



ENTHOUGHT
SCIENTIFIC COMPUTING SOLUTIONS

Traits UI User Guide (Draft)

Lyn Pierce

Version 1

Last rev: December 20, 2005

© 2005 Enthought, Inc.

All Rights Reserved.

Redistribution and use of this document in source and derived forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source or derived format (for example, Portable Document Format or Hypertext Markup Language) must retain the above copyright notice, this list of conditions and the following disclaimer.
- Neither the name of the Enthought nor the names of its contributors may be used to endorse or promote products derived from this document without specific prior written permission.

THIS DOCUMENT IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS DOCUMENT, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

All trademarks and registered trademarks are the property of their respective owners.

Enthought, Inc.
515 Congress Avenue
Suite 1614
Austin TX 78701
1.512.536.1057 (voice)
<http://www.enthought.com>
1.512.536.1059 (fax)
info@enthought.com

Table of Contents

1	Introduction	1
1.1	The Model-View-Controller (MVC) Design Pattern	1
1.1.1	The Model: HasTraits Subclasses and Objects	1
1.1.2	The View: View Objects	1
1.1.3	The Controller: Handler Subclasses and Objects	2
1.2	Structure of this Guide	2
2	Introduction to the View Object	3
3	The Building Blocks of a View	6
3.1	The Item Object	6
3.1.1	Content Attributes	6
3.1.2	Display Format Attributes	7
3.1.3	Content Format Attributes	8
3.1.4	Attributes for Overriding the Default Widget	8
3.1.5	Visibility and Status Attributes	8
3.1.6	User Help Attributes	9
3.1.7	Unique Id Attribute	9
3.2	The Group Object	9
3.2.1	Content Attributes	11
3.2.2	Group Display Format Attributes	11
3.2.3	Visibility and Status Attributes	12
3.2.4	Group-Level User Help Attributes	12
3.2.5	Unique Id Attribute	13
4	Customizing a View	14
4.1	Specifying Window Type: the ‘kind’ Attribute	14
4.1.1	Dialog boxes: ‘modal’, ‘live’, ‘livemodal’ and ‘nonmodal’	14
4.1.2	Wizards	15
4.1.3	Panels and Subpanels	15
4.2	Command Buttons: the <i>buttons</i> Attribute	16
4.3	Window Display Details	18
4.4	Window Command Attributes	19
4.5	User Help Attributes	19
4.6	Unique Id Attribute	20
5	Advanced View Concepts	21
5.1	Internal Views	21
5.1.1	Defining a Default View	21
5.1.2	Defining Multiple Views Within the Model	23

5.2 Separating Model and View: External Views	24
5.3 Displaying a View	24
5.3.1 <code>configure_traits()</code>	25
5.3.2 <code>edit_traits()</code>	26
5.3.3 <code>ui()</code>	26
5.4 The View Context	26
5.4.1 Multi-Object Views	27
5.5 Include Objects	29
6 Controlling the Interface: the Handler	31
6.1 Backstage: Introducing the <code>UIInfo</code> Object	31
6.2 Writing Handler Methods	32
6.2.1 Overriding Standard Methods	32
6.2.2 Reacting to Trait Changes	33
6.2.3 Implementing Custom Window Commands	35
6.3 Assigning Handlers to Views	37
6.3.1 Binding a singleton Handler to a View	38
6.3.2 Linking Handler and View at edit time	38
6.3.3 Creating a default View within a Handler	38
7 Introduction to Trait Editors	40
7.1 Specifying an Alternate Trait Editor	41
7.1.1 Initializing Editors	43
7.2 Specifying an Editor Style	43
7.2.1 The “simple” style	44
7.2.2 The “custom” style	44
7.2.3 The “text” style	45
7.2.4 The “readonly” style	45
7.2.5 Using Editor Styles	46
8 The Standard Trait Editors	48
8.1 Basic Trait Editors	48
8.1.1 <code>BooleanEditor</code>	48
8.1.2 <code>ButtonEditor</code>	49
8.1.3 <code>CheckListEditor</code>	50
8.1.4 <code>CodeEditor</code>	51
8.1.5 <code>ColorEditor</code>	52
8.1.6 <code>CompoundEditor</code>	53
8.1.7 <code>DirectoryEditor</code>	54
8.1.8 <code>EnableRGBAColorEditor</code>	55
8.1.9 <code>EnumEditor</code>	56
8.1.10 <code>FileEditor</code>	57
8.1.11 <code>FontEditor</code>	59

8.1.12	KivaFontEditor	60
8.1.13	ListEditor	61
8.1.14	RangeEditor	62
8.1.15	RGBColorEditor	64
8.1.16	RGBAColorEditor	64
8.1.17	SetEditor	65
8.1.18	TextEditor	67
8.1.19	TupleEditor	69
8.2	Advanced Trait Editors	70
8.2.1	CustomEditor	70
8.2.2	DropEditor	70
8.2.3	ImageEnumEditor	70
8.2.4	InstanceEditor	71
8.2.5	PlotEditor	71
8.2.6	TableEditor	71
8.2.7	TreeEditor	71
9	Advanced Editor Concepts	72
9.1	Interacting with an Editor Through the UI Object	72
9.1.1	Accessing Trait Editors Using Item 'id's	72
9.1.2	Controlling Editor Status Using 'enabled/disabled'	72
9.2	Introduction to Editor Factories	72
9.3	Defining a Custom Editor	72
9.3.1	Defining the Editor Factory	72
9.3.2	Defining the Editor	72
10	Miscellaneous Advanced Topics	73
10.1	The UI Object	73
10.2	The UIInfo Object Revisited	73
10.3	Defining a Custom Help Handler	73
10.4	Saving and Restoring User Preferences	73
10.4.1	Enabling User Preferences for a View	73
11	Tips, Tricks and Gotchas	74
11.1	Getting/Setting Model View Elements	74
11.1.1	trait_views()	74
11.1.2	trait_view()	74
	Appendix I: Glossary of Terms	76
	Appendix II: Default Editors for Standard Traits	80

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 data 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 is based on 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 the `HasTraits` class in Traits and the `View` and `Handler` classes in Traits UI. 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 and/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 Chapter 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` may be implicit. For example, none of the examples in Chapters 2 through 5 include or require 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 concerned with `Handlers` at all. Nonetheless, custom handlers can be a powerful tool for building sophisticated application interfaces, as will be 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 the user-programmer 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.

¹ A third type of content object, `Include`, is discussed briefly in Chapter 5, but is not presently in common usage.

² Not to be confused with the `TraitHandler` class of the Traits package, which is an entirely different beast.

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

```
# Sample code to demonstrate configure_traits()
from enthought.traits 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:

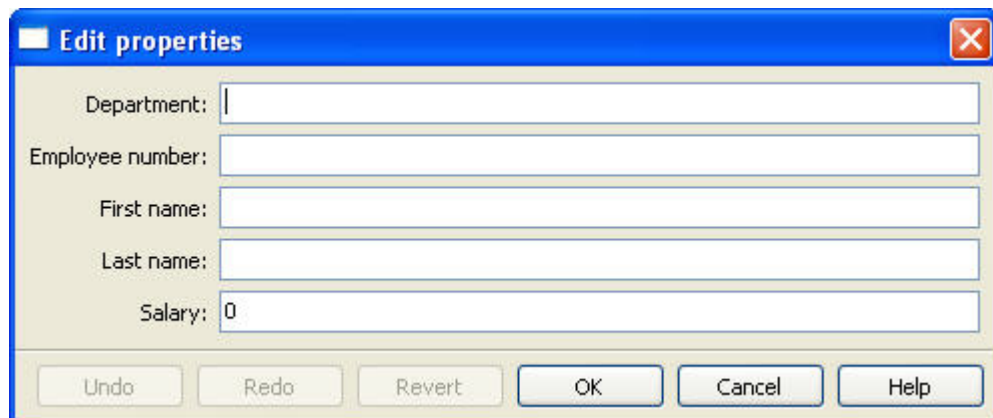


Figure 1

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

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 display⁴.

For example, suppose we 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, we would like to specify the order in which those fields appear. We can do this by defining a `View` object and passing it to the `configure_traits()` method:

Example 2

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

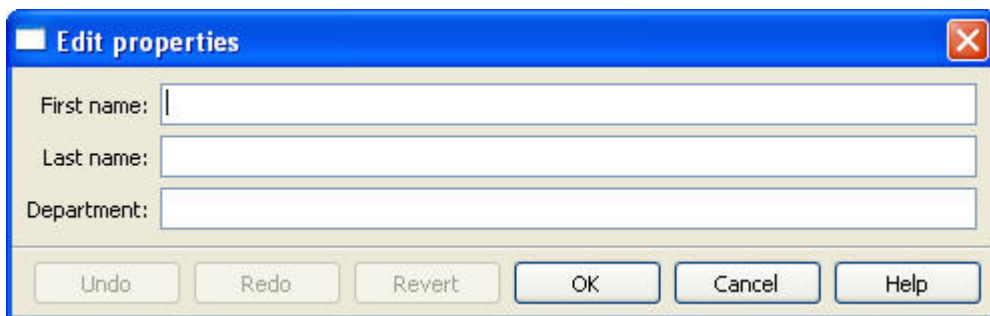


Figure 2

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

Chapters 3 through 5 explore the contents and capabilities of Views. An API summary for the View object appears in Appendix III.

For the time being, all example code will continue to use 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.

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` may 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 the example in Chapter 2.

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

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 will describe 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 we have already seen, simply specifying the name of the trait attribute to be displayed is often enough to produce a usable result.

An API summary for the `Item` class can be found in Appendix III.

3.1.1 Content Attributes

The content of an `Item` (i.e., the actual data to be displayed) is specified by means of two attributes: *name* and *object*.

name: The name of the trait attribute to be displayed.

object: A string identifying the `HasTraits` object to which the displayed trait attribute belongs.

Since an `Item` is essentially a template for displaying a single trait, its *name* attribute is nearly always specified. There is one trivial exception: if an `Item`'s *label* attribute (see Section 3.1.2) is specified but not its *name*, it means that the value of *label* should be displayed as a simple, non-editable string. (This feature can be useful for displaying, comments or instructions in a Traits UI window.)

By contrast, it is often not necessary to assign the *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`. For example, in Example 2, 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.

3.1.2 Display Format Attributes

In addition to specifying which trait attributes are to be displayed, a user may need to adjust the format of one or more of the resulting widgets. The `Item` class includes several attributes for this purpose, as described below.

label: The label to be used for the trait attribute display in the GUI. If this attribute is not set (as in Example 2) the label is simply the value of *name* with slight modifications: underscores are replaced by spaces, and the first letter is capitalized. However, a custom label can be specified by setting *label* to the desired string, e.g.: `Item(name='last_name', label='Surname')`. Such a label is used exactly as given; no capitalization or character substitution is performed. As previously mentioned, if an `Item`’s name is not specified, its label is represented in the GUI window as literal text.

width: The preferred width, in pixels, of the GUI widget corresponding to the `Item`. The actual width of the widget will be at least the maximum of *width* and the optimal width of the widget as calculated by the underlying GUI toolkit.⁵

In unusual cases, the user-programmer may need to specify an absolute width, ignoring the “optimal width” provided by the toolkit. This can be accomplished by specifying *width* as a negative integer. For example, if an `Item`’s *width* is set to 50 and the “optimal width” is 100, the widget will be displayed with a width of 100 pixels, but if *width* is set to -50, the widget will be displayed with a width of 50 pixels regardless of the GUI toolkit’s preferences. (Obviously, this feature should be used with caution.)

As a special case, setting *width* to -1 ensures that the default (“optimal”) width will be used. This is usually unnecessary, however, as the same result can be accomplished by simply not setting the *width* attribute at all.

Under some circumstances, this attribute is ignored altogether; see Section 3.2.1 for details.

height: The preferred height, in pixels, of the GUI widget corresponding to the `Item`. This attribute behaves exactly like *width*, above.

resizable: A Boolean attribute indicating whether the GUI widget should expand to fill any extra space that may be available in the display. This attribute defaults to **False**. If *resizable* is set to **True** for more than one `Item` in the same `View`, any extra space will be divided between them.

padding: The amount of extra space, in pixels, to place around (i.e., on each side of) the widget. The value of this attribute must be an integer between -15 and 15. (Negative values are used to *subtract* from the default spacing.)

⁵ As of this writing, Traits UI is implemented using the wx toolkit. However, other toolkits may be supported in future releases.

emphasized: A Boolean attribute indicating whether the label of the widget should appear in emphasized text. This attribute defaults to **False**. Note that if the label is not shown (see Section 3.2.2), this attribute is ignored.

3.1.3 Content Format Attributes

In some cases it may 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. There are two `Item` attributes that can be used for such formatting: *format_str* and *format_func*.

format_str: A Python-style format string. If this attribute is set, the specified format will be applied to the string representation of the trait being edited before it is displayed in the widget. If the widget does not use a string representation or if the `Item`'s *format_func* attribute is set, the *format_str* attribute is ignored.

format_func: A `Callable` object (i.e., a function or method; see *Traits User Manual*). If this attribute is set, the specified function (usually a custom routine) is used to create the string representation of the Trait to be edited. If the widget does not use a string representation, the *format_func* attribute is ignored.

3.1.4 Attributes for Overriding the Default Widget

There are two `Item` attributes that a user-programmer can use to override the widget automatically selected by Traits UI: *editor* and *style*. These relatively advanced options will not be discussed in detail until Chapters 7-9, but are mentioned here for the sake of completeness.

3.1.5 Visibility and Status Attributes

The `Item` attributes covered so far in this chapter are useful in setting up a static `View` layout. The more sophisticated an application is, however, the more likely it is to benefit from a dynamic GUI, in which the structure and behavior of the display itself alters in response to changes to the data it contains. While most dynamic behavior of this kind should be implemented in a custom `Handler` (see Chapter 6), the following attributes are available for simple cases:

enabled_when: A Boolean expression in string form, specifying the conditions under which the GUI widget is enabled. If the expression evaluates to **False**, the widget is disabled; i.e., it does not accept input. All *enabled_when* conditions are checked each time any trait attribute is edited in the display window; thus conditions based on trait values can be used to create a display in which widgets become active or inactive in response to user input.

visible_when: A Boolean expression in string form, specifying the conditions under which the GUI widget appears in the display. If the expression evaluates to **False**, the widget is invisible, and in fact disappears if it was previously visible. Likewise, if the value of the expression changes to **True**, the widget reappears.

As with *enabled_when*, the condition is checked whenever a trait is edited in the display window.

defined_when: A Boolean expression in string form, specifying the conditions under which the GUI widget is included in the window. This attribute is very similar to *visible_when*, except that it is evaluated only once, when the display is first constructed. It is therefore suitable for conditions based on attributes that vary from object to object but do not change over time (e.g., displaying a *maiden_name* attribute only for female employees in a company database).

3.1.6 User Help Attributes

tooltip: A string containing the “tooltip” text for the GUI widget. Tooltip text is displayed for a short time when the mouse pointer is left idle over the widget. If this attribute is not set, no tooltip is displayed.

Tooltip text should be kept as concise as possible. More detailed help can be made available using the *help* attribute.

help: A string containing help text for the widget. This help text appears in a pop-up window if the user clicks on the widget's label in the GUI. In addition, a *View*-level help window (see Section 4.5) displays the help text for all *Items* in the *View*. If the *help* attribute is not defined for an *Item*, a system-generated message is used instead.

help_id: A string attribute provided for use by a custom help handler (see Section 10.1). In applications for which no custom help utility is defined, this attribute is ignored.

3.1.7 Unique Id Attribute

id: A unique identifier string for the *Item*. This option is generally needed only for advanced Traits UI applications; see Section 9.1 for an example of its use. If this attribute is not set, it defaults to the value of *name* (see Section 3.1.1).

3.2 The Group Object

So far, we have seen how to construct GUI 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 may be visual (e.g., placing the widgets within a labeled border) or logical (activating/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

```
# Sample code to demonstrate configure traits()
from enthought.traits import HasTraits, Str, Int
from enthought.traits.ui 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 GUI window shows the same widgets as before, but they are now enclosed in a visible border with a text label:

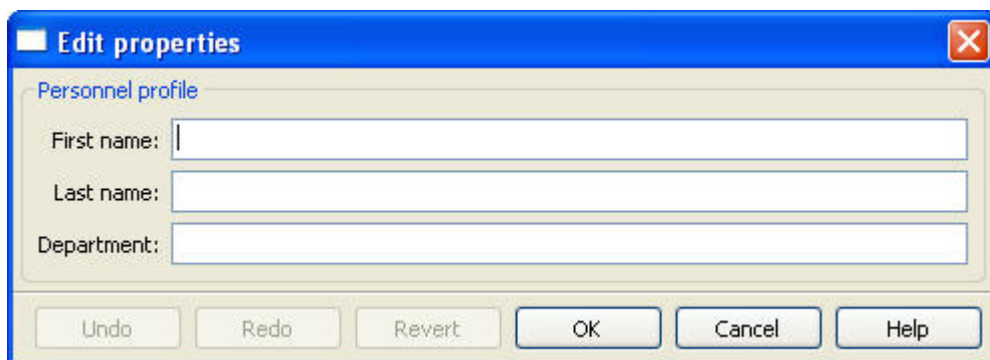


Figure 3

The remainder of this section describes the various attributes of the Group object, organized by categories of functionality. 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.

An API summary for the Group class appears in Appendix III.

3.2.1 Content Attributes

The content of a Group object is specified exactly like that of a View object. In other words, one or more Item and/or Group objects are given as arguments to the Group constructor, e.g., the three Items in the example above.⁶ For the remainder of this guide, the objects contained in a Group will be called the *elements* of that Group.

3.2.2 Group Display Format Attributes

There are a number of display instructions that can be specified at the Group level. For example, should the elements of a Group be laid out side by side or listed vertically? Should the Group be enclosed in a visible border? Should it have a visible title? These options and others are controlled by the attributes described in this section.

label: A string that will be displayed as a label for the Group. If *show_border* is set to **True** (see below), the label text is embedded in the border around the Group, as shown in Figure 3. If *show_border* is **False**, the label (if nonempty) appears as a banner above the elements of the Group. This attribute defaults to the empty string.

show_border: A Boolean attribute indicating whether the elements of the Group should be enclosed in a visible border. This attribute defaults to **False**.

show_labels: A Boolean attribute indicating whether the labels for any Items in the Group should be displayed.

Note that only Items directly contained by the Group are affected. In other words, if Group A contains Group B and ‘show_labels’ is set to **False** for Group A, the label of Group B and the labels of its elements are not affected.

show_left: A Boolean attribute indicating whether the labels for any Items in the Group should be displayed to the left of the corresponding widget (**True**) or the right (**False**). As with *show_labels*, only Items directly contained by the Group are affected. The default value for this attribute is **True**.

Note that if *show_labels* is set to **False**, the value of *show_left* is irrelevant.

padding: The amount of extra space, in pixels, to place around (i.e., on each side of) the widget for each Item in the Group. The value of this attribute must be an integer between 0 and 15.⁷ The padding for any individual widget is the sum of the padding for its Group, if any, the

⁶ 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).

⁷ Unlike the Item object, the Group object does not support negative padding.

padding for its `Item` (see Section 3.1.2), if any, and whatever default spacing is determined by the software.

layout: For Groups that contain two or more other Groups, this string-valued attribute specifies one of three possible layouts: ‘normal’, ‘split’, and ‘tabbed’. In a ‘normal’ layout, all the element Groups are displayed sequentially in a single panel. A ‘split’ layout is similar, except that the elements are separated by *splitter bars*, i.e., dividers that the user can move by dragging them with the mouse, thus adjusting the amount of space used by any particular Group. In a ‘tabbed’ layout, each of the element Groups appears on a separate tab (labeled with the Group’s *label* attribute, if specified). For Groups that contain Items or only a single Group, ‘layout’ is ignored. The default value of this attribute is ‘normal’.

selected: A Boolean attribute indicating whether the display for the Group will be uppermost if it is one of several contained in a View or in a Group whose ‘layout’ is set to ‘tabbed’. This attribute defaults to **False**.

orientation: A string-valued attribute indicating how the displays for the elements of the Group should be arranged in the GUI window. The possible values for this attribute are ‘horizontal’ (side-by-side) or ‘vertical’ (top-to-bottom). The default value is ‘vertical’.

style: A string-valued attribute for modifying the widget selection for all Items in the Group. See Section 7.2 for details.

3.2.3 Visibility and Status Attributes

The attributes **enabled_when**, **visible_when**, and **defined_when** of the Group class are exactly like those for the Item class (see Section 3.1.4) except that they apply to the entire Group. Thus if *visible_when* is set to some Boolean expression *<bool_exp>* for a given Group, the Group (including border and label, if any) appears or disappears depending on the value of *<bool_exp>*.

3.2.4 Group-Level User Help Attributes

help: A string containing help text that describes the Group as a whole. This help text appears in the View-level help window (see Section 4.5) for any View that contains *only* that Group. As of this writing, Group-level help is ignored for nested Groups and multiple top-level Groups. For example, suppose we have defined help text for a Group called *group1*. The following View will show this text in its help window:

```
View(group1)
```

while the following two will not:

```
View(group1, group2)
```

```
View(Group(group1))
```

help_id: A string-valued attribute provided for use by a custom help handler (see Section 10.1). In applications for which no custom help utility is defined, this attribute is ignored.

3.2.5 Unique Id Attribute

id: A unique identifier string for the Group. This option is generally needed only for advanced Traits UI applications; see Section 9.1 for an example of its use.

4 Customizing a View

As we have seen in the previous two chapters, it is possible to specify a GUI windows in Traits UI simply by creating a `View` object with the appropriate contents. In designing real-life applications, however, a user-programmer usually needs to be able to control the appearance and behavior of the GUI 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 different types of GUI windows may be used to display the same data content. A form may appear in a dialog box, a wizard, or an embedded panel; dialog boxes may be *modal* (i.e., stop all other program processing until the box is dismissed) or not, and may interact with live data or with a buffered copy. In Traits UI, a single `View` may 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 may vary over two significant degrees of freedom. First, it may be *modal*, meaning that when the dialog appears, all other GUI interaction is suspended until the dialog window is closed; if it is not modal, then both the window and the rest of the GUI remain active and responsive. Second, it may be *live*, meaning that any changes that the user makes to data in the window 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 pressing an ‘OK’ or ‘Apply’ button; see Section 4.2). The four possible combinations of these behaviors correspond to four of the possible values of the ‘kind’ attribute of the `View` object, as shown in the table below.

Table 1

	<i>not modal</i>	<i>modal</i>
<i>not live</i>	nonmodal	modal
<i>live</i>	live	livemodal

All four of these dialog types are identical in appearance. Also, all four 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 windows, and are primarily suitable for highly controlled user interactions such as software installation.

4.1.3 Panels and Subpanels

Both dialogs and wizards are individual GUI windows that appear separately from the main program display, if any. Often, however, it is necessary 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, except that it is embedded in a larger window, which need not be a Traits UI window. Like dialogs, panels support the *buttons* `View` attribute, as well as any menus and toolbars that are specified for the `View` (see Section 6.2.3). 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 GUI 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.⁸ Consider the following variation on Example 3:

Example 4

```
# Sample code to demonstrate configure_traits()
from enthought.traits import HasTraits, Str, Int
from enthought.traits.ui 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'),
             Item(name = 'department'),
             buttons = [OKbutton, CancelButton])

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

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

⁸ Actually, the value of the *buttons* attribute is really a list of *Actions*, from which GUI buttons are generated by Traits UI. The Action object will be described in Section 6.2.

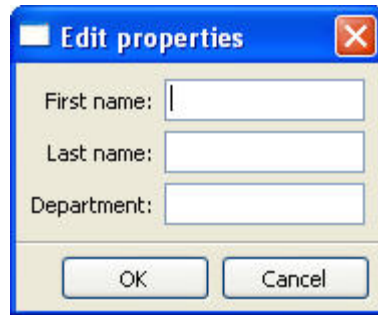


Figure 4

(Notice that the automatically sized window is significantly smaller than the one in the original example. See Section 4.3 for details on how to override automatic sizing.)

There are six standard buttons defined by Traits UI:

- **UndoButton** – Appears as two buttons: ‘Undo’ and ‘Redo’. When ‘Undo’ is pressed, the most recent change to the data is cancelled, restoring the previous value. ‘Redo’ cancels the most recent ‘Undo’.
- **ApplyButton** – Appears as a single button: ‘Apply’. When ‘Apply’ is pressed, all changes to the data window are applied to the model. (This option is only meaningful for modal windows.)
- **RevertButton** – Appears as a single button: ‘Revert’. When ‘Revert’ is pressed, all changes to the data form are cancelled and the original values restored. If the changes have been applied to the model (via ‘Apply’ or because the window is live), the model data is restored as well. The window remains open.
- **OKButton** – Appears as a single button: ‘OK’. When ‘OK’ is pressed, all changes to the data form are committed to the model, and the window is closed.
- **CancelButton** – Appears as a single button: ‘Cancel’. When ‘Cancel’ is pressed, all changes to the data form are discarded; if the window is live, the model data is restored to the values it held before the window was opened. The window is then closed.
- **HelpButton** – Appears as a single button: ‘Help’. When ‘Help’ is pressed, the pop-up help for the window is displayed.

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

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

Thus, we could rewrite the highlighted lines in the code sample above as follows, and the effect would be exactly the same:

```
from enthought.traits.ui.menu import OKCancelButtons
    .
    .
    .
    buttons = OKCancelButtons
```

Finally, the special constant *NoButtons* can be used to create a dialog 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 (Figures 1-3) are used.

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

4.3 Window Display Details

The `View` object also has attributes for controlling the visual properties of the GUI window itself, regardless of its content. These attributes are described in this section.

x: The horizontal position of the window. This attribute may be specified in four ways:

- A positive integer, indicating the number of pixels from the left edge of the screen to the left edge of the window
- A negative integer, indicating the number of pixels from the right edge of the screen to the right edge of the window
- A positive fractional value, indicating the fraction of total screen width between the left edge of the display and the left edge of the window
- A negative fractional value, indicating the fraction of total screen width between the right edge of the display and the right edge of the window

y: The vertical position of the window. This attribute behaves exactly like the *x* attribute, except that its value indicates the position of the window with respect to the top or bottom of the screen.

width: The width of the window, either as an absolute number of pixels, e.g.:

```
width = 150
```

This example specifies that the window should be exactly 150 pixels wide. Width may also be specified as a fraction of the screen width, e.g.:

```
width = 0.5
```

This example indicates that the window should be one-half the width of the screen.

height: The height of the window, either as an absolute number of pixels or as a fraction of the screen height. See *width* above for details.

title: A string-valued attribute specifying the contents of the window's title bar.

resizable: A Boolean attribute indicating whether the user can resize the window by dragging its border. The default value of this attribute is **False**.

scrollable: A Boolean attribute indicating whether the window as a whole is scrollable. If set to **True**, window-level scroll bars will appear whenever the window is too small to accommodate all its contents at once. If **False**, only individual widgets will have scroll bars. The default value of this attribute is **False**.

style: An attribute for modifying the widget selection for all `Items` in the `View`. See Section 7.2 for details.

dock: An attribute describing whether and how users are permitted to rearrange subelements (usually `Groups`) in the window. The default value, "fixed", specifies that no rearrangement is allowed. If *dock* is set to "horizontal" or "vertical", then moveable elements appear with a visual "handle" either to the left (horizontal) or above (vertical) by which the element can be dragged by holding the left mouse button down. If *dock* is set to "tabbed" then the moveable elements appear as tabbed panels which may be either stacked in an ordinary tabbed display or dragged around the window.

4.4 Window Command Attributes

handler: The `Handler` object that provides GUI logic for the window. It is necessary to set this attribute only if a custom `Handler` is being used; see Chapter 6 for a discussion of custom `Handlers`. If this attribute is not set, the default `Traits UI Handler` is used.

menubar: The menu bar for the window. `Traits UI` menus are generally implemented in conjunction with custom `Handlers`; see Section 6.2.3.2 for details.

toolbar: The toolbar for the window. Like menus, `aTraits UI` toolbars usually require a custom `Handler`; see Section 6.2.3.3 for details.

4.5 User Help Attributes

help: A string containing general help text for the window. This help text appears in the `View`-level help window when the user clicks the "Help" button (if present). This window also

displays the help text for all Items in the View. If the *help* attribute is not defined for the View, only the Item- and Group-level help is displayed in the window.

help_id: A string attribute provided for use by a custom help handler (see Section 10.1). In applications for which no custom help utility is defined, this attribute is ignored.

4.6 Unique Id Attribute

id: A unique identifier string for the View. This option is generally needed only for advanced Traits UI applications; see Section 7.4.1 for an example of its use.

5 Advanced View Concepts

The first four chapters of this Guide have given an overview of how to use the `View` class to quickly construct a simple GUI window for a single `HasTraits` object. In this chapter we will explore a number of more complex techniques that significantly increase the power and versatility of the `View` object .

5.1 Internal Views

In the examples we have seen 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 may 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

```
# Sample code to demonstrate the use of 'traits view'
from enthought.traits import HasTraits, Str, Int
from enthought.traits.ui 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 we saw in Figure 3:

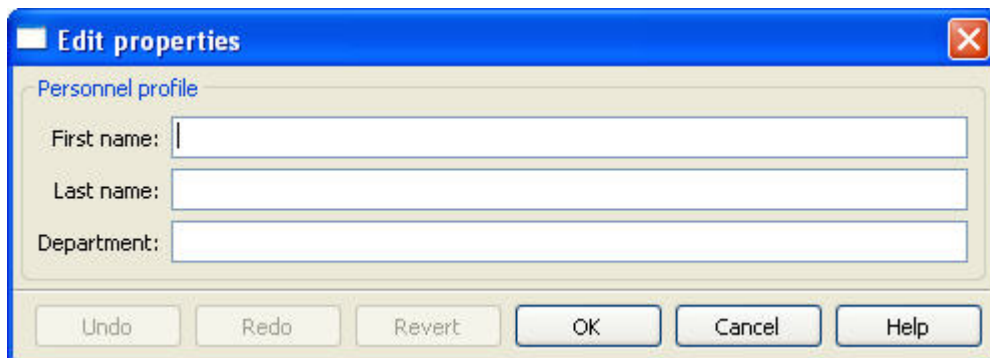


Figure 5

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 will always be treated as the default display template for the class. However, it is usually preferable to use the “`traits_view`” naming convention regardless, for reasons that are discussed in Section 5.1.2.

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 our `SimpleEmployee`, we might want to have both a “public information” view like the one above and an “all information” view. We can do this by simply adding a second `View` attribute:

Example 6

```
# Sample code to demonstrate the use of multiple views
from enthought.traits import HasTraits, Str, Int
from enthought.traits.ui 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 will produce a window based on the default `View` (`traits_view`). In order to use the alternate `View`, we 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, we must indicate which one is to be used as the default by naming it `traits_view`. Otherwise, Traits UI will give preference to none of them, and will instead try 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 – may unexpectedly change the behavior of the GUI.

5.2 Separating Model and View: External Views

In all the examples we have seen so far, 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; in other cases the `configure_traits()` method that produces a GUI 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. In this section we begin 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 attribute⁹. A View can even be defined in-line as a function or method argument, e.g.:

```
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.

We have already seen simple examples of external Views defined as variables in Examples 2-4. One advantage of this convention is that the variable 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 GUI window for that View.¹⁰ This technique will be 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.

⁹ 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.

¹⁰ Assuming there is one; as previously mentioned, not all GUIs require an explicitly defined `Handler`.

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 and restoring configuration information to/from a file.

There are six optional keyword arguments to `configure_traits()`:

- **view:** The name of the `View` object that is to be used to construct the GUI window. If the `View` is defined as a variable, the variable name is used without quotation marks, e.g.: `configure_traits(view=view1)`. If it is defined within the class of the object on which `configure_traits()` is called, the name should be enclosed in single or double quotes: `configure_traits(view='view2')`. If the *view* attribute is not set, the default `View` for the object is used. (See Section 5.1.1 for a discussion of default views.)
- **kind:** A string specifying the type of window to create from the `View`. The valid values are the same as for the *kind* attribute of the `View` object; see Section 4.1. This method argument, if specified, takes precedence over the *kind* attribute of the `View` being displayed.
- **context:** A dictionary used to match `Items` in the `View` to actual objects; see Section 5.4 for details. This argument may be omitted when all `Items` in the `View` correspond to attributes of the object whose `configure_traits()` method is called.
- **handler:** An instance of the `Handler` class, whose role is to implement any logic required for the desired behavior of the GUI window; see Chapter 6 for details. If this argument is omitted, the default `Handler` for Traits UI is used.
- **filename:** The name (including path) of a configuration information file for the object¹¹. When this attribute is specified, Traits UI reads the corresponding file (if it exists) to restore the saved values of the object's Traits before displaying them. When the GUI window is closed normally (via the 'OK' command button rather than the 'Cancel' button), the new values are written to the file. If this argument is not specified, the object values are loaded directly into the GUI window without modification, and are not saved to disk when the window is closed.
- **edit:** A Boolean value indicating whether a GUI window is to be created. If the *filename* argument is set to an existing configuration file, setting *edit* to **False** loads the saved values from that file into the object without requiring user interaction. The default value for this argument is **True**.

¹¹ In Python's `pickle` format.

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 and restoring a configuration file, as it is assumed that these options will be handled elsewhere in the application.

There are five optional keyword arguments to `edit_traits()`, the first four of which are identical in form and function to the corresponding arguments of `configure_traits()`:

- **view**
- **kind**
- **context**
- **handler**
- **parent**: The parent window of the new window or panel.

5.3.3 ui()

The `View` object includes a method called `ui()`, which performs the actual generation of the GUI 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.¹²

The `ui()` method has five keyword arguments: **kind**, **context**, **handler**, **parent** and **view_elements**. The first four are identical in form and function to the corresponding arguments of `edit_traits()`, except that *context* is not optional; we will see why in the next section.

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 the user-programmer. It contains a dictionary of the named `ViewElements` 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* argument. The description of this argument is given as “A dictionary used to match `Items` in the `View` to actual objects.” But what does this mean, exactly?

¹² 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”.

Recall the *object* attribute of the `Item` class, described in Section 3.1.1: “A label identifying the `HasTraits` object to which the displayed `Trait` attribute belongs.” A context is a Python-style dictionary whose entries are of the form `"label": <object name>`. Thus the following code behaves exactly like that of Example 3:

Example 7

```
# Sample code to demonstrate view contexts
from enthought.traits import HasTraits, Str, Int
from enthought.traits.ui 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',
                        object = 'object'),
                  Item(name = 'last name',
                        object = 'object'),
                  Item(name = 'department',
                        object = 'object'),
                  label = 'Personnel profile',
                  show border = True))

sam = SimpleEmployee()
sam.configure_traits(view=view1, context={'object':sam})
```

In fact, this is precisely what was going on behind the scenes in that example: in a call to `<obj>.configure_traits()` where the context is not specified, Traits UI automatically uses the context `{ 'object' : <obj> }`. Traits UI also assumes that the *object* attribute of all `Items` has the value “object” (i.e., the literal string “object”) unless otherwise specified. This is why *context* is not optional for the `ui()` method, as mentioned above: since `ui()` is called on a `View` rather than on a `HasTraits` object, there is no default context available.

So, if `configure_traits()` does all the work, why does the user have the option of explicitly specifying a view context 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 we 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 8

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

from enthought.traits import HasTraits, Str, Int, Bool
from enthought.traits.ui 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 GUI window has the desired appearance:¹³

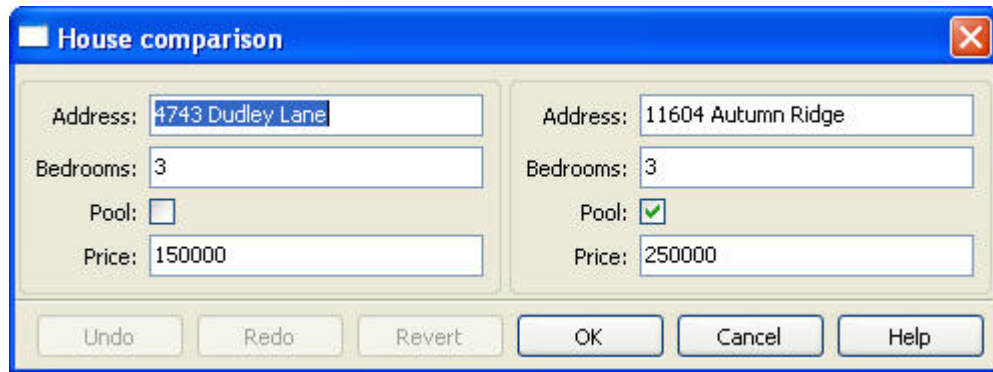


Figure 6

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 `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.

5.5 Include Objects

In addition to `Item` and `Group`, 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 in common use 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:

¹³ 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.

```
# 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` may be defined differently for different instances or subclasses of that class. This technique is called *view parameterization*.

Although the `Include` class in its present form is unsupported, its use cases are compelling enough that it will almost certainly be re-implemented as a supported feature in the future.

6 Controlling the Interface: the Handler

Most of the material in the first five chapters is concerned with the relationship between the “model” and “view” aspects of the MVC design pattern as supported by Traits UI. It is now time to look at 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 may 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.¹⁴

In the examples we have seen so far, 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).

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

As previously mentioned, 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 UI 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 – yet, in order to be useful, it must be able to observe and manipulate both. 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 object displayed in it. Each entry in the

¹⁴ Except those implemented via the *enabled_when*, *visible_when* and *defined_when* attributes of `Items` and `Groups`.

`View`'s context (see Section 5.4) becomes an attribute of the `UIInfo` object.¹⁵ For example, the `UIInfo` object created in Example 8 has attributes `h1` and `h2` whose values are the objects `house1` and `house2` respectively. In Examples 1-7, 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 we shall see in the next two sections.

6.2 Writing Handler Methods

6.2.1 Overriding Standard Methods

The `Handler` class provides six standard methods that are automatically executed at certain points in the lifespan of the UI window controlled by a given `Handler`. By overriding these methods, the programmer can implement a variety of custom window behaviors, as outlined below.

- **`init(self, info)`**

This method is called by the `Handler` after the window elements have been created, but before the window is displayed. The default method simply returns **True**, but can be overridden to perform specialized initialization and customization tasks. The child method should return **True** when it completes successfully, or **False** if the display operation should be aborted.

- **`position(self, info)`**

This method is called after the `init()` method, but before the window is displayed. The default method positions the window on the display device using the `x` and `y` attributes of the `View`. Because the user-programmer can control absolute window placement using these attributes, it is seldom necessary to override this method.

- **`setattr(self, info, object, name, value)`**

This method is called whenever the end user updates the value of a trait attribute using the GUI window. The `object` argument is the object whose attribute is being updated; `name` is the name (in string form) of the attribute being set, and `value` is the value to which the attribute is being set. The default method calls the Python `setattr()` function with these three arguments.

¹⁵ 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 will be discussed in Section 10.2.

The `setattr()` method may be overridden to perform additional tasks when a trait attribute is edited, e.g. updating a history file or modifying the display to highlight changes. In most cases, however, the default behavior (actually setting the attribute value) is also desirable. The child method should therefore usually either call the parent method via a command like:

```
super(<handlerclass>, self).setattr(info, object, name, value)
```

or include code to handle the update directly.

- **show_help(self, info, control=None)**

This method is called whenever the “Help” button is pressed on a Traits UI window or panel. The default method constructs and displays a pop-up help window using the *help* attributes for all `Items` and any top-level `Groups` in the `View` (see Sections 3.1.6 and 3.2.4).¹⁶ This method may be overridden to execute a user-defined help utility instead, or in addition (see Section 10.2).

- **close(self, info, is_ok)**

This method is called when a window close request is issued by the user (e.g., by pressing the “OK” or “Cancel” button or selecting “Close” on the Microsoft Windows menu), but before the window is actually destroyed. The default method simply returns **True**. If it is overridden, the child method should return **True** or **False** to indicate whether the window should be closed or not. One possible reason to override this method is to implement a confirmation dialog, i.e., a message box requiring the user to confirm or cancel the window closure.

The *is_ok* argument has the value **True** if the user confirmed any changes to the window data (usually by hitting the “OK” button) and **False** otherwise. Although Traits UI handles “OK” and “Cancel” intelligently without custom code, *is_ok* may be used by a custom version of `close()` to implement additional behaviors; see Chapter 11 for an example.

- **closed(self, info, is_ok)**

Like `close()`, the `closed()` method is called in response to a user-initiated window closure. However, `closed()` is run *after* the GUI window is destroyed. The default method does nothing; it may be overridden to perform any cleanup tasks required by the application. The *is_ok* argument is identical to that of `close()`.

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 writing methods that are automatically

¹⁶ If the *help* attribute is not specified for some `Item` or `Group`, the help text automatically generated by Traits is used instead; see *Traits User Manual* for details.

executed whenever the user edits a *particular* trait. Specifically, any Handler method with the signature `<objectname>_<traitname>_changed(self, info)` will be called whenever the attribute `<traitname>` of object `<objectname>` is changed, where `<objectname>` is the label of an object in the *context* of the View used to create the UI window (see Section 5.4).

Remember that if an object `<obj>` is displayed without an explicit View context, the default context `{"object": <obj>}` is used, so that `<objectname>` in this case is simply the literal string “object”. Thus, if we are writing a subclass of Handler to use with the `SimpleEmployee` class and its View from Example 2, and we want certain code to execute whenever the `salary` attribute is edited, we can place that code in the body of a method called `object_salary_changed()`. By contrast, a subclass of Handler for Example 8 might include a method called `h2_price_changed()` to be called whenever the price of the second house is edited.

(The experienced Traits programmer may notice that Handler methods of this type are very similar in functionality to `_<traitname>_changed()` methods for subclasses of `HasTraits`. See Chapter 11 for a discussion of this point.)

Important: `<objectname>_<traitname>_changed()` methods are also called when the UI window is first created, which can lead to unexpected results if the programmer is not aware of the fact. 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 a ‘*’ the first time a data element is updated, demonstrates a simple use of both an overridden `setattr()` method and an `<objectname>_<traitname>_changed()` method.

Example 9

```
from enthought.traits import HasTraits, Bool
from enthought.traits.ui 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

6.2.3.1 Actions

In Traits UI, window commands are implemented as instances of the `Action` class. An `Action` object has the following attributes:

name:¹⁷ A string to be used as the label for the `Action`. If the `Action` is used to create a command button, then this attribute becomes the label of the button. If the `Action` is included in a `Menu` or `ToolBar`, then this attribute is used as the label of the corresponding menu option.

¹⁷ The 'name' attribute actually belongs to the Pyface `Action` class, of which the Traits UI `Action` is a subclass.

action: The name of a `Handler` method to be called when the `Action` is invoked. This method must accept a single `UIInfo` argument. For example, suppose the following `Action` is included in a menu for a given `View`:

```
Action( name = 'foo', action = 'bar' )
```

When the menu item for this action is selected, the method `bar(info)` is called on the `Handler` for the window, where 'info' is the `UIInfo` object for the window (see Section 6.1).

As `Actions` are associated with `Views` rather than with `Handlers`, it is up to the programmer to ensure that the `Handler` object for a given window has an appropriately named method for each `Action` defined on the `View` for that window.

enabled_when, visible_when, defined_when: Boolean expressions indicating when the command button and/or menu or toolbar option for the `Action` should be enabled, visible or defined, respectively. These attributes are identical in form and function to the correspondingly named attributes of `Items` (see Section 3.1.5) and `Groups` (Section 3.2.3).

checked_when: A Boolean expression indicating when the menu option for the `Action` should appear with a checkmark beside it. This attribute applies only to `Actions` that are included in `Menus`; it is ignored in other cases.

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`. For example, suppose we want to build a GUI window with a custom "Recalculate" button as well as the standard "OK" and "Cancel" command buttons. Suppose further that we have defined a subclass of `Handler` called `MyHandler` to provide the logic for the window. There are three steps to adding the custom button:

1. Add a method to `MyHandler` that implements the command logic. This method may 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.:

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

3. Include the new `Action` in the `buttons` attribute for the `View`:

```
View ( <view contents>,  
      .  
      .  
      .  
      buttons = [ OKButton, CancelButton, MyAction ] )
```

6.2.3.3 Menus and Menu Bars

Another way to install an `Action` such as `MyAction` as a window command is to make it into a menu option. The first two steps (creating the `Action`) are the same as in Section 6.2.3.2; the remainder of the procedure is as follows:

3. If the `View` does not already include a `MenuBar`, create one and assign it to the `View`'s `menubar` attribute.
4. If the appropriate `Menu` does not yet exist, create it and add it to the `MenuBar`.
5. 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( MyAction,  
                name = 'My special menu' ) ) )
```

6.2.3.4 Tools and Tool Bars

A `ToolBar` class is under construction for Traits UI. It will be documented in this section when completed.

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

GUI window, it must be explicitly associated with the View for that window. Traits UI provides three ways of accomplishing this:

- making the Handler an attribute of the View
- providing the Handler as an argument to a display method such as `edit_traits()`
- defining the View as part of the Handler

The remainder of this section describes these alternatives.

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.

Suppose, for example, we replace the last line of Example 9 with:

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

The resulting window will be controlled by an instance of `SomeOtherHandler` rather than of `TC_Handler`.

6.3.3 Creating a default View within a Handler

Although it is theoretically possible to associate a single custom Handler with several different Views or vice versa, and there are cases where it is useful to be able to do so, it is seldom necessary. 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, we can rewrite Example 9 a little more concisely as follows:

Example 10

```
from enthought.traits import HasTraits, Bool
from enthought.traits.ui 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 argument is not optional.

7 Introduction to Trait Editors

The code samples we have seen so far 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: that 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*. A trait editor 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 that is automatically used whenever the trait is displayed unless the programmer specifies otherwise.

Consider the following script and the window it creates:

Example 11

```
from enthought.traits import HasTraits, Str, Range, Bool
from enthought.traits.ui 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()
```

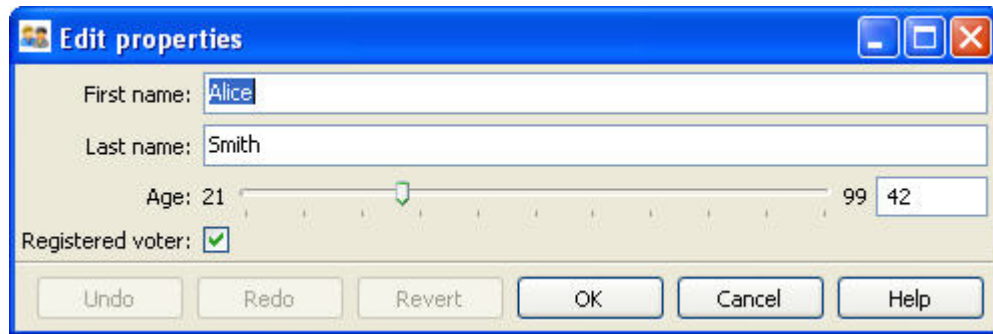


Figure 7

Notice that each trait is displayed in an appropriate widget although 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 is an implementation of the default trait editor (`TextEditor`, `RangeEditor` and `BooleanEditor` respectively) associated with the trait type.

Traits UI is by no means limited to these defaults, however. There are two ways to override the default representation of a trait attribute in a Traits UI window: explicitly specifying an alternate trait editor, and specifying an alternate style for the editor. The remainder of this chapter will examine these alternatives a little more closely.

7.1 Specifying an Alternate Trait Editor

As of this writing the Traits UI package includes twenty-six predefined trait editors:

Table 2

<code>BooleanEditor</code>	<code>ButtonEditor</code>	<code>CheckListEditor</code>	<code>CodeEditor</code>
<code>ColorEditor</code>	<code>CompoundEditor</code>	<code>CustomEditor</code>	<code>DirectoryEditor</code>
<code>DropEditor</code>	<code>EnableRGBAColorEditor</code>	<code>EnumEditor</code>	<code>FileEditor</code>
<code>FontEditor</code>	<code>KivaFontEditor</code>	<code>ImageEnumEditor</code>	<code>InstanceEditor</code>
<code>ListEditor</code>	<code>PlotEditor</code>	<code>RangeEditor</code>	<code>RGBColorEditor</code>
<code>RGBAColorEditor</code>	<code>SetEditor</code>	<code>TableEditor</code>	<code>TextEditor</code>
<code>TreeEditor</code>	<code>TupleEditor</code>		

These editors are described in detail in Chapter 8.

For most predefined traits (see *Traits User Manual*), there is exactly one predefined trait editor suitable for displaying it: the editor that is assigned as its default.¹⁸ There are exceptions, however; for example, there are two different editors for displaying an `RGBAColor` trait: the `RGBAColorEditor` and the `EnableRGBAColorEditor`. A `List` trait may be edited by means of a `ListEditor`, a `TableEditor` (if the `List` elements are `HasTraits` objects), a `CheckListEditor` or a `SetEditor`. Furthermore, Traits UI includes tools for building additional trait editors as needed; we will see how this works in Chapter 9.

To use an alternate editor 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 attribute. 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 attributes explicitly initialized; see Section 7.1.1.)

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

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

The resulting widget would then have the following appearance:

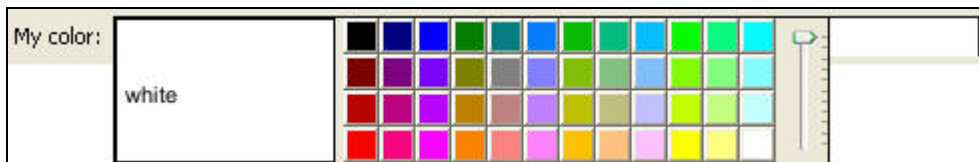


Figure 8

To use the `EnableRGBAColorEditor` instead, we would add the appropriate specification to the `Item`:

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

The resulting widget would then appear as follows:

¹⁸ Appendix II contains a table of the predefined trait types in the Traits package and their default trait editor types.

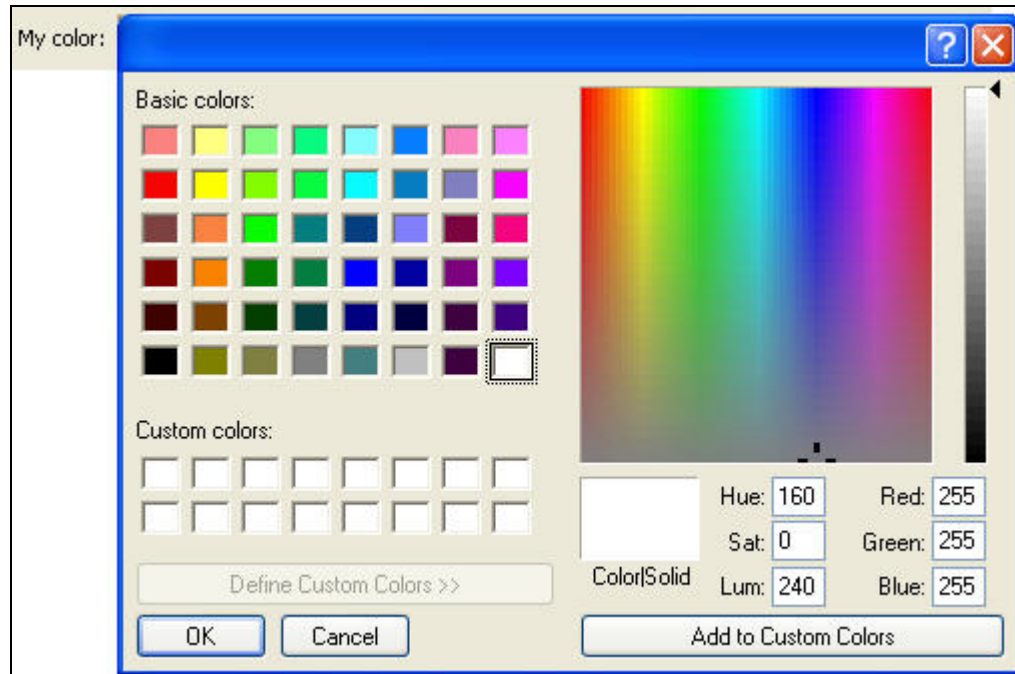


Figure 9

Caveat: 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 may 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.

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, the `CheckListEditor` and `SetEditor` both allow 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’s constructor, e.g.:

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

The trait editor descriptions 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 may be displayed 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 may look completely different in

one style than in another. However, different trait editors displayed in the same style will (usually) have noticable 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 will be discussed further in Section 7.2.5.

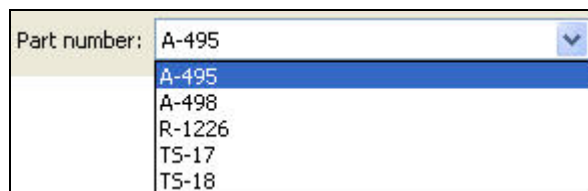
7.2.1 The “simple” style

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

In some cases, such as the `TextEditor` and `BooleanEditor` (see Section 8.1), the single line is fully sufficient. In others, such as the `ColorEditor` and `EnumEditor`, a more detailed interface is required; pop-ups or drop-downs are often used in such cases. For example, the “simple” version of the `EnumEditor` looks like this:



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



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

7.2.2 The “custom” style

The *custom* editor style generally represents 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 “custom” style the `EnumEditor` appears as a set of radio buttons rather than a drop-down list:

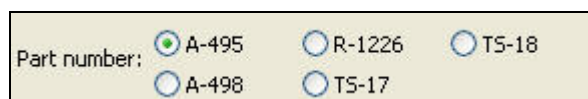


Figure 10

In the custom style of the `RGBAColorEditor`, the color palette is embedded in the window rather than appearing as a pop-up:

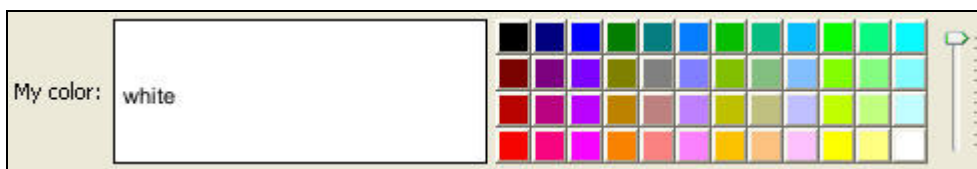


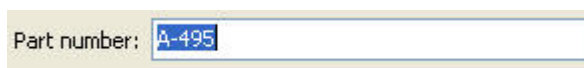
Figure 11

In general, the “custom” editor style can be very useful when there is no need to conserve window space, as it allows 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 editors. If the “custom” style is requested for an editor for which it is not defined, the “simple” style is used 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 presents a text representation of the trait in an editable box. Thus the `EnumEditor` in “text” style looks like the following:



For this type of editor, it is the end user’s responsibility to type in a valid value for the attribute. If an incorrect value is typed, the *trait validator* for the attribute (see *Traits User Manual*) notifies the user of the error.

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 3.1.3), if any, is called on the attribute.
- Otherwise, the function specified in the *format_func* attribute of the trait editor (see Section 9.3.1.2), if any, is called on the attribute.
- Otherwise, the Python-style formatting string specified in the *format_str* attribute of the `Item` (see Section 3.1.3), if any, is used to format the attribute.
- Otherwise, the Python-style formatting string specified in the *format_str* attribute of the trait editor (see Section 9.3.1.2), if any, is used to format the attribute.
- Otherwise, the Python `str()` function is called on the attribute..

7.2.4 The “readonly” style

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

Part number: A-495

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

7.2.5 Using Editor Styles

As we have already seen 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 may 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

```
from enthought.traits import HasTraits, Str, Enum
from enthought.traits.ui 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 the precedence rules we have just described:

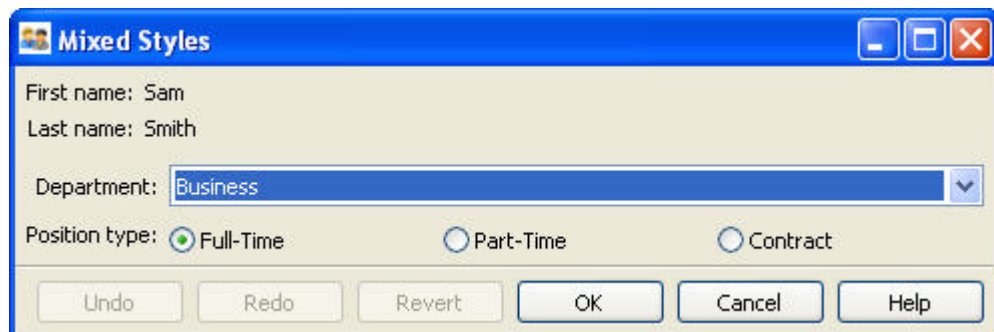


Figure 12

8 The Standard Trait Editors

This chapter contains individual descriptions of the twenty-six trait predefined trait editors provided by Traits UI. The majority of these editors 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 covers a smaller set of specialized editors with more complex interfaces. Please note that the exact appearance of the editors will depend on the underlying GUI toolkit; the screenshots included in this chapter are based on wxPython, which is the only currently supported GUI platform.

Rather than trying to memorize all the information in this chapter, the reader is encouraged to 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 Editors

8.1.1 BooleanEditor

Suitable for	Bool
Default for	Bool
Required initialization	(none)
Optional initialization	<i>mapping</i>

The `BooleanEditor` is one of the simplest of the built-in editors in Traits UI, and is used exclusively to edit and display `Bool` (i.e., **True/False**) traits. In the “simple” and “custom” styles it appears as a checkbox. In the “text” style the editor displays the trait value (as one would expect) as the string “True” or “False”. However, several variations are accepted as input: ‘True’, ‘true’, ‘t’, ‘yes’ or ‘y’ for **True**, and ‘False’, ‘false’, ‘f’, ‘no’ or ‘n’ for **False**.

The set of acceptable text inputs can be changed by initializing the `BooleanEditor` attribute *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 `BooleanEditor` 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 13 shows the four styles of the `BooleanEditor`.

Figure 13

8.1.2 ButtonEditor

Suitable for	Button, Event
Default for	Button
Required initialization	(none)
Optional initialization	<i>value, label</i>

The ButtonEditor is designed to be used with an Event or Button¹⁹ trait. When a user clicks on a ButtonEditor button, the associated event is fired. Because events are not readable objects, the “text” and “readonly” styles are not implemented for this editor. The “simple” and “custom” styles of this editor are identical, as shown in Figure 14.

Figure 14

By default, the label of the button is the name of the Button or Event trait to which it is linked.²⁰ However, this label may be set to any string by initializing the *label* attribute of the ButtonEditor with that string.

¹⁹ In Traits, a Button and an Event are essentially the same thing, except that Buttons are automatically associated with ButtonEditors.

²⁰ 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 3.1.2).

8.1.3 CheckListEditor

Suitable for	List
Default for	(none)
Required initialization	<i>values</i>
Optional initialization	<i>cols</i>

The `CheckListEditor` is designed to allow the user to edit a `List` trait by selecting elements from a “master list”, i.e., a list of possible values. The *values* trait of the `CheckListEditor` contains this master list, and must therefore be initialized when the editor is created.

The *values* trait may take either of two forms: a list of strings, or a list of tuples of the form (*<element>*, *<label>*), where *<element>* may 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.

The “custom” style of this editor is displayed as a set of checkboxes. By default, these checkboxes are displayed in a single column; however, the programmer initialize the *cols* attribute of the editor to any value between 1 and 20, in which case the corresponding number of columns will be used.

The “simple” style of the `CheckListEditor` appears as a drop-down list; in this style only one list element may be selected, resulting in a singleton list. The “text” and “readonly” 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 of the `CheckListEditor` are shown in Figure 15. Note that in this case the *cols* attribute has been initialized to 4.

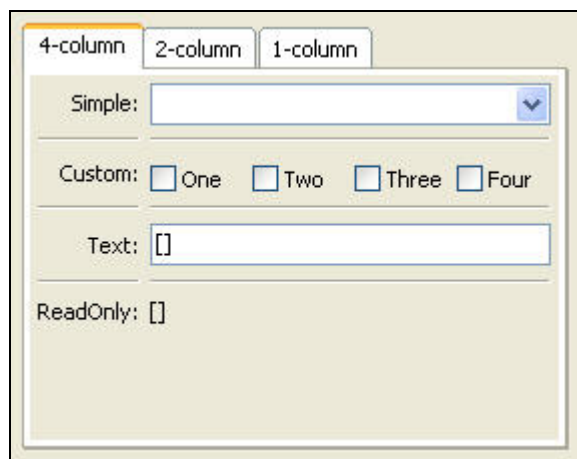


Figure 15

8.1.4 CodeEditor

Suitable for	Code, Str, String
Default for	Code
Required initialization	(none)
Optional initialization	<i>auto_set</i> , <i>key_bindings</i>

The purpose of the `CodeEditor` is, unsurprisingly, 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, nonwrapping lines with a horizontal scrollbar. The “text” style displays the trait using a single scrolling line with special characters to represent line breaks. The “readonly” 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).

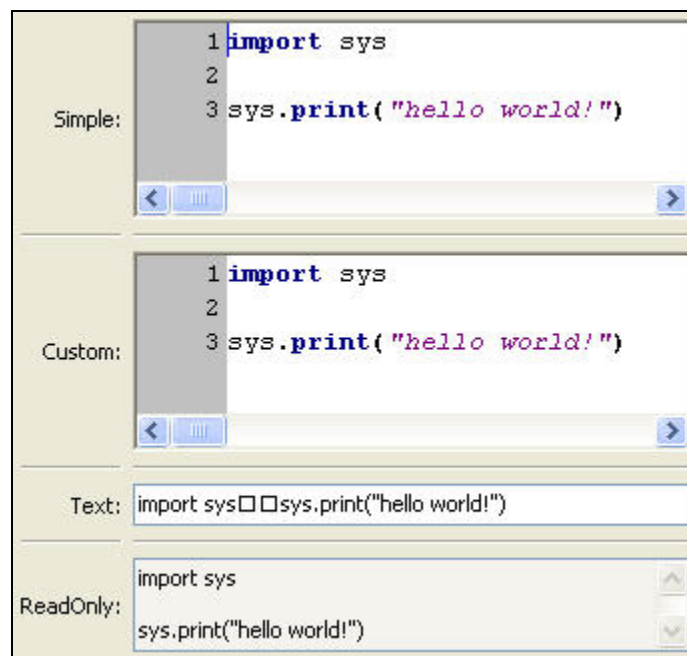


Figure 16

The *auto_set* keyword attribute 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 attribute is **True**.

8.1.5 ColorEditor

Suitable for	Color
Default for	Color
Required initialization	(none)
Optional initialization	<i>mapped</i>

The `ColorEditor` is designed to allow 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 this editor are shown in Figure 17.



Figure 17

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 brings up a pop-up window similar in appearance and function to the “custom” `ColorEditor` 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 interface to appear in a pop-up window, as shown in Figure 18.

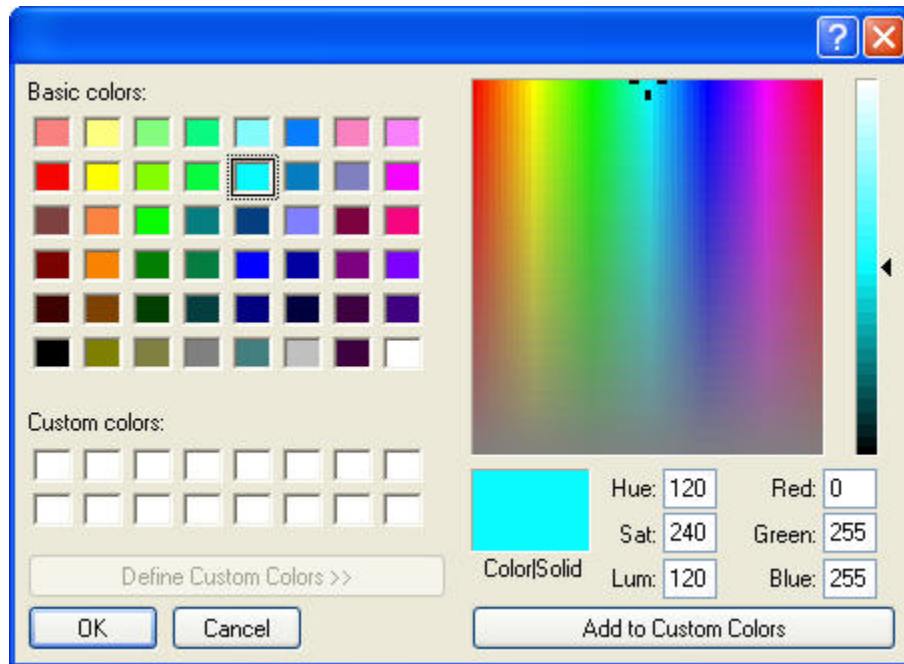


Figure 18

The “text” style of the `ColorEditor` looks exactly like the “simple” style, but the text box is editable (and clicking on it does not bring up a pop-up). It is the user’s responsibility to enter a recognized color name or a properly formatted (r, g, b) tuple.

The “readonly” 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 attribute of the `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 the `ColorEditor` is generally used only for `Color` traits, which are mapped (e.g., “cyan” to `wx.Colour(0, 255, 255)`), this attribute defaults to **True** and will not be of interest to most programmers. However, it is possible to define a custom color Trait that uses the `ColorEditor` but is not mapped (i.e., uses only one representation), which is why the attribute is available.

8.1.6 CompoundEditor

Suitable for	<i>special</i>
Default for	(none)
Required initialization	(none)
Optional initialization	<i>editors, auto_set</i>

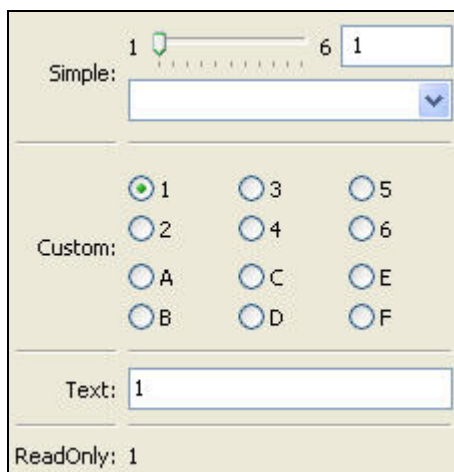


Figure 19

8.1.7 DirectoryEditor

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

The `DirectoryEditor` is designed to allow 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 19.

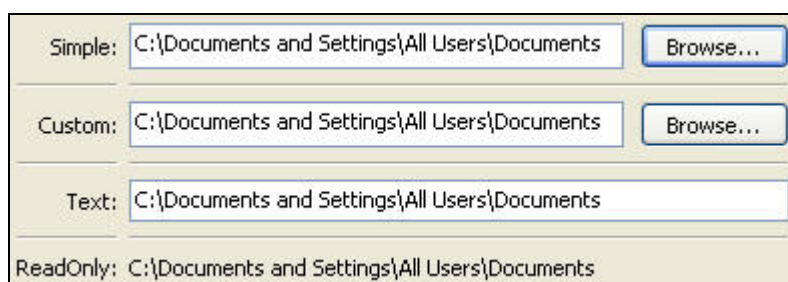


Figure 20

In the “simple” and “custom” styles (which are identical for this editor), the current value of the trait is displayed in a text box to the left of a “Browse...” button. The user may either type a new path directly into the text box or use the button to bring up a directory browser window as shown in Figure 20.

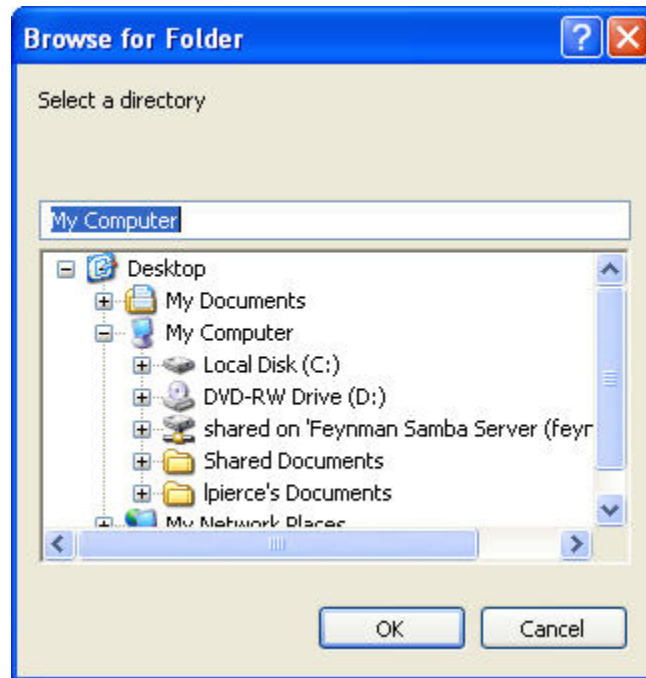


Figure 21

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

The “text” style of the editor is simply a text box into which the user may type a directory path. The “readonly” style is identical except that the text box is not editable.

No validation is performed on `Directory` traits; it is the user’s responsibility to ensure that a typed-in value is in fact an actual directory on the system.

8.1.8 EnableRGBAColorEditor

Suitable for	RGBAColor
Default for	(none)
Required initialization	(none)
Optional initialization	<i>auto_set, mode, edit_alpha, text, font</i>

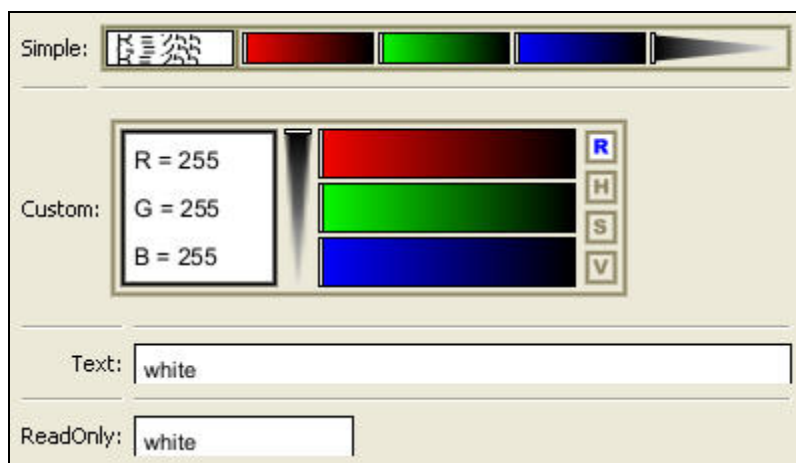


Figure 22

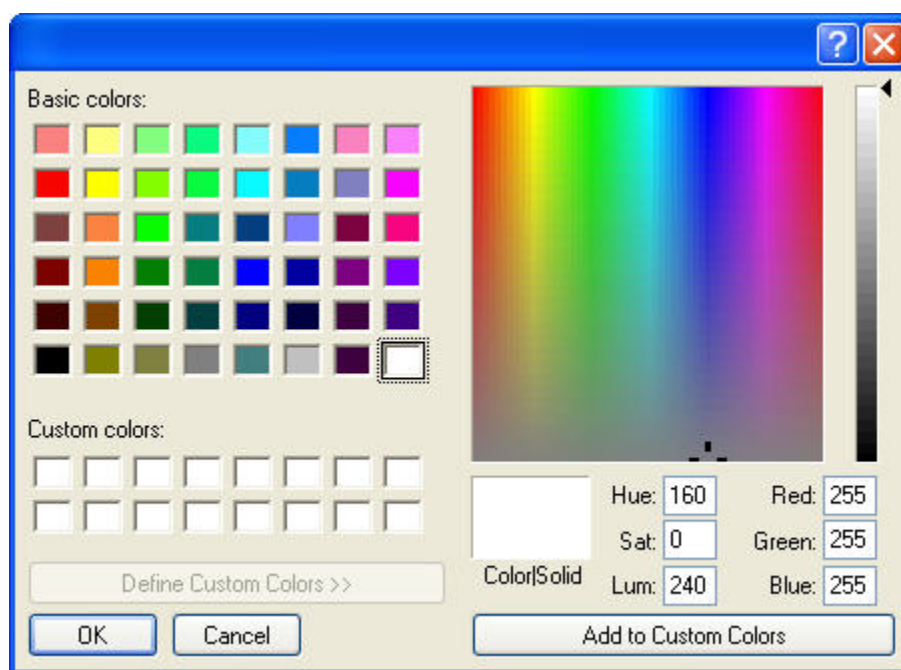


Figure 23

8.1.9 EnumEditor

Suitable for	Enum, Any
Default for	Enum
Required initialization	for non-Enum traits: <i>values or name,</i> <i>object</i>
Optional initialization	<i>evaluate, cols, mode</i>

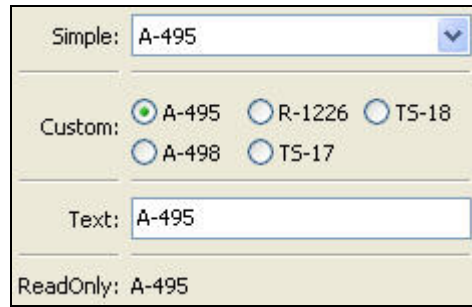


Figure 24

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

```
from enthought.traits import HasTraits, Trait
from enthought.traits.ui 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 EnumEditor strips off 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

The FileEditor is designed to allow 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 24.

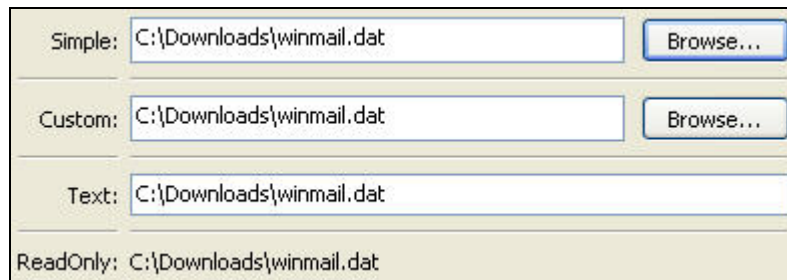


Figure 25

The behavior and appearance of this editor are very similar to those of the DirectoryEditor, except that the “Browse...” button activates a file browser rather than a directory browser, as shown in Figure 25.

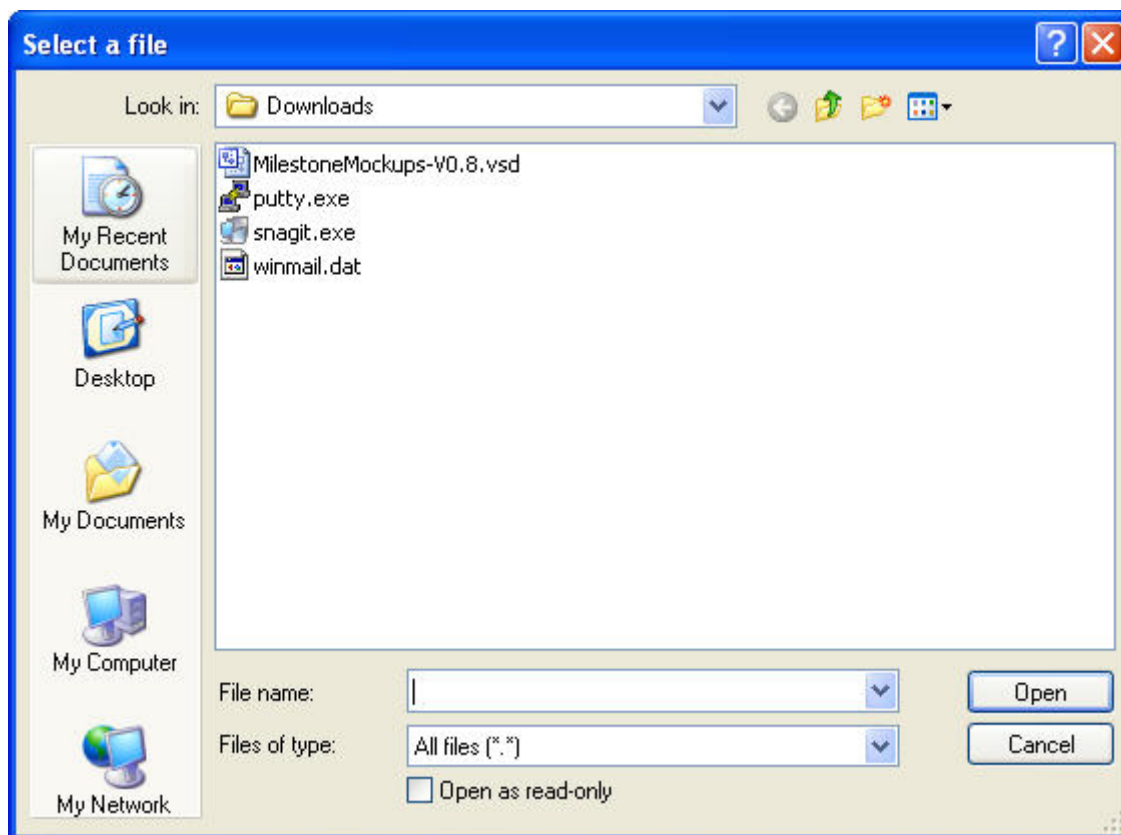


Figure 26

8.1.11 FontEditor

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

The `FontEditor` allows 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 26.

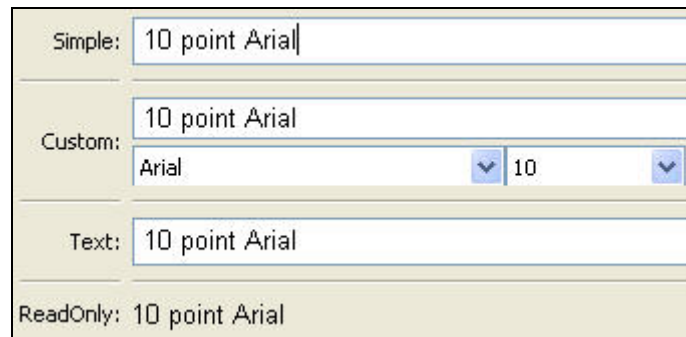


Figure 27

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 pop-up dialog appears with the detailed interface shown in Figure 27. Once the user hits “OK”, control returns to the editor, which then displays the newly selected font.

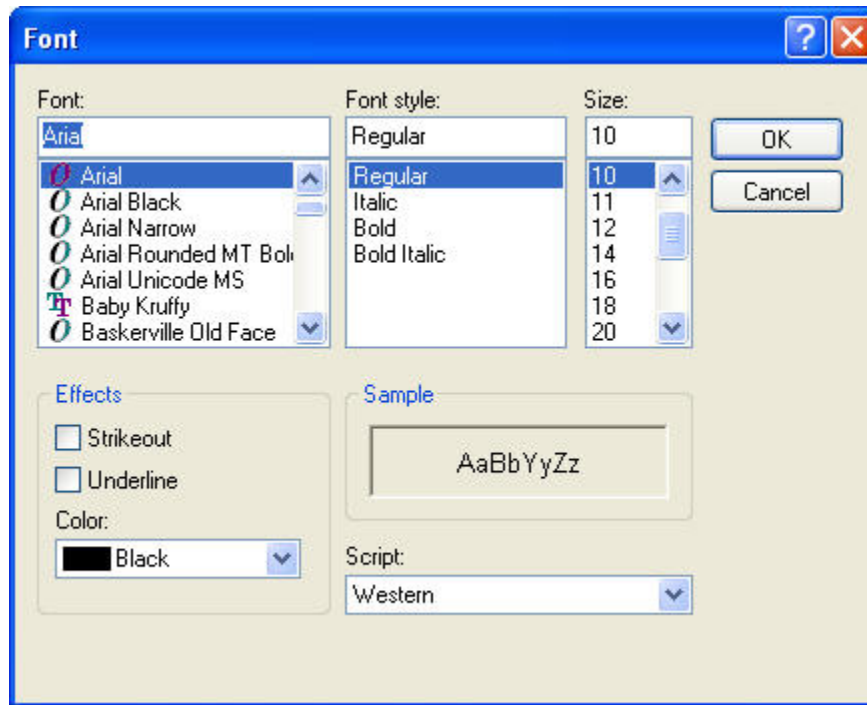


Figure 28

In the “custom” style, an abbreviated version of the font dialog is displayed in-line. The user may 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; it is the user’s responsibility to enter the correct name of an available font. The “readonly” style is identical except that the text box is not editable.

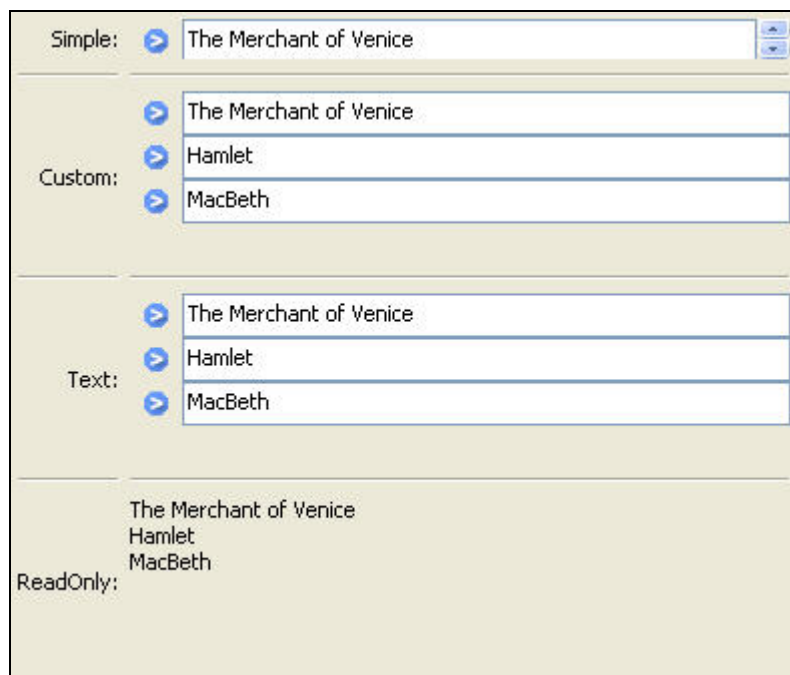
8.1.12 KivaFontEditor

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

This editor is identical in appearance and function to the `FontEditor`, except that it is used to display and edit `KivaFont` traits.

8.1.13 ListEditor

Suitable for	List
Default for	List ²¹



The screenshot displays the ListEditor interface with four distinct editing modes, each showing a list of three items: 'The Merchant of Venice', 'Hamlet', and 'MacBeth'.

- Simple:** A single text input field with a blue arrow icon on the left and a dropdown arrow on the right.
- Custom:** Three separate text input fields, each with a blue arrow icon on the left.
- Text:** Three separate text input fields, each with a blue arrow icon on the left.
- ReadOnly:** A single text area displaying the list items as plain text.

Figure 29

²¹ If a List is made up of HasTraits objects, the TableEditor is used as the default instead; see Section 8.2.6.

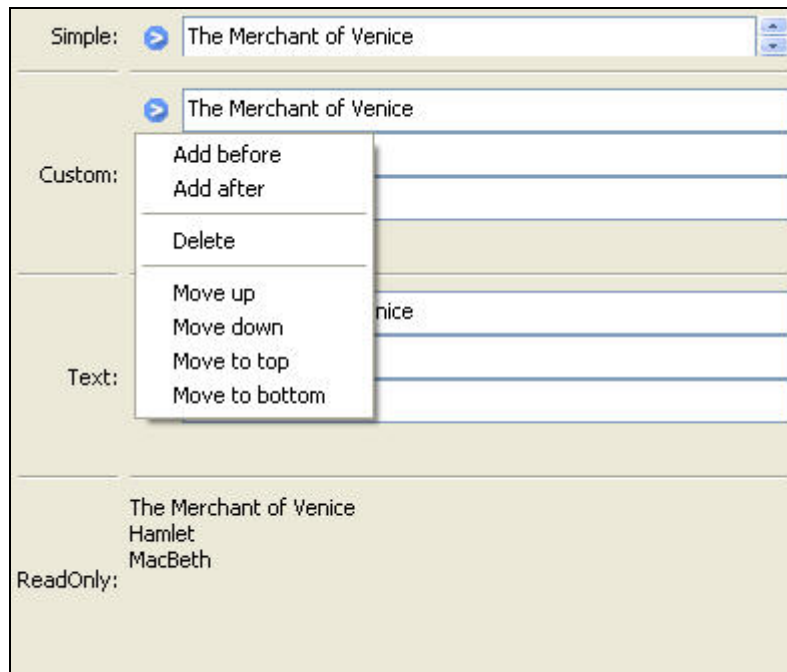


Figure 30

8.1.14 RangeEditor

Suitable for	Range
Default for	Range (all types)

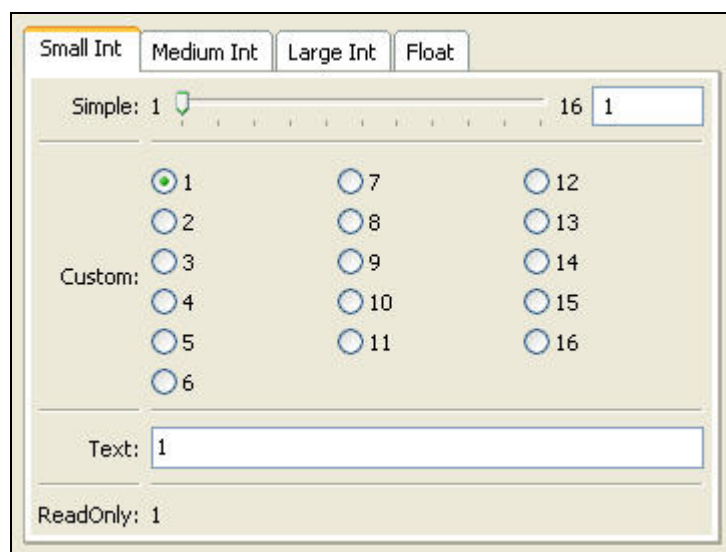
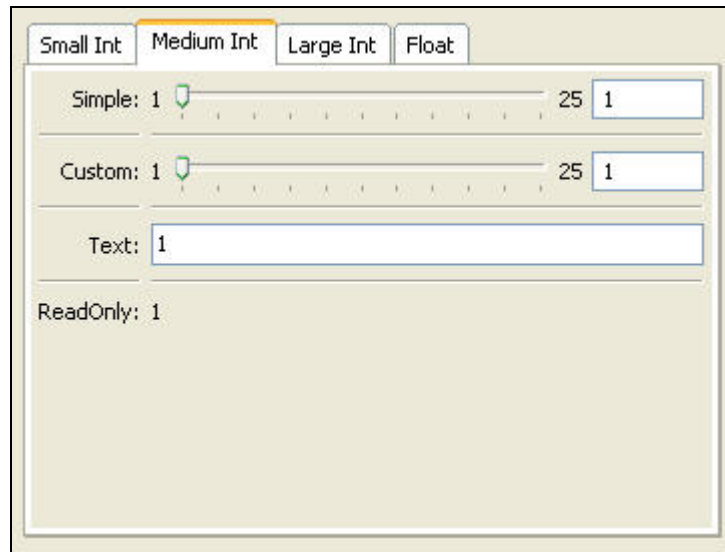


Figure 31



Small Int Medium Int Large Int Float

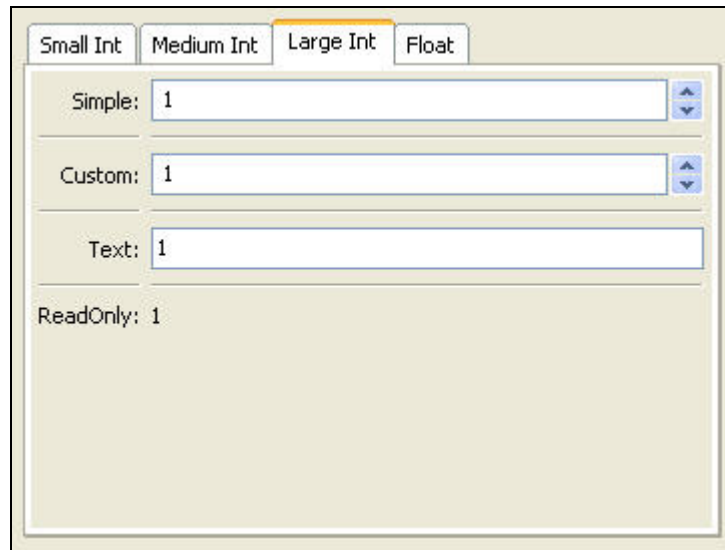
Simple: 1 25 1

Custom: 1 25 1

Text: 1

ReadOnly: 1

Figure 32



Small Int Medium Int Large Int Float

Simple: 1

Custom: 1

Text: 1

ReadOnly: 1

Figure 33

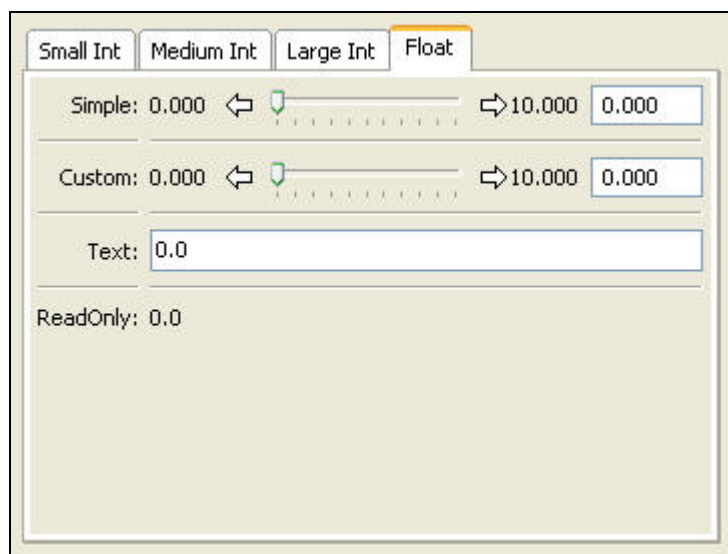


Figure 34

8.1.15 RGBColorEditor

Suitable for	RGBColor
Default for	RGBColor

(Identical in appearance to ColorEditor)

8.1.16 RGBAColorEditor

Suitable for	RGBAColor
Default for	RGBAColor

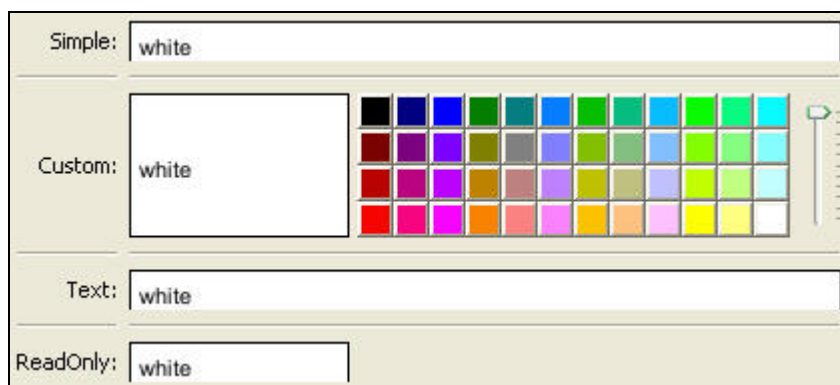


Figure 35

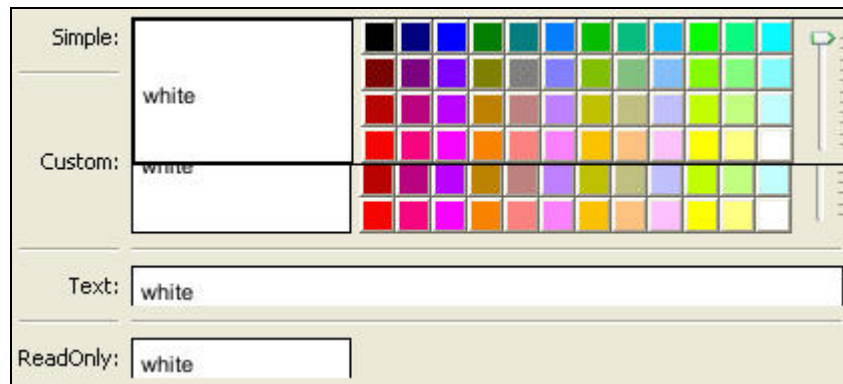


Figure 36

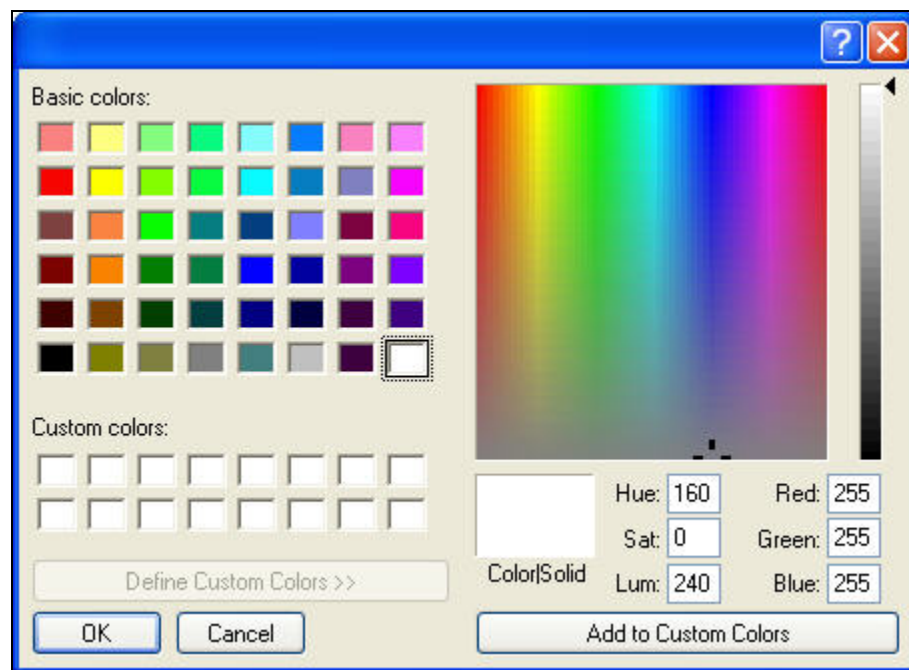


Figure 37

8.1.17 SetEditor

Suitable for	List
Default for	<i>none</i>

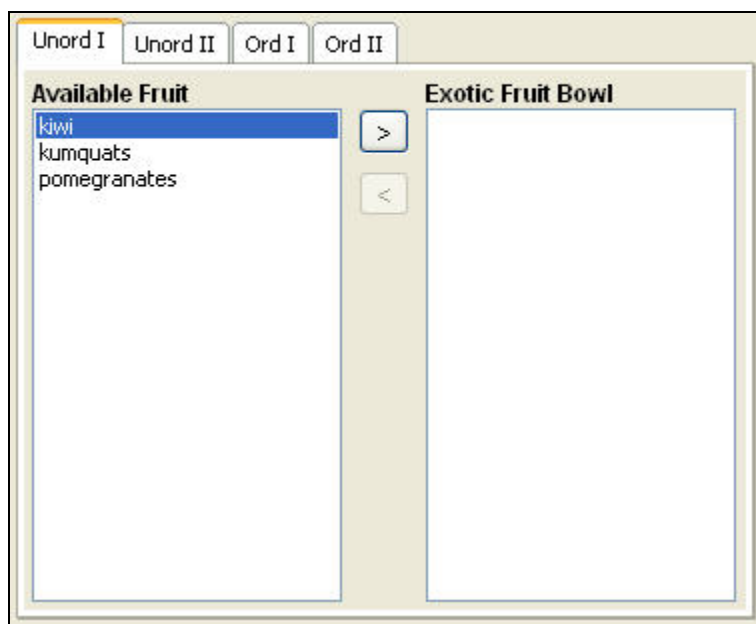


Figure 38

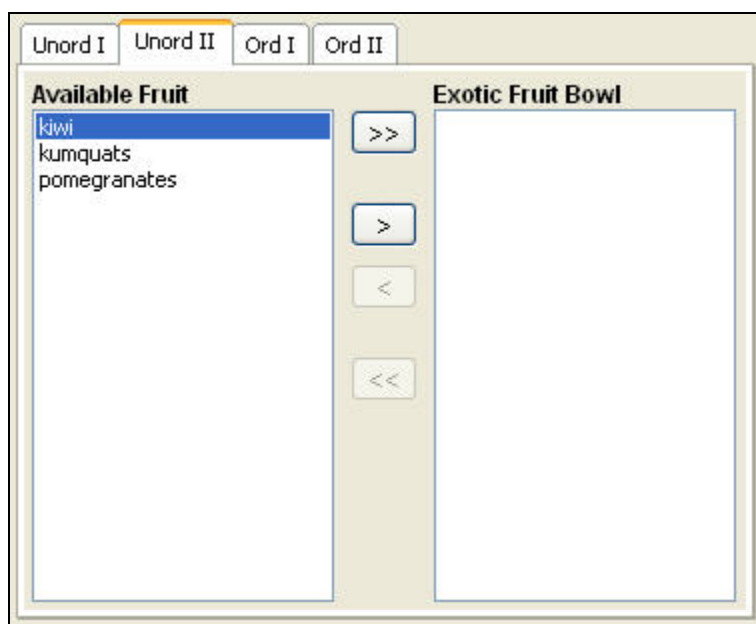


Figure 39

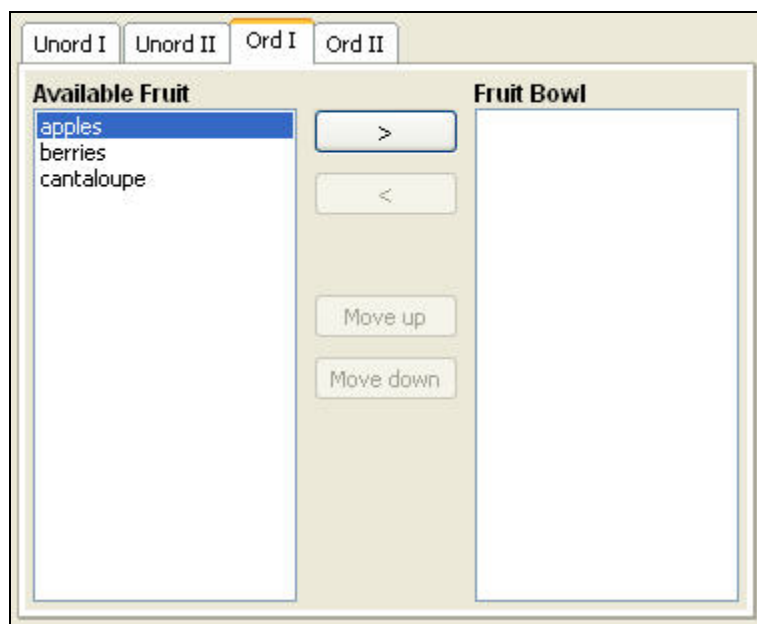


Figure 40

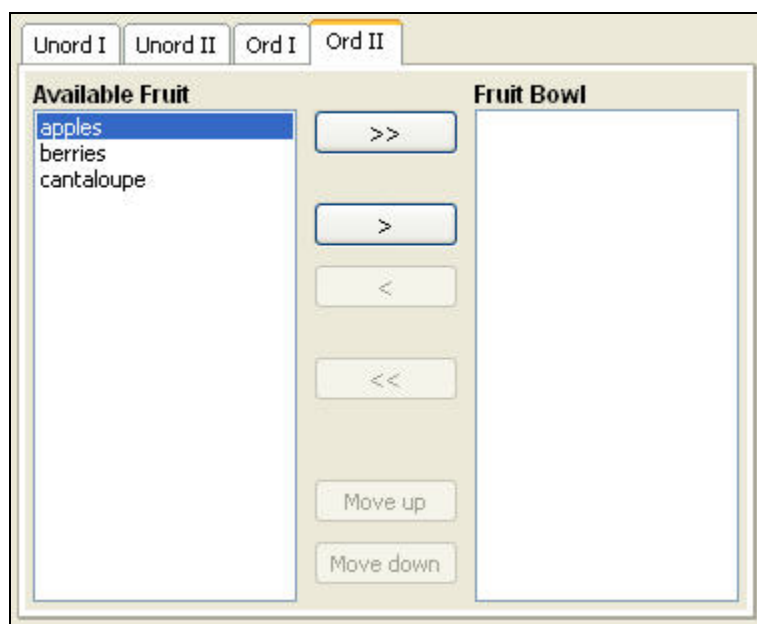


Figure 41

8.1.18 TextEditor

Suitable for	all
--------------	-----

Default for	Str, String, Password, Int, Float, <i>special</i>
--------------------	---



Figure 42

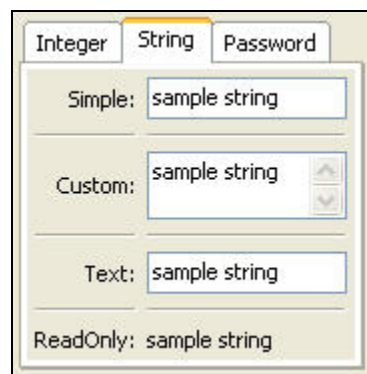


Figure 43

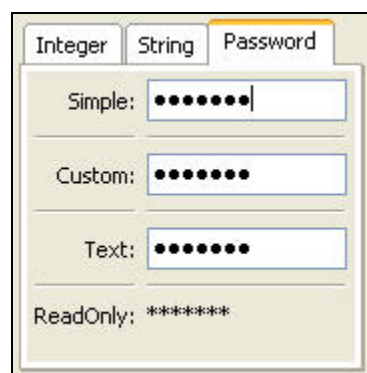
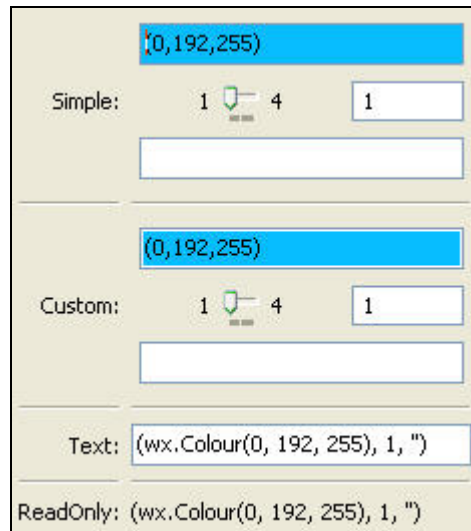


Figure 44

8.1.19 TupleEditor

Suitable for	Tuple
Default for	Tuple



The image shows a screenshot of the TupleEditor dialog box. It has four tabs: Simple, Custom, Text, and ReadOnly. The Simple and Custom tabs are currently selected and show a text input field with the value (0,192,255). The Text and ReadOnly tabs show the same value in a read-only text field. The Simple and Custom tabs also have a 'Simple:' or 'Custom:' label, a '1' in a small box, a green arrow icon, a '4' in a small box, and a '1' in a small box. Below these are empty text input fields.

Figure 45

8.2 Advanced Trait Editors

8.2.1 CustomEditor

8.2.2 DropEditor

8.2.3 ImageEnumEditor

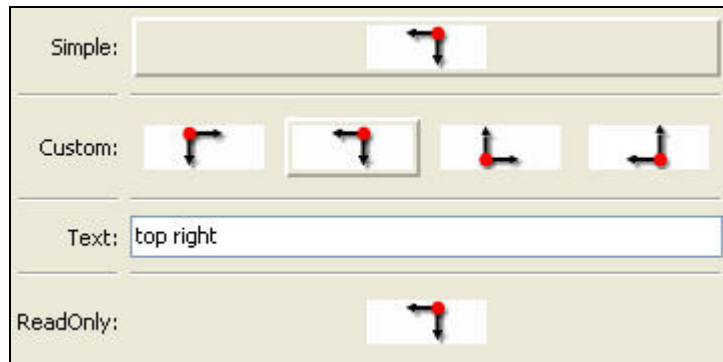
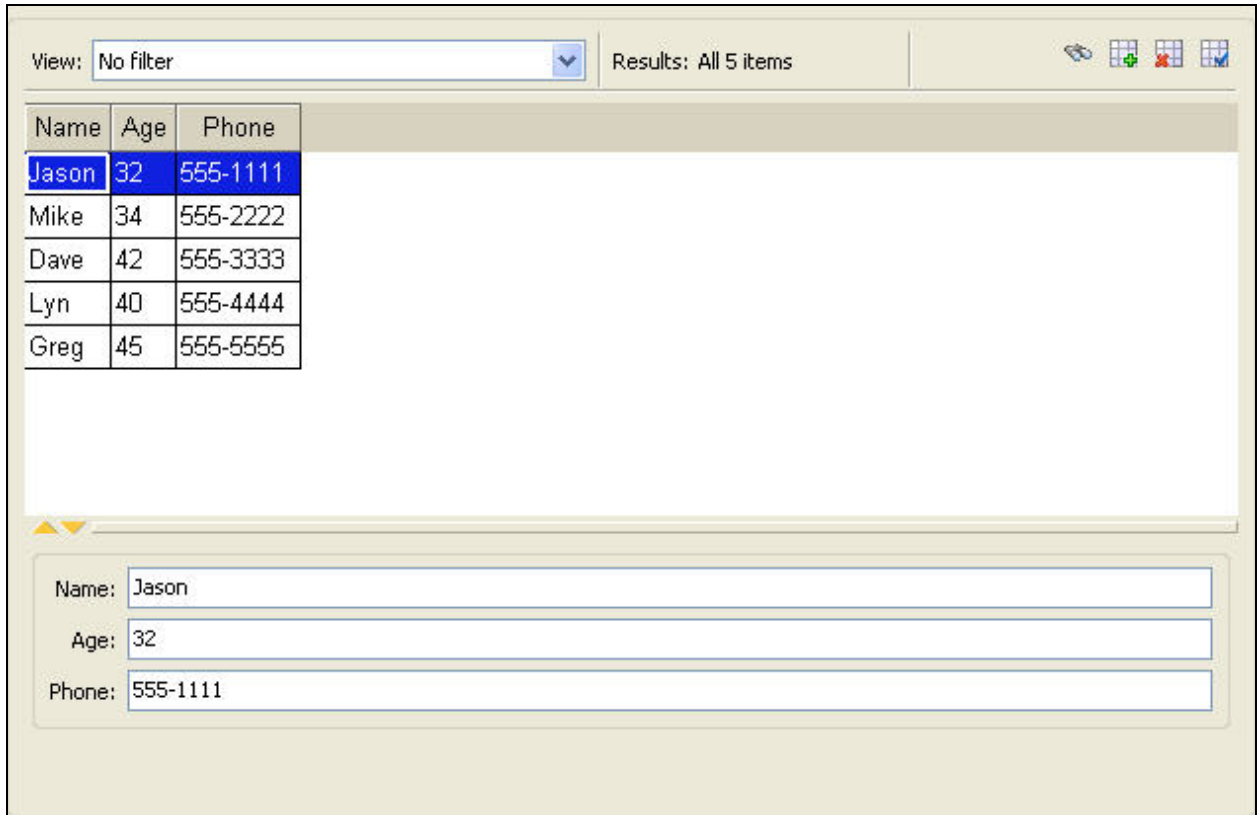


Figure 46

8.2.4 InstanceEditor

8.2.5 PlotEditor

8.2.6 TableEditor



View: Results: All 5 items

Name	Age	Phone
Jason	32	555-1111
Mike	34	555-2222
Dave	42	555-3333
Lyn	40	555-4444
Greg	45	555-5555

Name:

Age:

Phone:

Figure 47

8.2.7 TreeEditor

9 Advanced Editor Concepts

9.1 Interacting with an Editor Through the UI Object

9.1.1 Accessing Trait Editors Using Item ‘id’s

9.1.2 Controlling Editor Status Using ‘enabled/disabled’

9.2 Introduction to Editor Factories

9.3 Defining a Custom Editor

9.3.1 Defining the Editor Factory

9.3.1.1 Standard Traits: `format_str`, `format_func`, `is_grid_cell`

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

9.3.2 Defining the Editor

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

9.3.2.2 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

11 Tips, Tricks and Gotchas

11.1 Getting/Setting Model View Elements

For some applications, it may 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), the method call `sam.trait_views()` returns the list `['all_view', 'traits_view']`.

Alternatively, a call to `obj.trait_views(view_element_type)` returns a list of all named instances of class `view_element_type` defined in `obj`'s class, where the possible values of `view_element_type` are `View`, `Group`, `Item`, `ViewElement`, and `ViewSubElement`. Thus calling `trait_views(View)` is identical to calling `trait_views()`. Note that the call `sam.trait_views(Group)` returns an empty list, although 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 will 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).

Appendix I: Glossary of Terms

attribute: An element of data associated with all instances of a given class, and named at the class level.²² 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 window whose purpose is to allow a user to interact with a program, usually by entering data.

dictionary: A lookup table of the form {key1: value1, key2: value2, ... key_n: value_n}, implemented as 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 the user-programmer can override this selection with one of a number of editor types provided by the Traits UI package. In some cases an editor may 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`.

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

²² This is not always the case in Python, where attributes can be added to individual objects.

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

HasTraits: A class defined in Traits to specify objects whose attributes are typed (i.e., 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()`.²³

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 dialog box 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 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”.

²³ A method may, of course, have arguments. They are omitted from the sample syntax for simplicity.

panel: A construct 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` and/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 that will be available. 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: Default Editors for Standard Traits