This method constructs and displays a dialog box containing editable fields for each of the object's trait attributes. For example, the following sample code defines the SimpleEmployee class, creates an object of that class, and constructs and displays a GUI for the object:

Example 1: Using configure_traits()

```
# Sample code to demonstrate configure_traits()
from enthought.traits.api import HasTraits, Str, Int
import enthought.traits.ui

class SimpleEmployee(HasTraits):
    first_name = Str
    last_name = Str
    department = Str

    employee_number = Str
    salary = Int

sam = SimpleEmployee()
sam.configure_traits()
```

Unfortunately, the resulting form simply displays the attributes of the object **sam** in alphabetical order with little formatting, which is seldom what is wanted:

[3] If the code is being run from a program that already has a GUI defined, then use edit_traits() instead of configure_traits(). These methods are discussed in more detail in Section 5.4.

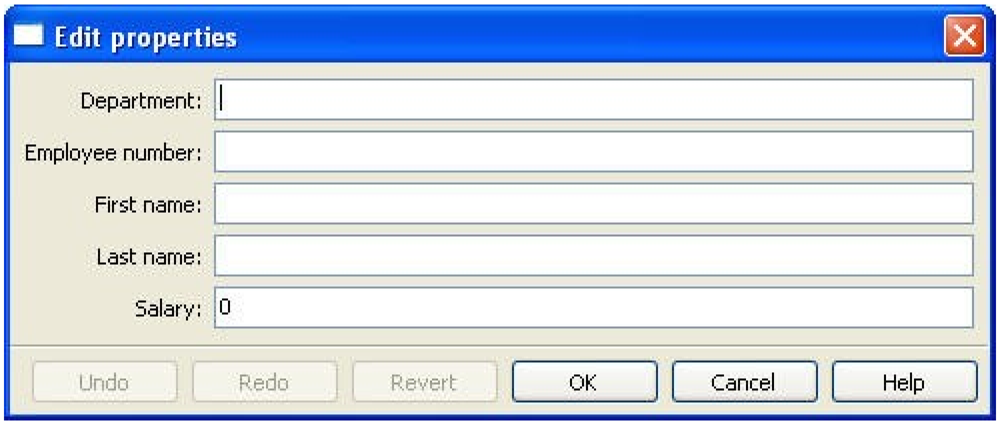


Figure 1: User interface for Example 1

In order to control the layout of the interface, it is necessary to define a View object. A View object is a template for a GUI window or panel. In other words, a View specifies the content and appearance of a TraitsUI window or panel display[4].

For example, suppose you want to construct a GUI window that shows only the first three attributes of a SimpleEmployee (e.g., because salary is confidential and the employee number should not be edited). Furthermore, you would like to specify the order in which those fields appear. You can do this by defining a View object and passing it to the configure_traits() method:

Example 2: Using configure_traits() with a View object

```
# Sample code to demonstrate configure_traits()
from enthought.traits.api import HasTraits, Str, Int
from enthought.traits.ui.api import View, Item
import enthought.traits.ui

class SimpleEmployee(HasTraits):
    first_name = Str
    last_name = Str
    department = Str
    employee_number = Str
    salary = Int

view1 = View(Item(name = 'first_name'),
             Item(name = 'last_name'),
             Item(name = 'department'))

sam = SimpleEmployee()
sam.configure_traits(view=view1)
```

The resulting window has the desired appearance:

[4] A `View` can also specify window behavior to a limited extent; see Sections 3.1.5 and 3.2.3.

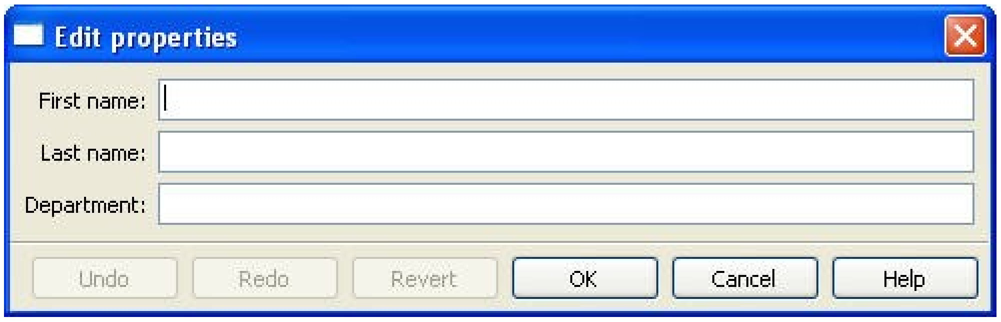


Figure 2: User interface for Example 2

Chapters 3 through 5 explore the contents and capabilities of Views. Refer to the *Traits API Reference* for details of the View class.

For the time being, all example code uses the configure_traits() method; a detailed description of this and other techniques for creating GUI displays from Views can be found in Section 5.3 on page 22.

3 The Building Blocks of a View

The contents of a View are specified primarily in terms of two basic building blocks: Item objects (which, as suggested by Example 2, correspond roughly to individual trait attributes), and Group objects. A given View can contain one or more objects of either of these types, which are specified as arguments to the View constructor as in the case of the three Items in Example 2.

The remainder of this chapter describes the Item and Group classes.

3.1 The Item Object

The simplest building block of a View is the Item object. An Item specifies a single interface widget, usually the display for a single trait attribute of a HasTraits object. The content, appearance, and behavior of the widget are controlled by means of the Item object's attributes, which are usually specified as keyword arguments to the Item constructor, as in the case of **name** in Example 2.

The remainder of this section describes the attributes of the Item object, grouped by categories of functionality. It is not necessary to understand all of these attributes in order to create useful Items; many of them can usually be left unspecified, as their default values are adequate for most purposes. Indeed, as demonstrated by earlier examples, simply specifying the name of the trait attribute to be displayed is often enough to produce a usable result.

Table 1 lists the attributes of the Item class, organized by functional categories. Refer to the *Traits API Reference* for details on the Item class.

Table 1: Attributes of Item, by category

Category	***Attributes***	***Description***
Content	• **name** • **object**	These attributes specify the actual data to be displayed by an item. Because an Item is essentially a template for displaying a single trait, its name attribute is nearly always specified. By contrast, it is often not necessary to assign the

Category	*Attributes*	*Description*
		object attribute: its default value is the literal string 'object', which is assumed to represent the object whose configure_traits() method is using the View. In Example 2 on page 5, the object attribute of all three items is assumed to be 'object', which represents the object **sam**. Scenarios where an Item's object attribute must be explicitly assigned are described in Chapter 5.
Display format	• **label** • **resizable** • **emphasized** • **padding** • **height** • **width**	In addition to specifying which trait attributes are to be displayed, you might need to adjust the format of one or more of the resulting widgets. If an Item's label attribute is specified but not its name, the value of label is displayed as a simple, non-editable string. (This feature can be useful for displaying comments or instructions in a Traits UI window.)
Content format	• **format_str** • **format_func**	In some cases it can be desirable to apply special formatting to a widget's contents rather than to the widget itself. Examples of such formatting might include rounding a floating-point value to two decimal places, or capitalizing all letter characters in a license plate number.

Category	*Attributes*	*Description*
Widget override	• **editor** • **style**	These attributes override the widget that is automatically selected by Traits UI. These relatively advanced options are discussed in Chapters 7 through 9.
Visibility and status	• **enabled_when** • **visible_when** • **defined_when**	Use these attributes to create a simple form of a dynamic GUI, which alters the display in response to changes in the data it contains. More sophisticated dynamic behavior can be implemented using a custom Handler (see Chapter 6).
User help	• **tooltip** • **help** • **help_id**	These attributes provide guidance to the user in using the user interface. If the **help** attribute is not defined for an Item, a system-generated message is used instead. The **help_id** attribute is ignored by the default help handler, but can be used by a custom help handler.
Unique identifier	• **id**	This option is needed only for advanced Traits UI applications; see Section 9.1 on page 103 for an example of its use. An id value can be used for persisting user preferences about the widget.

3.1.1 Subclasses of Item

The Traits UI package defines the following subclasses of Item:

- Label
- Heading
- Spring

These classes are intended to help with the layout of Traits UI View, and need not have a trait attribute associated with them. See the *Traits API Reference* for details.

3.2 The Group Object

The preceding sections have shown how to construct windows that display a simple vertical sequence of widgets using instances of the View and Item classes. For more sophisticated interfaces, though, it is often desirable to treat a group of data elements as a unit for reasons that might be visual (e.g., placing the widgets within a labeled border) or logical (activating or deactivating the widgets in response to a single condition, defining group-level help text). In Traits UI, such grouping is accomplished by means of the Group object.

Consider the following enhancement to Example 2:

Example 3: Using configure_traits() with a View and a Group object

```
# Sample code to demonstrate configure_traits()
from enthought.traits.api import HasTraits, Str, Int
from enthought.traits.ui.api import View, Item, Group
import enthought.traits.ui

class SimpleEmployee(HasTraits):
    first_name = Str
    last_name = Str
    department = Str

    employee_number = Str
    salary = Int

view1 = View(Group(Item(name = 'first_name'),
                   Item(name = 'last_name'),
                   Item(name = 'department'),
                   label = 'Personnel profile',
                   show_border = True))
```

```
sam = SimpleEmployee()
sam.configure_traits(view=view1)
```

The resulting window shows the same widgets as before, but they are now enclosed in a visible border with a text label:

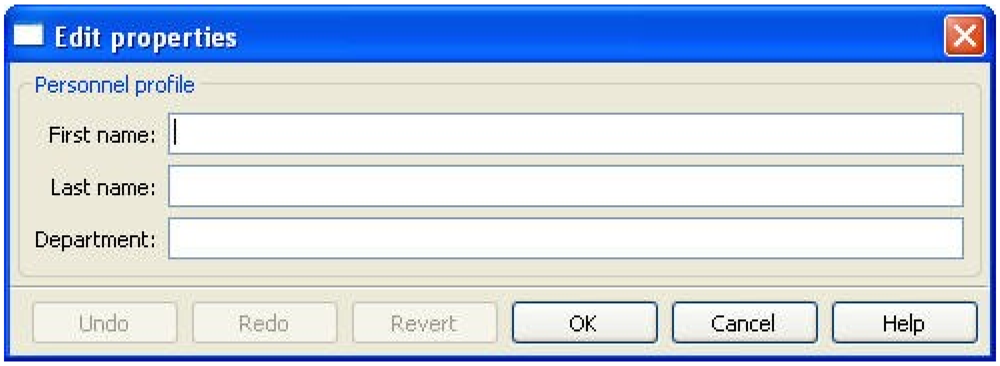


Figure 3: User interface for Example 3

3.2.1 Content of a Group

The content of a Group object is specified exactly like that of a View object. In other words, one or more Item or Group objects are given as arguments to the Group constructor, e.g., the three Items in Example 3.[5] The objects contained in a Group are called the *elements* of that Group.

3.2.2 Group Attributes

Table 2 lists the attributes of the Group class, organized by functional categories. As with Item attributes, many of these attributes can be left unspecified for any given Group, as the default values usually lead to acceptable displays and behavior.

See the *Traits API Reference* for details of the Group class.

[5] As with Views, it is possible for a Group to contain objects of more than one type, but not recommended (see Section 3.2.5).

Table 2: Attributes of Group, by category

Category	***Attributes***	***Description***
Display format	• **label** • **show_border** • **show_labels** • **show_left** • **padding** • **layout** • **selected.** • **orientation** • **style**	These attributes define display options for the group as a whole.
Visibility and status	• **enabled_when** • **visible_when** • **defined_when**	These attributes work similarly to the attributes of the same names on the Item class.

User help	• **help**
	• **help_id**
		The help text is used by the default help handler only if the group is the only top-level group for the current View. For example, suppose help text is defined for a Group called group1. The following View shows this text in its help window:
		`View(group1)`
		The following two do not:
		`View(group1, group2)`
		`View(Group(group1))`
		The **help_id** attribute is ignored by the default help handler, but can be used by a custom help handler.

Unique identifier	• **id**	This option is needed only for advanced Traits UI applications; see Section 9.1 on page 103 for an example of its use.

4 Customizing a View

As shown in the preceding two chapters, it is possible to specify a window in Traits UI simply by creating a View object with the appropriate contents. In designing real-life applications, however, you usually need to be able to control the appearance and behavior of the windows themselves, not merely their content. This chapter covers a variety of options for tailoring the appearance of a window that is created using a View, including the type of window that a View appears in, the command buttons that appear in the window, and the physical properties of the window.

4.1 Specifying Window Type: the kind Attribute

Many types of windows can be used to display the same data content. A form can appear in a dialog box, a wizard, or an embedded panel; dialog boxes can be *modal* (i.e., stop all other program processing until the box is dismissed) or not, and can interact with live data or with a buffered copy. In Traits UI, a single View can be used to implement any of these options simply by modifying its **kind** attribute. There are seven possible values of **kind**:

- 'modal'
- 'live'
- 'livemodal'
- 'nonmodal'
- 'wizard'
- 'panel'
- 'subpanel'

These alternatives are described below. If the **kind** attribute of a View object is not specified, the default value is 'modal'.

4.1.1 Dialog Boxes: 'modal', 'live', 'livemodal', and 'nonmodal'

As mentioned above, the behavior of a Traits UI dialog box can vary over two significant degrees of freedom. First, it can be *modal*, meaning that when the dialog box appears, all other GUI interaction is suspended until the dialog box is closed; if it is not modal, then both the dialog box and the rest of the GUI remain active and responsive. Second, it can be *live*, meaning that any changes that the user makes to data in the dialog box is applied directly and immediately to the underlying model object or objects; otherwise the changes are made to a copy of the model data, and are only copied to the model when the user commits them (usually by clicking an **OK** or **Apply** button; see Section 4.2 on page 15). The four possible combinations of these behaviors correspond to four of the possible values of the 'kind' attribute of the View object, as shown Table 3.

Table 3: Matrix of Traits UI dialog boxes

	not modal	modal
not live	nonmodal	modal
live	live	livemodal

All of these dialog box types are identical in appearance. Also, all types support the **buttons** attribute, which is described in Section 4.2.

4.1.2 Wizards

Unlike a dialog box, whose contents generally appear as a single page or a tabbed display, a wizard is presented as a series of pages that a user must navigate sequentially.

Traits UI Wizards are always modal and live. They always display a standard wizard button set; i.e., they ignore the **buttons** View attribute. In short, wizards are considerably less flexible than dialog boxes, and are primarily suitable for highly controlled user interactions such as software installation.

4.1.3 Panels and Subpanels

Both dialog boxes and wizards are secondary windows that appear separately from the main program display, if any. Often, however, you might need to create a window element that is embedded in a larger display. For such cases, the **kind** of the corresponding View object should be 'panel' or 'subpanel'.

A *panel* is very similar to a dialog box, except that it is embedded in a larger window, which need not be a Traits UI window. Like dialog boxes, panels support the **buttons** View attribute, as well as any menus and toolbars that are specified for the View (see Section 6.2.3 on page 32). Panels are always live and nonmodal.

A *subpanel* is almost identical to a panel. The only difference is that subpanels do not display command buttons even if the View specifies them.

4.2 Command Buttons: the buttons Attribute

A common feature of many windows is a row of command buttons along the bottom of the frame. These buttons have a fixed position outside any scrolled panels in the window, and are thus always visible while the window is displayed. They are usually used for window-level commands such as committing or cancelling the changes made to the form data, or displaying a help window.

In Traits UI, these command buttons are specified by means of the View object's **buttons** attribute, whose value is a list of buttons to display.[6] Consider the following variation on Example 3:

Example 4: Using a View object with buttons

```
# Sample code to demonstrate configure_traits()
from enthought.traits.api import HasTraits, Str, Int
from enthought.traits.ui.api import View, Item
from enthought.traits.ui.menu import OKbutton, CancelButton

class SimpleEmployee(HasTraits):
    first_name = Str
    last_name = Str
    department = Str

    employee_number = Str
    salary = Int

view1 = View(Item(name = 'first_name'),
             Item(name = 'last_name'),
```

[6] Actually, the value of the **buttons** attribute is really a list of Action objects, from which GUI buttons are generated by Traits UI. The Action class is described in Section 6.2.

```
             Item(name = 'department'),
             buttons = [OKButton, CancelButton])

sam = SimpleEmployee()
sam.configure_traits(view=view1)
```

The resulting dialog box has the same content as before, but only two buttons are displayed at the bottom: **OK** and **Cancel**:

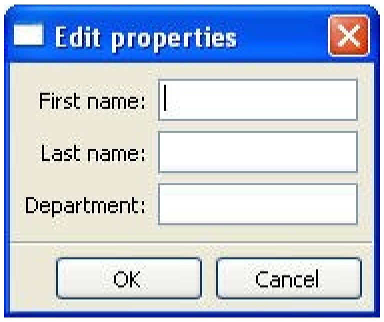


Figure 4: User interface for Example 4

Notice that the automatically sized dialog box is significantly smaller than the one in the original example. See Section 4.3 on page 17 for details on how to override automatic sizing.

There are six standard buttons defined by Traits UI:

- **UndoButton**
- **ApplyButton**
- **RevertButton**
- **OKButton**.
- **CancelButton** –
- **HelpButton**.

Alternatively, there are several pre-defined button lists that can be imported from **enthought.traits.ui.menu** and assigned to the **buttons** attribute:

- **OKCancelButtons** = [OKButton, CancelButton]
- **ModalButtons** = [ApplyButton, RevertButton, OKButton, CancelButton, HelpButton]
- **LiveButtons** = [UndoButton, RevertButton, OKButton, CancelButton, HelpButton]

Thus, one could rewrite the highlighted lines in Example 4 as follows, and the effect would be exactly the same:

```
            buttons = OKCancelButtons
```

Finally, the special constant **NoButtons** can be used to create a dialog box or panel without command buttons. Note that `buttons = NoButtons` is *not* equivalent to `buttons = []`. Setting the **buttons** attribute to an empty list has the same effect as not defining it at all: the default buttons (see Figure 1 through Figure 3) are used.

It is also possible to define custom buttons and add them to this list; see Section 6.2.3.2 on page 33 for details.

4.3 Other View Attributes

Table 4: Attributes of View, by category

Category	***Attributes***	***Description***
Window display	• **x** • **y** • **width** • **height** • **title** • **resizable** • **scrollable** • **style** • **dock**	These attributes control the visual properties of the window itself, regardless of its content.
Command	• **handler** • **menubar** • **toolbar**	Traits UI menus and toolbars are generally implemented in conjunction with custom Handlers; see Section 6.2.3.2 on page 33 for details.
User help	• **help** • **help_id**	The **help** attribute is a deprecated way to specify that the View has a Help button. Use the buttons attribute instead (see Section 4.2 on page 15 for details).
Unique identifier	• **id**	

5 Advanced View Concepts

The preceding chapters of this Guide give an overview of how to use the View class to quickly construct a simple window for a single HasTraits object. This chapter explores a number of more complex techniques that significantly increase the power and versatility of the View object.

- *Internal views:* Views can be defined as attributes of a HasTraits class; one class can have multiple views. View attributes can be inherited by subclasses.
- *External views:* A view can be defined as a module variable, inline as a function or method argument, or as an attribute of a Handler.
- *Ways of displaying views:* You can display a View by calling configure_traits() or edit_traits() on a HasTraits object, or by calling the ui() method on the View object.
- *View context:* You can pass a *context* to any of the methods for displaying views, which is a dictionary of labels and objects. In the default case, this dictionary contains only one object, referenced as 'object', but you can define contexts that contain multiple objects.
- *Include objects:* You can use an Include object as a placeholder for view items defined elsewhere.

5.1 Internal Views

In the examples thus far, the View objects have been external. That is to say, they have been defined outside the model (HasTraits object or objects) that they are used to display. This approach is in keeping with the separation of the two concepts prescribed by the MVC design pattern.

There are cases in which it is useful to define a View within a HasTraits class. In particular, it can be useful to associate one or more Views with a particular type of object so that they can be incorporated into other parts of the application with little or no additional programming. Further, a View that is defined within a model class is inherited by any subclasses of that class, a phenomenon called *visual inheritance*.

5.1.1 Defining a Default View

It is easy to define a default view for a HasTraits class: simply create a View attribute called **traits_view** for that class. Consider the following variation on Example 3:

Example 5: Using configure_traits() with a default View object

```
# Sample code to demonstrate the use of 'traits_view'
from enthought.traits.api import HasTraits, Str, Int
from enthought.traits.ui.api import View, Item, Group
import enthought.traits.ui

class SimpleEmployee2(HasTraits):
    first_name = Str
    last_name = Str
    department = Str

    employee_number = Str
    salary = Int

    traits_view = View(Group(Item(name = 'first_name'),
                             Item(name = 'last_name'),
                             Item(name = 'department'),
                             label = 'Personnel profile',
                             show_border = True))

sam = SimpleEmployee2()
sam.configure_traits()
```

In this example, configure_traits() no longer requires a *view* keyword argument; the **traits_view** attribute is used by default, resulting in the same display as in Figure 3:

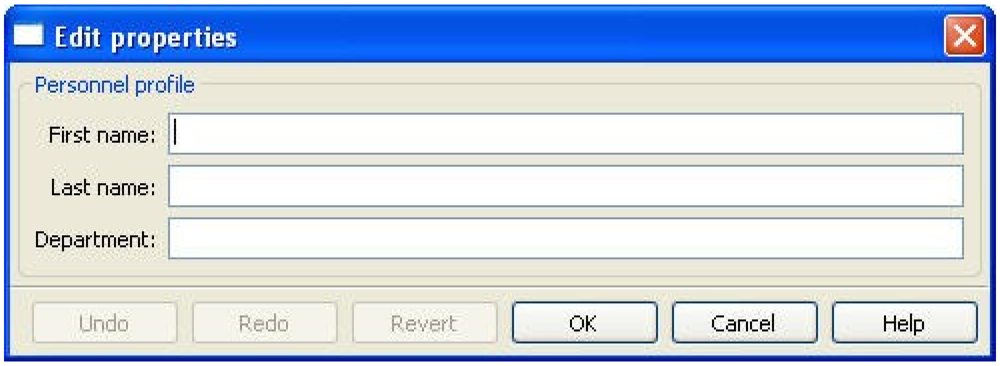


Figure 5: User interface for Example 5

It is not strictly necessary to call this View attribute **traits_view**. If exactly one View attribute is defined for a HasTraits class, that View is always treated as the default display template for the class. However, if there are multiple View attributes for the class (as discussed in the next section), if one is named "traits_view", it is always used as the default.

5.1.2 Defining Multiple Views Within the Model

Sometimes it is useful to have more than one pre-defined view for a model class. In the case of the SimpleEmployee class, one might want to have both a "public information" view like the one above and an "all information" view. One can do this by simply adding a second View attribute:

Example 6: Defining multiple View objects in a HasTraits class

```
# Sample code to demonstrate the use of multiple views
from enthought.traits.api import HasTraits, Str, Int
from enthought.traits.ui.api import View, Item, Group
import enthought.traits.ui

class SimpleEmployee3(HasTraits):
    first_name = Str
    last_name = Str
    department = Str

    employee_number = Str
    salary = Int

    traits_view = View(Group(Item(name = 'first_name'),
                             Item(name = 'last_name'),
                             Item(name = 'department'),
                             label = 'Personnel profile',
                             show_border = True))

    all_view = View(Group(Item(name = 'first_name'),
                          Item(name = 'last_name'),
                          Item(name = 'department'),
                          Item(name = 'employee_number'),
                          Item(name = 'salary'),
                          label = 'Personnel Database' +
                                  'Entry',
                          show_border = True))
```

As before, a simple call to configure_traits() for an object of this class produces a window based on the default View (**traits_view**). In order to use the alternate View, use the same syntax as for an external view, except that the View name is specified in single quotes to indicate that it is associated with the object rather than being a module-level variable:

`configure_traits(view='all_view').`

Note that if more than one View is defined for a model class, you must indicate which one is to be used as the default by naming it `traits_view`. Otherwise, Traits UI gives preference to none of them, and instead tries to construct a default View, resulting in a simple alphabetized display as described in Chapter 2. For this reason, it is usually preferable to name a model's default View `traits_view` even if there are no other Views; otherwise, simply defining additional Views—even if they are never used—can unexpectedly change the behavior of the GUI.

5.2 Separating Model and View: External Views

In all the preceding examples in this guide, the concepts of model and view have remained closely coupled. In some cases the view has been defined in the model class as in Section 5.1 on page 18; in other cases the configure_traits() method that produces a window from a View has been called from a HasTraits object. However, these strategies are simply conveniences; they are not an intrinsic part of the relationship between model and view in Traits UI. This section begins to explore how the Traits UI package truly supports the separation of model and view prescribed by the MVC design pattern.

An *external* view is one that is defined outside the model classes. In Traits UI, you can define a named View wherever you can define a variable or class attribute.[7] A View can even be defined in-line as a function or method argument, for example:

```
object.configure_traits(view=View(Group(Item(name='a'),
                                        Item(name='b'),
                                        Item(name='c')))
```

However, this approach is apt to obfuscate the code unless the View is very simple.

Example 2 through Example 4 demonstrate external Views defined as variables. One advantage of this convention is that the variable

[7] Note that although the definition of a View within a HasTraits class has the syntax of a trait attribute definition, the resulting View is not stored as an attribute of the class. See Section 11.1 for information on how to access such Views.

name provides an easily accessible "handle" for re-using the View. This technique does not, however, support visual inheritance.

A powerful alternative is to define a View within the controller (Handler) class that controls the window for that View.[8] This technique is described in Chapter 6.

5.3 Displaying a View

Traits UI provides three methods for creating a window or panel from a View object. The first two, configure_traits() and edit_traits(), are defined on the HasTraits class, which is a superclass of all Traits-based model classes, as well as of Handler and its subclasses. The third method, ui(), is defined on the View class itself.

5.3.1 configure_traits()

The configure_traits() method creates a standalone window for a given View object, i.e., it does not require an existing GUI to run in. It is therefore suitable for building command-line functions, as well as providing an accessible tool for the beginning Traits UI programmer.

The configure_traits() method also provides options for saving trait attribute values to and restoring them from a file. Refer to the *Traits API Reference* for details.

5.3.2 edit_traits()

The edit_traits() method is very similar to configure_traits(), with two major exceptions. First, it is designed to run from within a larger application whose GUI is already defined. Second, it does not provide options for saving data to and restoring data from a file, as it is assumed that these operations are handled elsewhere in the application.

[8] Assuming there is one; not all GUIs require an explicitly defined `Handler`.

5.3.3 ui()

The View object includes a method called ui(), which performs the actual generation of the window or panel from the View for both edit_traits() and configure_traits(). The ui() method is also available directly through the Traits UI API, though it is usually preferable to use one of the other two methods.[9]

The ui() method has five keyword parameters:

- *kind*
- *context*
- *handler*
- *parent*
- *view_elements*

The first four are identical in form and function to the corresponding arguments of edit_traits(), except that *context* is not optional; the following section explains why.

The fifth argument, *view_elements*, is used only in the context of a call to ui() from a model object method, i.e., from configure_traits() or edit_traits(), Therefore it is irrelevant in the rare cases when ui() is used directly by client code. It contains a dictionary of the named ViewElement objects defined for the object whose configure_traits() (or edit_traits()) method was called..

5.4 The View Context

All three of the methods described in Section 5.3 have a *context* parameter. This parameter can be a single object or a dictionary of string/object pairs; the object or objects are the model objects whose traits attributes are to be edited. In general a “context” is a Python dictionary whose keys are strings; the key strings are used to look up the values. In the case of the *context* parameter to the ui() method, the dictionary values are objects. In the special case where only one object is relevant, it can be passed directly instead of wrapping it in a dictionary.

[9] One possible exception is the case where a View object is defined as a variable (i.e., outside any class) or within a custom Handler, and is associated more or less equally with multiple model objects; see Section 5.4.1: “Multi-Object Views”.

When the ui() method is called from configure_traits() or edit_traits() on a HasTraits object, the relevant object is the HasTraits object whose method was called. For this reason, you do not need to specify the *context* argument in most calls to configure_traits() or edit_traits(). However, when you call the ui() method on a View object, you *must* specify the *context* parameter, so that the ui() method receives references to the objects whose trait attributes you want to modify.

So, if configure_traits() figures out the relevant context for you, why call ui() at all? One answer lies in *multi-object views*.

5.4.1 Multi-Object Views

A multi-object view is any view whose contents depend on multiple "independent" model objects, i.e., objects that are not attributes of one another. For example, suppose you are building a real estate listing application, and want to display a window that shows two properties side by side for a comparison of price and features. This is straightforward in Traits UI, as the following example shows:

Example 7: Using a multi-object view with a context

```
#Sample code to show multi-object view with context

from enthought.traits.api import HasTraits, Str, Int, Bool
from enthought.traits.ui.api import View, Group, Item

# Sample class
class House(HasTraits):
   address = Str
   bedrooms = Int
   pool = Bool
   price = Int

# View object designed to display two objects of class 'House'
comp_view = View(Group(Group(Item(name = 'address',
                                  object='h1',
                                  resizable=True),
                             Item(name = 'bedrooms',
                                  object='h1'),
                             Item(name = 'pool',
                                  object='h1'),
                             Item(name = 'price',
                                  object='h1'),
                             show_border = True),
                       Group(Item(name = 'address',
                                  object='h2',
                                  resizable=True),
                             Item(name = 'bedrooms',
                                  object='h2'),
                             Item(name = 'pool',
                                  object='h2'),
                             Item(name = 'price',
                                  object='h2'),
                             show_border = True),
                       orientation = 'horizontal'),
                 title = 'House comparison')

# A pair of houses to demonstrate the View
house1 = House(address='4743 Dudley Lane',
               bedrooms=3,
               pool=False,
               price=150000)
house2 = House(address='11604 Autumn Ridge',
               bedrooms=3,
               pool=True,
               price=200000)

# ...And the actual display command
house1.configure_traits(view=comp_view, context={'h1':house1,
                                                 'h2':house2})
```

The resulting window has the desired appearance:[10]

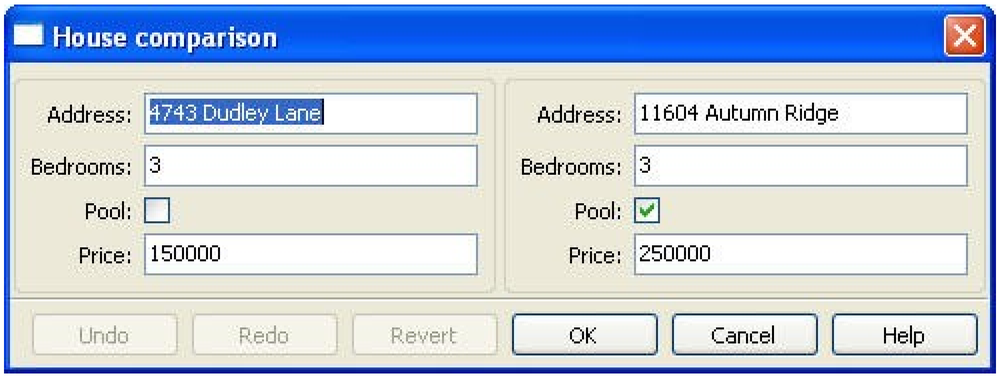


Figure 6: User interface for Example 7

For the purposes of this particular example, it makes sense to create a separate Group for each model object, and to use two model objects of the same class. Note, however, that neither is a requirement. It is only necessary for the **object** attribute of each

[10] If the script were designed to run within an existing GUI, it would make sense to replace the last line with "`comp_view.ui(context={'h1': house1, 'h2': house2})`", since neither object particularly dominates the view. However, the examples in this Guide are designed to be fully executable from the Python command line, which is why configure_traits() was used instead.

Item to be represented in the context, and for the **name** of each Item to be an existing attribute of the corresponding object.

Another use for multi-object views is shown in Section 8.1.9 on page 60.

5.5 Include Objects

In addition to the Item and Group class, a third building block class for Views exists in Traits UI: the Include class. For the sake of completeness, this section gives a brief description of Include objects and their purpose and usage. However, they are not commonly used as of this writing, and should be considered unsupported pending redesign.

In essence, an Include object is a placeholder for a named Group or Item object that is specified outside the Group or View in which it appears. For example, the following two definitions, taken together, are equivalent to the third:

Example 8: Using an Include object

```
# This fragment...
my_view = View(Group(Item(name='a'),
                     Item(name='b')),
               Include('my_group'))

# ...plus this fragment...
my_group = Group(Item(name='c'),
                 Item(name='d'),
                 Item(name='e'))

#...are equivalent to this:
my_view = View(Group(Item(name='a'),
                     Item(name='b')),
               Group(Item(name='c'),
                     Item(name='d'),
                     Item(name='e'))
```

This opens an interesting possibility when a View is part of a model class: any Include objects belonging to that View can be defined differently for different instances or subclasses of that class. This technique is called *view parameterization*.

6 Controlling the Interface: the Handler

Most of the material in the preceding chapters is concerned with the relationship between the model and view aspects of the MVC design pattern as supported by Traits UI. This chapter examines the third aspect: the *controller*, implemented in Traits UI as an instance of the Handler class.

A controller for an MVC-based application is essentially an event handler for GUI events, i.e., for events that are generated through or by the program interface. Such events can require changes to one or more model objects (e.g., because a data value has been updated) or manipulation of the interface itself (e.g., window closure, dynamic interface behavior). In Traits UI, such actions are performed by a Handler object.[11]

In the preceding examples in this guide, the Handler object has been implicit: Traits UI provides a default Handler that takes care of a common set of GUI events including window initialization and closure, data value updates, and button press events for the standard Traits UI window buttons (see Section 4.2 on page 15).

This chapter explains the features of the Traits UI Handler, and shows how to implement custom GUI behaviors by building and instantiating custom subclasses of the Handler class. The final section of the chapter describes several techniques for linking a custom Handler to the window or windows it is designed to control.

6.1 Backstage: Introducing the UIInfo Object

Traits UI supports the MVC design pattern by maintaining the model, view and controller as separate entities. A single View object can be used to construct windows for multiple model objects; likewise a single Handler can handle GUI events for windows created using different Views. Thus there is no static link between a Handler and any particular window or model object. However, in

[11] Except those implemented via the **enabled_when**, **visible_when**, and **defined_when** attributes of Items and Groups.

order to be useful, a Handler must be able to observe and manipulate both its corresponding window and model objects. In Traits UI, this is accomplished by means of the UIInfo object.

Whenever Traits UI creates a window or panel from a View, a UIInfo object is created to act as the Handler's reference to that window and to the objects whose trait attributes are displayed in it. Each entry in the View's context (see Section 5.4 on page 23) becomes an attribute of the UIInfo object.[12] For example, the UIInfo object created in Example 7 (on page 24) has attributes **h1** and **h2** whose values are the objects **house1** and **house2** respectively. In Example 1 through Example 6, the created UIInfo object has an attribute **object** whose value is the object **sam**.

Whenever a window event causes a Handler method to be called, Traits UI passes the corresponding UIInfo object as one of the method arguments. This gives the Handler the information necessary to perform its tasks, as described in the next two sections.

6.2 Writing Handler Methods

If you create a custom Handler subclass, depending on the behavior you want to implement, you might override the standard methods of Handler, or you might create methods that respond to changes to specific trait attributes.

6.2.1 Overriding Standard Methods

The Handler class provides methods that are automatically executed at certain points in the lifespan of the window controlled by a given Handler. By overriding these methods, you can implement a variety of custom window behaviors. The following sequence shows the points at which the Handler methods are called.

1. A UIInfo object is created
2. The Handler's init_info() method is called. Override this method if the handler needs access to viewable traits on the

[12] Other attributes of the UIInfo object include a UI object (see Section 10.1) and any *trait editors* contained in the window (see Chapters 7-9). The use of these UIInfo attributes is discussed in Section 10.2.

UIInfo object whose values are properties that depend on items in the context being edited.

3. The UI object is created, and generates the actual window.
4. The init() method is called. Override this method if you need to initialize or customize the window.
5. The position() method is called. Override this method to modify the position of the window (if setting the x and y attributes of the View is insufficient).
6. The window is displayed.

Table 5: When Handler methods are called, and when to override them

Method	*Called When*	*Override When?*
`apply(self, info)`	The user clicks the **Apply** button, and after the changes have been applied to the context objects.	To perform additional processing at this point.
`close(self, info, is_ok)`	The user requests to close the window, clicking **OK**, **Cancel**, or the window close button, menu, or icon.	To perform additional checks before destroying the window.
`closed(self, info, is_ok)`	The window has been destroyed	To perform additional clean-up tasks.
`revert(self, info)`	The user clicks the **Revert** button, or clicks **Cancel** in a live dialog box.	To perform additional processing.

Method	*Called When*	*Override When?*
`setattr(self, info, object, name, value)`	The user changes a trait attribute value through the user interface	To perform additional processing, such as keeping a change history. Make sure that the overriding method actually sets the attribute.
`show_help(self, info, control=None)`	The user clicks the **Help** button.	To call a custom help handler in addition to or instead of the global help handler, for this window.

6.2.2 Reacting to Trait Changes

The setattr() method described above is called whenever any trait value is changed in the UI. However, Traits UI also provides a mechanism for calling methods that are automatically executed whenever the user edits a *particular* trait. While you can use static notification handler methods on the HasTraits object, one might want to implement behavior that concerns only the user interface. In that case, following the MVC pattern dictates that such behavior should not be implemented in the "model" part of the code. In keeping with this pattern, Traits UI supports "user interface notification" methods, which must have a signature with the following format:

```
objectname_traitname_changed(self, info)
```

This method is called whenever a change is made to the attribute *traitname* of the object whose key is *objectname* in the **context** of the View used to create the window (see Section 5.4 on page 23).

Remember that if an object is displayed without an explicit View context, it implicitly uses the literal string 'object' as its context key.

Thus, if you are writing a subclass of Handler to use with the SimpleEmployee class and its View from Example 2 (on page 5), and you want certain code to execute whenever the **salary** attribute is edited, you can place that code in the body of a method called object_salary_changed(). By contrast, a subclass of Handler for Example 8 (on page 26) might include a method called h2_price_changed() to be called whenever the price of the second house is edited.

Important: These methods are called on window creation.

User interface notification methods are called when the window is first created.

To differentiate between code that should be executed when the window is first initialized and code that should be executed when the trait actually changes, use the **initialized** attribute of the UIInfo object (i.e., of the *info* argument):

```
def object_foo_changed(self, info):

    if not info.initialized:
        #code to be executed only when the window is
        #created
    else:
        #code to be executed only when 'foo' changes after
        #window initialization}

    #code to be executed in either case
```

The following script, which annotates its window's title with an asterisk ('*') the first time a data element is updated, demonstrates a simple use of both an overridden setattr() method and user interface notification method.

Example 9: Using a Handler that overrides setattr() and that has a user interface notification method

```
from enthought.traits.api import HasTraits, Bool
from enthought.traits.ui.api import View, Handler

class TC_Handler(Handler):

    def setattr(self, info, object, name, value):
        Handler.setattr(self, info, object, name, value)
        info.object._updated = True

    def object__updated_changed(self, info):
        if info.initialized:
            info.ui.title += "*"
```

```
class TestClass(HasTraits):
    b1 = Bool
    b2 = Bool
    b3 = Bool
    _updated = Bool(False)

view1 = View('b1', 'b2', 'b3',
             title="Alter Title on Update",
             handler=TC_Handler())

tc = TestClass()
tc.configure_traits(view=view1)
```

6.2.3 Implementing Custom Window Commands

Another purpose that you can use a Handler for is to define custom window actions, which can be presented as buttons, menu items, or toolbar buttons.

6.2.3.1 Actions

In Traits UI, window commands are implemented as instances of the Action class. Actions can be used in command buttons, menus, and toolbars.

Suppose you want to build a window with a custom **Recalculate** action. Suppose further that you have defined a subclass of Handler called MyHandler to provide the logic for the window. To create the action:

1. Add a method to MyHandler that implements the command logic. This method can have any name (e.g., do_recalc()), but must accept exactly one argument: a UIInfo object.
2. Create an Action instance using the name of the new method, e.g.:

```
recalc = Action(name = "Recalculate",
                action = "do_recalc")
```

6.2.3.2 Custom Command Buttons

The simplest way to turn an Action into a window command is to add it to the **buttons** attribute for the View. It appears in the button area of the window, along with any standard buttons you specify.

1. Define the handler method and action, as described in Section 6.2.3.1.
2. Include the new Action in the **buttons** attribute for the View:

```
View ( #view contents,
       # …,
       buttons = [ OKButton, CancelButton, recalc ])
```

6.2.3.3 Menus and Menu Bars

Another way to install an Action such as **recalc** as a window command is to make it into a menu option.

1. Define the handler method and action, as described in Section 6.2.3.1.
2. If the View does not already include a MenuBar, create one and assign it to the View's **menubar** attribute.
3. If the appropriate Menu does not yet exist, create it and add it to the MenuBar.
4. Add the Action to the Menu.

These steps can be executed all at once when the View is created, as in the following code:

```
View ( #view contents,
       # …,
       menubar = MenuBar(
          Menu( my_action,
                name = 'My Special Menu')))
```

6.2.3.4 Toolbars

A third way to add an action to a Traits View is to make it a button on a toolbar. Adding a toolbar to a Traits View is similar to adding a menu bar, except that toolbars do not contain menus; they directly contain actions.

1. Define the handler method and the action, as in Section 6.2.3.1 o page 32, including a tooltip and an image to display on the

toolbar. The image must be a Pyface ImageResource instance; if a path to the image file is not specified, it is assumed to be in an `images` subdirectory of the directory where ImageResource is used.

```
From enthought.pyface.api import ImageResource

recalc = Action(name = "Recalculate",
                action = "do_recalc",
                toolip = "Recalculate the results",
                image = ImageResource("recalc.png"))
```

2. If the View does not already include a ToolBar, create one and assign it to the View's **toolba**r attribute.
3. Add the Action to the ToolBar.

As with a MenuBar, these steps can be executed all at once when the View is created, as in the following code:

```
View ( #view contents,
       # …,
       toolbar = ToolBar( my_action))
```

6.3 Assigning Handlers to Views

In accordance with the MVC design pattern, Handlers and Views are separate entities belonging to distinct classes. In order for a custom Handler to provide the control logic for a window, it must be explicitly associated with the View for that window. The Traits UI package provides three ways to accomplish this:

- Make the Handler an attribute of the View.
- Provide the Handler as an argument to a display method such as edit_traits().
- Define the View as part of the Handler.

6.3.1 Binding a Singleton Handler to a View

To associate a given custom Handler with all windows produced from a given View, assign an instance of the custom Handler class to the View’s **handler** attribute. This is the technique used in Example 9, with the result that the window created by the configure_traits() call (and any other window built using **view1**) is automatically controlled by an instance of TC_Handler.

6.3.2 Linking Handler and View at Edit Time

It is also possible to associate a custom Handler with a specific window without assigning it permanently to the View. Each of the three Traits UI window-building methods (the configure_traits() and edit_traits() methods of the HasTraits class and the ui() method of the View class) has a *handler* keyword argument. Assigning an instance of Handler to this argument gives that instance control *only of the specific window being created by the method*. This assignment overrides the View's **handler** attribute.

For example, you can replace the last line of Example 9 with:

```
tc.configure_traits(view=view1, handler=SomeOtherHandler())
```

The resulting window is controlled by an instance of SomeOtherHandler rather than of TC_Handler.

6.3.3 Creating a Default View Within a Handler

You seldom need to associate a single custom Handler with several different Views or vice versa, although you can in theory and there are cases where it is useful to be able to do so. In most real-life scenarios, a custom Handler is tailored to a particular View with which it is always used. One way to reflect this usage in the program design is to define the View as part of the Handler. For example, one can rewrite Example 9 (on page 31) a little more concisely as follows:

Example 10: Defining a default view in a Handler class

```
from enthought.traits.api import HasTraits, Bool
from enthought.traits.ui.api import View, Handler

class TC_Handler(Handler):

    def setattr(self, info, object, name, value):
        Handler.setattr(self, info, object, name, value)
        info.object._updated = True
    def object__updated_changed(self, info):
        if info.initialized:
            info.ui.title += "*"

    traits_view = View('b1', 'b2', 'b3',
                       title="Alter Title on Update")

class TestClass(HasTraits):
    b1 = Bool
    b2 = Bool
    b3 = Bool
    _updated = Bool(False)

tc = TestClass()
TC_Handler().configure_traits(context={"object":tc})
```

The Handler class, which is a subclass of HasTraits, overrides the standard configure_traits() and edit_traits() methods; the child versions are identical to the originals except that the Handler object on which they are called becomes the default Handler for the resulting windows. Note that for these versions of the display methods, the *context* keyword parameter is not optional.

7 Introduction to Trait Editor Factories

The preceding code samples in this User Guide have been surprisingly simple considering the sophistication of the interfaces that they produce. In particular, no code at all has been required to produce appropriate widgets for the Traits to be viewed or edited in a given window. This is one of the strengths of Traits UI: usable interfaces can be produced simply and with a relatively low level of UI programming expertise.

An even greater strength lies in the fact that this simplicity does not have to be paid for in lack of flexibility. Where a novice Traits UI programmer can ignore the question of widgets altogether, a more advanced one can select from a variety of predefined interface components for displaying any given Trait. Furthermore, a programmer who is comfortable both with Traits UI and with UI programming in general can harness the full power and flexibility of the underlying GUI toolkit from within Traits UI.

The secret behind this combination of simplicity and flexibility is a Traits UI construct called a *trait editor factory*. A trait editor factory encapsulates a set of display instructions for a given trait type, hiding GUI-toolkit-specific code inside an abstraction with a relatively straightforward interface. Furthermore, every predefined trait type in the Traits package has a predefined trait editor factory that is automatically used whenever the trait is displayed, unless you specify otherwise.

Consider the following script and the window it creates:

Example 11: Using default trait editors

```
from enthought.traits.api import HasTraits, Str, Range, Bool
from enthought.traits.ui.api import View, Item

class Adult(HasTraits):
    first_name = Str
    last_name = Str
    age = Range(21,99)
    registered_voter = Bool

    traits_view = View(Item(name='first_name'),
                       Item(name='last_name'),
                       Item(name='age'),
                       Item(name='registered_voter'))
```

```
alice = Adult(first_name='Alice',
              last_name='Smith',
              age=42,
              registered_voter=True)

alice.configure_traits()
```

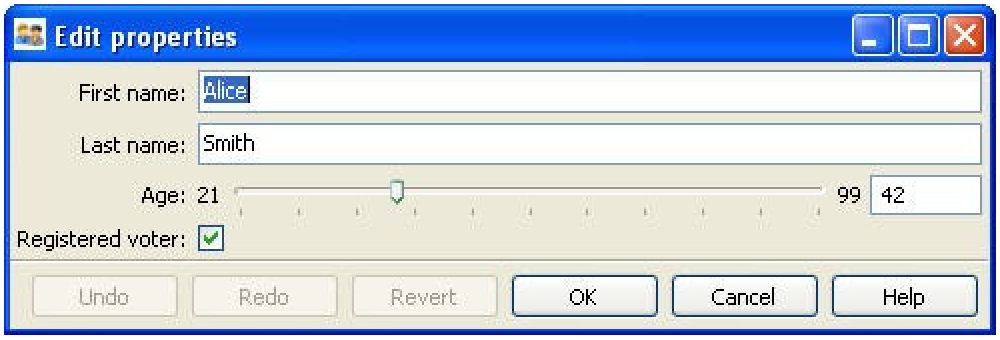


Figure 7: User interface for Example 11

Notice that each trait is displayed in an appropriate widget, even though the code does not explicitly specify any widgets at all. The two Str traits appear in text boxes, the Range is displayed using a combination of a text box and a slider, and the Bool is represented by a checkbox. Each implementation is generated by the default trait editor factory (TextEditor, RangeEditor and BooleanEditor respectively) associated with the trait type.

Traits UI is by no means limited to these defaults. There are two ways to override the default representation of a trait attribute in a Traits UI window:

- Explicitly specifying an alternate trait editor factory
- Specifying an alternate style for the editor generated by the factory

The remainder of this chapter examines these alternatives more closely.

7.1 Specifying an Alternate Trait Editor Factory

As of this writing the Traits UI package includes the following predefined trait editor factories:

- ArrayEditor()
- BooleanEditor()
- ButtonEditor()
- CheckListEditor()

- CodeEditor()
- ColorEditor()
- CompoundEditor()
- CustomEditor()
- DNDEditor()
- DirectoryEditor()
- DropEditor()
- EnableRGBAColorEditor()
- EnumEditor()
- FileEditor()
- FontEditor()
- HTMLEditor()
- ImageEnumEditor()
- InstanceEditor()
- KeyBindingsEditor()
- KivaFontEditor()
- ListEditor()
- NullEditor()
- RangeEditor()
- RGBColorEditor()
- RGBAColorEditor()
- SetEditor()
- ShellEditor()
- TableEditor()
- TextEditor()
- TreeEditor()
- TupleEditor()
- ValueEditor()

For a current complete list of editor factories, refer to the *Traits API Reference*. These editor factories are described in detail in Chapter 8.

For most predefined traits (see *Traits User Manual*), there is exactly one predefined trait editor factory suitable for displaying it: the editor factory that is assigned as its default.[13] There are exceptions, however; for example, there are two different editor factories for displaying an RGBAColor trait: RGBAColorEditor and EnableRGBAColorEditor. A List trait can be edited by means of ListEditor, TableEditor (if the List elements are HasTraits objects), CheckListEditor or SetEditor. Furthermore, the Traits UI package includes tools for building additional trait editors and factories for them as needed; these tools are described in Chapter 9.

To use an alternate editor factory for a trait in a Traits UI window, you must specify it in the View for that window. This is done at the Item level, using the *editor* keyword parameter. The syntax of the specification is `editor =` *editor_name*`()`. (This syntax is also used for specifying that the default editor should be used, but with certain keyword parameters explicitly specified; see Section 7.1.1 on page 41).

For example, to display an RGBAColor trait called **my_color** using the default editor factory (RGBAColorEditor()), the View might contain the following Item:

```
Item(name='my_color')
```

[13] Appendix II contains a table of the predefined trait types in the Traits package and their default trait editor types.

The resulting widget would have the following appearance:

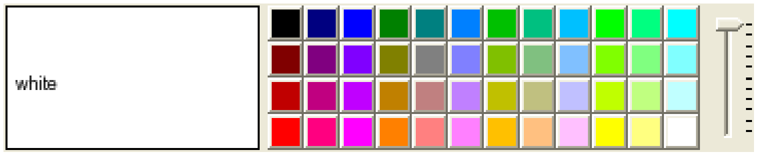


Figure 8: Default editor for an RGBAColor trait

To use the EnableRGBAColorEditor factory instead, add the appropriate specification to the Item:

```
Item( name='my_color', editor=EnableRGBAColorEditor() )
```

The resulting widget appears as in Figure 9:

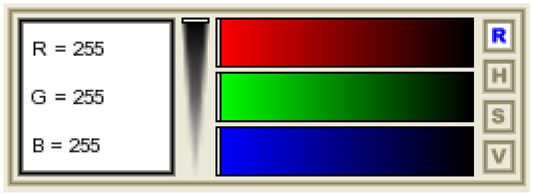


Figure 9: Editor generated by EnableRGBAColorEditor()

NOTE: Traits UI does not check editors for appropriateness.

Traits UI does not police the *editor* argument to ensure that the specified editor is appropriate for the trait being displayed. Thus there is nothing to prevent one from trying to, say, display a Float trait using a ColorEditor. The results of such a mismatch are unlikely to be helpful, and can even crash the application; it is up to the programmer to choose an editor sensibly. Chapter 8 is a useful reference for selecting an appropriate editor for a given task.

It is possible to specify the trait editor for a trait in other ways:

- You can specify a trait editor when you define a trait, by passing the result of a trait editor factory to the *editor* keyword parameter of the Trait() function. However, this approach commingles the “view” of a trait with its “model”.
- You can specify the **editor** attribute of a TraitHandler object. This approach commingles the “view” of a trait with its “controller”.

Use these approaches very carefully, if at all, as they muddle the MVC design pattern.

7.1.1 Initializing Editors

Many of the Traits UI trait editors can be used "straight from the box" as in the example above. There are some editors, however, that must be initialized in order to be useful. For example, a checklist editor (from CheckListEditor()) and a set editor (from SetEditor()) both enable the user to edit a List by selecting elements from a specified set; the contents of this set must, of course, be known to the editor. This sort of initialization is usually performed by means of one or more keyword arguments to the editor factory, for example:

```
Item(name='my_list',editor=CheckListEditor(values=["opt1","
opt2","opt3"]))
```

The descriptions of trait editor factories in Chapter 8 include a list of required and optional initialization keywords for each editor.

7.2 Specifying an Editor Style

In Traits UI, any given trait editor can be generated in any of four different styles: *simple*, *custom*, *text* or *readonly*. These styles, which are described in general terms below, represent different "flavors" of data display, so that a given trait editor can look completely different in one style than in another. However, different trait editors displayed in the same style (usually) have noticeable characteristics in common. This is useful because editor style, unlike individual editors, can be set at the Group or View level, not just at the Item level. This point is discussed further in Section 7.2.5 on page 44.

7.2.1 The 'simple' Style

The *simple* editor style is designed to be as functional as possible while requiring minimal space within the window. In simple style, most of the Traits UI editors take up only a single line of space in the window in which they are embedded.

In some cases, such as the text editor and Boolean editor (see Section 8.1 on page 46), the single line is fully sufficient. In others, such as the (plain) color editor and the enumeration editor, a more detailed interface is required; pop-up panels, drop-down lists, or dialog boxes are often used in such cases. For example, the simple version of the enumeration editor for the wxWindows toolkit looks like this:

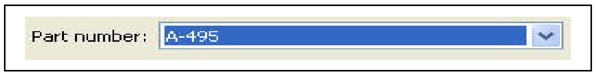


Figure 10: Simple style of enumeration editor

However, when the user clicks on the widget , a drop-down list appears:

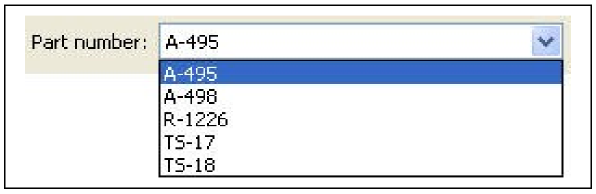


Figure 11: Simple enumeration editor with expanded list

The simple editor style is most suitable for windows that must be kept small and concise.

7.2.2 The 'custom' Style

The *custom* editor style generally generates the most detailed version of any given editor. It is intended to provide maximal functionality and information without regard to the amount of window space used. For example, in the wxWindows toolkit, the custom style the enumeration editor appears as a set of radio buttons rather than a drop-down list:

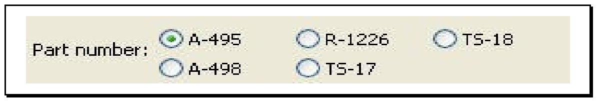


Figure 12: Custom style of enumeration editor

In the custom style of the RGBA color editor for wxWindows, the color palette is embedded in the window rather than appearing as a pop-up panel:

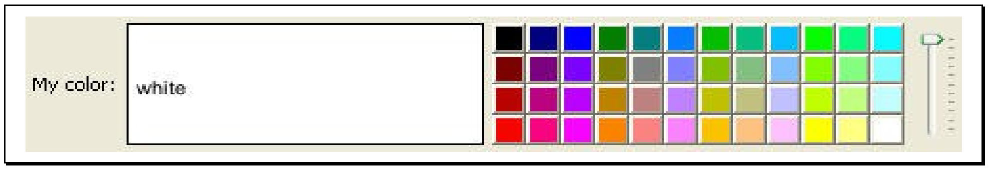


Figure 13: Custom style of RGBA color editor

In general, the custom editor style can be very useful when there is no need to conserve window space, as it enables the user to see as much information as possible without having to interact with the widget. It also usually provides the most intuitive interface of the four.

Note that this style is not defined explicitly for all trait editor implementations. If the custom style is requested for an editor for which it is not defined, the simple style is generated instead.

7.2.3 The 'text' Style

The *text* editor style is the simplest of the editor styles. When applied to a given trait attribute, it generates a text representation of the trait value in an editable box. Thus the enumeration editor in text style looks like the following:

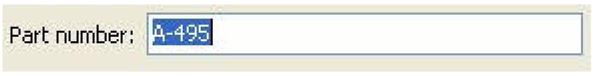


Figure 14: Text style of enumeration editor

For this type of editor, the end user must type in a valid value for the attribute. If the user types an invalid value, the *trait validator* for the attribute (see *Traits User Manual*) notifies the user of the error (for example, by shading the background of the text box red).

The text representation of an attribute to be edited in a text style editor is created in one of the following ways, listed in order of priority:

- The function specified in the **format_func** attribute of the Item (see Section 3.1 on page 7), if any, is called on the attribute value.
- Otherwise, the function specified in the *format_func* parameter of the trait editor factory (see Section 9.2.1.1 on page 103), if any, is called on the attribute value.

- Otherwise, the Python-style formatting string specified in the **format_str** attribute of the Item (see Section 3.1 on page 7), if any, is used to format the attribute value.
- Otherwise, the Python-style formatting string specified in the **format_str** parameter of the trait editor factory (see Section 9.2.1.1 on page 103), if any, is used to format the attribute value.
- Otherwise, the Python str() function is called on the attribute value.

7.2.4 The 'readonly' style

The *readonly* editor style is usually identical in appearance to the text style, except that the value appears as static text rather than in an editable box:

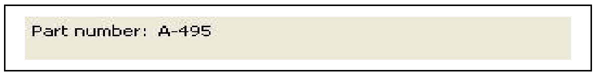


Figure 15: Read-only style of enumeration editor

This editor style is used to display data values without allowing the user to change them.

7.2.5 Using Editor Styles

As discussed in Chapters 3 and 4, the Item, Group and View objects of Traits UI all have a **style** attribute. The style of editor used to display the Items in a View is determined as follows:

1. The editor style used to display a given Item is the value of its **style** attribute if specifically assigned. Otherwise the editor style of the Group or View that contains the Item is used.
2. The editor style of a Group is the value of its **style** attribute if assigned. Otherwise, it is the editor style of the Group or View that contains the Group.
3. The editor style of a View is the value of its **style** attribute if specified, and simple otherwise.

In other words, editor style can be specified at the Item, Group or View level, and in case of conflicts the style of the smaller scope takes precedence. For example, consider the following script:

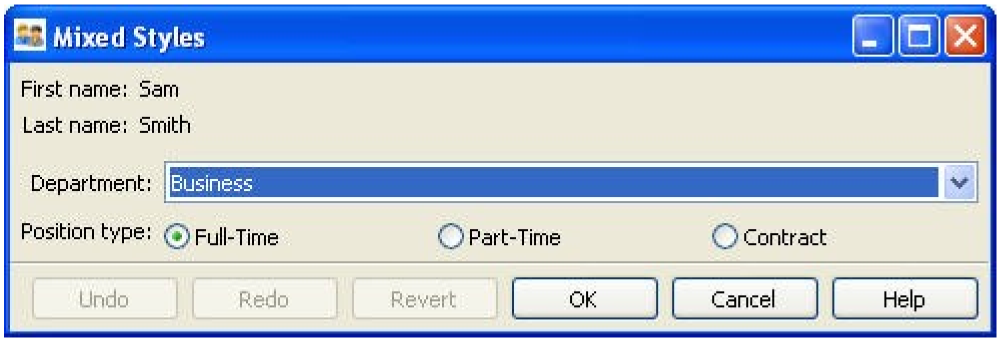
Example 12: Using editors styles at various levels

```
from enthought.traits.api import HasTraits, Str, Enum
from enthought.traits.ui.api import View, Group, Item

class MixedStyles(HasTraits):
   first_name = Str
   last_name = Str

   department = Enum("Business", "Research", "Admin")
   position_type = Enum("Full-Time",
                        "Part-Time",
                        "Contract")

   traits_view = View(Group(Item(name='first_name'),
                            Item(name='last_name'),
                            Group(Item(name='department'),
                                  Item(name=
                                          'position_type',
                                       style='custom'),
                                  style='simple')),
                      title='Mixed Styles',
                      style='readonly')

ms = MixedStyles(first_name='Sam', last_name='Smith')
ms.configure_traits()
```

Notice how the editor styles are set for each attribute:

- **position_type** at the Item level
- **department** at the Group level
- **first_name** and **last_name** at the View level

The resulting window demonstrates these precedence rules:


Mixed Styles

First name: Sam

Last name: Smith

Department: Business

Position type: (● Full-Time) (Part-Time) (Contract)

Figure 16: User interface for Example 12

8 The Predefined Trait Editor Factories

This chapter contains individual descriptions of the predefined trait editor factories provided by Traits UI. Most of these editor factories are straightforward and can be used easily with little or no expertise on the part of the programmer or end user; these are described in Section 8.1. Section 8.2 (on page 81) covers a smaller set of specialized editors that have more complex interfaces or that are designed to be used along with complex editors.

NOTE: Examples are toolkit-specific.

The exact appearance of the editors depends on the underlying GUI toolkit. The screenshots and descriptions in this chapter are based on wxWindows, which is the only currently supported GUI toolkit.

Rather than trying to memorize all the information in this chapter, you might skim it to get a general idea of the available trait editors and their capabilities, and to use it as a reference thereafter.

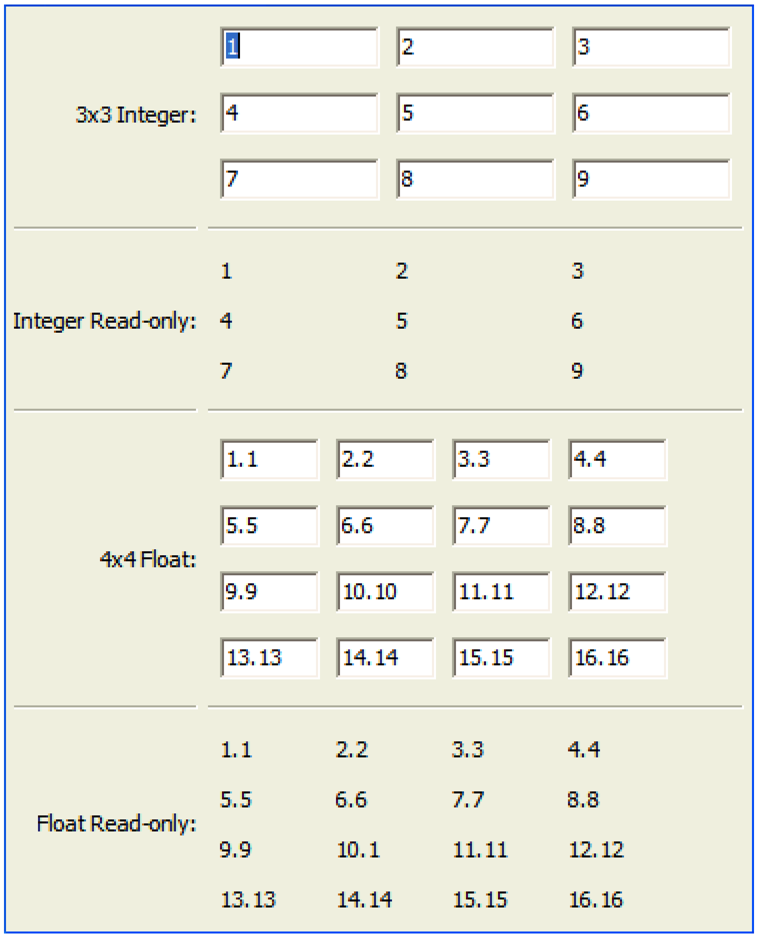
8.1 Basic Trait Editor Factories

The editor factories described in the following sections are straightforward to use. You can pass the editor object returned by the editor factory as the value of the *editor* keyword parameter when defining a trait.

8.1.1 ArrayEditor()

Suitable for	2-D Array, 2-D CArray
Default for	Array, CArray (if 2-D)
Required parameters	(none)
Optional parameters	*width*

The editors generated by ArrayEditor() provide text fields (or static text for the read-only style) for each cell of a two-dimensional Numeric array. Only the simple and read-only styles are supported by the wxWindows implementation. You can specify the width of the text fields with the *width* parameter.



3x3 Integer:
1 2 3
4 5 6
7 8 9

Integer Read-only:
1 2 3
4 5 6
7 8 9

4x4 Float:
1.1 2.2 3.3 4.4
5.5 6.6 7.7 8.8
9.9 10.10 11.11 12.12
13.13 14.14 15.15 16.16

Float Read-only:
1.1 2.2 3.3 4.4
5.5 6.6 7.7 8.8
9.9 10.1 11.11 12.12
13.13 14.14 15.15 16.16

Figure 17: Array editors

The following code generates the editors shown in Figure 17.

Example 13: Demonstration of array editors

```
from enthought.util.numerix import Int, Float
from enthought.traits.api import HasPrivateTraits, Array
from enthought.traits.ui.api \
    import View, ArrayEditor, Item
from enthought.traits.ui.menu import NoButtons

class ArrayEditorTest ( HasPrivateTraits ):

    three = Array((3,3), Int)
    four  = Array((4,4),
                  Float,
                  editor = ArrayEditor(width = -50))
    view = View( Item('three', label='3x3 Integer'),
                 '_',
                 Item('three',
                      label='Integer Read-only',
                      style='readonly'),
                 '_',
                 Item('four', label='4x4 Float'),
                 '_',
                 Item('four',
                      label='Float Read-only',
                      style='readonly'),
                 buttons   = NoButtons,
                 resizable = True )


if __name__ == '__main__':
    ArrayEditorTest().configure_traits()
```

8.1.2 BooleanEditor()

Suitable for	Bool, CBool
Default for	Bool, CBool
Required parameters	(none)
Optional parameters	*mapping*

BooleanEditor is one of the simplest of the built-in editor factories in the Traits UI package. It is used exclusively to edit and display Boolean (i.e, True/False) traits. In the simple and custom styles, it generates a checkbox. In the text style, the editor displays the trait value (as one would expect) as the strings `True` or `False`. However, several variations are accepted as input:

- `True`
- `True`
- `T`
- `Yes`
- `y`
- `False`
- `False`
- `F`
- `No`
- `n`

The set of acceptable text inputs can be changed by setting the BooleanEditor() parameter *mapping* to a dictionary whose entries are of the form *str* : *val*, where *val* is either True or False and *str* is a string that is acceptable as text input in place of that value. For example, to create a Boolean editor that accepts only `yes` and `no` as appropriate text values, you might use the following expression:

```
editor=BooleanEditor(mapping={"yes":True, "no":False})
```

Note that in this case, the strings `True` and `False` would *not* be acceptable as text input.

Figure 18 shows the four styles generated by BooleanEditor().

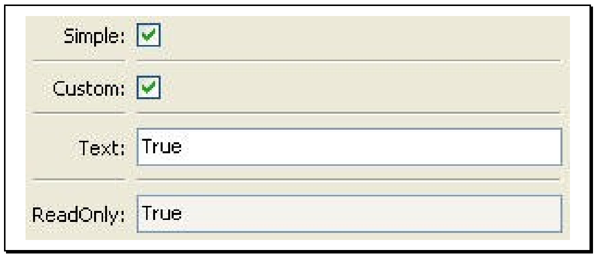


Figure 18: Boolean editor styles

8.1.3 ButtonEditor()

Suitable for	Button, Event, ToolbarButton
Default for	Button, ToolbarButton
Required parameters	(none)
Optional parameters	*label*, *value*

The ButtonEditor() factory is designed to be used with an Event or Button[14] trait. When a user clicks on a button editor, the associated event is fired. Because events are not printable objects, the text and read-only styles are not implemented for this editor. The simple and custom styles of this editor are identical, as shown in Figure 19.

[14] In Traits, a Button and an Event are essentially the same thing, except that Buttons are automatically associated with ButtonEditors.

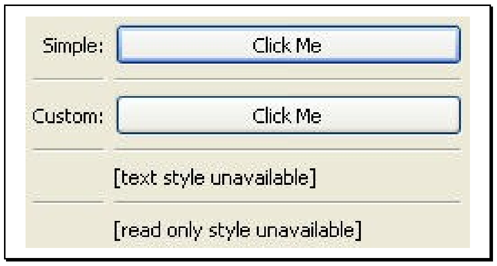


Figure 19: Button editor styles

By default, the label of the button is the name of the Button or Event trait to which it is linked.[15] However, this label can be set to any string by specifying the *label* parameter of ButtonEditor() as that string.

You can specify a value for the trait to be set to, using the *value* parameter. If the trait is an Event, then the value is not stored, but might be useful to an event listener.

8.1.4 CheckListEditor()

Suitable for	List
Default for	(none)
Required parameters	*values*
Optional parameters	*cols*, *names*

The editors generated by the CheckListEditor() factory are designed to enable the user to edit a List trait by selecting elements from a "master list", i.e., a list of possible values. The *values* parameter of CheckListEditor() contains this master list, and must therefore be specified when calling the factory.

The *values* parameter can take either of two forms:

- A list of strings
- A list of tuples of the form (*element*, *label*), where *element* can be of any type and *label* is a string.

In the latter case, the user selects from the labels, but the underlying trait is a List of the corresponding *element* values.

Alternatively, you can use the *names* parameter to specify the string labels for the values.

[15] Traits UI makes minor modifications to the name, capitalizing the first letter and replacing underscores with spaces, as in the case of a default Item label (see Section 3.1).

The custom style of editor from this factory is displayed as a set of checkboxes. By default, these checkboxes are displayed in a single column; however, you can initialize the *cols* parameter of the editor factory to any value between 1 and 20, in which case the corresponding number of columns is used.

The simple style generated by CheckListEditor() appears as a drop-down list; in this style, only one list element can be selected, so it returns a list with a single item. The text and read-only styles represent the current contents of the attribute in Python-style text format; in these cases the user cannot see the master list values that have not been selected.

The four styles generated by CheckListEditor() are shown in Figure 20. Note that in this case the *cols* parameter has been set to 4.

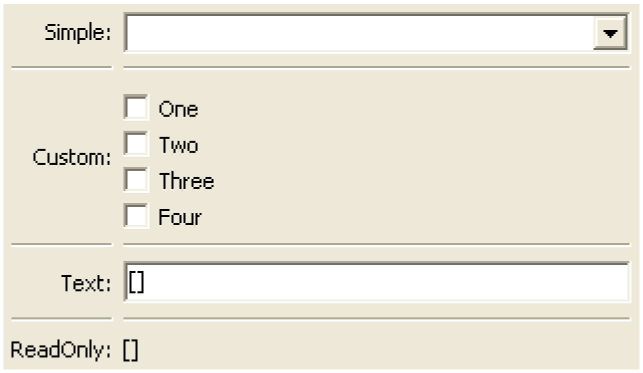
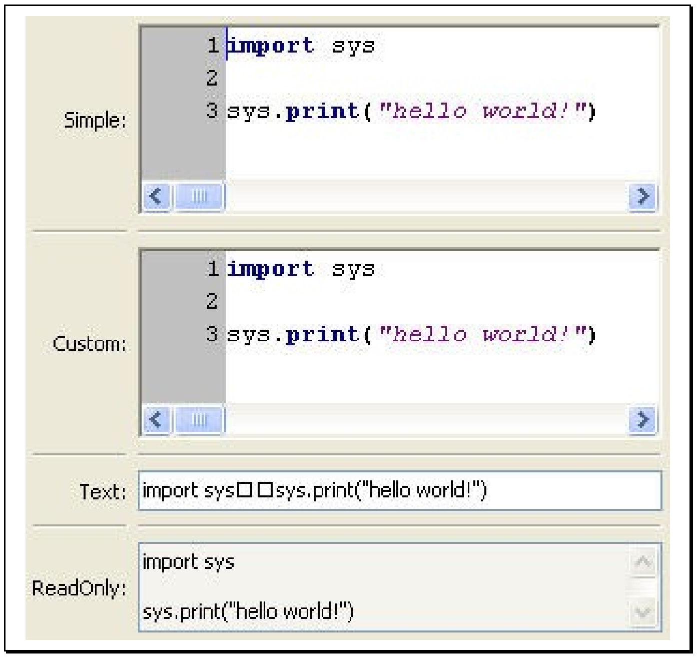


Figure 20: Checklist editor styles

8.1.5 CodeEditor()

Suitable for	Code, Str, String
Default for	Code
Required parameters	(none)
Optional parameters	*auto_set*, *key_bindings*

The purpose of a code editor is to display and edit Code traits, though it can be used with the Str and String trait types as well. In the simple and custom styles (which are identical for this editor), the text is displayed in numbered, non-wrapping lines with a horizontal scrollbar. The text style displays the trait value using a single scrolling line with special characters to represent line breaks. The read-only style is similar to the simple and custom styles except that line numbers are not displayed and the text is wrapped (and, of course, not editable).



```
import sys

sys.print("hello world!")
```

```
import sys

sys.print("hello world!")
```

```
import sys□□sys.print("hello world!")
```

```
import sys

sys.print("hello world!")
```

Figure 21: Code editor styles

The *auto_set* keyword parameter is a Boolean value indicating whether the trait being edited should be updated with every keystroke (True) or only when the editor loses focus, i.e., when the user tabs away from it or closes the window (False). The default value of this parameter is True.

The *key_bindings* keyword parameter is a reference to a KeyBindings object, which is a set of KeyBinding objects, which associate key strokes with Handler methods. See Section 8.2.2 for more information about key bindings.

8.1.6 Color Editors

The following sections describe the editor factories that are available to created editors for color traits.

8.1.6.1 ColorEditor()

Suitable for	Color
Default for	Color
Required parameters	(none)
Optional parameters	*mapped*

The editors generated by ColorEditor() are designed to enable the user to display a Color trait or edit it by selecting a color from the palette available in the underlying GUI toolkit. The four styles of color editor are shown in Figure 22.

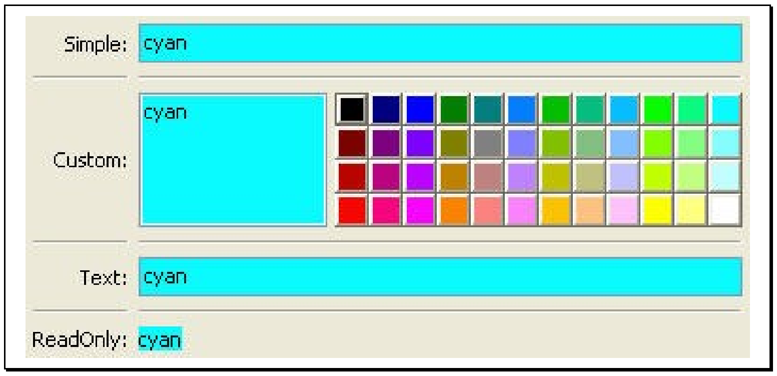


Figure 22: Color editor styles

In the simple style, the editor appears as a text box whose background is a sample of the currently selected color. The text in the box is either a color name or a tuple of the form (*r*,*g*,*b*) where *r*, *g*, and *b* are the numeric values of the red, green and blue color components respectively. (Which representation is used depends on how the value was entered.) The text value is not directly editable in this style of editor; instead, clicking on the text box displays a pop-up panel similar in appearance and function to the custom style.

The custom style includes a labeled color swatch on the left, representing the current value of the Color trait, and a palette of common color choices on the right. Clicking on any tile of the palette changes the color selection, causing the swatch to update accordingly. Clicking on the swatch itself causes a more detailed, platform-specific interface to appear in a dialog box, such as is shown in Figure 23.

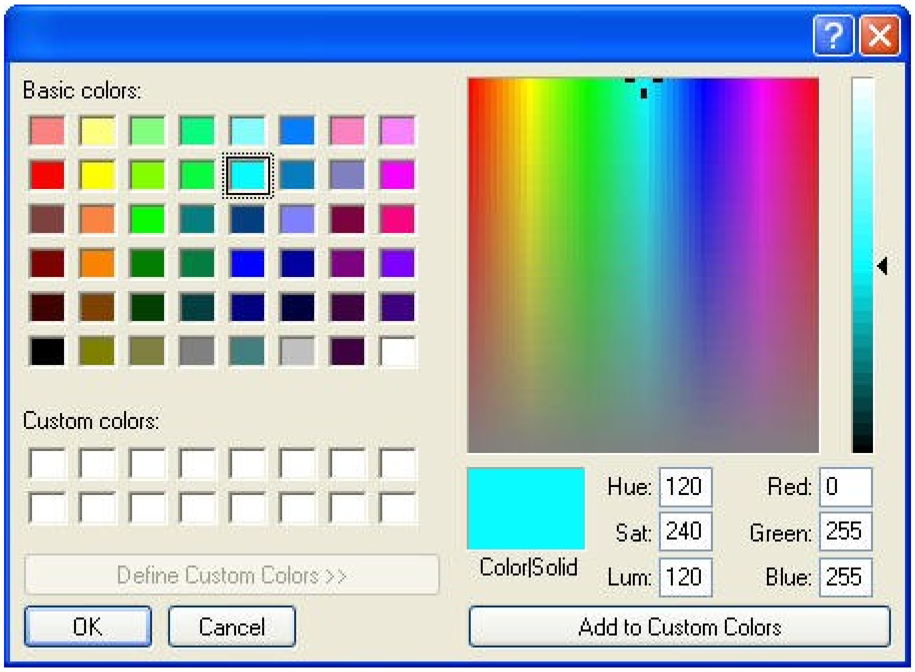


Figure 23: Custom color selection dialog box for Microsoft Windows XP

The text style of editor looks exactly like the simple style, but the text box is editable (and clicking on it does not open a pop-up panel). The user must enter a recognized color name or a properly formatted (*r*,*g*,*b*) tuple.

The read-only style displays the text representation of the currently selected Color value (name or tuple) on a minimally-sized background of the corresponding color.

For advanced users: The *mapped* keyword parameter of ColorEditor() is a Boolean value indicating whether the trait being edited has a built-in mapping of user-oriented representations (e.g., strings) to internal representations. Since ColorEditor() is generally used only for Color traits, which are mapped (e.g., `cyan` to wx.Colour(0,255,255)), this parameter defaults to True and is not of interest to most programmers. However, it is possible to define a custom color Trait that uses ColorEditor() but is not mapped (i.e., uses only one representation), which is why the attribute is available.

8.1.6.2 EnableRGBAColorEditor()

Suitable for	RGBAColor
Default for	(none)
Required parameters	(none)
Optional parameters	*auto_set*, *edit_alpha*, *mode*, *text*, *font*

The editors generated by EnableRGBAColorEditor() use an Enable (enthought.enable) widget to modify colors.

Figure 24 shows the editors generated by EnableRGBAColorEditor(). The text and read-only styles are identical to those generated by RGBAColorEditor() (see Section 8.1.6.4 on page 57).

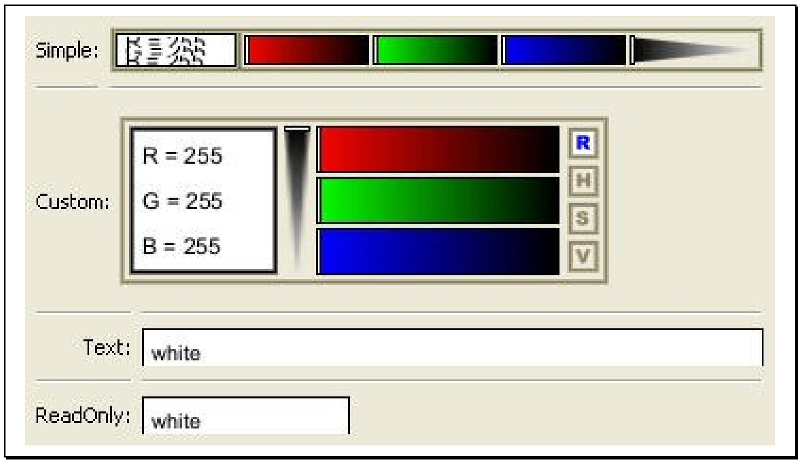
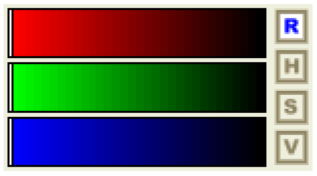
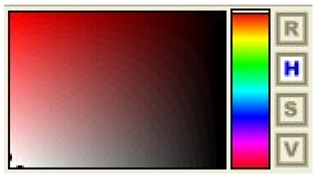
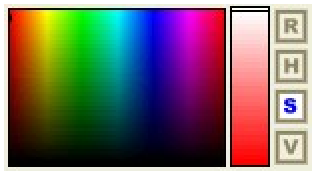
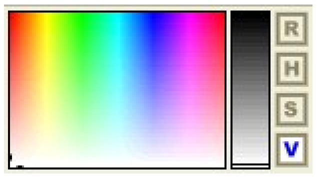


Figure 24: "Enable" editor styles for RGBAColor traits

The simple editor style shows a color swatch, and three graphical sliders, one each for red, green, blue, and alpha (transparency). The color swatch displays (in barely readable form) the red, green, and blue values. Clicking on the swatch itself causes a more detailed, platform-specific interface to appear in a dialog box, as shown in Figure 23 on page 54.

The custom editor style displays a larger color swatch with the red, green, and blue values, a vertical alpha slider, and an area for specifying the color. Small buttons to the right of the color area control what appears in that area, as shown in Table 6:

Table 6: Color selector for Enable custom editor

Button Selecte d	***Appearance***	***Description***
R	 R H S V	Horizontal sliders for red, green, and blue
H	 R H S V	2-D selector for saturation and value; vertical slider for hue.
S	 R H S V	2-D selector for value and hue; vertical slider for saturation.
V	 R H S V	2-D selector for saturation and hue; vertical slider for value.

By default, the editor opens with the "R" button selected. Use the *mode* parameter to specify which mode is selected when the editor opens. The possible values are 'rgb', 'hsv', 'hsv2', and 'hsv3'.

You can also specify text to display in the color swatch (which can use Python string formatting), and the font to use for the text, with the *text* and *font* parameters, respectively.

To disable editing of the alpha value, set the *edit_alpha* parameter to False.

8.1.6.3 RGBColorEditor()

Suitable for	RGBColor
Default for	RGBColor
Required parameters	(none)
Optional parameters	*mapped*

Editors generated by RGBColorEditor() are identical in appearance to those generated by ColorEditor(), but they are used for RGBColor traits. See Section 8.1.6.1 on page 53 for details.

8.1.6.4 RGBAColorEditor()

Suitable for	RGBAColor
Default for	RGBAColor
Required parameters	(none)
Optional parameters	*mapped*

The editors generated by RGBAColorEditor() are similar to those generated by ColorEditor(), with the addition of a vertical slider for the alpha value. The operation of the editors is otherwise identical. See Section 8.1.6.1 on page 53 for details.

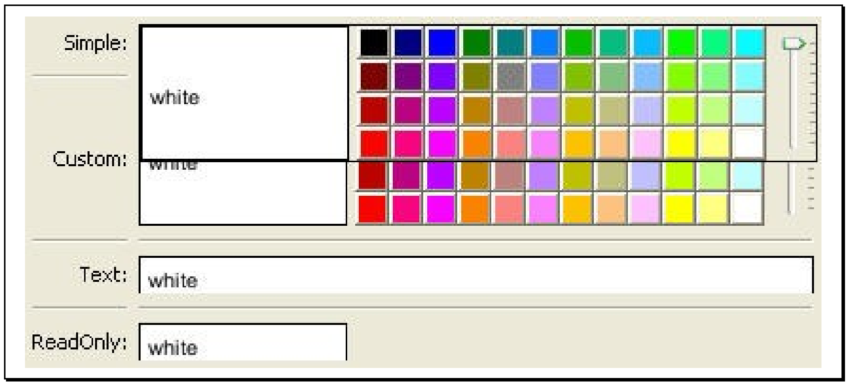


Figure 25: RGBAColor editor showing pop-up panel

8.1.7 CompoundEditor()

Suitable for	special
Default for	"compound" traits
Required parameters	(none)
Optional parameters	*auto_set*

An editor generated by CompoundEditor() consists of a combination of the editors for trait types that compose the compound trait. The widgets for the compound editor are of the style specified for the compound editor (simple, custom, etc.). The editors shown in Figure 26 are for the following trait, whose value can be an integer between 1 and 6, or any of the letters 'a' through 'f':

```
compound_trait = Trait( 1, Range( 1, 6 ), 'a', 'b', 'c',
                                          'd', 'e', 'f')
```

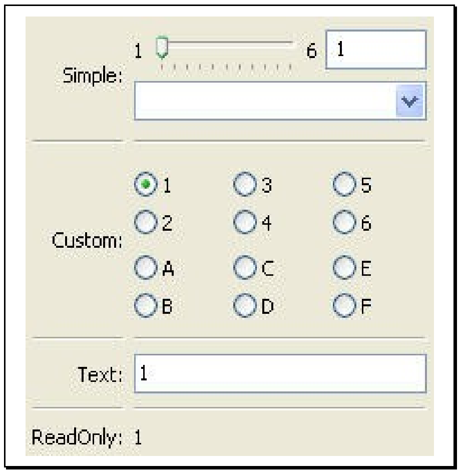


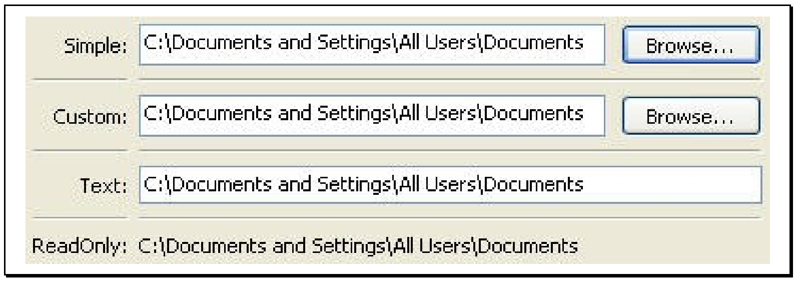
Figure 26: Example compound editor styles

The *auto_set* keyword parameter is a Boolean value indicating whether the trait being edited should be updated with every keystroke (True) or only when the editor loses focus, i.e., when the user tabs away from it or closes the window (False). The default value of this parameter is True.

8.1.8 DirectoryEditor()

Suitable for	Directory
Default for	Directory
Required parameters	(none)
Optional parameters	(none)

A directory editor enables the user to display a Directory trait or set it to some directory in the local system hierarchy. The four styles of this editor are shown in Figure 27.



Simple:	C:\Documents and Settings\All Users\Documents	Browse...
Custom:	C:\Documents and Settings\All Users\Documents	Browse...
Text:	C:\Documents and Settings\All Users\Documents	
ReadOnly:	C:\Documents and Settings\All Users\Documents	

Figure 27: Directory editor styles

In the simple and custom styles (which are identical), the current value of the trait is displayed in a text box to the left of a **Browse** button. The user can either type a new path directly into the text box or use the button to bring up a platform-specific directory browser dialog box, such as is shown in Figure 28.

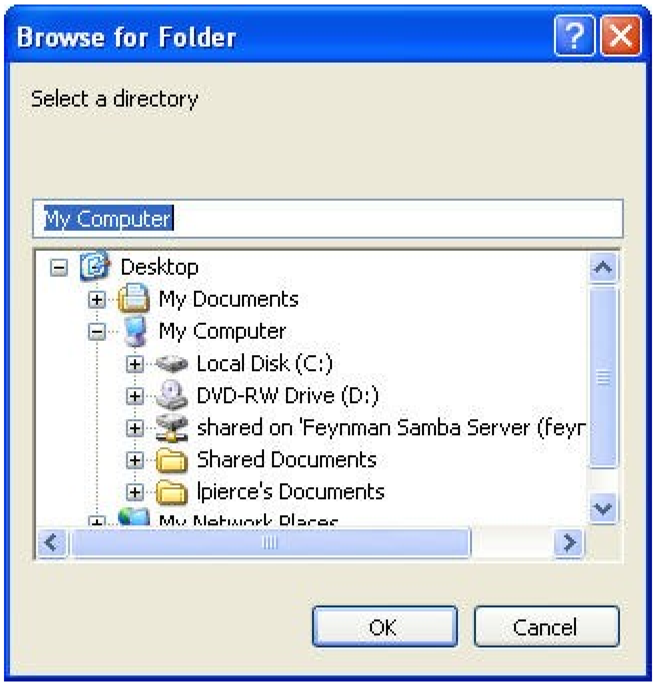


Figure 28: Example directory browser dialog box for Microsoft Windows

When the user selects a directory in this browser and clicks **OK**, control is returned to the original editor widget, which is automatically populated with the new path string.

The text style of editor is simply a text box into which the user can type a directory path. The ‘readonly style is identical to the text style, except that the text box is not editable.

No validation is performed on Directory traits; the user must ensure that a typed-in value is in fact an actual directory on the system.

8.1.9 EnumEditor()

Suitable for	Enum, Any
Default for	Enum
Required parameters	for non-Enum traits: *values*, or *name* and *object*
Optional parameters	*cols*, *evaluate*, *mode*

The editors generated by EnumEditor() enable the user to pick a single value from a closed set of values.

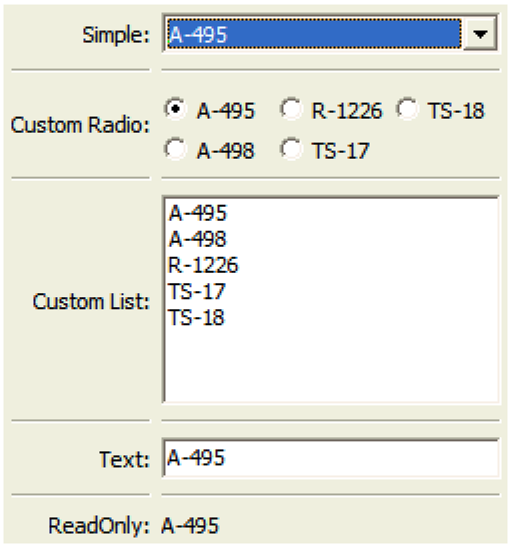


Figure 29: Enumeration editor styles

The simple style of editor is a drop-down list box.

The custom style can be a set of radio buttons, or a single-selection list box. The default is radio buttons; specify `mode='list'` to use a list box. Use the *cols* parameter to specify the number of columns of radio buttons.

The text style is an editable text field; if the user enters a value that is not in enumerated set, the background of the field turns red, to indicate an error. You can specify a function to evaluate text input, using the *evaluate* parameter.

The read-only style is the value of the trait as static text.

If the trait attribute that is being edited is not an enumeration, you must specify either the trait attribute (with the *object* and *name* parameters), or the set of values to display (with the *values* parameter). The *values* parameter can be a list, tuple, or dictionary, or a "mapped" trait.

By default, an enumeration editor sorts its values alphabetically. To specify a different order for the items, give it a mapping from the normal values to ones with a numeric tag. The enumeration editor sorts the values based on the numeric tags, and then strips out the tags. For example:

Example 14: Enumeration editor with mapped values

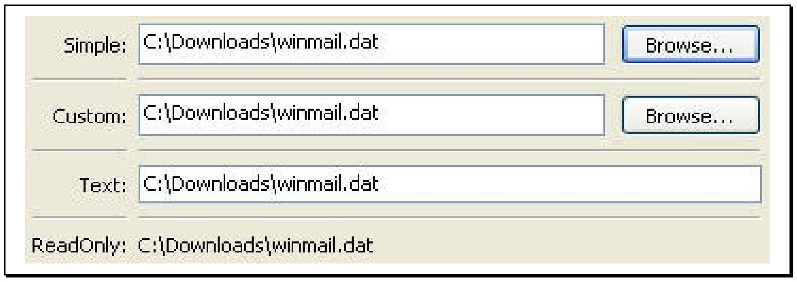
```
from enthought.traits.api import HasTraits, Trait
from enthought.traits.ui.api import EnumEditor

Class EnumExample(HasTraits):
    priority = Trait('Medium', 'Highest',
                               'High',
                               'Medium',
                               'Low',
                               'Lowest')

    view = View( Item(name='priority',
                      editor=EnumEditor(values={
                          'Highest' : '1:Highest',
                          'High'    : '2:High',
                          'Medium'  : '3:Medium',
                          'Low'     : '4:Low',
                          'Lowest'  : '5:Lowest', })))
```

The enumeration editor strips the characters up to and including the colon. It assumes that all the items have the colon in the same column; therefore, if some of your tags have multiple digits, you should use zeros to pad the items with fewer digits.

8.1.10 FileEditor()

Suitable for	File
Default for	File
Required parameters	(none)
Optional parameters	*filter*, *truncate_ext*, *auto_set*

A file editor enables the user to display a File trait or set it to some file in the local system hierarchy. The four styles of this editor are shown in Figure 30.



Simple: C:\Downloads\winmail.dat Browse...

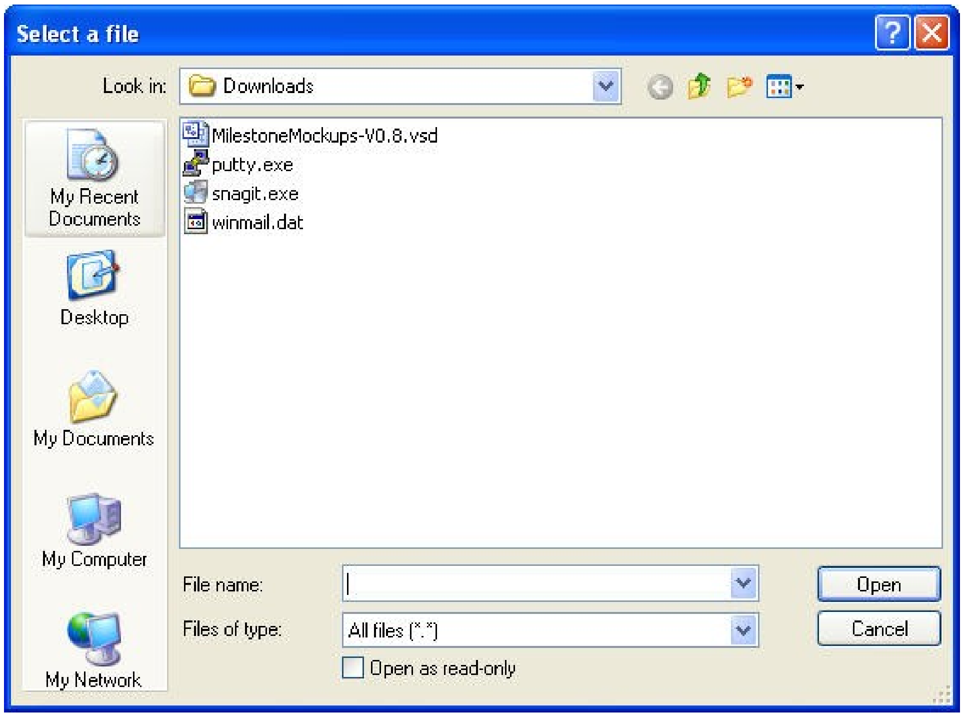
Custom: C:\Downloads\winmail.dat Browse...

Text: C:\Downloads\winmail.dat

ReadOnly: C:\Downloads\winmail.dat

Figure 30: File editor styles

The behavior and appearance of file editors are very similar to those of directory editors, except that the **Browse** button activates a platform-specific file browser dialog box, rather than a directory browser, such as is shown in Figure 31.



Select a file

Look in: Downloads

MilestoneMockups-V0.8.vsd
putty.exe
snagit.exe
winmail.dat

File name:

Files of type: All files (*.*)

Open as read-only

Figure 31: Example file browser dialog box for Microsoft Windows

If the *filter* or *auto_set* arguments are passed to the File() trait factory, they are automatically passed through to FileEditor() for the default editor for the trait. The *filter* parameter is a wildcard string to filter filenames in the file dialog box. The *auto_set* parameter is a Boolean that indicates whether the file dialog box updates its selection after every keystroke.

The *truncate_ext* parameter is a Boolean that indicates whether the file extension is removed from the returned filename. It is False by default, meaning that the filename is not modified before it is returned.

8.1.11 FontEditor()

Suitable for	Font
Default for	Font
Required parameters	(none)
Optional parameters	(none)

A font editor enables the user to display a Font trait or edit it by selecting one of the fonts provided by the underlying GUI toolkit. The four styles of this editor are shown in Figure 32.

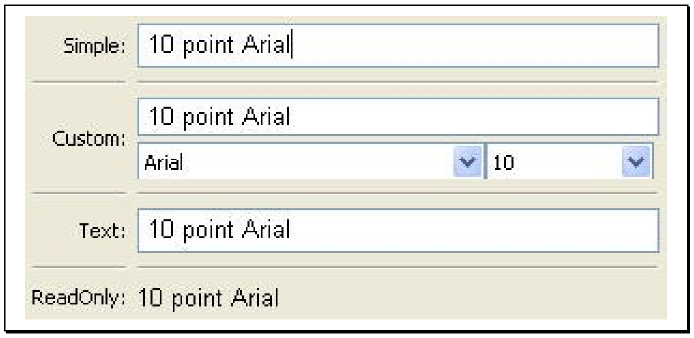

Simple: 10 point Arial
Custom: 10 point Arial
Arial 10
Text: 10 point Arial
ReadOnly: 10 point Arial

Figure 32: Font editor styles

In the simple style, the currently selected font appears in a display similar to a text box, except that when the user clicks on it, a platform-specific dialog box appears with a detailed interface, such as is shown in Figure 33. When the user clicks **OK**, control returns to the editor, which then displays the newly selected font.

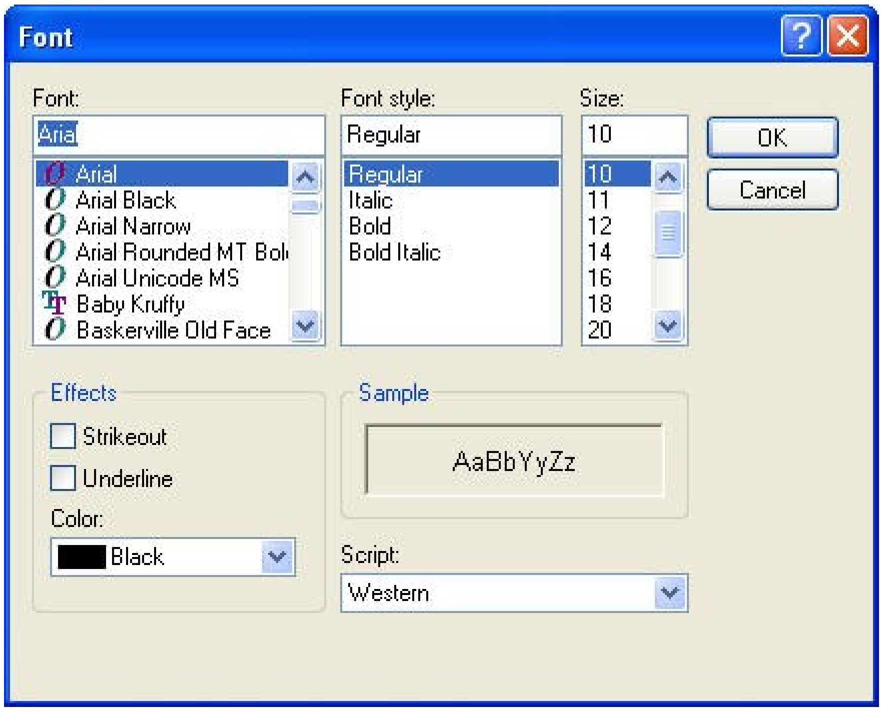


Figure 33: Example font dialog box for Microsoft Windows

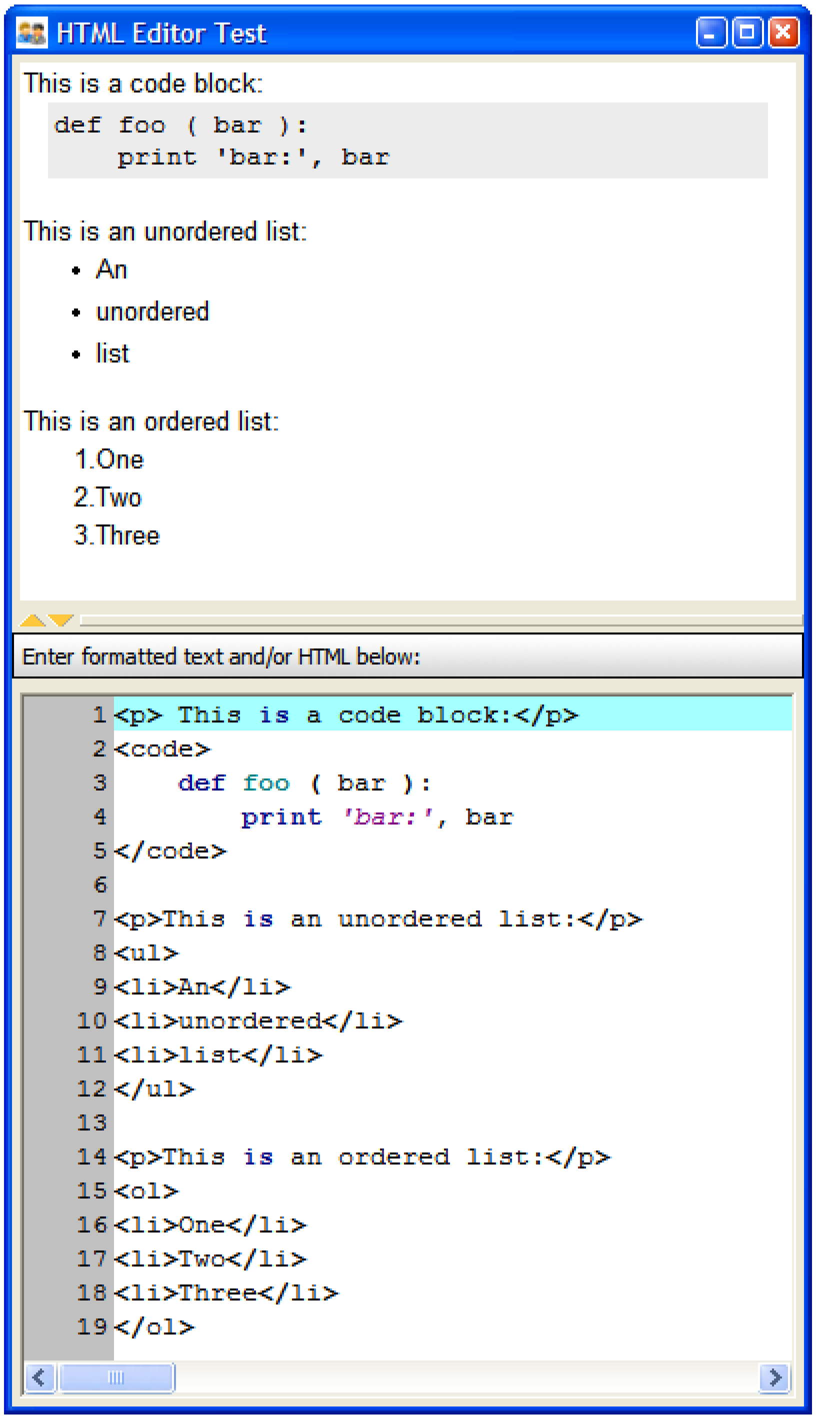
In the simple style, an abbreviated version of the font dialog box is displayed in-line. The user can either type the name of the font in the text box or use the two drop-down lists to select a typeface and size.

In the text style, the user *must* type the name of a font in the text box provided. No validation is performed; the user must enter the correct name of an available font. The read-only style is identical except that the text box is not editable.

8.1.12 HTMLEditor()

Suitable for	HTML, Str, Unicode
Default for	HTML
Required parameters	(none)
Optional parameters	*format_text*

The "editor" generated by HTMLEditor() interprets and displays text as HTML. It does not support the user editing the text that it displays. It generates the same type of editor, regardless of the style specified. Figure 34 shows an HTML editor in the upper pane, with a code editor in the lower pane, displaying the uninterpreted text.



This is a code block:

```
def foo ( bar ):
    print 'bar:', bar
```

This is an unordered list:

- An
- unordered
- list

This is an ordered list:

1. One
2. Two
3. Three

Enter formatted text and/or HTML below:

```
<p> This is a code block:</p>
<code>
    def foo ( bar ):
        print 'bar:', bar
</code>

<p>This is an unordered list:</p>
<ul>
<li>An</li>
<li>unordered</li>
<li>list</li>
</ul>

<p>This is an ordered list:</p>
<ol>
<li>One</li>
<li>Two</li>
<li>Three</li>
</ol>
```

Figure 34: Example HTML editor, with code editor showing original text

NOTE: HTML support is limited in the wxWindows toolkit.
The set of tags supported by the wxWindows implementation of the HTML editor is a subset of the HTML 3.2 standard. It does not support style sheets or complex formatting. Refer to the wxWindows documentation for details.

If the *format_text* argument is True, then the HTML editor supports basic implicit formatting, which it converts to HTML before passing the text to the HTML interpreter. The implicit formatting follows these rules:

- Indented lines that start with a dash ('-') are converted to unordered lists.
- Indented lines that start with an asterisk ('*') are converted to ordered lists.
- Indented lines that start with any other character are converted to code blocks.
- Blank lines are converted to paragraph separators.

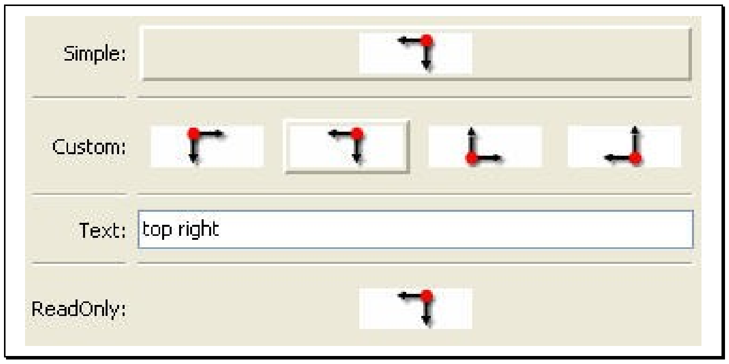
The following text produces the same displayed HTML as in Figure 34, when *format_text* is True:

```
This is a code block:

    def foo ( bar ):
        print 'bar:', bar

This is an unordered list:
 - An
 - unordered
 - list

This is an ordered list:
 * One
 * Two
 * Three
```

8.1.13 ImageEnumEditor()

Suitable for	Enum, Any
Default for	(none)
Required parameters	for non-Enum traits: *values*, or *name* and *object*
Optional parameters	*path*, *klass*, or *module*; *cols*, *evaluate*, *suffix*

The editors generated by ImageEnumEditor() enable the user to select an item in an enumeration by selecting an image that represents the item.



top right

Figure 35: Editor styles for image enumeration

The custom style of editor displays a set of images; the user selects one by clicking it, and it becomes highlighted to indicate that it is selected.

The simple style displays a button with an image for the currently selected item. When the user clicks the button, a pop-up panel displays a set of images, similar to the custom style. The user clicks an image, which becomes the new image on the button.

The text style does not display images; it displays the text representation of the currently selected item. The user must type the text representation of another item to select it.

The read-only style displays the image for the currently selected item, which the user cannot change.

The ImageEnumEditor() function accepts the same parameters as the EnumEditor() function (see Section 8.1.9 on page 60), as well as some additional parameters.

NOTE: Image enumeration editors do not use ImageResource.
Unlike most other images in the Traits and Traits UI packages, images in the wxWindows implementation of image enumeration editors do not use the PyFace ImageResource class.

In the wxWindows implementation, image enumeration editors use the following rules to locate images to use:

1. Only GIF (`.gif`) images are currently supported.
2. The base file name of the image is the string representation of the value, with spaces replaced by underscores and the *suffix* argument, if any, appended. Note that *suffix* is not a file extension, but rather a string appended to the base file name. For example, if *suffix* is '_origin' and the value is 'top left', the image file name is `top_left_origin.gif`.
3. If the *path* parameter is defined, it is used to locate the file. It can be absolute or relative to the file where the image enumeration editor is defined.
4. If *path* is not defined and the *klass* parameter is defined, it is used to locate the file. The *klass* parameter must be a reference to a class. The editor searches for an `images` subdirectory in the following locations:
    - o The directory that contains the module that defines the class.
    - o If the class was executed directly, the current working directory.
5. If *path* and *klass* are not defined, and the *module* parameter is defined, it is used to locate the file. The *module* parameter must be a reference to a module. The editor searches for an `images` subdirectory of the directory that contains the module.
6. If *path*, *klass*, and *module* are not defined, the editor searches for an `images` subdirectory of the **enthought.traits.ui.wx** package.
7. If none of the above paths are defined, the editor searches for an `images` directory that is a sibling of the directory from which the application was run.

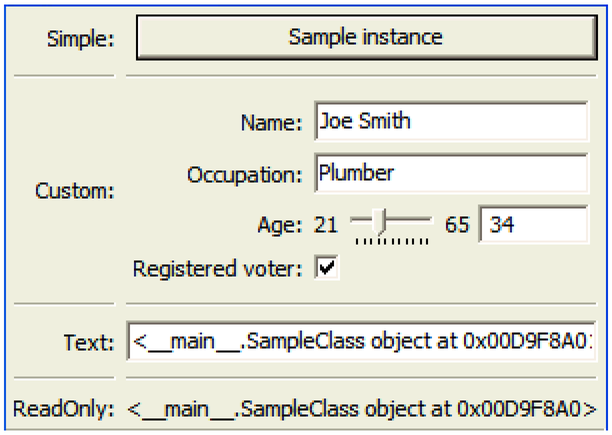
8.1.14 InstanceEditor()

Suitable for	Instance, Property, self, ThisClass, This
Default for	Instance, self, ThisClass, This
Required parameters	(none)
Optional parameters	*cachable*, *editable*, *id*, *kind*, *label*, *name*, *object*, *orientation*, *values*, *view*

The editors generated by InstanceEditor() enable the user to select an instance, or edit an instance, or both.

8.1.14.1 Editing a Single Instance

In the simplest case, the user can modify the trait attributes of an instance assigned to a trait attribute, but cannot modify which instance is assigned.



Joe Smith

Plumber

34

<__main__.SampleClass object at 0x00D9F8A0:

<__main__.SampleClass object at 0x00D9F8A0>

Figure 36: Editor styles for instances

The custom style displays a user interface panel for editing the trait attributes of the instance. The simple style displays a button, which when clicked, opens a dialog box containing a user interface for the instance. The *kind* parameter specifies the kind of dialog box to open (see Section 4.1 on page 13). The *label* parameter specifies a label for the button in the simple interface. The *view* parameter specifies a view to use for the referenced instance's user interface; if this is not specified, the default view for the instance is used (see Section 5.1.1 on page 19).

The text and read-only styles display the string representation of the instance. They therefore cannot be used to modify the attributes of the instance. A user could modify the assigned instance if they happened to know the memory address of another instance of the same type, which is unlikely. These styles can useful for prototyping and debugging, but not for real applications.

8.1.14.2 Selecting Instances

You can add an option to select a different instance to edit. Use the *name* and *object* parameters to specify the name of a trait attribute in the context that contains a list of instances that can be selected or edited. The *object* parameter defaults to 'object', and need not be specified for a single-object context. (See Section 5.4 on page 23 for an explanation of contexts.) Using these parameters results in a drop-drown list box containing a list of text representations of the available instances. If the instances have a **name** trait attribute, it is used for the string in the list; otherwise, a user-friendly version of the class name is used.

For example, the following code defines a Team class and a Person class. A Team has a roster of Persons, and a captain. In the view for a team, the user can pick a captain and edit that person's information.

Example 15: Instance editor with instance selection

```
from enthought.traits.api    \
    import HasStrictTraits, Int, Regex, Str,

class Person ( HasStrictTraits ):
    name  = Str
    age   = Int
    phone = Regex( value = '000-0000',
                   regex = '\d\d\d[-]\d\d\d\d' )

    traits_view = View( 'name', 'age', 'phone' )

people = [
  Person( name = 'Dave',   age = 39, phone = '555-1212' ),
  Person( name = 'Mike',   age = 28, phone = '555-3526' ),
  Person( name = 'Joe',    age = 34, phone = '555-6943' ),
  Person( name = 'Tom',    age = 22, phone = '555-7586' ),
  Person( name = 'Dick',   age = 63, phone = '555-3895' ),
  Person( name = 'Harry',  age = 46, phone = '555-3285' ),
  Person( name = 'Sally',  age = 43, phone = '555-8797' ),
  Person( name = 'Fields', age = 31, phone = '555-3547' )
]

class Team ( HasStrictTraits ):

    name    = Str
    captain = Instance( Person )
    roster  = List( Person )

    traits_view =
        View( Item('name'),
              Item('_'),
              Item( 'captain',
                    label='Team Captain',
                    editor =
                        InstanceEditor( name = 'roster',
                                        editable = True),
                          style = 'custom',
                         ),
              buttons = ['OK'])

if __name__ == '__main__':
```

```
Team( name    = 'Vultures',
      captain = people[0],
      roster  = people ).configure_traits()
```

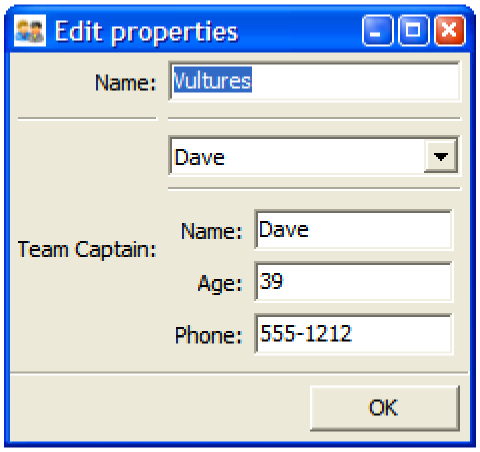


Figure 37: User interface for Example 15

If you want the user to be able to select instances, but not modify their contents, set the *editable* parameter to False. In that case, only the selection list for the instances appears, without the user interface for modifying instances.

8.1.14.3 Allowing Instances

You can specify what types of instances can be edited in an instance editor, using the *values* parameter. This parameter is a list of items describing the type of selectable or editable instances. These items must be instances of subclasses of **enthought.traits.ui.api.InstanceChoiceItem**. If you want to generate new instances, put an InstanceFactoryChoice instance in the *values* list that describes the instance to create. If you want certain types of instances to be dropped on the editor, use an InstanceDropChoice instance in the values list.

8.1.15 KivaFontEditor()

Suitable for	KivaFont
Default for	KivaFont
Required parameters	(none)
Optional parameters	(none)

The editors generated by KivaFontEditor() are identical in appearance and function to those generated by FontEditor() (see Section 8.1.11 on page 63). They are used to display and edit KivaFont traits.

8.1.16 ListEditor()

Suitable for	List
Default for	List[16]
Required parameters	(none)
Optional parameters	*editor*, *rows*, *style*, *trait_handler*, *use_notebook* The following parameters are used only if *use_notebook* is True: *deletable*, *dock_style*, *export*, *page_name*, *select*, *view*

The editors generated by ListEditor() enable the user to modify the contents of a list, both by editing the individual items and by adding, deleting, and reordering items within the list.

[16] If a List is made up of HasTraits objects, a table editor is used instead; see Section 8.2.3 on page 84.

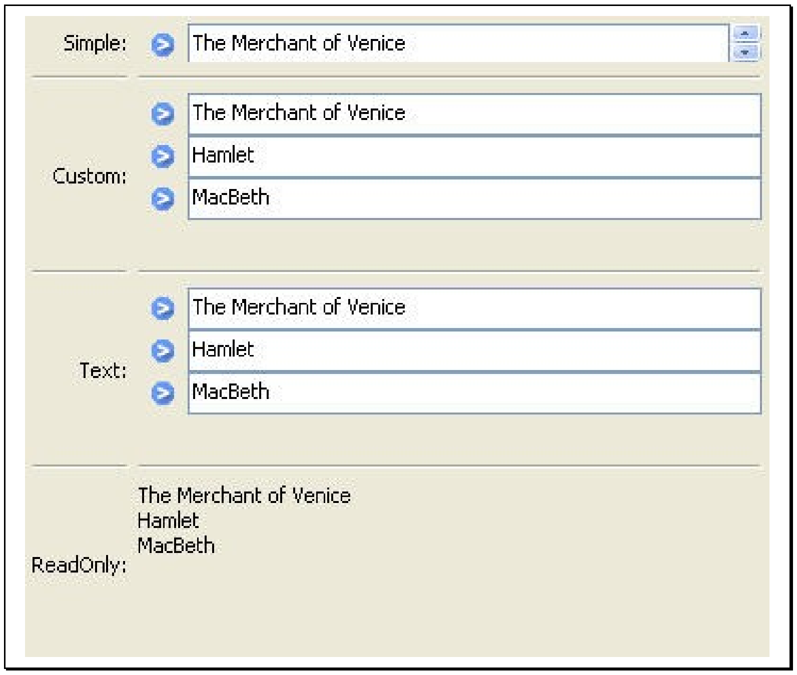


Figure 38: List editor styles

The simple style displays a single item at a time, with small arrows on the right side to scroll the display. The custom style shows multiple items. The number of items displayed is controlled by the *rows* parameter; if the number of items in the list exceeds this value, then the list display scrolls. The editor used for each item in the list is determined by the *editor* and *style* parameters. The text style of list editor is identical to the custom style, except that the editors for the items are text editors. The read-only style displays the contents of the list as static text.

By default, the items use the trait handler appropriate to the type of items in the list. You can specify a different handler to use for the items using the *trait_handler* parameter.

For the simple, custom, and text list editors, a button appears to the left of each item editor; clicking this button opens a context menu for modifying the list, as shown in Figure 39.

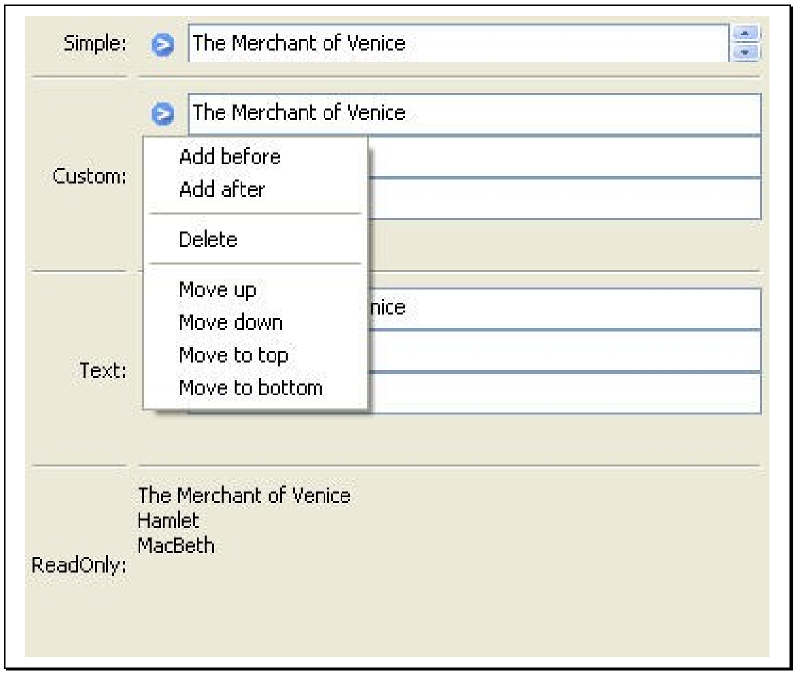
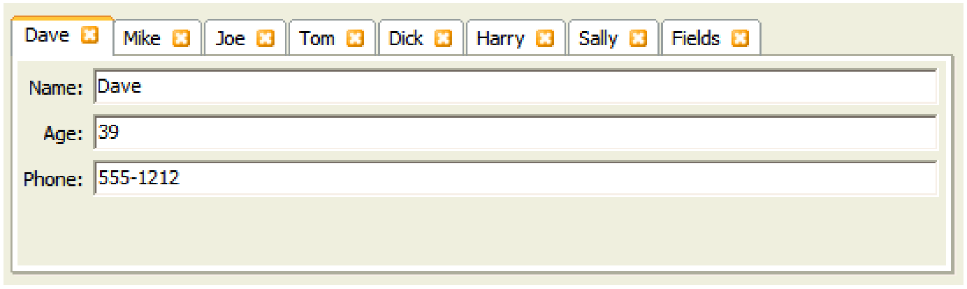


Figure 39: List editor showing context menu

In addition to the four standard styles for list editors, fifth list editor user interface option is available. If *use_notebook* is True, then the list editor displays the list as a “notebook” of tabbed pages, one for each item in the list, as shown in Figure 40. This style can be useful in cases where the list items are instances with their own views. Items in the list can be deleted if the *deletable* parameter is True; items cannot be added through this style of editor.



Name: Dave
Age: 39
Phone: 555-1212

Figure 40: Notebook list editor

8.1.17 NullEditor()

Suitable for	controlling layout
Default for	(none)

Required parameters	(none)
Optional parameters	(none)

The NullEditor() factory generates a completely empty panel. It is used by the Spring subclass of Item, to generate a blank space that uses all available extra space along its layout orientation. You can also use it to create a blank area of a fixed height and width.

8.1.18 RangeEditor()

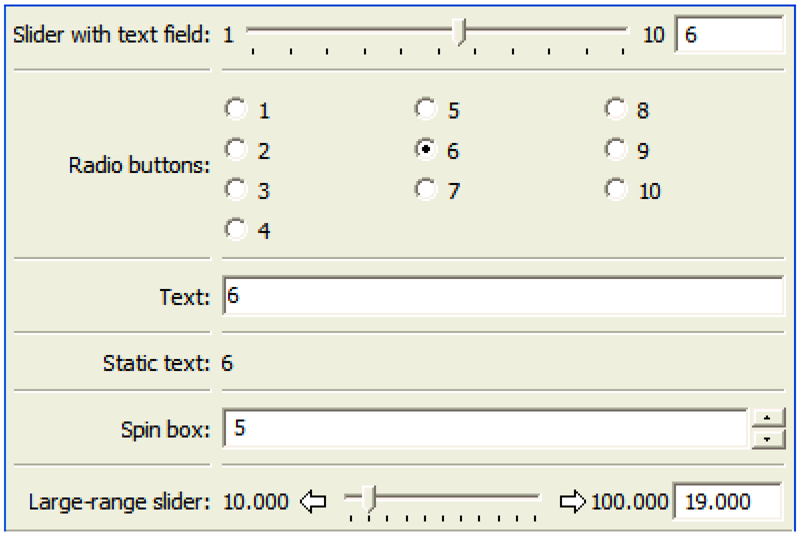
Suitable for	Range
Default for	Range
Required parameters	(none)
Optional parameters	*auto_set*, *cols*, *enter_set*, *format*, *high_label*, *high_name*, *label_width*, *low_label*, *low_name*, *mode*

The editors generated by RangeEditor() enable the user to specify numeric values within a range. The widgets used to display the range vary depending on both the numeric type and the size of the range, as described in Table 7 and shown in Figure 41. If one limit of the range is unspecified, then a text editor is used.

Table 7: Range editor widgets

	Simple	Custom	Text	Read-only
Integer: Small Range (Size 0-16)	Slider with text field	Radio buttons	Text field	Static text
Integer: Medium Range (Size 17-101)	Slider with text field	Slider with text field	Text field	Static text
Integer: Large Range (Size > 101)	Spin box	Spin box	Text field	Static text
Floating Point: Small Range (Size <= 100.0)	Slider with text field	Slider with text field	Text field	Static text

	Simple	Custom	Text	Read-only
Floating Point: Large Range (Size > 100.0)	Large-range slider	Large-range slider	Text field	Static text



Slider with text field: 1 10 6
Radio buttons: 1 2 3 4 5 6 7 8 9 10
Text: 6
Static text: 6
Spin box: 5
Large-range slider: 10.000 100.000 19.000

Figure 41: Range editor widgets

In the large-range slider, the arrows on either side of the slider move the editable range, so that the user can move the slider more precisely to the desired value.

You can override the default widget for each type of editor using the *mode* parameter, which can have the following values:

- 'auto'—The default widget, as described in Table 7
- 'slider'—Simple slider with text field
- 'xslider'—Large-range slider with text field
- 'spinner'—Spin box
- 'enum'—Radio buttons
- 'text'—Text field

You can set the limits of the range dynamically, using the *low_name* and *high_name* parameters to specify trait attributes that contain the low and high limit values; use *low_label*, *high_label* and *label_width* to specify labels for the limits.

8.1.19 SetEditor()

Suitable for	List
Default for	none
Required parameters	Either *values* or *name*
Optional parameters	*can_move_all*, *left_column_title*, *object*, *ordered*, *right_column_title*,

In the editors generated by SetEditor(), the user can select a subset of items from a larger set. The two lists are displayed in list boxes, with the candidate set on the left and the selected set on the right. The user moves an item from one set to the other by selecting the item and clicking a direction button (> for left-to-right and < for right-to-left).

Additional buttons can be displayed, depending on two Boolean parameters:

- If *can_move_all* is True, additional buttons appear, whose function is to move all items from one side to the other (>> for left-to-right and << for right-to-left).
- If *ordered* is True, additional buttons appear, labeled **Move up** and **Move down**, which affect the position of the selected item within the set in the right list box.

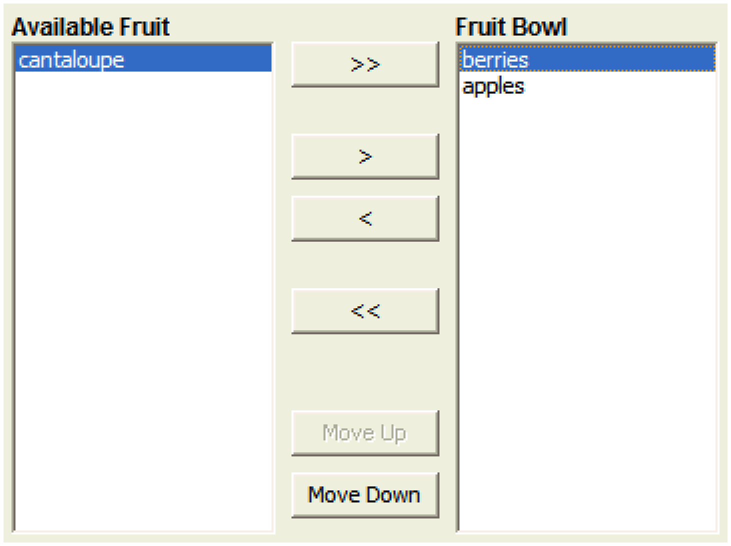


Figure 42: Set editor showing all possible buttons

You can specify the set of candidate items in either of two ways:

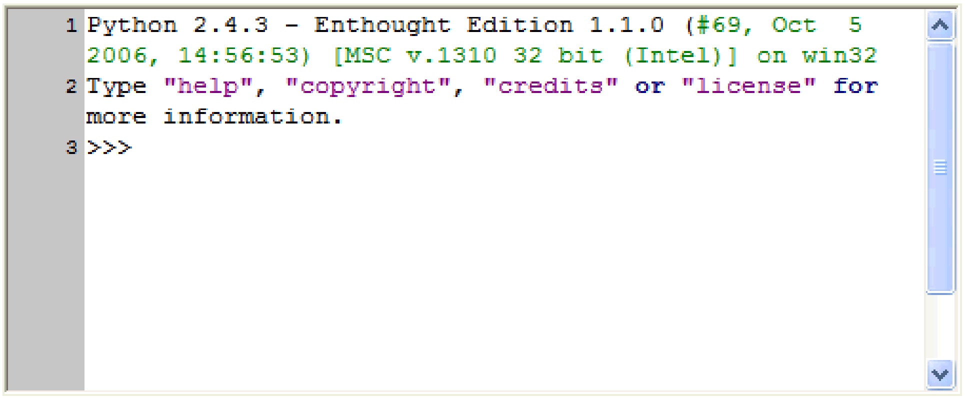
- Set the *values* parameter to a list, tuple, dictionary, or mapped trait.

- Set the *name* parameter to the name of a trait attribute that contains the list; use the *object* parameter to specify the object in the context whose attribute is name. The default value for *object* is 'object', so you do not need to specify it if the target object is the only one in the context. (See Section 5.4 on page 23 for information on contexts.)

8.1.20 ShellEditor()

Suitable for	special
Default for	PythonValue
Required parameters	(none)
Optional parameters	(none)

The editor generated by ShellEditor() displays an interactive Python shell.



```
Python 2.4.3 - Enthought Edition 1.1.0 (#69, Oct  5
2006, 14:56:53) [MSC v.1310 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for
more information.
>>>
```

Figure 43: Python shell editor

8.1.21 TextEditor()

Suitable for	all
Default for	Str, String, Password, Unicode, Int, Float, Dict, CStr, CUnicode, and any trait that does not have a specialized TraitHandler
Required parameters	(none)
Optional parameters	*auto_set*, *enter_set*, *evaluate*, *evaluate_name*, *mapping*, *multi_line*, *password*

The editor generated by TextEditor() displays a text box. For the custom style, it is a multi-line field; for the read-only style, it is static text. If *password* is True, the text is obscured.

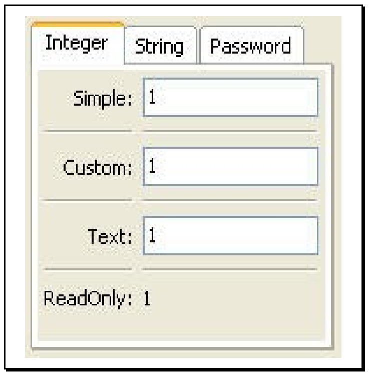


Figure 44: Text editor styles for integers

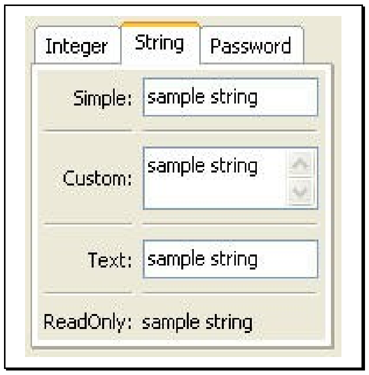


Figure 45: Text editor styles for strings

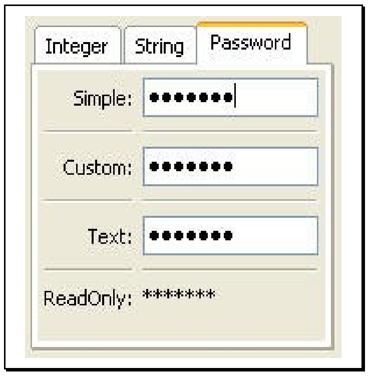


Figure 46: Text editor styles for passwords

You can specify whether the trait being edited is updated on every keystroke (`auto_set=True`) or when the user presses the Enter key (`enter_set=True`). If *auto_set* and *enter_set* are False, the trait is updated when the user shifts the input focus to another widget.

You can specify a mapping from user input values to other values with the *mapping* parameter. You can specify a function to evaluate user input, either by passing a reference to it in the *evaluate* parameter, or by passing the name of a trait that references it in the *evaluate_name* parameter.

8.1.22 TupleEditor()

Suitable for	Tuple
Default for	Tuple
Required parameters	(none)
Optional parameters	*cols*, *labels*, *traits*

The simple and custom editors generated by TupleEditor() provide a widget for each slot of the tuple being edited, based on the type of data in the slot. The text and read-only editors edit or display the text representation of the tuple.

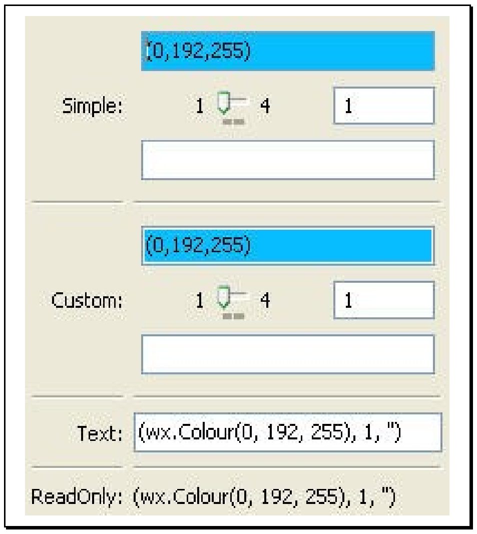

(0,192,255)
Simple:
1
4
1
(0,192,255)
Custom:
1
4
1
Text:
(wx.Colour(0, 192, 255), 1, '')
ReadOnly:
(wx.Colour(0, 192, 255), 1, '')

Figure 47: Tuple editor styles

You can specify the number of columns to use to lay out the widgets with the *cols* parameter. You can specify labels for the widgets with the *labels* parameter. You can also specify trait definitions for the slots of the tuple; however, this is usually implicit in the tuple being edited.

8.2 Advanced Trait Editors

The editor factories described in the following sections are more advanced than those in the previous section. In some cases, they require writing additional code; in others, the editors they generate are intended for use in complex user interfaces, in conjunction with other editors.

8.2.1 CustomEditor()

Suitable for	Special cases
Default for	(none)
Required parameters	*factory*
Optional parameters	*args*

Use CustomEditor() to create an "editor" that is a non-Traits-based custom control. The *factory* parameter must be a function that generates the custom control. The function must have the following signature:

```
factory_function(window_parent, editor, *args,
    **kwargs)
```

- *window_parent*—The parent window for the control
- *editor*—The editor object created by CustomEditor()

Additional arguments, if any, can be passed as a tuple in the *args* parameter of CustomEditor().

For an example of using CustomEditor(), run `NumericModelExplorer_demo.py` in the "`demos/Traits UI Demo/Advanced`" subdirectory of the Traits UI package directory. Examine the implementation of the NumericModelExplorer class in the **enthought.model.numeric_model_explorer** module. CustomEditor() is used to generate the plots in the user interface.

8.2.2 KeyBindingEditor()

The KeyBindingEditor() factory differs from other trait editor factories because it generates an editor, not for a single attribute, but for an object of a particular class, **enthought.traits.ui.key_bindings.KeyBindings**. A KeyBindings object is a list of bindings between key codes and handler methods.

A key bindings editor is a separate dialog box that displays the string representation of each key code and a description of the corresponding method. The user can click a text box, and then press a key or key combination to associate that key press with a method.

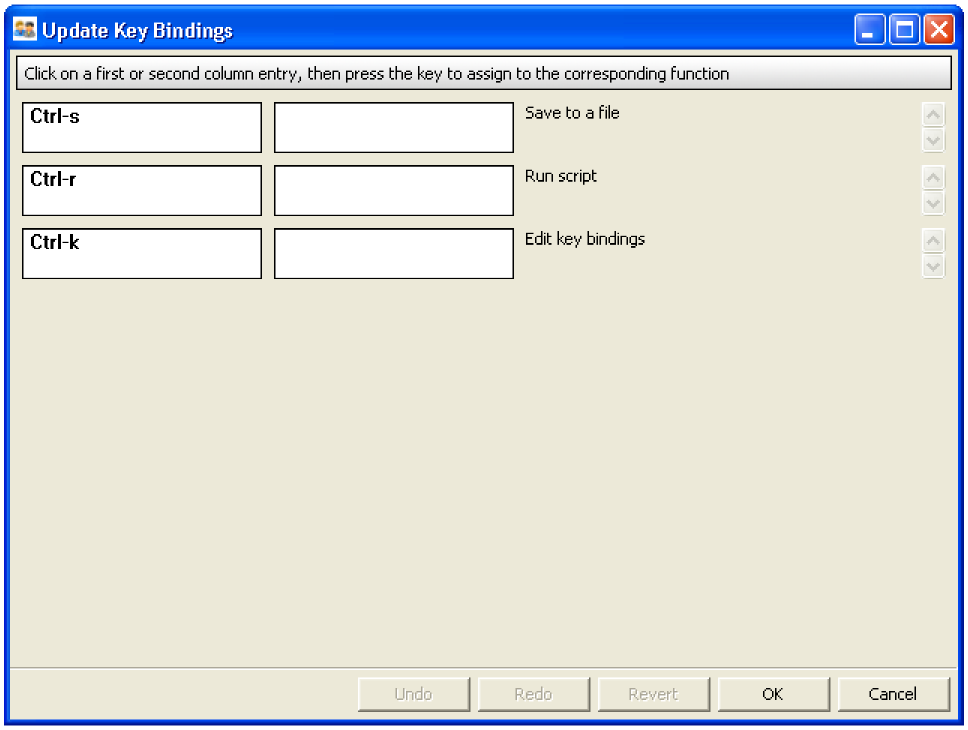


Figure 48: Key binding editor dialog box

The following code example creates a user interface containing a code editor with associated key bindings, and a button that invokes the key binding editor.

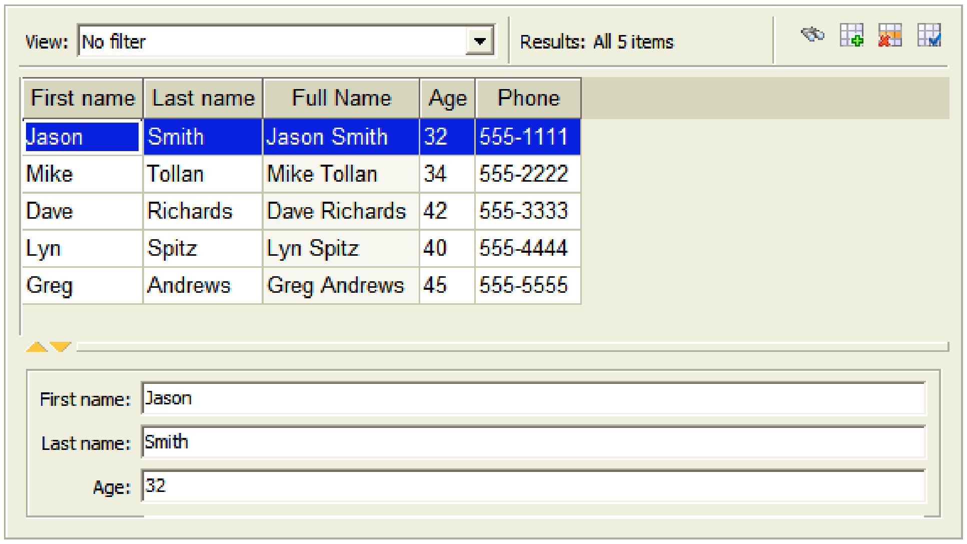
Example 16: Code editor with key binding editor

```
from enthought.traits.api \
    import Button, Code, HasPrivateTraits, Str

from enthought.traits.ui.api \
    import View, Item, Group, Handler, CodeEditor

from enthought.traits.ui.key_bindings \
    import KeyBinding, KeyBindings

from enthought.traits.ui.menu \
    import NoButtons

key_bindings = KeyBindings(
    KeyBinding( binding1    = 'Ctrl-s',
                description = 'Save to a file',
                method_name = 'save_file' ),
    KeyBinding( binding1    = 'Ctrl-r',
                description = 'Run script',
                method_name = 'run_script' ),
    KeyBinding( binding1    = 'Ctrl-k',
                description = 'Edit key bindings',
                method_name = 'edit_bindings' )
)

# Traits UI Handler class for bound methods
class CodeHandler ( Handler ):

    def save_file ( self, info ):
        info.object.status = "save file"

    def run_script ( self, info ):
        info.object.status = "run script"

    def edit_bindings ( self, info ):
        info.object.status = "edit bindings"
        key_bindings.edit_traits()

class KBCodeExample ( HasPrivateTraits ):

    code   = Code
    status = Str
    kb    = Button(label='Edit Key Bindings')

    view = View( Group (
                 Item( 'code',
                       style     = 'custom',
                       resizable = True,
                       editor = CodeEditor(
                                    key_bindings =
                                        key_bindings ) ),
                 Item('status', style='readonly'),
                 'kb',
                 orientation = 'vertical',
                 show_labels = False,
                 ),
               id = 'KBCodeExample',
               title = 'Code Editor With Key Bindings',
               resizable = True,
               buttons   = NoButtons,
               handler   = CodeHandler() )

    def _kb_fired( self, event ):
        key_bindings.edit_traits()


if __name__ == '__main__':
    KBCodeExample().configure_traits()
```

8.2.3 TableEditor()

Suitable for	List(*InstanceType*)
Default for	(none)
Required parameters	*columns* or *columns_name*
Optional parameters	See *Traits API Reference*, **enthought.traits.ui.wx.table_editor .ToolkitEditorFactory** attributes.

TableEditor() generates a editor that displays instances in a list as rows in a table, with attributes of the instances as values in columns. You must specify the columns in the table. Optionally, you can provide filters for filtering the set of displayed items, and you can specify a wide variety of options for interacting with and formatting the table.



First name	Last name	Full Name	Age	Phone
Jason	Smith	Jason Smith	32	555-1111
Mike	Tollan	Mike Tollan	34	555-2222
Dave	Richards	Dave Richards	42	555-3333
Lyn	Spitz	Lyn Spitz	40	555-4444
Greg	Andrews	Greg Andrews	45	555-5555

First name: Jason
Last name: Smith
Age: 32

Figure 49: Table editor

To see the code that results in Figure 49, refer to `TableEditor_demo.py` in the `demos/Traits UI Demo/Standard Editors` subdirectory of the Traits UI package. This example demonstrates object columns, expression columns, filters, searching, and adding and deleting rows.

The parameters for TableEditor() can be grouped in several broad categories, described in the following sections.

- Specifying columns
- Managing items

- Editing the table or items
- Controlling the table display
- Controlling table formatting

8.2.3.1 Specifying Columns

You must provide the TableEditor() factory with a list of columns for the table. You can specify this list directly, as the value of the *columns* parameter, or indirectly, in a context attribute referenced by the *columns_name* parameter.

The items in the list must be instances of **enthought.traits.ui.api.TableColumn**, or of a subclass of TableColumn. Some subclasses of TableColumn that are provided by the Traits UI package include ObjectColumn, ListColumn, NumericColumn, and ExpressionColumn. (See the *Traits API Reference* for details about these classes.) In practice, most columns are derived from one of these subclasses, rather than from TableColumn. For the usual case of editing trait attributes on objects in the list, use ObjectColumn. You must specify the *name* parameter to the ObjectColumn() constructor, referencing the name of the trait attribute to be edited.

You can specify additional columns that are not initially displayed using the *other_columns* parameter. If the *configurable* parameter is True (the default), a **Set user preferences for table** icon (▦) appears on the table's toolbar. When the user clicks this icon, a dialog box opens that enables the user to select and order the columns displayed in the table, as shown in Figure 50. (The dialog box is implemented using a set editor; see Section 8.1.19 on page 77.) Any columns that were specified in the *other_columns* parameter are listed in the left list box of this dialog box, and can be displayed by moving them into the right list box.

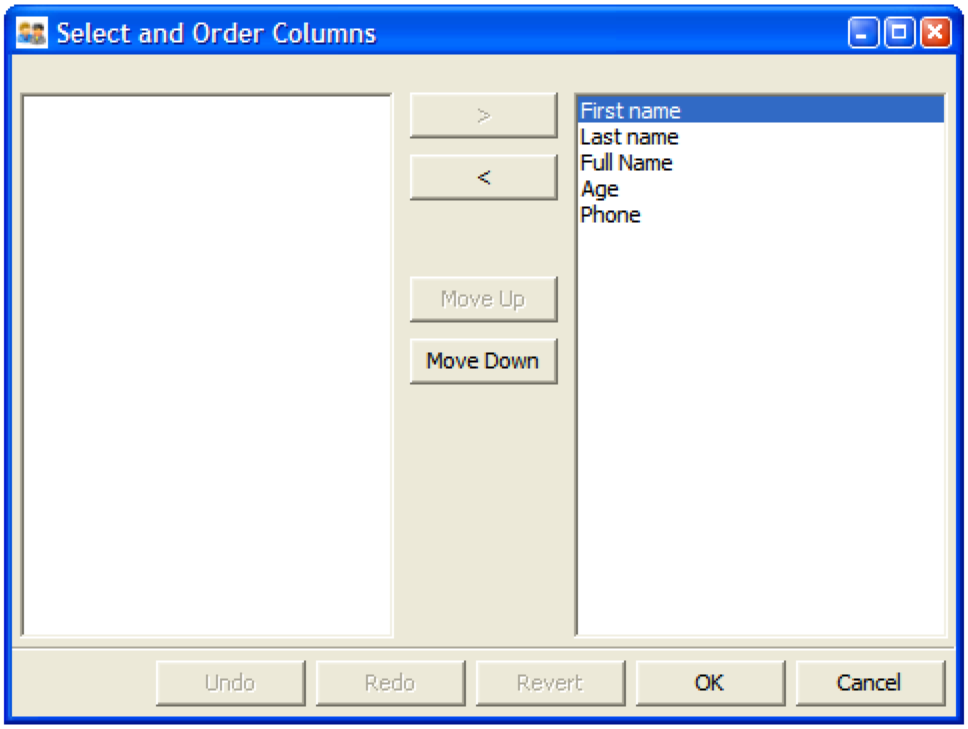


Figure 50: Column selection dialog box for a table editor

8.2.3.2 Managing Items

Table editors support several mechanisms to help users locate items of interest.

8.2.3.2.1 Organizing Items

Table editors provide two mechanisms for the user to organize the contents of a table: sorting and reordering. The user can sort the items based on the values in a column, or the user can manually order the items. Usually, only one of these mechanisms is used in any particular table, although the Traits UI package does not enforce a separation. If the user has manually ordered the items, sorting them would throw away that effort.

If the *reorderable* parameter is True, **Move up** (⬆) and **Move down** (⬇) icons appear in the table toolbar. Clicking one of these icons changes the position of the selected item.

If the *sortable* parameter is True (the default), then the user can sort the items in the table based on the values in a column by clicking the header of that column.

- On the first click, the items are sorted in ascending order. The characters “>>” appear in the column header to indicate that the table is sorted ascending on this column’s values.
- On the second click, the items are sorted descending order. The characters “<<” appear in the column header to indicate that the table is sorted descending on this column’s values.
- On the third click, the items are restored to their original order, and the column header is undecorated.

If the *sort_model* parameter is true, the items in the list being edited are sorted when the table is sorted. The default value is False, in which case, the list order is not affected by sorting the table.

If *sortable* is True and *sort_model* is False, then a **Do not sort columns** icon () appears in the table toolbar. Clicking this icon restores the original sort order.

If the *reverse* parameter is True, then the items in the underlying list are maintained in the reverse order of the items in the table (regardless of whether the table is sortable or reorderable).

8.2.3.2.2 Filtering and Searching

You can provide an option for the user to apply a filter to a table, so that only items that pass the filter are displayed. This feature can be very useful when dealing with lengthy lists. To add the filtering feature, use the *filters* parameter to specify a list of table filters, which must be instances of **enthought.traits.ui.api.TableFilter**, or of a subclass of TableFilter. Some subclasses of TableFilter that are provided by the Traits UI package include EvalTableFilter, RuleTableFilter, and MenuTableFilter. (See the *Traits API Reference* for details about these classes.) The Traits UI package also provides instances of these filter classes as “templates”, which cannot be edited or deleted, but which can be used as models for creating new filters.

When *filters* is specified, a drop-down list box appears in the table toolbar, containing the filters that are available for the user to apply. When the user selects a filter, it is automatically applied to the table. A status message to the right of the filters list indicates what subset of the items in the table is currently displayed. A special item in the filter list, named **Customize**, is always provided; clicking this item opens a dialog box that enables the user to create new filters, or to edit or delete existing filters (except templates).

If you want a particular filter to be applied to the table when it is first displayed, use the *filter* parameter to specify it. If this parameter is None or unspecified, all items are displayed in the table, and an item named **No filter** is selected in the filter list.

You can also provide an option for the user to use filters to search the table. If you set the *search* parameter to an instance of TableFilter (or of a subclass), a **Search table** icon () appears on the table toolbar. Clicking this icon opens a **Search for** dialog box, which enables the user to specify filter criteria, to browse through matching items, or select all matching items.

8.2.3.2.3 Interacting with Items

As the user clicks in the table, you may wish to enable certain program behavior.

You can use the *selected* parameter to specify the name of a trait attribute on the current context object to synchronize with the user's current selection. For example, you can enable or disable menu items or toolbar icons depending on which item is selected. The synchronization is two-way; you can set the attribute referenced by *selected* to force the table to select a particular item.

The *on_select* and *on_dclick* parameters are callables to invoke when the user selects or double-clicks an item, respectively.

You can define a shortcut menu that opens when the user right-clicks an item. Use the *menu* parameter to specify a Traits UI or PyFace Menu, containing Action objects for the menu commands.

8.2.3.3 Editing the Table

The Boolean *editable* parameter controls whether the table or its items can be modified in any way. This parameter defaults to True, except when the style is 'readonly'. Even when the table as a whole is editable, you can control whether individual columns are editable through the **editable** attribute of TableColumn.

8.2.3.3.1 Adding Items

To enable users to add items to the table, specify as the *row_factory* parameter a callable that generates an object that can be added to the list in the table; for example, the class of the objects in the table.

When *row_factory* is specified, an **Insert new item** icon () appears in the table toolbar, which generates a new row in the table. Optionally, you can use *row_factory_args* and *row_factory_kw* to specify positional and keyword arguments to the row factory callable.

To save users the trouble of mousing to the toolbar, you can enable them to add an item by selecting the last row in the table. To do this, set *auto_add* to True. In this case, the last row is blank until the user sets values. Pressing Enter creates the new item and generates a new, blank last row.

8.2.3.3.2 Deleting Items

The *deletable* parameter controls whether items can be deleted from the table. This parameter can be a Boolean (defaulting to False) or a callable; the callable must take an item as an argument and handle deleting it. If *deletable* is not False, a **Delete current item** icon () appears on the table toolbar; clicking it deletes the item corresponding to the row that is selected in the table.

8.2.3.3.3 Modifying Items

The user can modify items in two ways.

- For columns that are editable, the user can change an item's value directly in the table. The editor used for each attribute in the table is the simple style of editor for the corresponding trait.
- Alternatively, you can specify a View for editing instances, using the *edit_view* parameter. The resulting user interface appears in a sub-panel to the right or below the table (depending on the *orientation* parameter). You can specify a handler to use with the view, using *edit_view_handler*. You can also specify the subpanel's height and width, with *edit_view_height* and *edit_view_width*.

8.2.3.4 Defining the Layout

Some of the parameters for the TableEditor() factory affect global aspects of the display of the table.

- *auto_size*—If True, the cells of the table automatically adjust to the optimal size based on their contents.

- *orientation*—The layout of the table relative to its associated editor pane. Can be 'horizontal' or 'vertical'.
- *rows*—The number of visible rows in the table.
- *show_column_labels*—If True (the default), displays labels for the columns. You can specify the labels to use in the column definitions; otherwise, a "user friendly" version of the trait attribute name is used.

8.2.3.5 Defining the Format

The TableEditor() factory supports a variety of parameters to control the visual formatting of the table, such as colors, fonts, and sizes for lines, cells, and labels. For details, refer to the *Traits API Reference*, **enthought.traits.ui.wx.table_editor.ToolkitEditorFactory** attributes.

You can also specify formatting options for individual table columns when you define them.

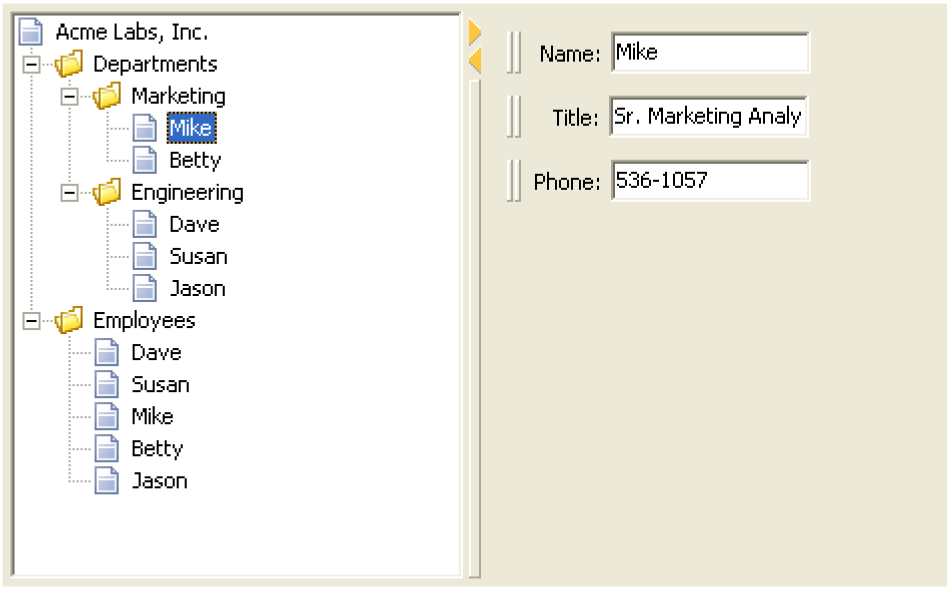
8.2.4 TreeEditor()

Suitable for	Instance
Default for	(none)
Required parameters	*nodes* (required except for shared editors; see Section 8.2.4.2.5)
Optional parameters	*auto_open, editable, editor, hide_root, icon_size, lines_mode, on_dclick, on_select, orientation, selected, shared_editor, show_icons*

TreeEditor() generates a hierarchical tree control, consisting of nodes. It is useful for cases where objects contain lists of other objects.

The tree control is displayed in one pane of the editor, and a user interface for the selected object is displayed in the other pane. The layout orientation of the tree and the object editor is determined by the *orientation* parameter of TreeEditor(), which can be 'horizontal' or 'vertical'.

You must specify the types of nodes that can appear in the tree using the *nodes* parameter, which must be a list of instances of TreeNode (or of subclasses of TreeNode).



Name: Mike

Title: Sr. Marketing Analy

Phone: 536-1057

Figure 51: Tree editor

The following example shows the code that produces the editor shown in Figure 51.

Example 17: Code for example tree editor

```
from enthought.traits.api \
    import HasTraits, Str, Regex, List, Instance
from enthought.traits.ui.api \
    import TreeEditor, TreeNode, View, Item, VSplit, \
           HGroup, Handler, Group
from enthought.traits.ui.menu \
    import Menu, Action, Separator
from enthought.traits.ui.wx.tree_editor \
    import NewAction, CopyAction, CutAction, \
           PasteAction, DeleteAction, RenameAction

# DATA CLASSES

class Employee ( HasTraits ):
    name  = Str( '<unknown>' )
    title = Str
    phone = Regex( regex = r'\d\d\d-\d\d\d\d' )

    def default_title ( self ):
        self.title = 'Senior Engineer'

class Department ( HasTraits ):
    name      = Str( '<unknown>' )
    employees = List( Employee )
```

```
class Company ( HasTraits ):
    name        = Str( '<unknown>' )
    departments = List( Department )
    employees   = List( Employee )

class Owner ( HasTraits ):
    name    = Str( '<unknown>' )
    company = Instance( Company )

# INSTANCES

jason = Employee(
     name  = 'Jason',
     title = 'Engineer',
     phone = '536-1057' )

mike = Employee(
     name  = 'Mike',
     title = 'Sr. Marketing Analyst',
     phone = '536-1057' )

dave = Employee(
     name  = 'Dave',
     title = 'Sr. Engineer',
     phone = '536-1057' )

susan = Employee(
     name  = 'Susan',
     title = 'Engineer',
     phone = '536-1057' )

betty = Employee(
     name  = 'Betty',
     title = 'Marketing Analyst' )

owner = Owner(
    name    = 'wile',
    company = Company(
        name = 'Acme Labs, Inc.',
        departments = [
            Department(
                name = 'Marketing',
                employees = [ mike, betty ]
            ),
            Department(
                name = 'Engineering',
                employees = [ dave, susan, jason ]
            )
        ],
        employees = [ dave, susan, mike, betty, jason ]
    )
)
```

```
# View for objects that aren't edited
no_view = View()

# Actions used by tree editor context menu

def_title_action = Action(name='Default title',
                          action = 'object.default')

dept_action = Action(
    name='Department',
    action='handler.employee_department(editor,object)')

# View used by tree editor
employee_view = View(
    VSplit(
        HGroup( '3', 'name' ),
        HGroup( '9', 'title' ),
        HGroup( 'phone' ),
        id = 'vsplit' ),
    id = 'enthought.traits.doc.example.treeeditor',
    dock = 'vertical' )

class TreeHandler ( Handler ):

    def employee_department ( self, editor, object ):
        dept = editor.get_parent( object )
        print '%s works in the %s department.' %\
            ( object.name, dept.name )

# Tree editor
tree_editor = TreeEditor(
    nodes = [
        TreeNode( node_for  = [ Company ],
                  auto_open = True,
                  children  = '',
                  label     = 'name',
                  view      = View( Group('name',
                                 orientation='vertical',
                                 show_left=True )) ),
        TreeNode( node_for  = [ Company ],
                  auto_open = True,
                  children  = 'departments',
                  label     = '=Departments',
                  view      = no_view,
                  add       = [ Department ] ),
        TreeNode( node_for  = [ Company ],
                  auto_open = True,
                  children  = 'employees',
                  label     = '=Employees',
                  view      = no_view,
                  add       = [ Employee ] ),
        TreeNode( node_for  = [ Department ],
                  auto_open = True,
                  children  = 'employees',
                  label     = 'name',
                  menu      = Menu( NewAction,
                                    Separator(),
                                    DeleteAction,
                                    Separator(),
                                    RenameAction,
                                    Separator(),
                                    CopyAction,
                                    CutAction,
                                    PasteAction ),
                  view      = View( Group ('name',
                                   orientation='vertical',
                                   show_left=True )),
                  add       = [ Employee ] ),
        TreeNode( node_for  = [ Employee ],
                  auto_open = True,
                  label     = 'name',
                  menu=Menu( NewAction,
                             Separator(),
                             def_title_action,
                             dept_action,
                             Separator(),
                             CopyAction,
                             CutAction,
                             PasteAction,
                             Separator(),
                             DeleteAction,
                             Separator(),
                             RenameAction ),
                  view = employee_view )
    ]
)
# The main view
view = View(
           Group(
               Item(
                    name = 'company',
                    id = 'company',
                    editor = tree_editor,
                    resizable = True ),
                orientation = 'vertical',
                show_labels = True,
                show_left = True, ),
            title = 'Company Structure',
            id = \
             'enthought.traits.ui.tests.tree_editor_test',
            dock = 'horizontal',
            drop_class = HasTraits,
            handler = TreeHandler(),
            buttons = [ 'Undo', 'OK', 'Cancel' ],
            resizable = True,
            width = .3,
            height = .3 )

if __name__ == '__main__':
    owner.configure_traits( view = view )
```

8.2.4.1 Defining Nodes

For details on the attributes of the TreeNode class, refer to the *Traits API Reference*.

You must specify the classes whose instances the node type applies to. Use the **node_for** attribute of TreeNode to specify a list of classes; often, this list contains only one class. You can have more than one node type that applies to a particular class; in this case, each object of that class is represented by multiple nodes, one for each applicable node type. In Figure 51, one Company object is represented by the nodes labeled “Acme Labs, Inc.”, “Departments”, and “Employees”.

8.2.4.1.1 A Node Type without Children

To define a node type without children, set the **children** attribute of TreeNode to the empty string. In Example 16, the following lines define the node type for the node that displays the company name, with no children:

```
TreeNode( node_for  = [ Company ],
          auto_open = True,
          children  = '',
          label     = 'name',
          view      = View( Group('name',
                                 orientation='vertical',
                                 show_left=True )) ),
```

8.2.4.1.2 A Node Type with Children

To define a node type that has children, set the **children** attribute of TreeNode to the name of a trait on the object that this is a node for; the named trait contain a list of the object’s children. In Example 16, the following lines define the node type for the node that contains the departments of a company. The node type is for instances of Company, and ‘departments’ is a trait attribute of Company.

```
TreeNode( node_for  = [ Company ],
          auto_open = True,
          children  = 'departments',
          label     = '=Departments',
          view      = no_view,
          add       = [ Department ] ),
```

8.2.4.1.3 Setting the Label of a Tree Node

The **label** attribute of Tree Node can work one of two ways: as a trait attribute name, or as a literal string.

If the value is a simple string, it is interpreted as the name of a trait attribute on the object that the node is for, whose value is used as the label. This approach is used in the code snippet in Section 8.2.4.1.1.

If the value is a string that begins with an equals sign ('='), the rest of the string is used as the literal label. This approach is used in Section 8.2.4.1.2.

You can also specify a callable to format the label of the node, using the **formatter** attribute of TreeNode.

8.2.4.2 Defining Operations on Nodes

You can use various attributes of TreeNode to define operations or behavior of nodes.

8.2.4.2.1 Shortcut Menus on Nodes

Use the **menu** attribute of TreeNode to define a shortcut menu that opens when the user right-clicks on a node. The value is a Traits UI or PyFace menu containing Action objects for the menu commands. In Example 16, the following lines define the node type for employees, including a shortcut menu for employee nodes:

```
        TreeNode( node_for  = [ Department ],
                  auto_open = True,
                  children  = 'employees',
                  label     = 'name',
                  menu      = Menu( NewAction,
                                    Separator(),
                                    DeleteAction,
                                    Separator(),
                                    RenameAction,
                                    Separator(),
                                    CopyAction,
                                    CutAction,
                                    PasteAction ),
                  view      = View( Group ('name',
                                     orientation='vertical',
                                     show_left=True )),
                  add       = [ Employee ] ),
```

8.2.4.2.2 Allowing the Hierarchy to Be Modified

If a node contains children, you can allow objects to be added to its set of children, through operations such as dragging and dropping, copying and pasting, or creating new objects. Two attributes control these operations: **add** and **move**. Both are lists of classes. The **add** attribute contains classes that can be added by any means, including creation. The code snippet in the preceding section (8.2.4.2.1) includes an example of the **add** attribute. The **move** attribute contains classes that can be dragged and dropped, but not created. The **move** attribute need not be specified if all classes that can be moved can also be created (and therefore are specified in the **add** value).

NOTE: The 'add' attribute alone is not enough to create objects. Specifying the **add** attribute makes it possible for objects of the specified classes to be created, but by itself, it does not provide a way for the user to do so. In the code snippet in the preceding section (8.2.4.2.1), 'NewAction' in the Menu constructor call defines a **New > Employee** menu item that creates Employee objects.

In the example tree editor, users can create new employees using the **New > Employee** shortcut menu item, and they can drag an employee node and drop it on a department node. The corresponding object becomes a member of the appropriate list.

You can specify the label that appears on the **New** submenu when adding a particular type of object, using the **name** attribute of TreeNode. Note that you set this attribute on the tree node type that will be added by the menu item, not the node type that contains the menu item. For example, to change **New > Employee** to **New > Worker**, set `name = 'Worker'` on the tree node whose **node_for** value contains Employee. If this attribute is not set, the class name is used.

You can determine whether a node or its children can be copied, renamed, or deleted, by setting the following attributes on TreeNode:

- **copy**: If True (the default), the object's children can be copied.
- **delete**: If True (the default), the object's children can be deleted.
- **delete_me**: If True (the default), the object can be deleted.
- **rename**: If True (the default), the object's children can be renamed.
- **rename_me**: If True (the default), the object can be renamed.

As with **add**, you must also define actions to perform these operations.

8.2.4.2.3 Behavior on Nodes

As the user clicks in the tree, you may wish to enable certain program behavior.

You can use the *selected* parameter to specify the name of a trait attribute on the current context object to synchronize with the user's current selection. For example, you can enable or disable menu items or toolbar icons depending on which node is selected. The synchronization is two-way; you can set the attribute referenced by *selected* to force the tree to select a particular node.

The *on_select* and *on_dclick* parameters are callables to invoke when the user selects or double-clicks a node, respectively.

8.2.4.2.4 Expanding and Collapsing Nodes

You can control some aspects of expanding and collapsing of nodes in the tree.

The integer *auto_open* parameter of TreeEditor() determines how many levels are expanded below the root node, when the tree is first displayed. For example, if *auto_open* is 2, then two levels below the root node are displayed (whether or not the root node itself is displayed, which is determined by *hide_root*).

The Boolean **auto_open** attribute of TreeNode determines whether nodes of that type are expanded when they are displayed (at any time, not just on initial display of the tree). For example, suppose that a tree editor has *auto_open* setting of 2, and contains a tree node at level 3 whose **auto_open** attribute is True. The nodes at level 3 are not displayed initially, but when the user expands a level 2 node, displaying the level 3 node, that's nodes children are automatically displayed also. Similarly, the number of levels of nodes initially displayed can be greater than specified by the tree editor's *auto_open* setting, if some of the nodes have **auto_open** set to True.

If the **auto_close** attribute of TreeNode is set to True, then when a node is expanded, any siblings of that node are automatically closed. In other words, only one node of this type can be expanded at a time.

8.2.4.2.5 Editing Objects

One pane of the tree editor displays a user interface for editing the object that is selected in the tree. You can specify a View to use for each node type using the **view** attribute of TreeNode. If you do not specify a view, then the default view for the object is displayed. To suppress the editor pane, set the *editable* parameter of TreeEditor() to False; in this case, the objects represented by the nodes can still be modified by other means, such as shortcut menu commands.

You can define multiple tree editors that share a single editor pane. Each tree editor has its own tree pane. Each time the user selects a different node in any of the sharing tree controls, the editor pane updates to display the user interface for the selected object. To establish this relationship, do the following:

1. Call TreeEditor() with the *shared_editor* parameter set to True, without defining any tree nodes. The object this call returns defines the shared editor pane. For example:

```
my_shared_editor_pane = TreeEditor(shared_editor=True)
```

2. For each editor that uses the shared editor pane:

   - Set the *shared_editor* parameter of TreeEditor() to True.
   - Set the editor parameter of TreeEditor() to the object returned in Step 1.

   For example:

```
shared_tree_1 = TreeEditor(shared_editor = True,
                           editor = my_shared_editor_pane,
                           nodes = [ TreeNode( # …
                                             )
                                   ]
                           )
shared_tree_2 = TreeEditor(shared_editor = True,
                           editor = my_shared_editor_pane,
                           nodes = [ TreeNode( # …
                                              )
                                   ]
                           )
```

8.2.4.3 Defining the Format

Several parameters to TreeEditor() affect the formatting of the tree control:

- *show_icons*: If True (the default), icons are displayed for the nodes in the tree.
- *icon_size*: A two-integer tuple indicating the size of the icons for the nodes.
- *lines_mode*: Determines whether lines are displayed between related nodes. The valid values are 'on', 'off', and 'appearance' (the default). When set to 'appearance', lines are displayed except on Posix-based platforms.
- *hide_root*: If True, the root node in the hierarchy is not displayed. If this parameter were specified as True in Example 16, the node in Figure 51 that is labeled "Acme Labs, Inc." would not appear.

Additionally, several attributes of TreeNode also affect the display of the tree:

- **icon_path**: A directory path to search for icon files. This path can be relative to the module it is used in.
- **icon_item**: The icon for a leaf node.
- **icon_open**: The icon for a node with children whose children are displayed.
- **icon_group**: The icon for a node with children whose children are not displayed.

The wxWindows implementation automatically detects the bitmap format of the icon.

8.2.5 DropEditor()

Suitable for	Instance traits
Default for	(none)
Required parameters	(none)
Optional parameters	*binding*, *klass*, *readonly*

DropEditor() generates an editor that is a text field containing a string representation of the trait attribute's value. The user can change the value assigned to the attribute by dragging and dropping an object on the text field, for example, a node from a tree editor. If the *readonly* parameter is True (the default), the user cannot modify the value by typing in the text field.

You can restrict the class of objects that can be dropped on the editor by specifying the *klass* parameter.

You can specify that the dropped object must be a binding (**enthought.naming.api.Binding**) by setting the *binding* parameter to True. If so, the bound object is retrieved and checked to see if it can be assigned to the trait attribute.

If the dropped object (or the bound object associated with it) has a method named drop_editor_value(), it is called to obtain the value to assign to the trait attribute. Similarly, if the object has a method named drop_editor_update(), it is called to update the value displayed in the text editor. This method requires one parameter, which is the GUI control for the text editor.

8.2.6 DNDEditor()

Suitable for	Instance traits
Default for	(none)
Required parameters	(none)
Optional parameters	*drag_target, drop_target, image*

DNDEditor() generates an editor that represents a file or a HasTraits instance as an image that supports dragging and dropping. Depending on the editor style, the editor can be a *drag source* (the user can set the value of the trait attribute by dragging a file or object onto the editor, for example, from a tree editor), or *drop target* (the user can drag from the editor onto another target).

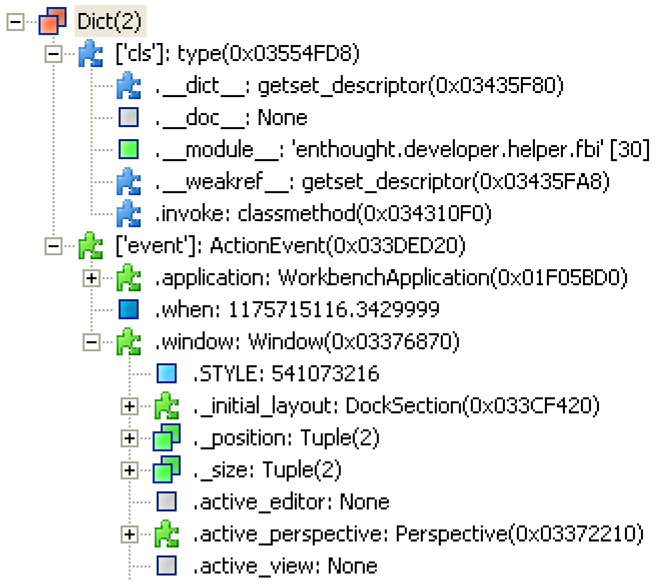
Table 8: Drag-and-drop editor style variations

Editor Style	*Drag Source?*	*Drop Target?*
Simple	Yes	Yes
Custom	No	Yes
Read-only	Yes	No

8.2.7 ValueEditor()

Suitable for	(any)
Default for	(none)
Required parameters	(none)
Optional parameters	*auto_open*

ValueEditor() generates a tree editor that displays Python values and objects, including all the objects' members. For example, Figure 52 shows a value editor that appears in the Frame-Based Inspector (FBI), in this case showing the local variables when FBI is invoked.



```
Dict(2)
    ['cls']: type(0x03554FD8)
        .__dict__: getset_descriptor(0x03435F80)
        .__doc__: None
        .__module__: 'enthought.developer.helper.fbi' [30]
        .__weakref__: getset_descriptor(0x03435FA8)
        .invoke: classmethod(0x034310F0)
    ['event']: ActionEvent(0x033DED20)
        .application: WorkbenchApplication(0x01F05BD0)
        .when: 1175715116.3429999
        .window: Window(0x03376870)
            .STYLE: 541073216
            ._initial_layout: DockSection(0x033CF420)
            ._position: Tuple(2)
            ._size: Tuple(2)
            .active_editor: None
            .active_perspective: Perspective(0x03372210)
            .active_view: None
```

Figure 52: Value editor from FBI

9 Advanced Editor Concepts

9.1 Interacting with an Editor Through the UI Object

9.1.1 Accessing Trait Editors Using Item IDs

9.1.2 Controlling Editor Status Using ‘enabled’ and ‘disabled’

9.2 Defining a Custom Editor

9.2.1 Defining the Editor Factory

9.2.1.1 Standard Traits: format_str, format_func, is_grid_cell

9.2.1.2 Standard methods: init(), simple_editor(), custom_editor(), text_editor(), readonly_editor(), format_func()

9.2.2 Defining the Editor

9.2.2.1 Standard Traits: ui, object, name, old_value, description, control, enabled, factory, updating, value, str_value

9.2.2.2 Standard methods: init(), dispose(), update_editor(), error(), string_value(), save_prefs(), restore_prefs(), get_undo_item()

10 Miscellaneous Advanced Topics

10.1 The UI Object

10.2 The UIInfo Object Revisited

10.3 Defining a Custom Help Handler

10.4 Saving and Restoring User Preferences

(which preferences can be saved, how to save and restore preferences)

10.4.1 Enabling User Preferences for a View

10.4.1.1 The View id

10.4.1.2 The Item ID

11Tips, Tricks and Gotchas

11.1Getting and Setting Model View Elements

For some applications, it can be necessary to retrieve or manipulate the View objects associated with a given model object. The HasTraits class defines two methods for this purpose: trait_views() and trait_view().

11.1.1 trait_views()

The trait_views() method, when called without arguments, returns a list containing the names of all Views defined in the object's class. For example, if **sam** is an object of type SimpleEmployee3 (from Example 6 on page 20), the method call `sam.trait_views()` returns the list `['all_view', 'traits_view']`.

Alternatively, a call to trait_views(*view_element_type*) returns a list of all named instances of class *view_element_type* defined in the object's class. The possible values of *view_element_type* are:

- View
- Group
- Item,
- ViewElement
- ViewSubElement

Thus calling `trait_views(View)` is identical to calling `trait_views()`. Note that the call `sam.trait_views(Group)` returns an empty list, even though both of the Views defined in SimpleEmployee contain Groups. This is because only *named* elements are returned by the method.

Group and Item are both subclasses of ViewSubElement, while ViewSubElement and View are both subclasses of ViewElement. Thus, a call to `trait_views(ViewSubElement)` returns a list of named Items and Groups, while `trait_views(ViewElement)` returns a list of named Items, Groups and Views.

11.1.2 trait_view()

The trait_view() method is used for three distinct purposes:

- To retrieve the default View associated with an object
- To retrieve a particular named ViewElement (i.e., Item, Group or View)
- To define a new named ViewElement

For example:

- `obj.trait_view()` returns the default View associated with object *obj*. For example, `sam.trait_view()` returns the View object called `traits_view`. Note that unlike trait_views(), trait_view() returns the View itself, not its name.
- `obj.trait_view('my_view')` returns the view element named `my_view` (or None if `'my_view'` is not defined).
- `obj.trait_view('my_group', Group('a', 'b'))` defines a Group with the name `my_group`. Note that although this Group can be retrieved using trait_view(), its name does not appear in the list returned by traits_view(Group). This is because `my_group` is associated with *obj* itself, rather than with its class.

This last usage will be particularly useful in conjunction with *parameterized views* once they are fully supported (see Section 5.5 on page 26).

Appendix I: Glossary of Terms

attribute: An element of data that is associated with all instances of a given class, and is named at the class level.[17] In most cases, attributes are stored and assigned separately for each instance (for the exception, see *class attribute*). Synonyms include "data member" and "instance variable".

Bool: The trait type of a *Boolean* attribute.

Boolean: Having only True or False as possible values.

controller: The element of the *MVC* ("model-view-controller") design pattern that manages the transfer of information between the data *model* and the *view* used to observe and edit it.

data member: Synonym for *attribute*.

dialog box: A secondary window whose purpose is for a user to specify additional information when entering a command.

dictionary: A lookup table of the form:

```
{key1: value1, key2: value2, … key_n: value_n},
```

Dictionaries are a built-in type in the Python language. Values are inserted and retrieved by key rather than by offset (as they would be in Python lists or in C-style arrays).

editor: A user interface component for editing the value of a trait attribute. Each type of trait has a default editor, but you can override this selection with one of a number of editor factories provided by the Traits UI package. In some cases an editor can include multiple widgets, e.g., a slider and a text box for a Range trait attribute.

editor factory: An instance of the Traits class EditorFactory. Editor factories generate the actual widgets used in a user interface. You can use an editor factory without knowing what the underlying GUI toolkit is.

factory: An object used to produce other objects at run time without necessarily assigning them to named variables or

[17] This is not always the case in Python, where attributes can be added to individual objects.

attributes. A single factory is often parameterized to produce instances of different classes as needed.

Group: An object that specifies an ordered set of Items and other Groups for display in a Traits UI View. Various display options can be specified by means of attributes of this class, including a border, a group label, and the orientation of elements within the Group. An instance of the Traits UI class Group.

Handler: A Traits UI object that implements GUI logic (data manipulation and dynamic window behavior) for one or more user interface windows. A Handler instance fills the role of *controller* in the MVC design pattern. An instance of the Traits UI class *Handler*.

HasTraits: A class defined in the Traits package to specify objects whose attributes are typed (i.e., whose attributes are "trait attributes").

instance: A concrete entity belonging to an abstract category such as a class. In object-oriented programming terminology, an entity with allocated memory storage whose structure and behavior are defined by the class to which it belongs. Often called an "object".

instance method: A method that is performed on (and called through) a specific instance, usually using a syntax like `object1.do_this()`.[18]

instance variable: Synonym for *attribute*.

Item: A non-subdividable element of a Traits user interface specification (View), usually specifying the display options to be used for a single trait attribute. An instance of the Traits UI class Item.

live: A term used to describe a window that is linked directly to the underlying model data, so that changes to data in the interface are reflected immediately in the model. A dialog box that is not live displays and manipulates a copy of the model data until the user confirms any changes.

livemodal: A term used to describe a dialog box that is both *live* and *modal*.

MVC: A design pattern for interactive software applications. The initials stand for "Model-View-Controller", the three distinct

[18] A method can, of course, have arguments. They are omitted from the sample syntax for simplicity.

entities prescribed for designing such applications. (See the glossary entries for *model*, *view*, and *controller*.)

modal: A term used to describe a dialog box that causes the remainder of the application to be suspended, so that the user can interact only with the dialog box until it is closed.

model: A component of the *MVC* design pattern for interactive software applications. The model consists of the set of classes and objects that define the underlying data of the application, as well as any internal (i.e., non-GUI-related) methods or functions on that data.

nonmodal: A term used to describe a dialog box that is neither *live* nor *modal*.

object: Synonym for *instance*.

object method: Synonym for *instance method*.

panel: A user interface region similar to a window except that it is embedded in a larger window rather than existing independently.

predefined trait: Any trait type that is built into the Traits package.

regular expression: A way of specifying a set of strings by encoding its syntax requirements as a single string rather than by enumerating all its members.

subpanel: A variation on a *panel* that ignores (i.e., does not display) any command buttons.

trait: A term used loosely to refer to either a *trait type* or a *trait attribute*.

trait attribute: An attribute whose type is specified and checked by means of the Traits package.

trait type: A type-checked data type, either built into or implemented by means of the Traits package.

Traits: An open source package engineered by Enthought, Inc. to perform manifest typing in Python.

Traits UI: A high-level user interface toolkit designed to be used with the Traits package.

tuple: An ordered set of Python objects, not necessarily of the same type. The syntax for tuples differs from that of Python lists in that parentheses are used (and , in some contexts, omitted) rather than square brackets, e.g. `tuple1 = ("hello", 3, True).`

View: A template object for constructing a GUI window or panel for editing a set of traits. The structure of a View is defined by one or more Group or Item objects; a number of attributes are defined for specifying display options including height and width, menu bar (if any), and the set of buttons (if any) that are displayed. A member of the Traits UI class View.

view: A component of the *MVC* design pattern for interactive software applications. The view component encompasses the visual aspect of the application, as opposed to the underlying data (the *model*) and the application's behavior (the *controller*).

ViewElement: A View, Group or Item object. The ViewElement class is the parent of all three of these subclasses.

widget: An interactive element in a graphical user interface, e.g., a scrollbar, button, pull-down menu or text box.

wizard: An interface composed of a series of *dialog boxes*, usually used to guide a user through an interactive task such as software installation.

wx: A shorthand term for the low-level GUI toolkit on which TraitsUI and PyFace are currently based (wxWidgets) and its Python wrapper (wxPython).

Appendix II: Editor Factories for Predefined Trait Factories

Trait Factory	*Default Editor Factory*	*Other Possible Editor Factories*
Any	TextEditor	EnumEditor, ImageEnumEditor, ValueEditor
Array	ArrayEditor (for 2-D arrays)	
Bool	BooleanEditor	
Button	ButtonEditor	
CArray	ArrayEditor (for 2-D arrays)	
CBool	BooleanEditor	
CComplex, CFloat, CInt, CLong	TextEditor	
Code	CodeEditor	
Color	ColorEditor	
Complex	TextEditor	
CStr, CUnicode	TextEditor(multi_line=True)	CodeEditor, HTMLEditor
Dict	TextEditor	ValueEditor
Directory	DirectoryEditor	
Enum	EnumEditor	ImageEnumEditor
Event	(none)	ButtonEditor, ToolbarButtonEditor
Expression	TextEditor	
File	FileEditor	

Trait Factory	*Default Editor Factory*	*Other Possible Editor Factories*
Float	TextEditor	
Font	FontEditor	
HTML	HTMLEditor	
Instance	InstanceEditor	TreeEditor, DropEditor, DNDEditor, ValueEditor
Int	TextEditor	
KivaFont	KivaFontEditor	
List	TableEditor for lists of HasTraits objects; ListEditor for all other lists.	CheckListEditor, SetEditor, ValueEditor
Long	TextEditor	
Password	TextEditor(password=True)	
PythonValue	ShellEditor	
Range	RangeEditor	
Regex	TextEditor	
RGBAColor	RGBAColorEditor	
RGBColor	RGBColorEditor	EnableRGBColorEditor
Str	TextEditor(multi_line=True)	CodeEditor, HTMLEditor
String	TextEditor	CodeEditor
ToolbarButton	ButtonEditor	
Tuple	TupleEditor	
UIDebugger	ButtonEditor (button calls the UIDebugEditor factory)	
Unicode	TextEditor(multi_line=True)	HTMLEditor

Trait Factory	***Default Editor Factory***	***Other Possible Editor Factories***
WeakRef	InstanceEditor	