



TraitsUI 4 User Manual

Release 6.0.0

Enthought, Inc.

April 12, 2018

Contents

| | | |
|----------|---|------------|
| 1 | TraitsUI 6.0 User Manual | 1 |
| 1.1 | TraitsUI 6.0 User Manual | 1 |
| 1.2 | Introduction | 2 |
| 1.3 | The View and Its Building Blocks | 4 |
| 1.4 | Customizing a View | 11 |
| 1.5 | Advanced View Concepts | 16 |
| 1.6 | Controlling the Interface: the Handler | 23 |
| 1.7 | Introduction to Trait Editor Factories | 30 |
| 1.8 | The Predefined Trait Editor Factories | 36 |
| 1.9 | Advanced Trait Editors | 63 |
| 1.10 | “Extra” Trait Editor Factories | 84 |
| 1.11 | Advanced Editor Adapters | 85 |
| 1.12 | Tips, Tricks and Gotchas | 95 |
| 1.13 | Appendix I: Glossary of Terms | 96 |
| 1.14 | Appendix II: Editor Factories for Predefined Traits | 98 |
| 2 | TraitsUI 6.0 API Reference | 101 |
| 2.1 | traitsui package | 101 |
| 3 | TraitsUI 6.0 Tutorials | 181 |
| 3.1 | Writing a graphical application for scientific programming using TraitsUI 6.0 | 181 |
| 4 | TraitsUI 6.0 Demos | 201 |
| 4.1 | Standard Editors | 201 |
| 4.2 | Advanced Demos | 202 |
| 5 | Traits UI Changelog | 205 |
| 5.1 | Release 6.0.0 | 205 |
| 5.2 | Release 5.2.0 | 207 |
| 5.3 | Release 5.1.0 | 207 |
| 5.4 | Release 5.0.0 | 208 |
| 5.5 | Release 4.5.1 | 209 |
| 5.6 | Release 4.5.0 | 209 |
| 5.7 | Release 4.4.0 | 209 |
| 5.8 | Traits 3.5.0 (Oct 15, 2010) | 210 |
| 5.9 | Traits 3.4.0 (May 26, 2010) | 210 |
| 5.10 | Traits 3.3.0 (Feb 24, 2010) | 211 |

| | | |
|----------|---|------------|
| 5.11 | Traits 3.2.0 (July 15, 2009) | 211 |
| 6 | TraitsUI: Traits-capable windowing framework | 213 |
| 6.1 | Example | 213 |
| 6.2 | Important Links | 214 |
| 6.3 | Installation | 214 |
| 6.4 | Running the Test Suite | 215 |
| 7 | Indices and tables | 217 |
| | Python Module Index | 219 |

1.1 TraitsUI 6.0 User Manual

Authors Lyn Pierce, Janet Swisher, and Enthought Developers

Version Document Version 4

Copyright 2005, 2008-2018 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 Enthought, Inc., nor the names of 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 2100

Austin TX 78701
1.512.536.1057 (voice)
1.512.536.1059 (fax)
<http://www.enthought.com>
info@enthought.com

1.2 Introduction

This guide is designed to act as a conceptual guide to *TraitsUI*, an open-source package built and maintained by Enthought, Inc. The TraitsUI package is a set of GUI (Graphical User Interface) tools designed to complement *Traits*, another Enthought open-source package that provides explicit 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. This guide discusses many but not all features of TraitsUI. For complete details of the TraitsUI API, refer to the *TraitsUI API Reference*.

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

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

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

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

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

The Model: HasTraits Subclasses and Objects

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

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 *The View and Its Building Blocks*; 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 TraitsUI construct.

¹ A third type of content object, Include, is discussed briefly in *Include Objects*, but presently is not commonly used.

The Controller: Handler Subclasses and Objects

The controller for a Traits-based application is defined in terms of the *Handler* class.² Specifically, the relationship between any given View instance and the underlying model is managed by an instance of the Handler class. For simple interfaces, the Handler can be implicit. For example, none of the examples in the first four chapters includes or requires any specific Handler code; they are managed by a default Handler that performs the basic operations of window initialization, transfer of data between GUI and model, and window closing. Thus, a programmer new to TraitsUI need not be concerned with Handlers at all. Nonetheless, custom handlers can be a powerful tool for building sophisticated application interfaces, as discussed in *Controlling the Interface: the Handler*.

1.2.2 Toolkit Selection

The TraitsUI package is designed to be toolkit-independent. Programs that use TraitsUI do not need to explicitly import or call any particular GUI toolkit code unless they need some capability of the toolkit that is not provided by TraitsUI. However, *some* particular toolkit must be installed on the system in order to actually display GUI windows.

TraitsUI uses a separate package, `traits.etsconfig`, to determine which GUI toolkit to use. This package is also used by other Enthought packages that need GUI capabilities, so that all such packages “agree” on a single GUI toolkit per application. The `etsconfig` package contains a singleton object, **ETSTConfig** (importable from `traits.etsconfig.api`), which has a string attribute, **toolkit**, that signifies the GUI toolkit.

The values of **ETSTConfig.toolkit** that are supported by TraitsUI version 6.0 are:

- ‘qt4’: `PyQt`, which provides Python bindings for the `Qt` framework version 4.
- ‘wx’: `wxPython`, which provides Python bindings for the `wxWidgets` toolkit.
- ‘null’: A do-nothing toolkit, for situations where neither of the other toolkits is installed, but Traits is needed for non-UI purposes.

The default behavior of TraitsUI is to search for available toolkit-specific packages in the order listed, and uses the first one it finds. The programmer or the user can override this behavior in any of several ways, in the following order of precedence:

1. The program can explicitly set **ETSTConfig.toolkit**. It must do this before importing from any other Enthought Tool Suite component, including `traits`. For example, at the beginning of a program:

```
from traits.etsconfig.api import ETSTConfig
ETSTConfig.toolkit = 'wx'
```

2. The user can define a value for the `ETS_TOOLKIT` environment variable.

Warning: The default order of toolkits changed in TraitsUI 5.0 to prefer ‘qt4’ over ‘wx’.

1.2.3 Structure of this Manual

The intent of this guide is to present the capabilities of the TraitsUI package in usable increments, so that you can create and display gradually more sophisticated interfaces from one chapter to the next.

- *The View and Its Building Blocks*, *Customizing a View*, and *Advanced View Concepts* 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.

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

- *Controlling the Interface: the Handler* explains how to use the Handler class to implement custom GUI behaviors, as well as menus and toolbars.
- *Introduction to Trait Editor Factories* and *The Predefined Trait Editor Factories* show how to control GUI widget selection by means of trait *editors*.
- *Tips, Tricks and Gotchas* covers miscellaneous additional topics.
- Further reference materials, including a *Appendix I: Glossary of Terms* and an API summary for the TraitsUI classes covered in this Manual, are located in the Appendices.

1.3 The View and Its Building Blocks

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 window containing editable fields for each of the object's *trait attributes*. For example, the following sample code⁴ defines the `SimpleEmployee` class, creates an object of that class, and constructs and displays a GUI for the object:

Example 1: Using `configure_traits()`

```
# configure_traits.py -- Sample code to demonstrate
#                               configure_traits()
from traits.api import HasTraits, Str, Int
import traitsui

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:

1.3.1 The View Object

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

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

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

⁴ All code examples in this guide that include a file name are also available as examples in the `tutorials/doc_examples/examples` subdirectory of the Traits docs directory. You can run them individually, or view them in a tutorial program by running: `python Traits_dir/tutorials/tutor.py Traits_dir/docs/tutorials/doc_examples`



Fig. 1.1: Figure 1: User interface for Example 1

Example 2: Using `configure_traits()` with a View object

```
# configure_traits_view.py -- Sample code to demonstrate
#                             configure_traits()

from traits.api import HasTraits, Str, Int
from traitsui.api import View, Item
import traitsui

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:



Fig. 1.2: Figure 2: User interface for Example 2

A View object can have a variety of attributes, which are set in the View definition, following any Group or Item objects.

The sections on [Contents of a View](#) through [Advanced View Concepts](#) explore the contents and capabilities of Views. Refer to the [Traits API Reference](#) for details of the View class.

Except as noted, all example code uses the `configure_traits()` method; a detailed description of this and other techniques for creating GUI displays from Views can be found in *Displaying a View*.

1.3.2 Contents 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 definition can contain one or more objects of either of these types, which are specified as arguments to the View constructor, as in the case of the three Items in Example 2.

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

The Item Object

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

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

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

Attributes of Item, by category

Content These attributes specify the actual data to be displayed by an item. Because an Item is essentially a template for displaying a single trait, its **name** attribute is nearly always specified.

name: str The name of the trait being edited.

Display format In addition to specifying which trait attributes are to be displayed, you might need to adjust the format of one or more of the resulting widgets.

If an Item's **label** attribute is specified but not its name, the value of **label** is displayed as a simple, non-editable string. (This feature can be useful for displaying comments or instructions in a TraitsUI window.)

dock: Docking style for the item.

emphasized: bool Should label text be emphasized?

export: Category of elements dragged from view.

height: Requested height as pixels ($\text{height} > 1$) or proportion of screen ($0 < \text{height} < 1$)

image: Image to show on tabs.

label: str The label to display on the item.

padding: int Amount of extra space, in pixels, to add around the item. Values must be integers between -15 and 15. Use negative values to subtract from the default spacing.

resizable: bool Can the item be resized to use extra space. The default is False.

show_label: bool Whether to show the label or not (defaults to True).

springy: bool Use extra space in the parent layout? The default is False.

width: float Requested width as pixels ($\text{width} > 1$) or proportion of screen ($0 < \text{width} < 1$).

Content format In some cases it can be desirable to apply special formatting to a widget's contents rather than to the widget itself. Examples of such formatting might include rounding a floating-point value to two decimal places, or capitalizing all letter characters in a license plate number.

format_str: str '% format' string for text.

format_func: func Format function for text.

Widget override These attributes override the widget that is automatically selected by TraitsUI. These options are discussed in *Introduction to Trait Editor Factories* and *The Predefined Trait Editor Factories*.

editor: ItemEditor Editor to use.

style: {'simple', 'custom', 'text', 'readonly'} The editor style (see *Specifying an Editor Style*).

Visibility and status Use these attributes to create a simple form of a dynamic GUI, which alters the display in response to changes in the data it contains. More sophisticated dynamic behavior can be implemented using a custom *Handler* (see *Controlling the Interface: the Handler*).

enabled_when: str Expression that determines whether of group can be edited.

visible_when: str Expression that determines visibility of group.

defined_when: str Expression that determines inclusion of group in parent.

has_focus: bool Should this item get initial focus?

User help These attributes provide guidance to the user in using the user interface.

tooltip: str Tooltip to display on mouse-over.

help: If the **help** attribute is not defined for an Item, a system-generated message is used instead.

help_id: It is ignored by the default help handler, but can be used by a custom help handler.

Unique identifier

id: Used as a key for saving user preferences about the widget. If **id** is not specified, the value of the **name** attribute is used.

Subclasses of Item

The TraitsUI package defines the following subclasses of Item, which are helpful shorthands for defining certain types of items. Label, Heading and Spring are intended to help with the layout of a TraitsUI View, and need not have a trait attribute associated with them. For example, `Spring()` and `Label("This is a label")` are valid code.

| Sub-class | Description | Equivalent To |
|-----------|--|--|
| Label | An item that is just a label and doesn't require a trait name associated with it | |
| Heading | A fancy label | |
| Spring | A item that expands to take as much space as necessary | <code>Item(name='spring', springy=True, show_label=False)</code> |
| Custom | An item with a custom editor style | <code>Item(style='custom')</code> |
| Read-only | An item with a readonly editor style | <code>Item(style='readonly')</code> |
| UItem | An item with no label | <code>Item(show_label=False)</code> |
| UCustom | A Custom item with no label | <code>Item(style='custom', show_label=False)</code> |
| UReadonly | A Readonly item with no label | <code>Item(style='readonly', show_label=False)</code> |

The Group Object

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

Consider the following enhancement to Example 2:

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

```
# configure_traits_view_group.py -- Sample code to demonstrate
#                                     configure_traits()
from traits.api import HasTraits, Str, Int
from traitsui.api import View, Item, Group
import traitsui

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

    employee_number = Str
    salary = Int

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

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

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



Fig. 1.3: Figure 3: User interface for Example 3

Content of a Group

The content of a Group object is specified exactly like that of a View object. In other words, one or more Item or Group objects are given as arguments to the Group constructor, e.g., the three Items in Example 3.⁵ The objects contained in a Group are called the *elements* of that Group. Groups can be nested to any level.

Group Attributes

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

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

Attributes of Group, by category

Content

object: References the object whose traits are being edited by members of the group; by default this is ‘object’, but could be another object in the current context.

content: list List of elements in the group.

Display format These attributes define display options for the group as a whole.

columns: The number of columns in the group.

dock: Dock style of sub-groups.

dock_theme: The theme to use for the dock.

export: Category of elements dragged from view.

image: Image to show on tabs.

label: The label to display on the group.

layout: {‘normal’, ‘flow’, ‘split’, ‘tabbed’} Layout style of the group, which can be one of the following:

- ‘normal’ (default): Sub-groups are displayed sequentially in a single panel.

⁵ As with Views, it is possible for a Group to contain objects of more than one type, but it is not recommended.

- ‘flow’: Sub-groups are displayed sequentially, and then “wrap” when they exceed the available space in the **orientation** direction.
- ‘split’: Sub-groups are displayed in a single panel, separated by “splitter bars”, which the user can drag to adjust the amount of space for each sub-group.
- ‘tabbed’: Each sub-group appears on a separate tab, labeled with the sub-group’s *label* text, if any.

This attribute is ignored for groups that contain only items, or contain only one sub-group.

orientation: {‘vertical’, ‘horizontal’} The orientation of the subgroups.

padding: **int** Amount of extra space, in pixels, to add around the item. Values must be integers between -15 and 15. Use negative values to subtract from the default spacing.

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

show_border: **bool** Should a border be shown or not

show_labels: Show the labels of items?

show_left: **bool** Show labels on the left or the right.

springy: **bool** Use extra space in the parent layout? The default is False.

style: {‘simple’, ‘custom’, ‘text’, ‘readonly’} Default editor style of items in the group.

Visibility and status These attributes work similarly to the attributes of the same names on the Item class.

enabled_when: **str** Expression that determines whether of group can be edited.

visible_when: **str** Expression that determines visibility of group.

defined_when: **str** Expression that determines inclusion of group in parent.

User help The help text is used by the default help handler only if the group is the only top-level group for the current View. For example, suppose help text is defined for a Group called **group1**. The following View shows this text in its help window:

```
View(group1)
```

The following two do not:

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

help: **str** Help message. If the **help** attribute is not defined, a system-generated message is used instead.

help_id: The **help_id** attribute is ignored by the default help handler, but can be used by a custom help handler.

Unique identifier

id: **str** The **id** attribute is used as a key for saving user preferences about the widget. If **id** is not specified, the **id** values of the elements of the group are concatenated and used as the group identifier.

Subclasses of Group

The TraitsUI package defines the following subclasses of Group, which are helpful shorthands for defining certain types of groups. Refer to the *Traits API Reference* for details.

Subclasses of Group

| Sub-class | Description | Equivalent To |
|-----------|---|--|
| HGroup | A group whose items are laid out horizontally. | <code>Group(orientation='horizontal')</code> |
| HFlow | A horizontal group whose items “wrap” when they exceed the available horizontal space. | <code>Group(orientation='horizontal', layout='flow', show_labels=False)</code> |
| HSplit | A horizontal group with splitter bars to separate it from other groups. | <code>Group(orientation='horizontal', layout='split')</code> |
| Tabbed | A group that is shown as a tab in a notebook. | <code>Group(orientation='horizontal', layout='tabbed', springy=True)</code> |
| VGroup | A group whose items are laid out vertically. | <code>Group(orientation='vertical')</code> |
| VFlow | A vertical group whose items “wrap” when they exceed the available vertical space. | <code>Group(orientation='vertical', layout='flow', show_labels=False)</code> |
| VFold | A vertical group in which items can be collapsed (i.e., folded) by clicking their titles. | <code>Group(orientation='vertical', layout='fold', show_labels=False)</code> |
| VGrid | A vertical group whose items are laid out in two columns. | <code>Group(orientation='vertical', columns=2)</code> |
| VSplit | A vertical group with splitter bars to separate it from other groups. | <code>Group(orientation='vertical', layout='split')</code> |

1.4 Customizing a View

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

1.4.1 Specifying Window Type: the **kind** Attribute

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

- ‘modal’
- ‘live’
- ‘livemodal’
- ‘nonmodal’
- ‘wizard’
- ‘panel’
- ‘subpanel’

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

Stand-alone Windows

The behavior of a stand-alone TraitsUI window can vary over two significant degrees of freedom. First, it can be *modal*, meaning that when the window appears, all other GUI interaction is suspended until the window is closed; if it is not modal, then both the window and the rest of the GUI remain active and responsive. Second, it can be *live*, meaning that any changes that the user makes to data in the 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 clicking an *OK* or *Apply* button; see *Command Buttons: the buttons Attribute*). 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 following table.

Matrix of TraitsUI windows

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

All of these window types are identical in appearance. Also, all types support the **buttons** attribute, which is described in *Command Buttons: the buttons Attribute*. Usually, a window with command buttons is called a *dialog box*.

Wizards

Unlike a window, 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.

TraitsUI 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 windows, and are primarily suitable for highly controlled user interactions such as software installation.

Panels and Subpanels

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

A *panel* is very similar to a window, except that it is embedded in a larger window, which need not be a TraitsUI window. Like windows, panels support the **buttons** View attribute, as well as any menus and toolbars that are specified for the View (see *Menus and Menu Bars*). 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.

1.4.2 Command Buttons: the buttons Attribute

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

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

Example 4: Using a View object with buttons

```
# configure_traits_view_buttons.py -- Sample code to demonstrate
#                                configure_traits()

from traits.api import HasTraits, Str, Int
from traitsui.api import View, Item
from traitsui.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 now two buttons are displayed at the bottom: *OK* and *Cancel*:



Fig. 1.4: Figure 4: User interface for Example 4

There are six standard buttons defined by TraitsUI. Each of the standard buttons has matching a string alias. You can either import and use the button names, or simply use their aliases:

⁶ Actually, the value of the **buttons** attribute is really a list of Action objects, from which GUI buttons are generated by TraitsUI. The Action class is described in [Actions](#).

Command button aliases

| Button Name | Button Alias |
|--------------|------------------------|
| UndoButton | 'Undo' |
| ApplyButton | 'Apply' |
| RevertButton | 'Revert' |
| OKButton | 'OK' (case sensitive!) |
| CancelButton | 'Cancel' |

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

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

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

```
from traitsui.menu import OKCancelButton

    buttons = OKCancelButton
```

The special constant `NoButtons` can be used to create a window or panel without command buttons. While this is the default behavior, `NoButtons` can be useful for overriding an explicit value for **buttons**. You can also specify `buttons = []` to achieve the same effect. Setting the **buttons** attribute to an empty list has the same effect as not defining it at all.

It is also possible to define custom buttons and add them to the **buttons** list; see [Custom Command Buttons](#) for details.

1.4.3 Other View Attributes

Attributes of View, by category

Window display These attributes control the visual properties of the window itself, regardless of its content.

dock: `{'fixed', 'horizontal', 'vertical', 'tabbed'}` The default docking style to use for sub-groups of the view. The following values are possible:

- `'fixed'`: No rearrangement of sub-groups is allowed.
- `'horizontal'`: Moveable elements have a visual “handle” to the left by which the element can be dragged.
- `'vertical'`: Moveable elements have a visual “handle” above them by which the element can be dragged.
- `'tabbed'`: Moveable elements appear as tabbed pages, which can be arranged within the window or “stacked” so that only one appears at a time.

height: `int` or `float` Requested height for the view window, as an (integer) number of pixels, or as a (floating point) fraction of the screen height.

icon: `str` The name of the icon to display in the dialog window title bar.

image: `Image` The image to display on notebook tabs.

resizable: bool Can the user resize the window?

scrollable: bool Can the user scroll the view? If set to True, window-level scroll bars appear whenever the window is too small to show all of its contents at one time. If set to False, the window does not scroll, but individual widgets might still contain scroll bars.

statusbar: Status bar items to add to the view's status bar. The value can be:

- **None:** No status bar for the view (the default).
- **string:** Same as `[StatusItem(name=string)]`.
- **StatusItem:** Same as `[StatusItem]`.
- `[[StatusItem|string], ...]`: Create a status bar with one field for each StatusItem in the list (or tuple). The status bar fields are defined from left to right in the order specified. A string value is converted to: `StatusItem(name=string)`:

style: The default editor style of elements in the view.

title: str Title for the view, displayed in the title bar when the view appears as a secondary window (i.e., dialog or wizard). If not specified, "Edit properties" is used as the title.

width: int or float Requested width for the view window, as an (integer) number of pixels, or as a (floating point) fraction of the screen width.

x, y: int or float The requested x and y coordinates for the window (positive for top/left, negative for bottom/right, either pixels or proportions)

Command TraitsUI menus and toolbars are generally implemented in conjunction with custom *Handlers*; see *Menus and Menu Bars* for details.

buttons: List of button actions to add to the view. The `traitsui.menu` module defines standard buttons, such as **OKButton**, and standard sets of buttons, such as **ModalButtons**, which can be used to define a value for this attribute. This value can also be a list of button name strings, such as `['OK', 'Cancel', 'Help']`. If set to the empty list, the view contains a default set of buttons (equivalent to **LiveButtons**: Undo/Redo, Revert, OK, Cancel, Help). To suppress buttons in the view, use the **NoButtons** variable, defined in `traitsui.menu`.

close_result: What result should be returned if the user clicks the window or dialog close button or icon?

handler: The Handler object that provides GUI logic for handling events in the window. Set this attribute only if you are using a custom Handler. If not set, the default Traits UI Handler is used.

key_bindings: The set of global key bindings for the view. Each time a key is pressed while the view has keyboard focus, the key is checked to see if it is one of the keys recognized by the KeyBindings object.

menubar: The menu bar for the view. Usually requires a custom **handler**.

model_view: The factory function for converting a model into a model/view object.

on_apply: Called when modal changes are applied or reverted.

toolbar: The toolbar for the view. Usually requires a custom **handler**.

updated: Event Event when the view has been updated.

Content The **imports** and **drop_class** attributes control what objects can be dragged and dropped on the view.

content: The top-level Group object for the view.

drop_class: Class of dropped objects that can be added.

export: The category of exported elements.

imports: The valid categories of imported elements.

object: The default object being edited.

User help

help: (deprecated) The **help** attribute is a deprecated way to specify that the View has a Help button. Use the **buttons** attribute instead (see *Command Buttons: the buttons Attribute* for details).

help_id: The **help_id** attribute is not used by Traits, but can be used by a custom help handler.

Unique

id: The **id** attribute is used as a key to save user preferences about a view, such as customized size and position, so that they are restored the next time the view is opened. The value of **id** must be unique across all Traits-based applications on a system. If no value is specified, no user preferences are saved for the view.

1.5 Advanced View Concepts

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

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

1.5.1 Internal Views

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

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

Defining a Default View

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

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

```
# default_traits_view.py -- Sample code to demonstrate the use of
#                               'traits_view'
from traits.api import HasTraits, Str, Int
from traitsui.api import View, Item, Group
import traitsui

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

    employee_number = Str
    salary = Int

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

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

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



Fig. 1.5: Figure 5: User interface for Example 5

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

Sometimes, it is necessary to build a view based on the state of the object when it is being built. In such cases, defining the view statically is limiting, so one can override the `default_traits_view()` method of a HasTraits object. The example above would be implemented as follows:

Example 5b: Building a default View object with `default_traits_view()`

```
# default_traits_view2.py -- Sample code to demonstrate the use of
#                               'default_traits_view'
from traits.api import HasTraits, Str, Int
from traitsui.api import View, Item, Group
import traitsui
```

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

    employee_number = Str
    salary = Int

    def default_traits_view(self):
        return View(Group(Item(name = 'first_name'),
                           Item(name = 'last_name'),
                           Item(name = 'department'),
                           label = 'Personnel profile',
                           show_border = True))

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

This pattern can be useful for situations where the layout of GUI elements depends on the state of the object. For instance, to populate the values of a *CheckListEditor()* with items read in from a file, it would be useful to build the default view this way.

Defining Multiple Views Within the Model

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

Example 6: Defining multiple View objects in a HasTraits class

```
# multiple_views.py -- Sample code to demonstrate the use of
#                               multiple views
from traits.api import HasTraits, Str, Int
from traitsui.api import View, Item, Group
import traitsui

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))

sam = SimpleEmployee3()
sam.configure_traits()
sam.configure_traits(view='all_view')

```

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

```
configure_traits(view='all_view').
```

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

1.5.2 Separating Model and View: External Views

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

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

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

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

Example 2 through *Example 4* demonstrate external Views defined as variables. One advantage of this convention is that the variable name provides an easily accessible “handle” for re-using the View. This technique does not, however, support visual inheritance.

A powerful alternative is to define a View within the *controller* (Handler) class that controls the window for that View.⁸ This technique is described in *Controlling the Interface: the Handler*.

1.5.3 Displaying a View

TraitsUI 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

⁷ Note that although the definition of a View within a HasTraits class has the syntax of a trait attribute definition, the resulting View is not stored as an attribute of the class.

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

of Handler and its subclasses. The third method, `ui()`, is defined on the View class itself.

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 TraitsUI programmer.

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

edit_traits()

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

ui()

The View object includes a method called `ui()`, which performs the actual generation of the window or panel from the View for both `edit_traits()` and `configure_traits()`. The `ui()` method is also available directly through the TraitsUI API; however, using one of the other two methods is usually preferable.⁹

The `ui()` method has five keyword parameters:

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

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

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

1.5.4 The View Context

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

When the `ui()` method is called from `configure_traits()` or `edit_traits()` on a HasTraits object, the relevant object is the HasTraits object whose method was called. For this reason, you do not need to specify the *context* argument in most calls to `configure_traits()` or `edit_traits()`. However, when you call the `ui()` method on a View object, you *must* specify

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

the *context* parameter, so that the `ui()` method receives references to the objects whose trait attributes you want to modify.

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

Multi-Object Views

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

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

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

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

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

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

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

```
pool=True,
price=200000)

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

The resulting window has the desired appearance:¹⁰



Fig. 1.6: Figure 6: User interface for Example 7

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

Notice that the Item definitions in Example 7 use the same type of extended trait attribute syntax as is supported for the `on_trait_change()` dynamic trait change notification method. In fact, Item **name** attributes can reference any trait attribute that is reachable from an object in the context. This is true regardless of whether the context contains a single object or multiple objects. For example:

```
Item('object.axle.chassis.serial_number')
```

where “*object*” is the literal name which refers to the top-level object being viewed. (Note that “*object*” is **not** some user-defined attribute name like “*axle*” in this example.) More precisely, “*object*” is the default name, in the view’s *context* dictionary, of this top-level viewed object (see [Advanced View Concepts](#)).

Because an Item can refer only to a single trait, do not use extended trait references that refer to multiple traits, since the behavior of such references is not defined. Also, avoid extended trait references where one of the intermediate objects could be None, because there is no way to obtain a valid reference from None.

Refer to the [Traits User Manual](#), in the chapter on trait notification, for details of the extended trait name syntax.

1.5.5 Include Objects

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

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

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

Example 8: Using an Include object

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

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

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

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

1.6 Controlling the Interface: the Handler

Most of the material in the preceding chapters is concerned with the relationship between the model and view aspects of the *MVC* design pattern as supported by TraitsUI. This chapter examines the third aspect: the *controller*, implemented in TraitsUI as an *instance* of the *Handler* class.¹¹

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

In the preceding examples in this guide, the Handler object has been implicit: TraitsUI 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 TraitsUI window buttons (see *Command Buttons: the buttons Attribute*).

This chapter explains the features of the TraitsUI 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.

1.6.1 Backstage: Introducing the UIInfo Object

TraitsUI supports the MVC design pattern by maintaining the model, view, and controller as separate entities. A single View object can be used to construct windows for multiple model objects; likewise a single Handler can handle GUI events for windows created using different Views. Thus there is no static link between a Handler and any particular window or model object. However, in order to be useful, a Handler must be able to observe and manipulate both its corresponding window and model objects. In TraitsUI, this is accomplished by means of the UIInfo object.

Whenever TraitsUI creates a window or panel from a View, a UIInfo object is created to act as the Handler's reference to that window and to the objects whose *trait attributes* are displayed in it. Each entry in the View's context (see *The*

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

View Context) becomes an attribute of the UIInfo object.¹² For example, the UIInfo object created in *Example 7* has attributes **h1** and **h2** whose values are the objects **house1** and **house2** respectively. In *Example 1* through *Example 6*, the created UIInfo object has an attribute **object** whose value is the object **sam**.

Whenever a window event causes a Handler method to be called, TraitsUI passes the corresponding UIInfo object as one of the method arguments. This gives the Handler the information necessary to perform its tasks.

1.6.2 Assigning Handlers to Views

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

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

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. The result of this technique, as shown in *Example 9*, is that the window created by the View object is automatically controlled by the specified handler instance.

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 TraitsUI 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 handler instance control *only of the specific window being created by the method call*. This assignment overrides the View's **handler** attribute.

Creating a Default View Within a Handler

You seldom need to associate a single custom Handler with several different Views or vice versa, although you can in theory and there are cases where it is useful to be able to do so. In most real-life scenarios, a custom Handler is tailored to a particular View with which it is always used. One way to reflect this usage in the program design is to define the View as part of the Handler. The same rules apply as for defining Views within `HasTraits` objects; for example, a view named `'trait_view'` is used as the default view.

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

1.6.3 Handler Subclasses

TraitsUI provides two Handler subclasses: `ModelView` and `Controller`. Both of these classes are designed to simplify the process of creating an MVC-based application.

¹² Other attributes of the UIInfo object include a UI object and any *trait editors* contained in the window (see *Introduction to Trait Editor Factories* and *The Predefined Trait Editor Factories*).

Both `ModelView` and `Controller` extend the `Handler` class by adding the following trait attributes:

- **model**: The model object for which this handler defines a view and controller.
- **info**: The `UIInfo` object associated with the actual user interface window or panel for the model object.

The **model** attribute provides convenient access to the model object associated with either subclass. Normally, the **model** attribute is set in the constructor when an instance of `ModelView` or `Controller` is created.

The **info** attribute provides convenient access to the `UIInfo` object associated with the active user interface view for the handler object. The **info** attribute is automatically set when the handler object's view is created.

Both classes' constructors accept an optional *model* parameter, which is the model object. They also can accept metadata as keyword parameters.

```
class ModelView ([model = None, **metadata ])
```

```
class Controller ([model = None, **metadata ])
```

The difference between the `ModelView` and `Controller` classes lies in the context dictionary that each one passes to its associated user interface, as described in the following sections.

Controller Class

The `Controller` class is normally used when implementing a standard MVC-based design, and plays the “controller” role in the MVC design pattern. The “model” role is played by the object referenced by the `Controller`'s **model** attribute; and the “view” role is played by the `View` object associated with the model object.

The context dictionary that a `Controller` object passes to the `View`'s `ui()` method contains the following entries:

- **object**: The `Controller`'s model object.
- **controller**: The `Controller` object itself.

Using a `Controller` as the handler class assumes that the model object contains most, if not all, of the data to be viewed. Therefore, the model object is used for the object key in the context dictionary, so that its attributes can be easily referenced with unqualified names (such as `Item('name')`).

ModelView Class

The `ModelView` class is useful when creating a variant of the standard MVC design pattern. In this variant, the `ModelView` subclass reformulates a number of trait attributes on its model object as properties on the `ModelView`, usually to convert the model's data into a format that is more suited to a user interface.

The context dictionary that a `ModelView` object passes to the `View`'s `ui()` method contains the following entries:

- **object**: The `ModelView` object itself.
- **model**: The `ModelView`'s model object.

In effect, the `ModelView` object substitutes itself for the model object in relation to the `View` object, serving both the “controller” role and the “model” role (as a set of properties wrapped around the original model). Because the `ModelView` object is passed as the context's object, its attributes can be referenced by unqualified names in the `View` definition.

1.6.4 Writing Handler Methods

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

Overriding Standard Methods

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

1. A UIInfo object is created
2. The Handler's `init_info()` method is called. Override this method if the handler needs access to viewable traits on the UIInfo object whose values are properties that depend on items in the context being edited.
3. The UI object is created, and generates the actual window.
4. The `init()` method is called. Override this method if you need to initialize or customize the window.
5. The `position()` method is called. Override this method to modify the position of the window (if setting the x and y attributes of the View is insufficient).
6. The window is displayed.

When Handler methods are called, and when to override them

| Method | Called When | Override When? |
|---|--|---|
| <code>apply(info)</code> | The user clicks the <i>Apply</i> button, and after the changes have been applied to the context objects. | To perform additional processing at this point. |
| <code>close(info, is_ok)</code> | The user requests to close the window, clicking <i>OK</i> , <i>Cancel</i> , or the window close button, menu, or icon. | To perform additional checks before destroying the window. |
| <code>closed(info, is_ok)</code> | The window has been destroyed. | To perform additional clean-up tasks. |
| <code>revert(info)</code> | The user clicks the <i>Revert</i> button, or clicks <i>Cancel</i> in a live window. | To perform additional processing. |
| <code>setattr(info, object, name, value)</code> | The user changes a trait attribute value through the user interface. | To perform additional processing, such as keeping a change history. Make sure that the overriding method actually sets the attribute. |
| <code>show_help(info, control=None)</code> | The user clicks the <i>Help</i> button. | To call a custom help handler in addition to or instead of the global help handler, for this window. |
| <code>perform(info, action, event)</code> | The user clicks a button or toolbar item, or selects a menu item. | To change the way that actions are handled, eg. to pass more info to a method. |

Reacting to Trait Changes

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

`extended_traitname_changed` (*info*)

This method is called whenever a change is made to the attribute specified by *extended_traitname* in the **context** of the View used to create the window (see *Multi-Object Views*), where the dots in the extended trait reference have been replaced by underscores. For example, for a method to handle changes on the **salary** attribute of the object whose context key is 'object' (the default object), the method name should be `object_salary_changed()`.

By contrast, a subclass of Handler for *Example 7* might include a method called `h2_price_changed()` to be called whenever the price of the second house is edited.

Note: These methods are called on window creation.

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

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

```
def object_foo_changed(self, info):

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

    #code to be executed in either case
```

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

Example 9: Using a Handler that reacts to trait changes

```
# handler_override.py -- Example of a Handler that overrides
#                          setattr(), and that has a user interface
#                          notification method

from traits.api import HasTraits, Bool
from traitsui.api import View, Handler

class TC_Handler(Handler):

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

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

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

view1 = View('b1', 'b2', 'b3',
             title="Alter Title",
```

```

        handler=TC_Handler(),
        buttons = ['OK', 'Cancel'])

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

```

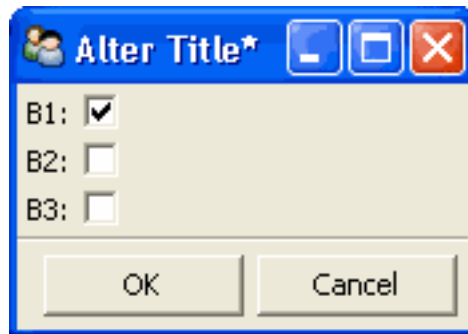


Fig. 1.7: Figure 7: Before and after views of Example 9

Implementing Custom Window Commands

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

Actions

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

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

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

```

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

```


Custom Command Buttons

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

1. Define the handler method and action, as described in [Actions](#).
2. Include the new Action in the **buttons** attribute for the View:

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

Menus and Menu Bars

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

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

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

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

Toolbars

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

1. Define the handler method and the action, as in [Actions](#), including a tooltip and an image to display on the toolbar. The image must be a Pyface ImageResource instance; if a path to the image file is not specified, it is assumed to be in an images subdirectory of the directory where ImageResource is used:

```
From pyface.api import ImageResource

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

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

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

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

1.7 Introduction to Trait Editor Factories

The preceding code samples in this User Manual 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 TraitsUI: 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 TraitsUI 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 TraitsUI and with UI programming in general can harness the full power and flexibility of the underlying GUI toolkit from within TraitsUI.

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

Consider the following script and the window it creates:

Example 12: Using default trait editors

```
# default_trait_editors.py -- Example of using default
#                             trait editors

from traits.api import HasTraits, Str, Range, Bool
from traitsui.api import View, Item

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

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

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

alice.configure_traits()
```

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

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

- Explicitly specifying an alternate trait editor factory



Fig. 1.8: Figure 12: User interface for Example 12

- Specifying an alternate style for the editor generated by the factory

The remainder of this chapter examines these alternatives more closely.

1.7.1 Specifying an Alternate Trait Editor Factory

As of this writing the TraitsUI package includes a wide variety of predefined trait editor factories, which are described in *Basic Trait Editor Factories* and *Advanced Trait Editors*. Some additional editor factories are specific to the wxWidgets toolkit and are defined in one of the following packages:

- traitsui.wx
- traitsui.wx.extra
- traitsui.wx.extra.windows (specific to Microsoft Windows)

These editor factories are described in *“Extra” Trait Editor Factories*.

For a current complete list of editor factories, refer to the *Traits API Reference*.

Other packages can define their own editor factories for their own traits. For example, `enthought.kiva.api.KivaFont` uses a `KivaFontEditor()` and `enthought.enable2.traits.api.RGBAColor` uses an `RGBAColorEditor()`.

For most *predefined trait types* (see *Traits User Manual*), there is exactly one predefined trait editor factory suitable for displaying it: the editor factory that is assigned as its default.¹⁵ There are exceptions, however; for example, a `Str` trait defaults to using a `TextEditor`, but can also use a `CodeEditor` or an `HTMLEditor`. A `List` trait can be edited by means of `ListEditor`, `TableEditor` (if the `List` elements are `HasTraits` objects), `CheckListEditor` or `SetEditor`. Furthermore, the TraitsUI package includes tools for building additional trait editors and factories for them as needed.

To use an alternate editor factory for a trait in a TraitsUI window, you must specify it in the View for that window. This is done at the Item level, using the `editor` keyword parameter. The syntax of the specification is `editor = editor_factory()`. (Use the same syntax for specifying that the default editor should be used, but with certain keyword parameters explicitly specified; see *Initializing Editors*).

For example, to display a `Str` trait called `my_string` using the default editor factory (`TextEditor()`), the View might contain the following Item:

```
Item(name='my_string')
```

The resulting widget would have the following appearance:

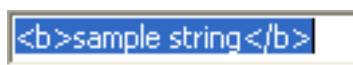


Fig. 1.9: Figure 13: Default editor for a Str trait

¹⁵ Appendix II contains a table of the predefined trait types in the Traits package and their default trait editor types.

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

```
Item( name='my_string', editor=HTMLEditor() )
```

The resulting widget appears as in Figure 14:



Fig. 1.10: Figure 14: Editor generated by HTMLEditor()

Note: TraitsUI does not check editors for appropriateness.

TraitsUI 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 you from trying to, say, display a Float trait using ColorEditor(). The results of such a mismatch are unlikely to be helpful, and can even crash the application; it is up to the programmer to choose an editor sensibly. *The Predefined Trait Editor Factories* is a useful reference for selecting an appropriate editor for a given task.

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

- You can specify a trait editor when you define a trait, by passing the result of a trait editor factory as the *editor* keyword parameter of the callable that creates the trait. However, this approach commingles the *view* of a trait with its *model*.
- You can specify the **editor** attribute of a TraitHandler object. This approach commingles the *view* of a trait with its *controller*.

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

Initializing Editors

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

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

The descriptions of trait editor factories in *The Predefined Trait Editor Factories* include a list of required and optional initialization keywords for each editor.

1.7.2 Specifying an Editor Style

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

The ‘simple’ Style

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

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



Fig. 1.11: Figure 15: Simple style of enumeration editor

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

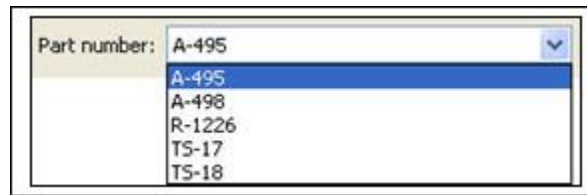


Fig. 1.12: Figure 16: Simple enumeration editor with expanded list

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

The ‘custom’ Style

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

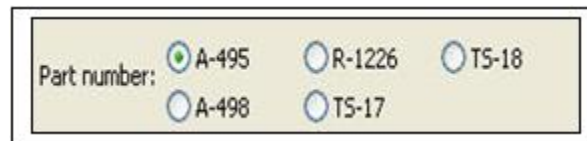


Fig. 1.13: Figure 17: Custom style of enumeration editor

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

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

The ‘text’ Style

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

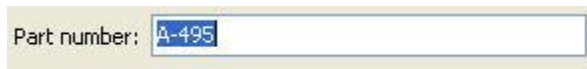


Fig. 1.14: Figure 18: Text style of enumeration editor

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

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

1. The function specified in the **format_func** attribute of the Item (see *The Item Object*), if any, is called on the attribute value.
2. Otherwise, the function specified in the *format_func* parameter of the trait editor factory, if any, is called on the attribute value.
3. Otherwise, the Python-style formatting string specified in the **format_str** attribute of the Item (see *The Item Object*), if any, is used to format the attribute value.
4. The Python-style formatting string specified in the *format_str* parameter of the trait editor factory, if any, is used to format the attribute value.
5. Otherwise, the Python `str()` function is called on the attribute value.

The ‘readonly’ style

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

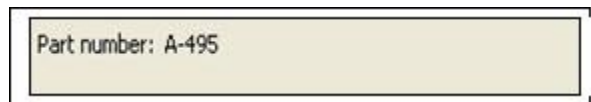


Fig. 1.15: Figure 19: Read-only style of enumeration editor

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

Using Editor Styles

As discussed in *Contents of a View* and *Customizing a View*, the Item, Group and View objects of TraitsUI all have a **style** attribute. The style of editor used to display the Items in a View is determined as follows:

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

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

Example 13: Using editor styles at various levels

```
# mixed_styles.py -- Example of using editor styles at
#                      various levels

from traits.api import HasTraits, Str, Enum
from traitsui.api import View, Group, Item

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

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

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

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

Notice how the editor styles are set for each attribute:

- **position_type** at the Item level (lines 19-20)
- **department** at the Group level (lines 18 and 21)
- **first_name** and **last_name** at the View level (lines 16, 17, and 23)

The resulting window demonstrates these precedence rules:

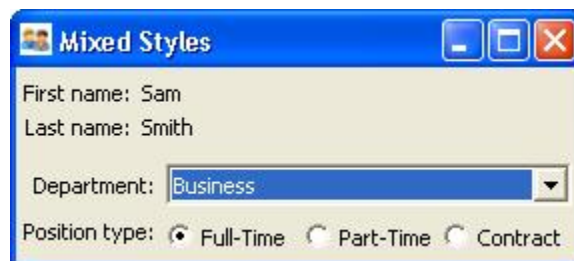


Fig. 1.16: Figure 20: User interface for Example 13

1.8 The Predefined Trait Editor Factories

This chapter contains individual descriptions of the predefined trait editor factories provided by TraitsUI. Most of these editor factories are straightforward and can be used easily with little or no expertise on the part of the programmer or end user; these are described in *Basic Trait Editor Factories*. The section *Advanced Trait Editors* covers a smaller set of specialized editors that have more complex interfaces or that are designed to be used along with complex editors.

Note: Examples are toolkit-specific.

The exact appearance of the editors depends on the underlying GUI toolkit. The screenshots and descriptions in this chapter are based on wxWindows. Another supported GUI toolkit is Qt, from TrollTech.

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

1.8.1 Basic Trait Editor Factories

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

ArrayEditor()

Suitable for 2-D Array, 2-D CArray

Default for Array, CArray (if 2-D)

Optional parameter *width*

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

The following code generates the editors shown in Figure 21.

Example 14: Demonstration of array editors

```
# array_editor.py -- Example of using array editors

import numpy as np
from traits.api import HasPrivateTraits, Array
from traitsui.api import \
    import View, ArrayEditor, Item
from traitsui.menu import NoButtons

class ArrayEditorTest ( HasPrivateTraits ):

    three = Array(np.int, (3,3))
    four  = Array(np.float,
                  (4,4),
                  editor = ArrayEditor(width = -50))

    view = View( Item('three', label='3x3 Integer'),
                  '_')
```


| | | | |
|--------------------|----------------------------------|----------------------------------|----------------------------------|
| | <input type="text" value="1"/> | <input type="text" value="0"/> | <input type="text" value="0"/> |
| 3x3 Integer: | <input type="text" value="0"/> | <input type="text" value="0"/> | <input type="text" value="0"/> |
| | <input type="text" value="0"/> | <input type="text" value="0"/> | <input type="text" value="0"/> |
| <hr/> | | | |
| | 0 | 0 | 0 |
| Integer Read-only: | 0 | 0 | 0 |
| | 0 | 0 | 0 |
| <hr/> | | | |
| | <input type="text" value="0.0"/> | <input type="text" value="0.0"/> | <input type="text" value="0.0"/> |
| 4x4 Float: | <input type="text" value="0.0"/> | <input type="text" value="0.0"/> | <input type="text" value="0.0"/> |
| | <input type="text" value="0.0"/> | <input type="text" value="0.0"/> | <input type="text" value="0.0"/> |
| | <input type="text" value="0.0"/> | <input type="text" value="0.0"/> | <input type="text" value="0.0"/> |
| <hr/> | | | |
| | 0.0 | 0.0 | 0.0 |
| Float Read-only: | 0.0 | 0.0 | 0.0 |
| | 0.0 | 0.0 | 0.0 |
| | 0.0 | 0.0 | 0.0 |

Fig. 1.17: Figure 21: Array editors

```

        Item('three',
            label='Integer Read-only',
            style='readonly'),
        '_',
        Item('four', label='4x4 Float'),
        '_',
        Item('four',
            label='Float Read-only',
            style='readonly'),
        buttons = NoButtons,
        resizable = True )

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

```

BooleanEditor()

Suitable for Bool, CBool

Default for Bool, CBool

Optional parameters *mapping*

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

- 'True'
- T
- Yes
- Y
- 'False'
- F
- No
- n

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

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

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

Figure 22 shows the four styles generated by BooleanEditor().

ButtonEditor()

Suitable for Button, Event, ToolbarButton

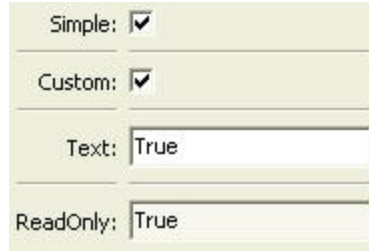


Fig. 1.18: Figure 22: Boolean editor styles

Default for Button, ToolbarButton

Optional parameters *height_padding, image, label, label_value, orientation, style, value, values_trait, view, width_padding*

The ButtonEditor() factory is designed to be used with an Event or Button¹⁶ trait. When a user clicks a button editor, the associated event is fired. Because events are not printable objects, the text and read-only styles are not implemented for this editor. The simple and custom styles of this editor are identical.

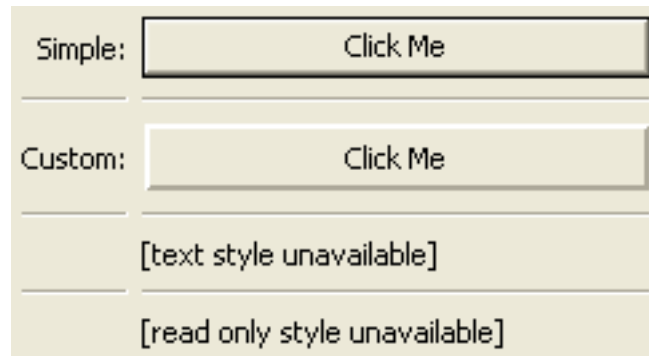


Fig. 1.19: Figure 23: Button editor styles

By default, the label of the button is the name of the Button or Event trait to which it is linked.¹⁷ However, this label can be set to any string by specifying the *label* parameter of ButtonEditor() as that string.

Alternatively, use *label_value* to specify the name of the trait to use as the button label.

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

Use *values_trait* to specify the name of the trait on the object that contains the list of possible values. If this is set, then the *value*, *label*, and *label_value* traits are ignored; instead, they will be set from this list. When this button is clicked, the value set will be the one selected from the drop-down.

CheckListEditor()

Suitable for List

Default for (none)

Optional parameters *cols, name, values*

¹⁶ In Traits, a Button and an Event are essentially the same thing, except that Buttons are automatically associated with button editors.

¹⁷ TraitsUI 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 *The View Object*).

The editors generated by the `CheckListEditor()` factory are designed to enable the user to edit a List trait by selecting elements from a “master list”, i.e., a list of possible values. The list of values can be supplied by the trait being edited, or by the *values* parameter.

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

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

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

Alternatively, you can use the *name* parameter to specify a trait attribute containing the label strings for the values.

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

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

The four styles generated by `CheckListEditor()` are shown in Figure 24. Note that in this case the *cols* parameter has been set to 4.

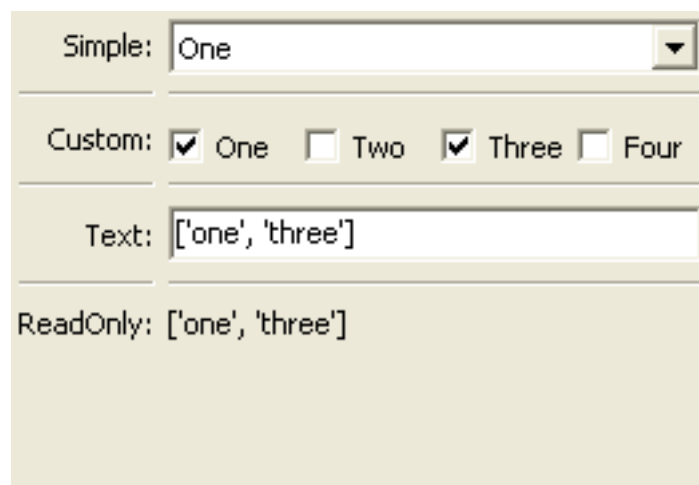


Fig. 1.20: Figure 24: Checklist editor styles

CodeEditor()

Suitable for Code, Str, String

Default for Code

Optional parameters *auto_set*

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

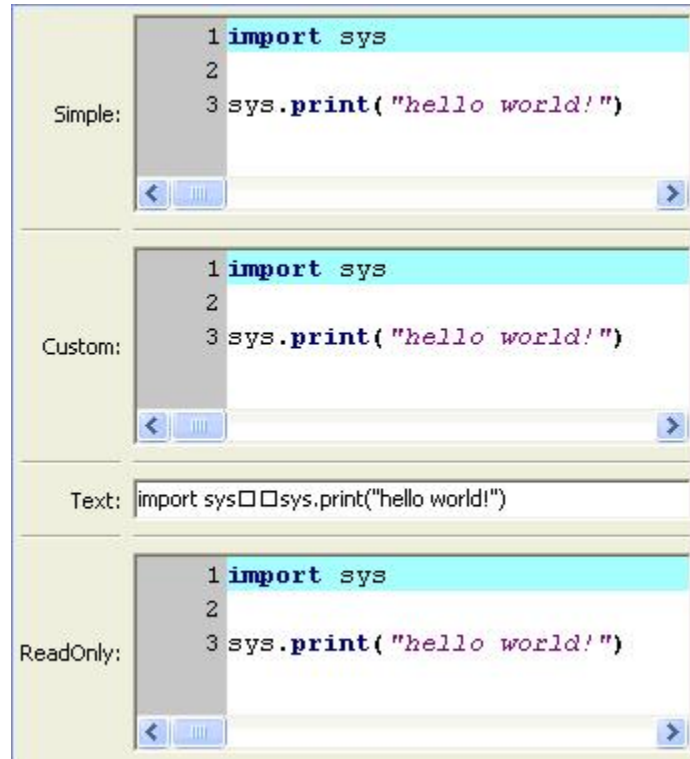


Fig. 1.21: Figure 25: Code editor styles

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

ColorEditor()

Suitable for Color

Default for Color

Optional parameters *mapped*

The editors generated by `ColorEditor()` are designed to enable the user to display a Color trait or edit it by selecting a color from the palette available in the underlying GUI toolkit. The four styles of color editor are shown in Figure 26.

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

The custom style includes a labeled color swatch on the left, representing the current value of the Color trait, and a palette of common color choices on the right. Clicking on any tile of the palette changes the color selection, causing the swatch to update accordingly. Clicking on the swatch itself causes a more detailed, platform-specific interface to appear in a dialog box, such as is shown in Figure 27.

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



Fig. 1.22: Figure 26: Color editor styles

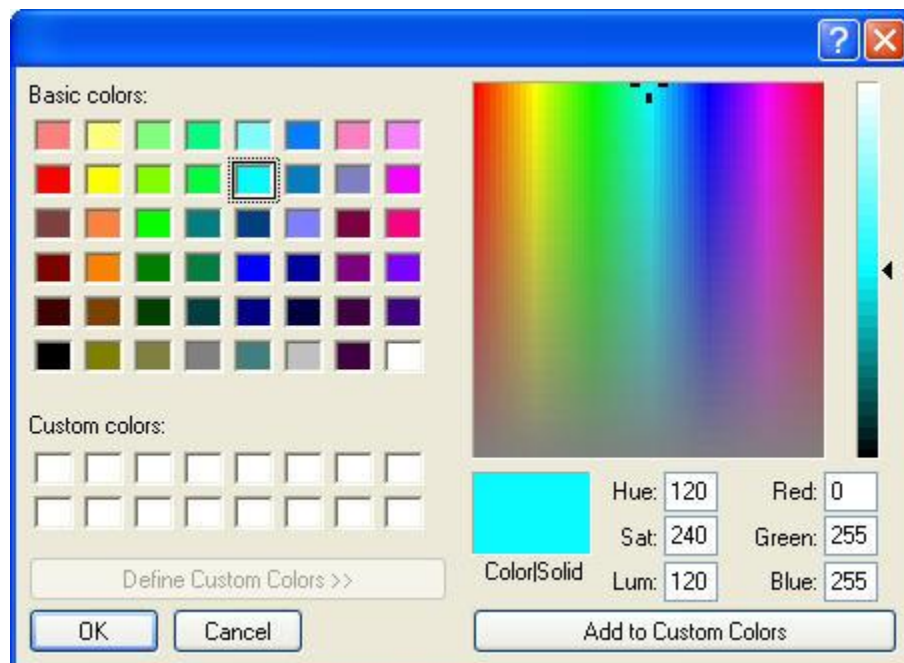


Fig. 1.23: Figure 27: Custom color selection dialog box for Microsoft Windows XP

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

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

CompoundEditor()

Suitable for special

Default for “compound” traits

Optional parameters *auto_set*

An editor generated by `CompoundEditor()` consists of a combination of the editors for trait types that compose the compound trait. The widgets for the compound editor are of the style specified for the compound editor (simple, custom, etc.). The editors shown in Figure 28 are for the following trait, whose value can be an integer between 1 and 6, or any of the letters 'a' through 'f':

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

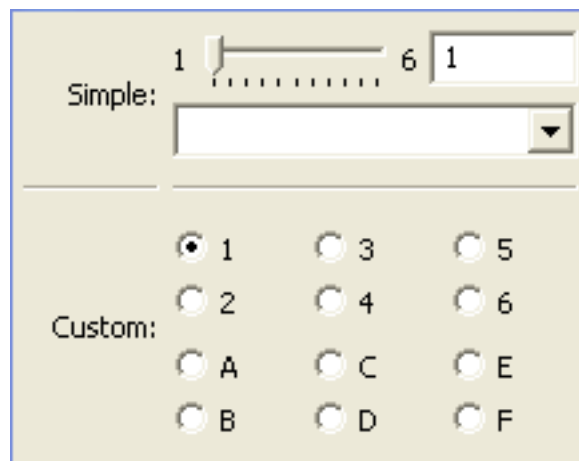


Fig. 1.24: Figure 28: Example compound editor styles

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

CSVListEditor()

Suitable for lists of simple data types

Default for none

Optional parameters *auto_set*, *enter_set*, *ignore_trailing_sep*, *sep*

This editor provides a line of text for editing a list of certain simple data types. The following List traits can be edited by a `CSVListEditor`:

- List(Int)
- List(Float)
- List(Str)
- List(Enum('string1', 'string2', etc))
- List(Range(low= *low value or trait name*, high= *high value or trait name*))

The 'text', 'simple' and 'custom' styles are all the same. They provide a single line of text in which the user can enter the list. The 'readonly' style provides a line of text that can not be edited by the user.

The default separator of items in the list is a comma. This can be overridden with the *sep* keyword parameter.

Parameters

auto_set [bool] If *auto_set* is True, each key pressed by the user triggers validation of the input, and if it is valid, the value of the object being edited is updated. *Default:* True

enter_set [bool] If *enter_set* is True, the input is updated when the user presses the *Enter* key. *Default:* False

sep [str or None] The separator of the list item in the text field. If *sep* is None, each contiguous span of whitespace is a separator. (Note: After the text field is split at the occurrences of *sep*, leading and trailing whitespace is removed from each item before converting to the underlying data type.) *Default:* ',' (a comma)

ignore_trailing_sep [bool] If *ignore_trailing_sep* is True, the user may enter a trailing separator (e.g. '1, 2, 3,') and it will be ignored. If this is False, a trailing separator is an error. *Default:* True

See Also

ListEditor, TextEditor

DefaultOverride()

Suitable for (any)

Default for (none)

The DefaultOverride() is a factory that takes the trait's default editor and customizes it with the specified parameters. This is useful when a trait defines a default editor using some of its data, e.g. Range or Enum, and you want to tweak some of the other parameters without having recreate that data.

For example, the default editor for Range(low=0, high=1500) has '1500' as the upper label. To change it to 'Max' instead, use:

```
View(Item('my_range', editor=DefaultOverride(high_label='Max'))
```

DirectoryEditor()

Suitable for Directory

Default for Directory

Optional parameters *entries, filter, filter_name, reload_name, truncate_ext, dclick_name*

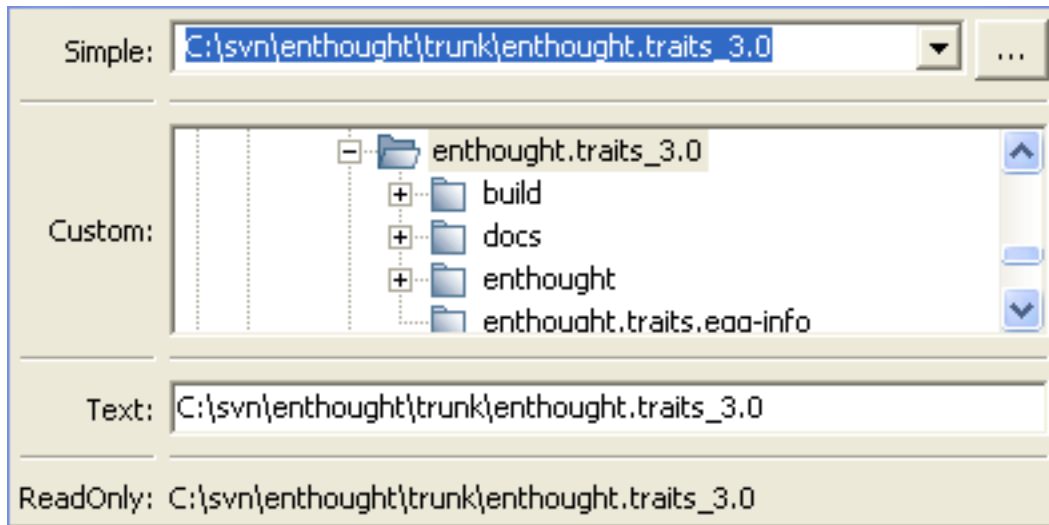


Fig. 1.25: Figure 29: Directory editor styles

A directory editor enables the user to display a Directory trait or set it to some directory in the local system hierarchy. The four styles of this editor are shown in Figure 29.

In the simple style, the current value of the trait is displayed in a combo box to the left of a button labeled ‘...’. The user can type a new path directly into the text box, select a previous value from the droplist of the combo box, or use the button to bring up a directory browser panel similar to the custom style of editor.

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

The user can also drag and drop a directory object onto the simple style editor.

The custom style displays a directory browser panel, in which the user can expand or collapse directory structures, and click a folder icon to select a directory.

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

The optional parameters are the same as the FileEditor.

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

EnumEditor()

Suitable for Enum, Any

Default for Enum

Required parameters for non-Enum traits: *values* or *name*

Optional parameters *cols*, *evaluate*, *mode*

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

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

The custom style is a set of radio buttons. Use the *cols* parameter to specify the number of columns of radio buttons.

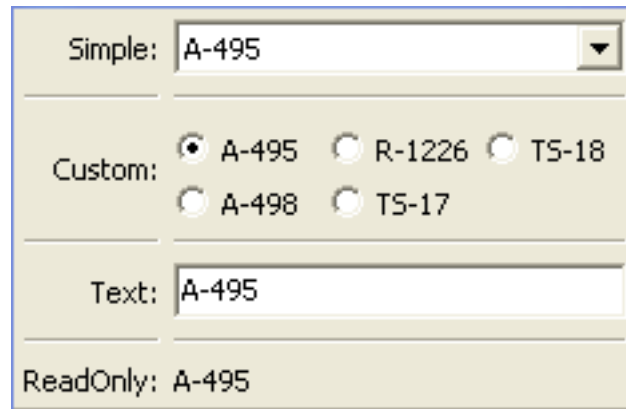


Fig. 1.26: Figure 30: Enumeration editor styles

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

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

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

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

Example 15: Enumeration editor with mapped values

```
# enum_editor.py -- Example of using an enumeration editor
from traits.api import HasTraits, Enum
from traitsui.api import EnumEditor

class EnumExample(HasTraits):
    priority = Enum('Medium', 'Highest',
                   'High',
                   'Medium',
                   'Low',
                   'Lowest')

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

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

FileEditor()

Suitable for File

Default for File

Optional parameters *entries, filter, filter_name, reload_name, truncate_ext, dclick_name*

A file editor enables the user to display a File trait or set it to some file in the local system hierarchy. The styles of this editor are shown in Figure 31.

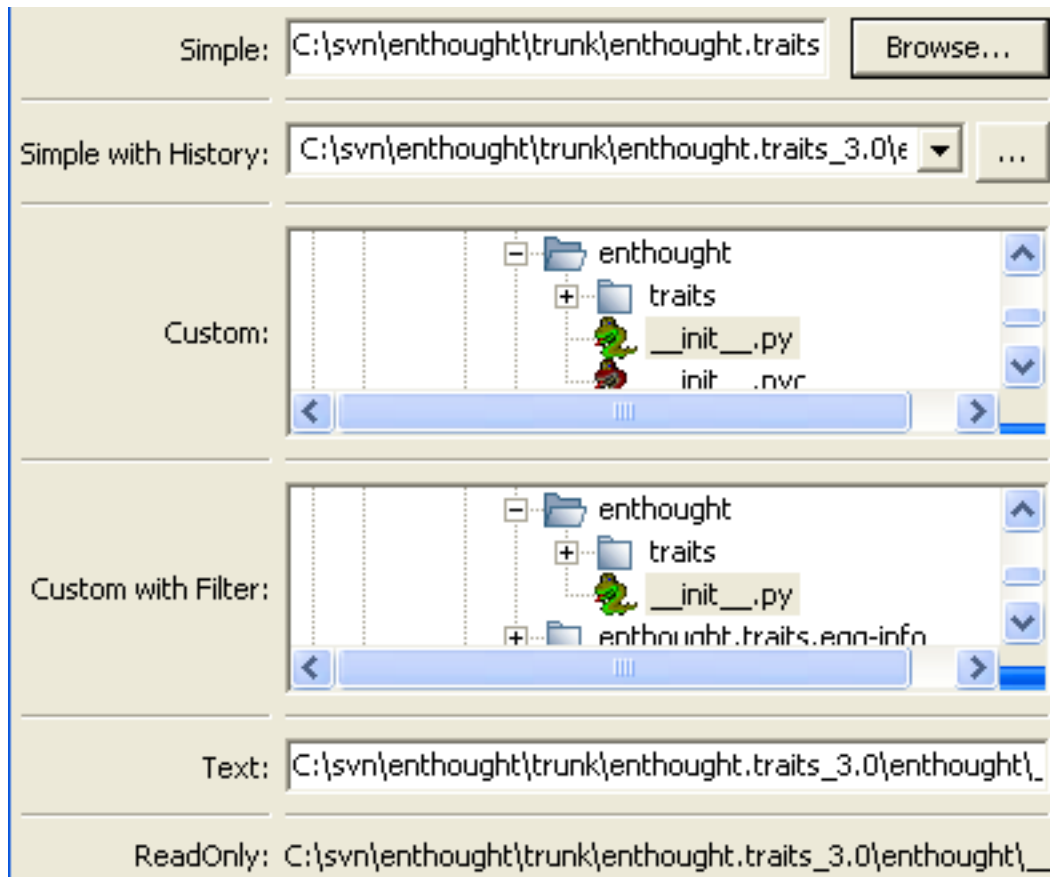


Fig. 1.27: Figure 31: File editor styles

The default version of the simple style displays a text box and a *Browse* button. Clicking *Browse* opens a platform-specific file selection dialog box. If you specify the *entries* keyword parameter with an integer value to the factory function, the simple style is a combo box and a button labeled The user can type a file path in the combo box, or select one of *entries* previous values. Clicking the ... button opens a browser panel similar to the custom style of editor. When the user selects a file in this browser, the panel collapses, and control is returned to the original editor widget, which is automatically populated with the new path string.

For either version of the simple style, the user can drag and drop a file object onto the control.

The custom style displays a file system browser panel, in which the user can expand or collapse directory structures, and click an icon to select a file.

You can specify a list of filters to apply to the file names displayed, using the *filter* keyword parameter of the factory function. In Figure 31, the “Custom with Filter” editor uses a *filter* value of `['*.py']` to display only Python source files. You can also specify this parameter for the simple style, and it will be used in the file selection dialog box or

pop-up file system browser panel. Alternatively, you can specify *filter_name*, whose value is an extended trait name of a trait attribute that contains the list of filters.

The *reload_name* parameter is an extended trait name of a trait attribute that is used to notify the editor when the view of the file system needs to be reloaded.

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

The *dclick_name* parameter is an extended trait name of a trait event which is fired when the user double-clicks on a file name when using the custom style.

FontEditor()

Suitable for Font

Default for Font

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

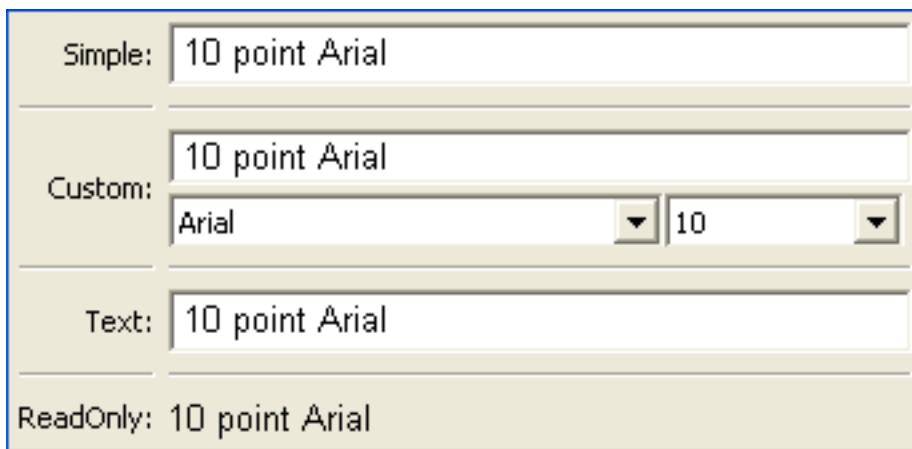


Fig. 1.28: Figure 32: Font editor styles

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

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

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

HTMLEditor()

Suitable for HTML, string traits

Default for HTML

Optional parameters *format_text*

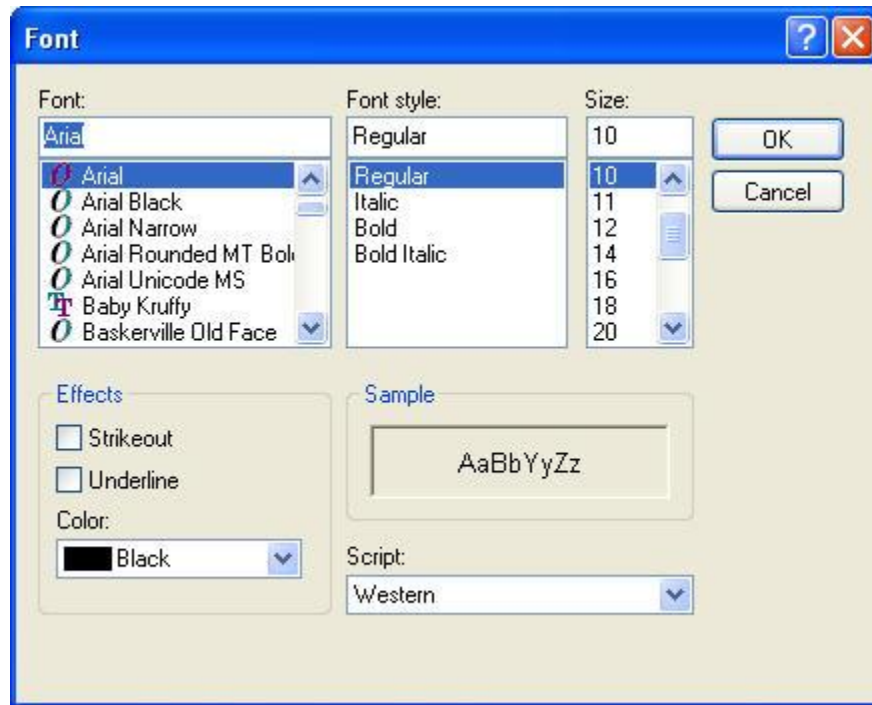


Fig. 1.29: Figure 33: Example font dialog box for Microsoft Windows

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

Note: HTML support is limited in the wxWidgets toolkit.

The set of tags supported by the wxWidgets implementation of the HTML editor is a subset of the HTML 3.2 standard. It does not support style sheets or complex formatting. Refer to the [wxWidgets documentation](#) for details.

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

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

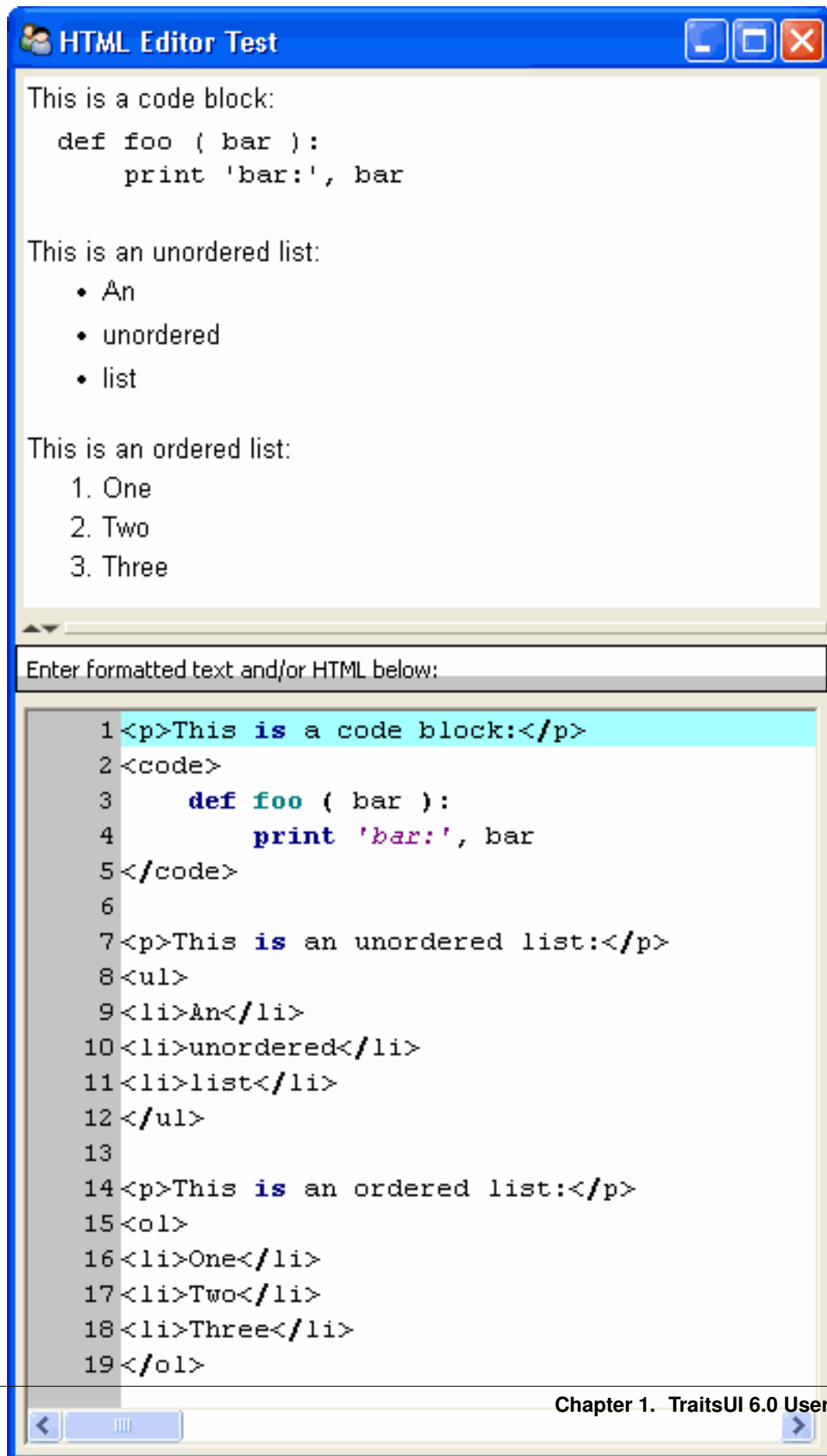
The following text produces the same displayed HTML as in Figure 34, when `format_text` is `True`:

This **is** a code block:

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

This **is** an unordered list:

- An
- unordered
- list



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

ImageEnumEditor()

Suitable for Enum, Any

Default for (none)

Required parameters for non-Enum traits: *values* or *name*

Optional parameters *path*, *klass* or *module*, *cols*, *evaluate*, *suffix*

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

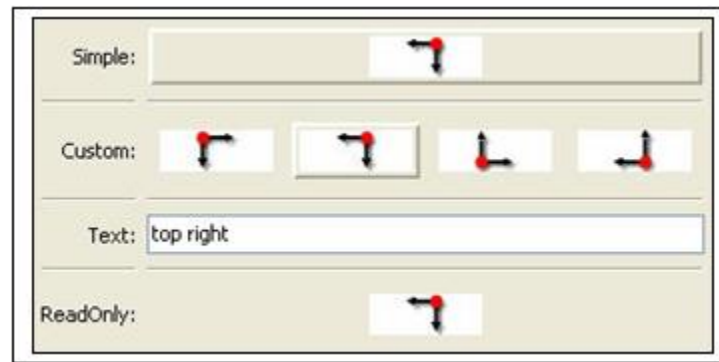


Fig. 1.31: Figure 35: Editor styles for image enumeration

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

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

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

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

The ImageEnumEditor() function accepts the same parameters as the EnumEditor() function (see [EnumEditor\(\)](#)), as well as some additional parameters.

Note: Image enumeration editors do not use ImageResource.

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

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

1. Only GIF (.gif) images are currently supported.

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

InstanceEditor()

Suitable for Instance, Property, self, ThisClass, This

Default for Instance, self, ThisClass, This

Optional parameters *cachable, editable, id, kind, label, name, object, orientation, values, view*

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

Editing a Single Instance

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

Fig. 1.32: Figure 36: Editor styles for instances

The custom style displays a user interface panel for editing the trait attributes of the instance. The simple style displays a button, which when clicked, opens a window containing a user interface for the instance. The *kind* parameter specifies the kind of window to open (see [Stand-alone Windows](#)). The *label* parameter specifies a label for the button in the simple interface. The *view* parameter specifies a view to use for the referenced instance's user interface; if this is not specified, the default view for the instance is used (see [Defining a Default View](#)).

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

Selecting Instances

You can add an option to select a different instance to edit. Use the *name* parameter to specify the extended name of a trait attribute in the context that contains a list of instances that can be selected or edited. (See [The View Context](#) for an explanation of contexts.) Using these parameters results in a drop-down list box containing a list of text representations of the available instances. If the instances have a **name** trait attribute, it is used for the string in the list; otherwise, a user-friendly version of the class name is used.

For example, the following code defines a Team class and a Person class. A Team has a roster of Persons, and a captain. In the view for a team, the user can pick a captain and edit that person's information. Example 16: Instance editor with instance selection

```
# instance_editor_selection.py -- Example of an instance editor
#                               with instance selection

from traits.api    \
    import HasStrictTraits, Int, Instance, List, Regex, Str
from traitsui.api \
    import View, Item, InstanceEditor

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

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

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

class Team ( HasStrictTraits ):

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

    traits_view = View( Item('name'),
```

```

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

if __name__ == '__main__':
    Team( name = 'Vultures',
        captain = people[0],
        roster = people ).configure_traits()

```

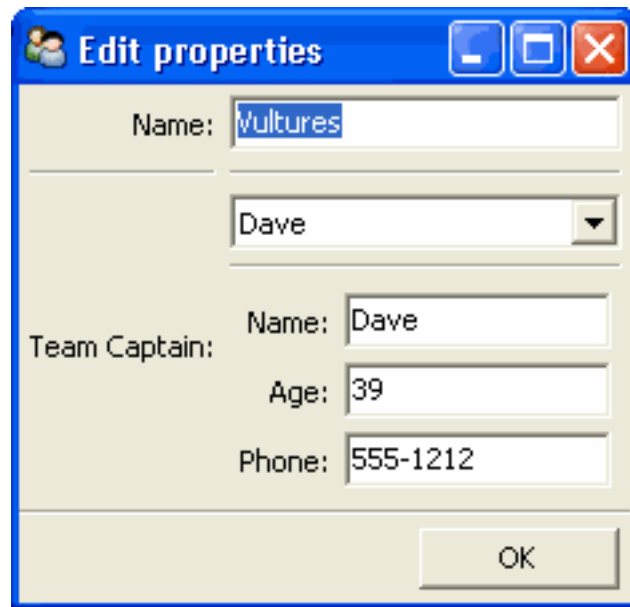


Fig. 1.33: Figure 37: User interface for Example 16

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

Allowing Instances

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

ListEditor()

Suitable for List

Default for List¹⁸

Optional parameters *editor, rows, style, trait_handler, use_notebook*

The following parameters are used only if *use_notebook* is True: *deletable, dock_style, export, page_name, select, view*

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

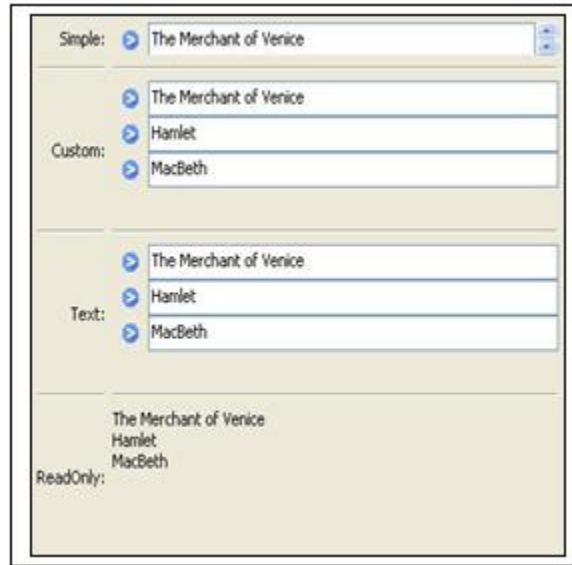


Fig. 1.34: Figure 38: List editor styles

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

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

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

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

ListStrEditor()

Suitable for ListStr or List of values mapped to strings

Default for (none)

¹⁸ If a List is made up of HasTraits objects, a table editor is used instead; see [TableEditor\(\)](#).

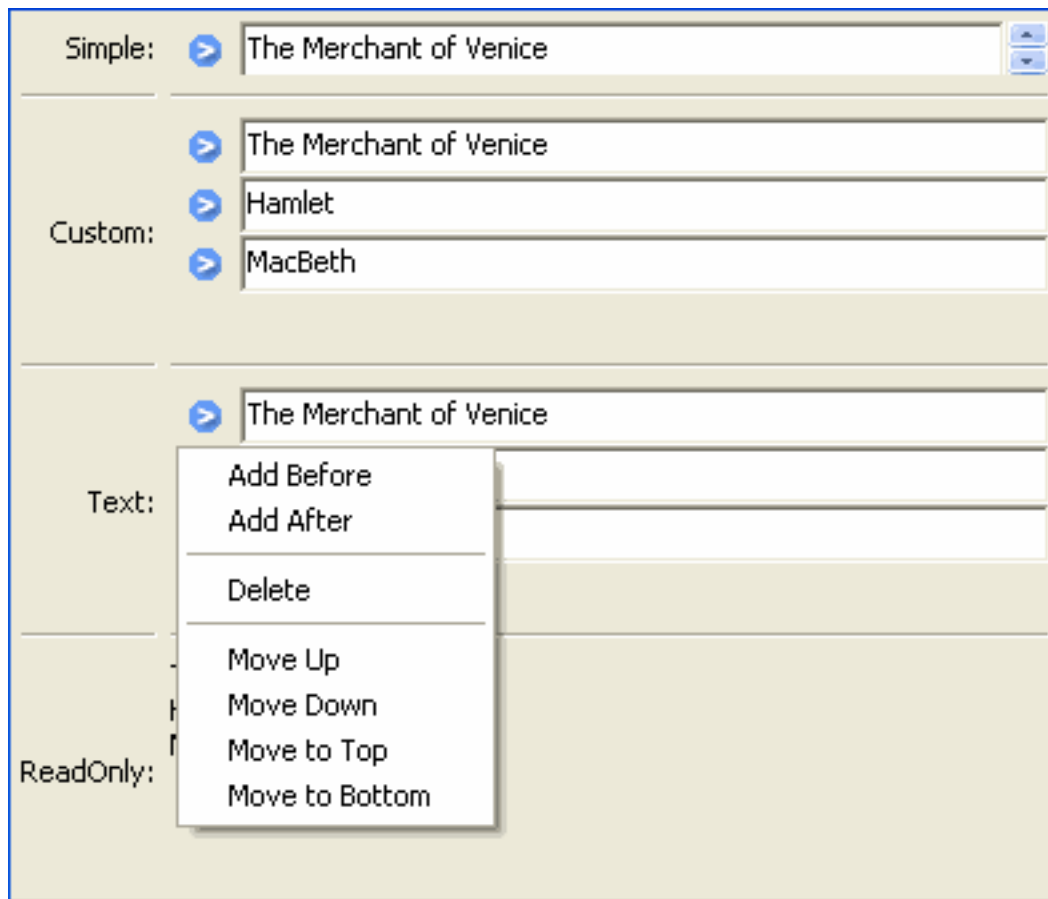


Fig. 1.35: Figure 39: List editor showing context menu



Fig. 1.36: Figure 40: Notebook list editor

Optional parameters *activated, activated_index, adapter, adapter_name, auto_add, drag_move, editable, horizontal_lines, images, multi_select, operations, right_clicked, right_clicked_index, selected, selected_index, title, title_name*

ListStrEditor() generates a list of selectable items corresponding to items in the underlying trait attribute. All styles of the editor are the same. The parameters to ListStrEditor() control aspects of the behavior of the editor, such as what operations it allows on list items, whether items are editable, and whether more than one can be selected at a time. You can also specify extended references for trait attributes to synchronize with user actions, such as the item that is currently selected, activated for editing, or right-clicked.



Fig. 1.37: Figure 41: List string editor

NullEditor()

Suitable for controlling layout

Default for (none)

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

RangeEditor()

Suitable for Range

Default for Range

Optional parameters *auto_set, cols, enter_set, format, high_label, high_name, label_width, low_label, low_name, mode*

The editors generated by RangeEditor() enable the user to specify numeric values within a range. The widgets used to display the range vary depending on both the numeric type and the size of the range, as described in Table 8 and shown in Figure 42. If one limit of the range is unspecified, then a text editor is used.

Table 8: Range editor widgets

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

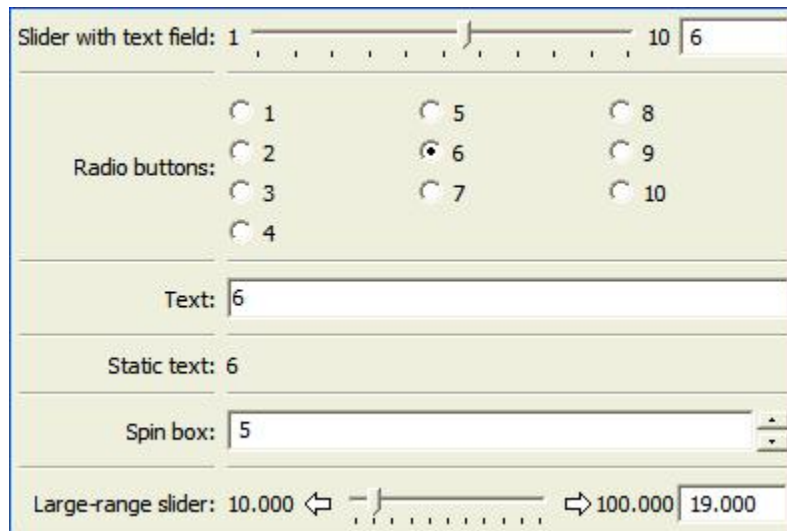


Fig. 1.38: Figure 42: Range editor widgets

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

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

- ‘auto’: The default widget, as described in Table 8
- ‘slider’: Simple slider with text field
- ‘xslider’: Large-range slider with text field
- ‘spinner’: Spin box with increment/decrement buttons
- ‘enum’: Radio buttons
- ‘text’: Text field

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

RGBColorEditor()

Suitable for RGBColor

Default for RGBColor

Optional parameters *mapped*

Editors generated by RGBColorEditor() are identical in appearance to those generated by ColorEditor(), but they are used for RGBColor traits. See [ColorEditor\(\)](#) for details.

SetEditor()

Suitable for List

Default for (none)

Required parameters Either *values* or *name*

Optional parameters *can_move_all, left_column_title, object, ordered, right_column_title*

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

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

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

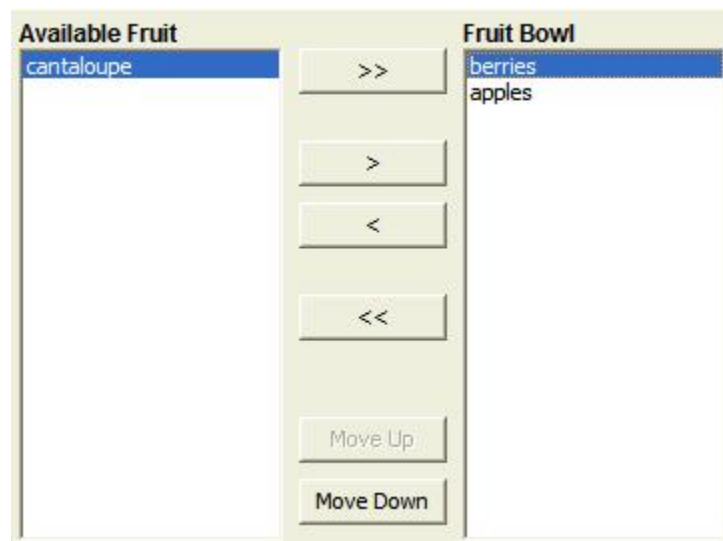


Fig. 1.39: Figure 43: Set editor showing all possible buttons

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

- Set the *values* parameter to a list, tuple, dictionary, or mapped trait.
- Set the *name* parameter to the extended name of a trait attribute that contains the list.

ShellEditor()

Suitable for special

Default for PythonValue

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

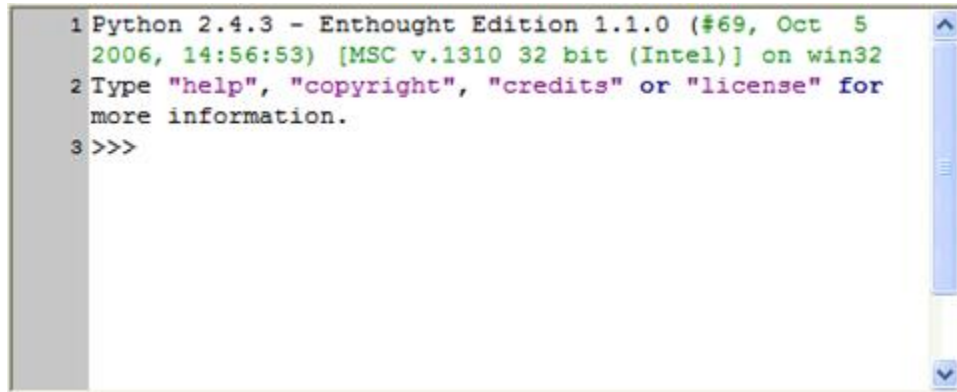


Fig. 1.40: Figure 44: Python shell editor

TextEditor()

Suitable for all

Default for Str, String, Password, Unicode, Int, Float, Dict, CStr, CUnicode, and any trait that does not have a specialized TraitHandler

Optional parameters *auto_set*, *enter_set*, *evaluate*, *evaluate_name*, *mapping*, *multi_line*, *password*

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

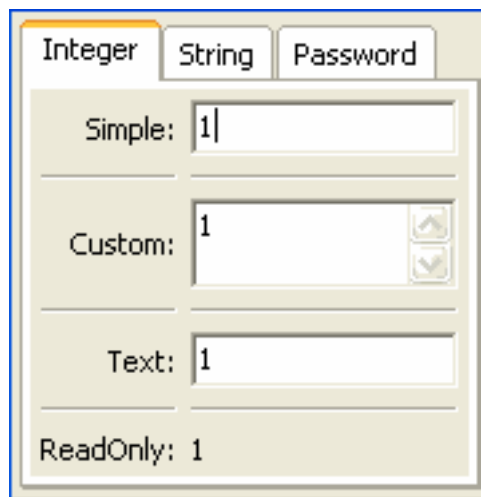


Fig. 1.41: Figure 45: Text editor styles for integers

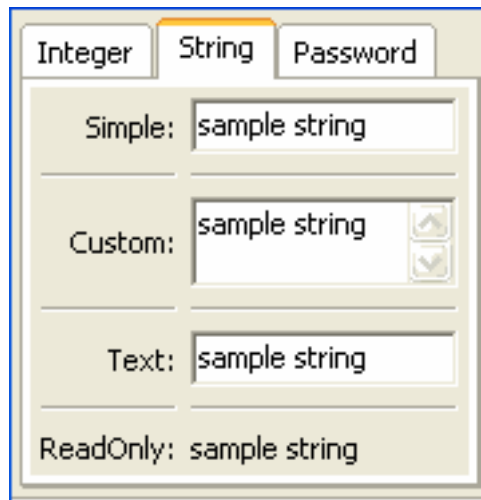


Fig. 1.42: Figure 46: Text editor styles for strings

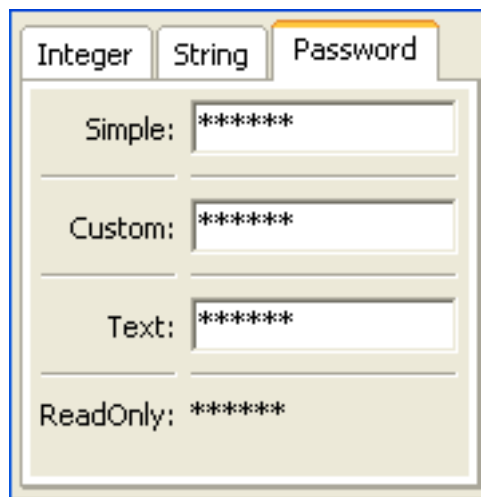


Fig. 1.43: Figure 47: Text editor styles for passwords

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

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

TitleEditor()

Suitable for string traits

Default for (none)

TitleEditor() generates a read-only display of a string value, formatted as a heading. All styles of the editor are identical. Visually, it is similar to a Heading item, but because it is an editor, you can change the text of the heading by modifying the underlying attribute.

TupleEditor()

Suitable for Tuple

Default for Tuple

Optional parameters *cols, editors, labels, traits*

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

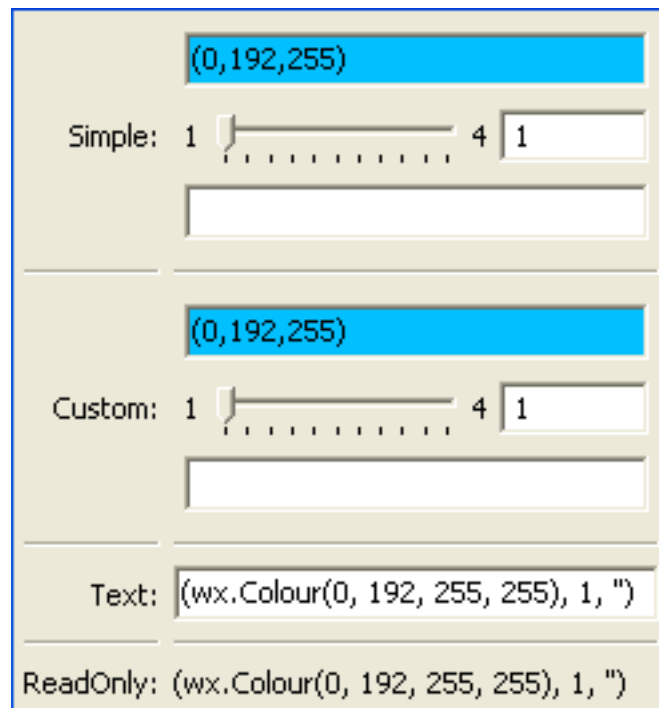


Fig. 1.44: Figure 48: Tuple editor styles

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

You can supply a list of editors to be used for each corresponding tuple slot. If the *editors* list is missing, or is shorter than the length of the tuple, default editors are used for any tuple slots not defined in the list. This feature allows you to substitute editors, or to supply non-default parameters for editors.

ValueEditor()

Suitable for (any)

Default for (none)

Optional parameters *auto_open*

ValueEditor() generates a tree editor that displays Python values and objects, including all the objects' members. For example, Figure 49 shows a value editor that is displayed by the “pickle viewer” utility in enthought.debug.

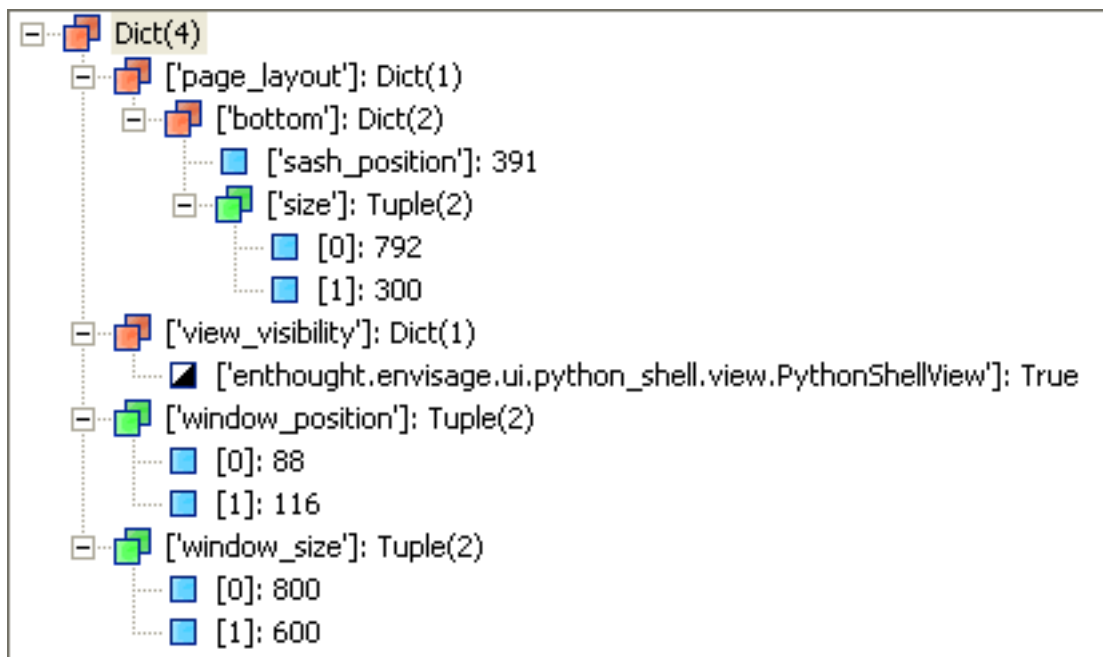


Fig. 1.45: Figure 49: Value editor from Pickle Viewer

1.9 Advanced Trait Editors

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

1.9.1 CustomEditor()

Suitable for Special cases

Default for (none)

Required parameters *factory*

Optional parameters *args*

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

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

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

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

For an example of using `CustomEditor()`, examine the implementation of the `NumericModelExplorer` class in the `enthought.model.numeric_model_explorer` module; `CustomEditor()` is used to generate the plots in the user interface.

1.9.2 DropEditor()

Suitable for Instance traits

Default for (none)

Optional parameters *binding, klass, readonly*

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

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

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

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

1.9.3 DNDEditor()

Suitable for Instance traits

Default for (none)

Optional parameters *drag_target, drop_target, image*

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

Table 9: Drag-and-drop editor style variations

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

1.9.4 KeyBindingEditor()

The `KeyBindingEditor()` factory differs from other trait editor factories because it generates an editor, not for a single attribute, but for an object of a particular class, `traitsui.key_bindings.KeyBindings`. A `KeyBindings` object is a list of bindings between key codes and handler methods. You can specify a `KeyBindings` object as an attribute of a `View`. When the user presses a key while a `View` has input focus, the user interface searches the `View` for a `KeyBindings` that contains a binding that corresponds to the key press; if such a binding does not exist on the `View`, it searches enclosing `Views` in order, and uses the first matching binding, if any. If it does not find any matching bindings, it ignores the key press.

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

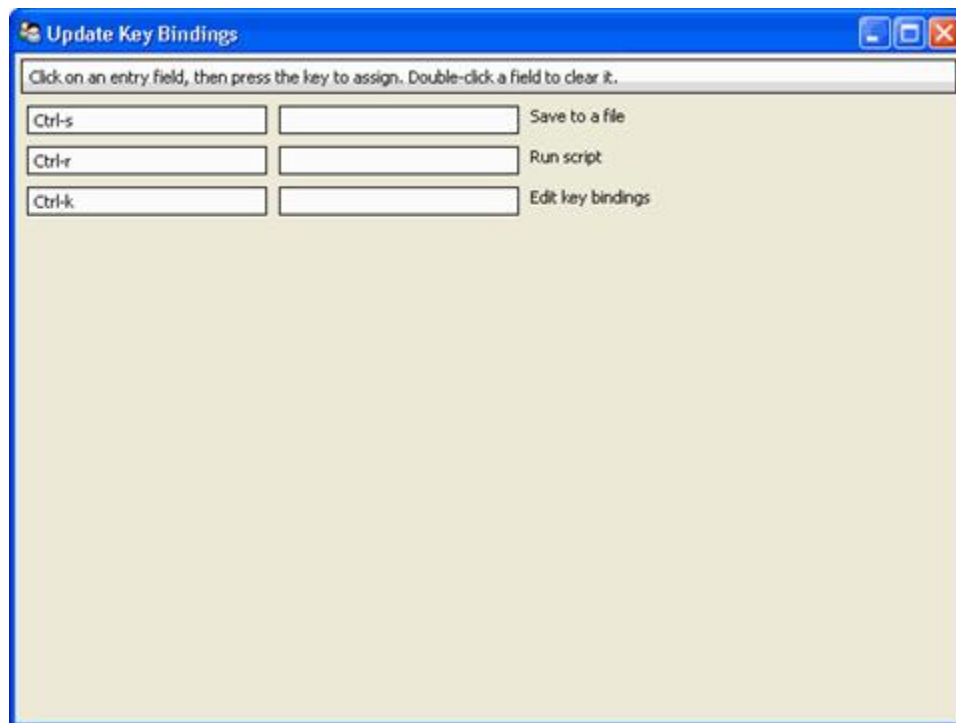


Fig. 1.46: Figure 50: Key binding editor dialog box

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

Example 17: Code editor with key binding editor

```
# key_bindings.py -- Example of a code editor with a
#                      key bindings editor

from traits.api \
    import Button, Code, HasPrivateTraits, Str
from traitsui.api \
    import View, Item, Group, Handler, CodeEditor
from traitsui.key_bindings \
    import KeyBinding, KeyBindings

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

# TraitsUI Handler class for bound methods
class CodeHandler ( Handler ):

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

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

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

class KBCodeExample ( HasPrivateTraits ):

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

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

        handler = CodeHandler() )
```

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

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

1.9.5 TableEditor()

Suitable for `List(InstanceType)`

Default for (none)

Required parameters `columns` or `columns_name`

Optional parameters See *Traits API Reference*, `traitsui.wx.table_editor.ToolkitEditorFactory` attributes.

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

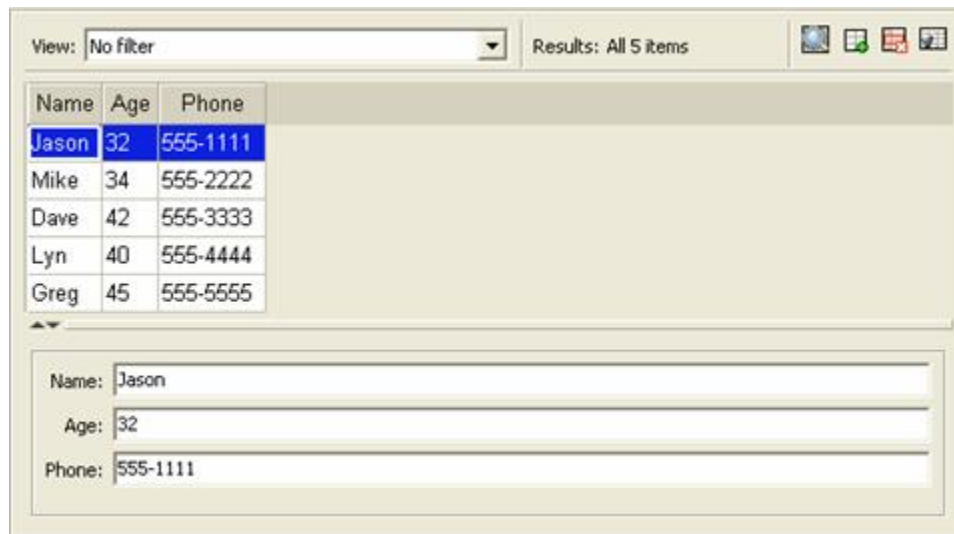


Fig. 1.47: Figure 51: Table editor

To see the code that results in Figure 51, refer to `TableEditor_demo.py` in the `demos/TraitsUI Demo/Standard Editors` subdirectory of the Traits UI package. This example demonstrates object columns, expression columns, filters, searching, and adding and deleting rows.


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

- *Specifying Columns*
- *Managing Items*
- *Editing the Table*
- *Defining the Layout*
- *Defining the Format*
- *Other User Interactions*

Specifying Columns

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

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

You can specify additional columns that are not initially displayed using the `other_columns` parameter. If the `configurable` parameter is `True` (the default), a *Set user preferences for table* icon () appears on the table's toolbar. When the user clicks this icon, a dialog box opens that enables the user to select and order the columns displayed in the table, as shown in Figure 52. (The dialog box is implemented using a set editor; see *SetEditor()*.) Any columns that were specified in the `other_columns` parameter are listed in the left list box of this dialog box, and can be displayed by moving them into the right list box.

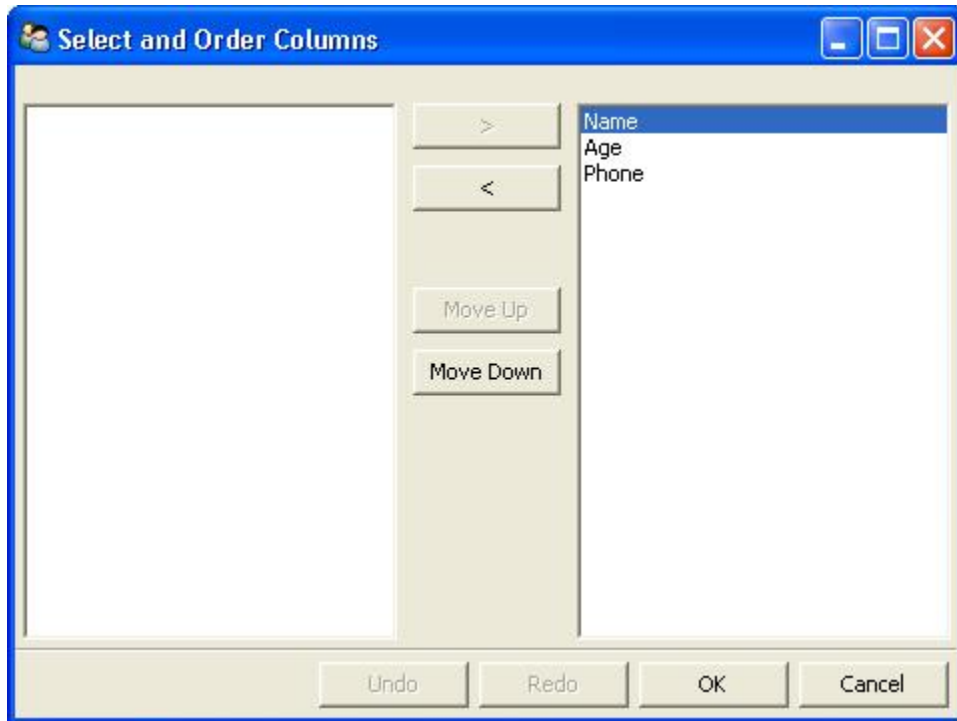




Fig. 1.48: Figure 52: Column selection dialog box for a table editor

Managing Items

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

Organizing Items


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

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

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

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

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

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


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

Filtering and Searching

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

The *filter* parameter specifies a filter that is applied to the table when it is first displayed. The *filter_name* parameter specifies an extended trait name for a trait that is either a table filter object or a callable that accepts an object and returns True if the object passes the filter criteria, or false if it does not. You can use *filter_name* to embed a view of a table filter in the same view as its table.

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

You can also provide an option for the user to use filters to search the table. If you set the *search* parameter to an instance of `TableFilter` (or of a subclass), a *Search table* icon () appears on the table toolbar. Clicking this icon

opens a *Search for* dialog box, which enables the user to specify filter criteria, to browse through matching items, or select all matching items.

Interacting with Items

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

The value of the *selection_mode* parameter specifies how the user can make selections in the grid:

- *cell*: A single cell at a time
- *cells*: Multiple cells
- *column*: A single column at a time
- *columns*: Multiple columns
- *row*: A single row at a time
- *rows*: Multiple rows

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

You can use the *selected_indices* parameter to specify the name of a trait attribute in the current context to synchronize with the indices of the table editor selection. The content of the selection depends on the *selection_mode* value:

- **cell**: The selection is a tuple of the form (*object*, *column_name*), where *object* is the object contains the selected cell, and *column_name* is the name of the column the cell is in. If there is no selection, the tuple is (None, '').
- *cells*: The selection is a list of tuples of the form (*object*, *column_name*), with one tuple for each selected cell, in order from top to bottom and left to right. If there is no selection, the list is empty.
- *column*: The selection is the name of the selected column, or the empty string if there is no selection.
- *columns*: The selection is a list containing the names of the selected columns, in order from left to right. If there is no selection, the list is empty.
- *row*: The selection is either the selected object or None if nothing is selected in the table.
- *rows*: The selection is a list of the selected objects, in ascending row order. If there is no selection, the list is empty.


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

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

Editing the Table

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


Adding Items

To enable users to add items to the table, specify as the *row_factory* parameter a callable that generates an object that can be added to the list in the table; for example, the class of the objects in the table. When *row_factory* is specified, an *Insert new item* icon () appears in the table toolbar, which generates a new row in the table. Optionally, you can use *row_factory_args* and *row_factory_kw* to specify positional and keyword arguments to the row factory callable.

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

Deleting Items

The *deletable* parameter controls whether items can be deleted from the table. This parameter can be a Boolean (defaulting to False) or a callable; the callable must take an item as an argument and handle deleting it. If *deletable* is

not False, a *Delete current item* icon () appears on the table toolbar; clicking it deletes the item corresponding to the row that is selected in the table.

Modifying Items

The user can modify items in two ways.

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

Defining the Layout

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

- *auto_size*: If True, the cells of the table automatically adjust to the optimal size based on their contents.
- *orientation*: The layout of the table relative to its associated editor pane. Can be 'horizontal' or 'vertical'.
- *rows*: The number of visible rows in the table.
- *show_column_labels*: If True (the default), displays labels for the columns. You can specify the labels to use in the column definitions; otherwise, a "user friendly" version of the trait attribute name is used.
- *show_toolbar*: If False, the table toolbar is not displayed, regardless of whether other settings would normally create a toolbar. The default is True.

Defining the Format

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

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

Other User Interactions

The table editor supports additional types of user interaction besides those controlled by the factory parameters.

- **Column dragging:** The user can reorganize the column layout of a table editor by clicking and dragging a column label to its new location. If you have enabled user preferences for the view and table editor (by specifying view and item IDs), the new column layout is persisted across user sessions.
- **Column resizing:** The user can resize a column by dragging the column separator (in one of the data rows) to a new position. Because of the column-dragging support, clicking the column separator in the column label row does not work.
- **Data dragging:** The user can drag the contents of any cell by clicking and dragging.

1.9.6 TabularEditor()

Suitable for lists, arrays, and other large sequences of objects

Default for (none)

Required parameters *adapter*

Optional parameters *activated, clicked, column_clicked, dclicked, drag_move, editable, horizontal_lines, images, multi_select, operations, right_clicked, right_dclicked, selected, selected_row, show_titles, vertical_lines*

The TabularEditor() factory can be used for many of the same purposes as the TableEditor() factory, that is, for displaying a table of attributes of lists or arrays of objects. While similar in function, the tabular editor has advantages and disadvantages relative to the table editor.

Advantages

- **Very fast:** The tabular editor uses a virtual model, which accesses data from the underlying model only as needed. For example, if you have a million-element array, but can display only 50 rows at a time, the editor requests only 50 elements of data at a time.
- **Very flexible data model:** The editor uses an adapter model to interface with the underlying data. This strategy allows it to easily deal with many types of data representation, from list of objects, to arrays of numbers, to tuples of tuples, and many other formats.
- **Supports useful data operations,** including:
 - Moving the selection up and down using the keyboard arrow keys.
 - Moving rows up and down using the keyboard.
 - Inserting and deleting items using the keyboard.
 - Initiating editing of items using the keyboard.
 - Dragging and dropping of table items to and from the editor, including support for both copy and move operations for single and multiple items.
- **Visually appealing:** The tabular editor, in general, uses the underlying operating system's native table or grid control, and as a result often looks better than the control used by the table editor.
- **Supports displaying text and images in any cell.** However, the images displayed must be all the same size for optimal results.

Disadvantages

- **Not as full-featured:** The table editor includes support for arbitrary data filters, searches, and different types of sorting. These differences may narrow as features are added to the tabular editor.
- **Limited data editing capabilities:** The tabular editor supports editing only textual values, whereas the table editor supports a wide variety of column editors, and can be extended with more as needed. This is due to limitations of the underlying native control used by the tabular editor.

TabularAdapter

The tabular editor works in conjunction with an adapter class, derived from `TabularAdapter`. The tabular adapter interfaces between the tabular editor and the data being displayed. The tabular adapter is the reason for the flexibility and power of the tabular editor to display a wide variety of data.

The most important attribute of `TabularAdapter` is **columns**, which is list of columns to be displayed. Each entry in the **columns** list can be either a string, or a tuple consisting of a string and another value, which can be of any type. The string is used as the label for the column. The second value in the tuple, called the *column ID*, identifies the column to the adapter. It is typically a trait attribute name or an integer index, but it can be any value appropriate to the adapter. If only a string is specified for an entry, then the index of the entry within the **columns** list is used as that entry's column ID.

Attributes on `TabularAdapter` control the appearance of items, and aspects of interaction with items, such as whether they can be edited, and how they respond to dragging and dropping. Setting any of these attributes on the adapter subclass sets the global behavior for the editor. Refer to the *Traits API Reference* for details of the available attributes.

You can also specify these attributes for a specific class or column ID, or combination of class and column ID. When the `TabularAdapter` needs to look up the value of one of its attributes for a specific item in the table, it looks for attributes with the following naming conventions in the following order:

1. *classname_columnid_attribute*
2. *classname_attribute*
3. *columnid_attribute*
4. *attribute*

For example, to find the **text_color** value for an item whose class is `Person` and whose column ID is 'age', the `get_text_color()` method looks for the following attributes in sequence, and returns the first value it finds:

1. **Person_age_text_color**
2. **Person_text_color**
3. **age_text_color**
4. **text_color**

Note that the *classname* can be the name of a base class, searched in the method resolution order (MRO) for the item's class. So for example, if the item were a direct instance of `Employee`, which is a subclass of `Person`, then the **Person_age_text_color** attribute would apply to that item (as long as there were no **Employee_age_text_color** attribute).

The Tabular Editor User Interface

Figure 53 shows an example of a tabular editor on Microsoft Windows, displaying information about source files in the Traits package. This example includes a column that contains an image for files that meet certain conditions.

| File Name | Size | Time | Date |
|----------------------|--------|-------------|------------|
| adapter.py | 7830 | 04:08:06 PM | 08/13/2007 |
| api.py | 5794 | 05:37:52 PM | 11/16/2007 |
| category.py | 4757 | 04:08:06 PM | 08/13/2007 |
| core_traits.py | 3758 | 04:08:06 PM | 08/13/2007 |
| has_dynamic_views.py | 15556 | 04:08:06 PM | 08/13/2007 |
| has_traits.py | 146863 | 03:29:33 PM | 11/28/2007 |
| standard.py | 13715 | 04:08:06 PM | 08/13/2007 |
| traits.py | 55643 | 05:37:52 PM | 11/16/2007 |
| traits_listener.py | 39379 | 03:29:33 PM | 11/28/2007 |
| trait_base.py | 15628 | 12:10:01 PM | 11/09/2007 |
| trait_db.py | 22879 | 04:08:06 PM | 08/13/2007 |
| trait_errors.py | 4058 | 12:11:35 PM | 09/11/2007 |
| trait_handlers.py | 112372 | 12:10:01 PM | 11/09/2007 |
| trait_notifiers.py | 27635 | 01:28:49 PM | 10/03/2007 |
| trait_numeric.py | 14070 | 12:10:01 PM | 11/09/2007 |

Fig. 1.49: Figure 53: Tabular editor on MS Windows

Depending on how the tabular editor is configured, certain keyboard interactions may be available. For some interactions, you must specify that the corresponding operation is allowed by including the operation name in the *operations* list parameter of `TabularEditor()`.

- Up arrow: Selects the row above the currently selected row.
- Down arrow: Selects the row below the currently selected row.
- Page down: Appends a new item to the end of the list ('append' operation).
- Left arrow: Moves the currently selected row up one line ('move' operation).
- Right arrow: Moves the currently selected row down one line ('move' operation).
- Backspace, Delete: Deletes from the list all items in the current selection ('delete' operation).
- Enter, Escape: Initiates editing on the current selection ('edit' operation).
- **Insert :: Inserts a new item before the current selection ('insert' operation).**

The 'append', 'move', 'edit', and 'insert' operations can occur only when a single item is selected. The 'delete' operation works for one or more items selected.

Depending on how the editor and adapter are specified, drag and drop operations may be available. If the user selects multiple items and drags one of them, all selected items are included in the drag operation. If the user drags a non-selected item, only that item is dragged.

The editor supports both "drag-move" and "drag-copy" semantics. A drag-move operation means that the dragged items are sent to the target and are removed from the list displayed in the editor. A drag-copy operation means that the dragged items are sent to the target, but are not deleted from the list data.

1.9.7 TreeEditor()

Suitable for Instance

Default for (none)

Required parameters *nodes* (required except for shared editors; see [Editing Objects](#))

Optional parameters *auto_open*, *editable*, *editor*, *hide_root*, *icon_size*, *lines_mode*, *on_dclick*, *on_select*, *orientation*, *selected*, *shared_editor*, *show_icons*

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

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

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

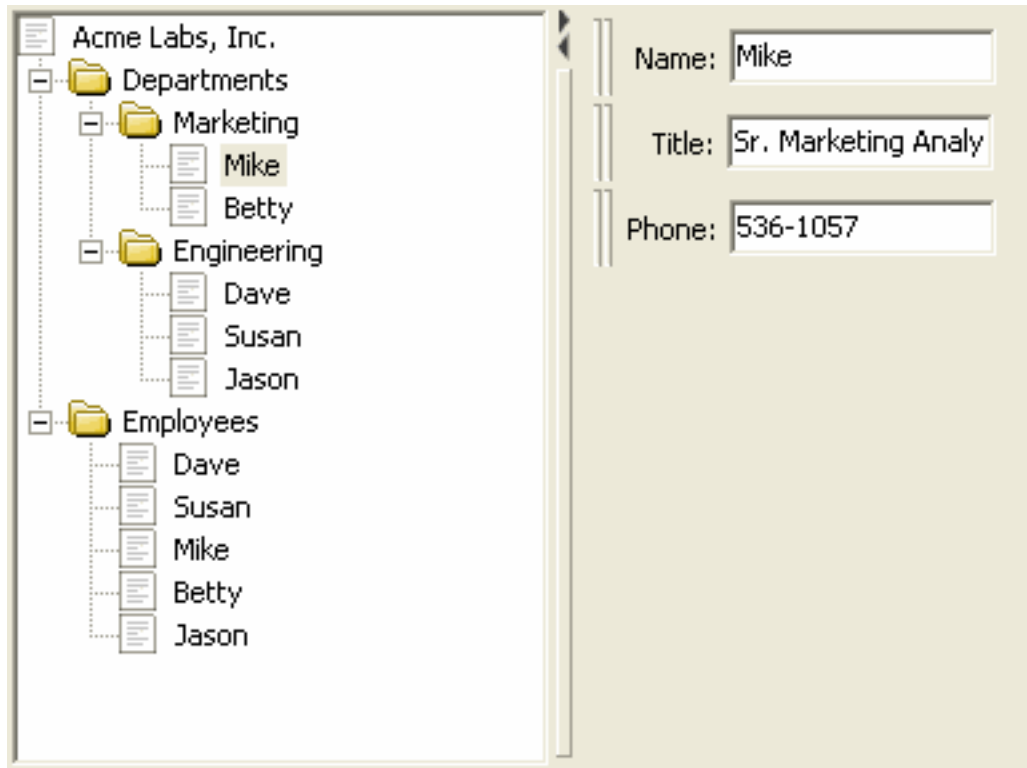


Fig. 1.50: Figure 54: Tree editor

The following example shows the code that produces the editor shown in Figure 54.

Example 18: Code for example tree editor

```
# tree_editor.py -- Example of a tree editor

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

# DATA CLASSES
```

```

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

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

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

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

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

# INSTANCES

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

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

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

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

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

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

```



```

        Department(
            name = 'Engineering',
            employees = [ dave, susan, jason ]
        )
    ],
    employees = [ dave, susan, mike, betty, jason ]
)

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

# Actions used by tree editor context menu

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

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

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

class TreeHandler ( Handler ):

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

# Tree editor
tree_editor = TreeEditor(
    nodes = [
        TreeNode( node_for = [ Company ],
                  auto_open = True,
                  children = '',
                  label = 'name',
                  view = View( Group('name',
                                   orientation='vertical',
                                   show_left=True )) ),
        TreeNode( node_for = [ Company ],
                  auto_open = True,
                  children = 'departments',
                  label = '=Departments',
                  view = no_view,
                  add = [ Department ] ),
        TreeNode( node_for = [ Company ],
                  auto_open = True,
                  children = 'employees',
                  label = '=Employees',

```

```

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

# The main view
view = View(
    Group(
        Item(
            name = 'company',
            id = 'company',
            editor = tree_editor,
            resizable = True ),
        orientation = 'vertical',
        show_labels = True,
        show_left = True, ),
    title = 'Company Structure',
    id = \
        'traitsui.tests.tree_editor_test',
    dock = 'horizontal',
    drop_class = HasTraits,
    handler = TreeHandler(),
    buttons = [ 'Undo', 'OK', 'Cancel' ],
    resizable = True,
    width = .3,

```

```

        height = .3 )

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

Defining Nodes

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

You must specify the classes whose instances the node type applies to. Use the **node_for** attribute of `TreeNode` to specify a list of classes; often, this list contains only one class. You can have more than one node type that applies to a particular class; in this case, each object of that class is represented by multiple nodes, one for each applicable node type. In Figure 54, one `Company` object is represented by the nodes labeled “Acme Labs, Inc.”, “Departments”, and “Employees”.

A Node Type without Children

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

```

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

A Node Type with Children

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

```

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

Setting the Label of a Tree Node

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

If the value is a simple string, it is interpreted as the extended trait name of an attribute on the object that the node is for, whose value is used as the label. This approach is used in the code snippet in *A Node Type without Children*.

If the value is a string that begins with an equals sign (=), the rest of the string is used as the literal label. This approach is used in the code snippet in *A Node Type with Children*.

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

Defining Operations on Nodes

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

Shortcut Menus on Nodes

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

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

Allowing the Hierarchy to Be Modified

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

Note: The **add** attribute alone is not enough to create objects.

Specifying the **add** attribute makes it possible for objects of the specified classes to be created, but by itself, it does not provide a way for the user to do so. In the code snippet in the preceding section (*Shortcut Menus on Nodes*), ‘NewAction’ in the Menu constructor call defines a *New > Employee* menu item that creates Employee objects.

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

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

'Worker' on the tree node whose **node_for** value contains Employee. If this attribute is not set, the class name is used.

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

| Attribute | If True, the ... | can be... |
|------------------|-------------------|-----------|
| copy | object's children | copied. |
| delete | object's children | deleted. |
| delete_me | object | deleted. |
| rename | object's children | renamed. |
| rename_me | object | renamed. |

All of these attributes default to True. As with **add**, you must also define actions to perform these operations.

Behavior on Nodes

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

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

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

Expanding and Collapsing Nodes

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

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

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

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

Editing Objects

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

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

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

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

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

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

For example:

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

Defining the Format

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

- *show_icons*: If `True` (the default), icons are displayed for the nodes in the tree.
- *icon_size*: A two-integer tuple indicating the size of the icons for the nodes.
- *lines_mode*: Determines whether lines are displayed between related nodes. The valid values are ‘on’, ‘off’, and ‘appearance’ (the default). When set to ‘appearance’, lines are displayed except on Posix-based platforms.
- *hide_root*: If `True`, the root node in the hierarchy is not displayed. If this parameter were specified as `True` in Example 16, the node in Figure 54 that is labeled “Acme Labs, Inc.” would not appear.

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

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

The `wxWidgets` implementation automatically detects the bitmap format of the icon.

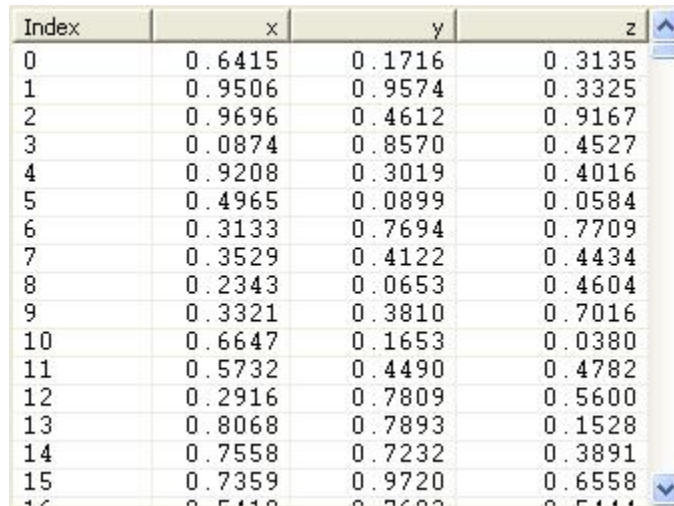
1.9.8 ArrayViewEditor()

Suitable for 2-D Array, 2-D CArray

Default for (none)

Optional parameters *format, show_index, titles, transpose*

ArrayViewEditor() generates a tabular display for an array. It is suitable for use with large arrays, which do not work well with the editors generated by ArrayEditor(). All styles of the editor have the same appearance.



| Index | x | y | z |
|-------|--------|--------|--------|
| 0 | 0.6415 | 0.1716 | 0.3135 |
| 1 | 0.9506 | 0.9574 | 0.3325 |
| 2 | 0.9696 | 0.4612 | 0.9167 |
| 3 | 0.0874 | 0.8570 | 0.4527 |
| 4 | 0.9208 | 0.3019 | 0.4016 |
| 5 | 0.4965 | 0.0899 | 0.0584 |
| 6 | 0.3133 | 0.7694 | 0.7709 |
| 7 | 0.3529 | 0.4122 | 0.4434 |
| 8 | 0.2343 | 0.0653 | 0.4604 |
| 9 | 0.3321 | 0.3810 | 0.7016 |
| 10 | 0.6647 | 0.1653 | 0.0380 |
| 11 | 0.5732 | 0.4490 | 0.4782 |
| 12 | 0.2916 | 0.7809 | 0.5600 |
| 13 | 0.8068 | 0.7893 | 0.1528 |
| 14 | 0.7558 | 0.7232 | 0.3891 |
| 15 | 0.7359 | 0.9720 | 0.6558 |

Fig. 1.51: Figure 55: Array view editor

1.9.9 DataFrameEditor()

Suitable for Pandas DataFrames

Default for (none)

Optional parameters *formats, show_index, show_titles, columns, fonts, selected, selected_row, selectable, activated, activated_row, clicked, dclicked, right_clicked, right_dclicked, column_clicked, column_right_clicked, editable, operations*

DataFrameEditor() generates a tabular display for a DataFrame. It is suitable for use with large DataFrames. All styles of the editor have the same appearance. Many of the optional parameters are identical to those of the TabularEditor().

The following have special meaning for the DataFrameEditor():

- **formats:** either a %-style formatting string for all entries, or a dictionary mapping DataFrame columns to formatting strings.
- **show_index:** whether or not to show the index as a column in the table.
- **show_titles:** whether or not to show column headers on the table.
- **fonts:** either a font for all entries, or a mapping of column id to fonts.

1.9.10 HistoryEditor()

Suitable for string traits

Default for (none)

Optional parameters *entries*

HistoryEditor() generates a combo box, which allows the user to either enter a text string or select a value from a list of previously-entered values. The same control is used for all editor styles. The *entries* parameter determines how many entries are preserved in the history list. This type of control is used as part of the simple style of file editor; see *FileEditor()*.

1.9.11 ImageEditor()

Suitable for (any)

Default for (none)

Optional parameters *image, scale, preserve_aspect_ratio, allow_upscaling, allow_clipping*

ImageEditor() generates a read-only display of an image. The image to be displayed is determined by the *image* parameter, or by the value of the trait attribute being edited, if *image* is not specified. In either case, the value must be a PyFace ImageResource (pyface.api.ImageResource), or a string that can be converted to one. If *image* is specified, then the type and value of the trait attribute being edited are irrelevant and are ignored.

For the Qt backend *scale, preserve_aspect_ratio, allow_upscaling, and allow_clipping* control whether the image should be scaled or not, and how to perform that scaling.

1.10 “Extra” Trait Editor Factories

The traitsui.wx package defines a few editor factories that are specific to the wxWidgets toolkit, some of which are also specific to the Microsoft Windows platform. These editor factories are not necessarily implemented for other GUI toolkits or other operating system platforms.

1.10.1 AnimatedGIFEditor()

Suitable for File

Default for (none)

Optional parameters *playing*

AnimatedGIFEditor() generates a display of the contents of an animated GIF image file. The Boolean *playing* parameter determines whether the image is animated or static.

1.10.2 FlashEditor()

Suitable for string traits, Enum(string values)

Default for (none)

FlashEditor() generates a display of an Adobe Flash Video file, using an ActiveX control (if one is installed on the system). This factory is available only on Microsoft Windows platforms. The attribute being edited must have a value whose text representation is the name or URL of a Flash video file. If the value is a Unicode string, it must contain only characters that are valid for filenames or URLs.

1.10.3 IEHTMLEditor()

Suitable for string traits, Enum(string values)

Default for (none)

Optional parameters *back, forward, home, html, page_loaded, refresh, search, status, stop, title*

IEHTMLEditor() generates a display of a web page, using Microsoft Internet Explorer (IE) via ActiveX to render the page. This factory is available only on Microsoft Windows platforms. The attribute being edited must have value whose text representation is a URL. If the value is a Unicode string, it must contain only characters that are valid for URLs.

The *back, forward, home, refresh, search* and *stop* parameters are extended names of event attributes that represent the user clicking on the corresponding buttons in the standard IE interface. The IE buttons are not displayed by the editor; you must create buttons separately in the View, if you want the user to be able to actually click buttons.

The *html, page_loaded, status*, and *title* parameters are the extended names of string attributes, which the editor updates with values based on its own state. You can display these attributes elsewhere in the View.

- *html*: The current page content as HTML (as would be displayed by the *View > Source* command in IE).
- *page_loaded*: The URL of the currently displayed page; this may be different from the URL represented by the attribute being edited.
- *status*: The text that would appear in the IE status bar.
- *title*: The title of the currently displayed page.

1.10.4 LEDEditor()

Suitable for numeric traits

Default for (none)

Optional parameters *alignment, format_str*

LEDEditor() generates a display that resembles a “digital” display using light-emitting diodes. All styles of this editor are the same, and are read-only.

The *alignment* parameter can be ‘left’, ‘center’, or ‘right’ to indicate how the value should be aligned within the display. The default is right-alignment.



Fig. 1.52: Figure 56: LED Editor with right alignment

1.11 Advanced Editor Adapters

A number of trait editors provide a way for code to adapt objects to the expected API for the editor, and this can be used by Traits UI code to provide strongly customized views of the data. The editors which provide this facility are the ListStrEditor, the TabularEditor and the TreeEditor. In this section we will look more closely at each of these and discuss how they can be customized as needed.

1.11.1 The TreeEditor and TreeNodes

The *TreeEditor* internally associates with each node in the tree a pair consisting of the object that is associated with the node and something that adheres to the *TreeNode* interface. The *TreeNode* interface is not explicitly

laid out, but it corresponds to the “overridable” public methods of the *TreeNode* class, such as *get_label()* and *get_children()*.

This means that the tree editor expects one of the following three things to be offered as values associated with a node, such as the value of the root node trait or values that might be returned by the *get_children()* method:

- an explicit pair of the object and a *TreeNode* instance for that object
- an object that has *is_node_for()* return True for at least one of the factory’s nodes items.
- an object that provides or can be adapted to the *ITreeNode* interface using Traits adaptation.

There is a crucial distinction between the way that *TreeNode* and *ITreeNode* work. *TreeNode* is generic—it is designed to work with certain types of objects, but doesn’t hold references to those objects—instead they rely on the *TreeEditor* to keep track of the association between the objects and the *TreeNode* to use with that object. *ITreeNode*, on the other hand, is an interface and uses adapters associated with individual objects rather than types of objects. This means that *ITreeNode*-based approaches are generally more heavyweight: you end up with at least one additional class instance for each displayed node (and most likely two additional instances) vs. a tuple. On the other hand, because *ITreeNode* uses Traits adaptation, you can extend the set of classes that are supported by adding more *ITreeNode* adapters, for example via Envisage extension points.

Specializing TreeNode Behaviour

In general using *TreeNode*s works well when you have a hierarchy of *HasTraits* objects, which is probably the most common situation. And while the *TreeNode* is fairly generic, there are times when you want to override the default behaviour of one or more aspects of the object. In this case it may be that the best way to do this is to simply subclass *TreeNode* and adjust it to behave the way that you want.

For example, the default behaviour of the *TreeNode* is to show one of 3 different icons depending on whether the node has children or not and whether it has been expanded. But you might want to display a different icon based on some attribute of the object being viewed, and that would require a new *TreeNode* subclass to override that behaviour.

Concretely, if we had different document types, identified by file extension:

```
class DocumentTreeNode(TreeNode):

    icons = Dict({
        '.npy': ImageResource('document-table'),
        '.txt': ImageResource('document-text'),
        '.rst': ImageResource('document-text'),
        '.png': ImageResource('document-image'),
        '.jpg': ImageResource('document-image'),
    })

    def get_icon(self, object, is_expanded):
        icon = self.icons.get(object.extension, self.icon_item)
        return icon
```

This *TreeNode* subclass can now be used with any compatible class to give a richer set of icons.

Common use cases for this approach would include:

- more customized icon display, as above.
- having the label built from multiple traits, which requires overriding *get_label()*, *when_label_changed()* and possibly *set_label()*.
- having the children come from multiple traits, which requires overriding *allows_children()*, *get_children()*, *when_children_replaced()*, *when_children_changed()* and possibly

`append_child()`, `insert_child()` and `delete_child()` (although there may be better ways to handle this situation by using multiple `TreeNode`s for the class).

- being more selective about what objects to use for the node. For example, requiring not only that an object be of a certain class, but that it also have an attribute with a certain value. This requires overriding `is_node_for()`.
- customization of menus on a per-object basis, or other UI behaviour like drag and drop, selection and clicking.

This has the advantage that most of the time the behaviour that you want is built into the `TreeNode` class, and you only need to change the things which are not to your requirements.

Where `TreeNode` classes are generally weak is when the object you are trying to view is not a `HasTraits` instance, or where you don't know the full set of classes that you need to display in the tree when writing the UI. You can overcome these obstacles by careful subclassing, taking particular care to avoid things like trying to set traits listeners on non-`HasTraits` objects or adapting the object to a desired interface before using it. But in these cases it may be better to use a different approach.

ITreeNodes and ITreeNodeAdapters

These are most useful for situations where you don't know the full set of classes that may be displayed in a tree. This is a common situation when writing complex applications using libraries like Envisage that allow new functionality to be added to the application via plug-ins (potentially during run-time!). It is also useful in situations where the model object that is being viewed isn't a `HasTraits` object, or where you may need some UI state in the node that doesn't belong on the underlying model object (for example, caching quantities which are expensive to compute).

Before using this approach, you should make sure that you understand the way that traits adaptation works.

To make writing code which satisfies the `ITreeNode` interface easier, there is an `ITreeNodeAdapter` class which provides basic functionality and which can be subclassed to provide an adapter class for your own nodes. This adapter is minimalistic and not complete. You will at a minimum need to override the `get_label()` method, and probably many others to get the desired behaviour. Since the `ITreeNodeAdapter` is an `Adapter` subclass, the object being adapted is available as the `adaptee` attribute. This means that the methods might look similar to the ones for `TreeNode`, but they don't expect to be passed the object as a parameter.

Once you have written the `ITreeNodeAdapter` subclass, you have to register the adapter with traits using the `Traits.register_factory()` function. You are not required to use `ITreeNodeAdapter` if you don't wish to. You can instead write a class which `@provides` the `ITreeNode` interface directly, or create an alternative adapter class.

Note that currently the tree editor infrastructure uses the deprecated `Traits.adapts()` class advisor and the default traits adapter registry which means that you can't have multiple different `ITreeNode` adapters for a given object to use in different editors within a given application. This is likely to be fixed in a future release of TraitsUI. In the mean-time you can work around this somewhat by having the trait being edited and/or the `get_children()` method return pre-adapted objects, rather than relying on traits adaptation machinery to find and adapt the object.

ObjectTreeNodes and TreeNodeObjects

Another approach to adapting objects, particularly non-`HasTraits` objects is used by the `ValueEditor`, but is available for general tree editors to use as well. In this approach you write one or more `TreeNodeObject` classes that wrap the model objects that you want to display, and then use instances of the `TreeNodeObject` classes within the tree editor, both as the root node being edited, and the objects returned by the `tno_get_children()` methods. To fit these with the expected `TreeNode` classes used by the `TreeEditor`, there is the `ObjectTreeNode` class which knows how to call the appropriate `TreeNodeObjects` and which can be given a list of `TreeNodeObject` classes that it understands.

For example, it is possible to represent a tree structure in Python using nested dictionaries with strings as keys. A `TreeNodeObject` for such a structure might look like this:

```
class DictNode(TreeNodeObject):

    #: The parent of the node
    parent = Instance('DictNode')

    #: The label for the node
    label = Str

    #: The value for this node
    value = Any

    def tno_get_label(self, node):
        return self.label

    def tno_allows_children(self, node):
        return isinstance(self.value, dict)

    def tno_has_children(self, node):
        return bool(self.value)

    def tno_get_children(self, node):
        return [DictNode(parent=self, label=key, value=value)
                for key, value in sorted(self.value.items())]
```

and so forth. There is additional work if you want to be able to modify the structure of the tree, for example. In addition to defining the *TreeNodeObject* subclass, you also need provide the nodes for the editor something like this:

```
dict_tree_editor = TreeEditor(
    editable=False,
    nodes=[
        ObjectTreeNode(
            node_for=[DictNode],
            rename=False,
            rename_me=False,
            copy=False,
            delete=False,
            delete_me=False,
        )
    ]
)
```

The *ObjectTreeNode* is a *TreeNode* subclass that delegates operations to the *TreeNodeObject*, but the default *TreeNodeObject* methods try to behave in the same way as the base *TreeNode*, so you can specify global behaviour on the *ObjectTreeNode* in the same way that you can for a *TreeNode*.

The last piece to make this approach work is that the root node when editing has to be a *DictNode* instance, so you may need to provide a property that wraps the raw tree structure in a *DictNode* to get started: unlike the *ITreeNodeAdapter* approaches this wrapping not automatically provided for you.

Examples

There are a number of examples of use of the *TreeEditor* in the TraitsUI demos:

- [TreeEditor](#)
- [Adapted TreeEditor](#)

- [HDF5 Tree](#)

1.11.2 The TabularAdapter Class

The power and flexibility of the tabular editor is mostly a result of the *TabularAdapter* class, which is the base class from which all tabular editor adapters must be derived.

The *TabularEditor* object interfaces between the underlying toolkit widget and your program, while the *TabularAdapter* object associated with the editor interfaces between the editor and your data.

The design of the *TabularAdapter* base class is such that it tries to make simple cases simple and complex cases possible. How it accomplishes this is what we'll be discussing in the following sections.

The TabularAdapter *columns* Trait

First up is the *TabularAdapter* *columns* trait, which is a list of values which define, in presentation order, the set of columns to be displayed by the associated *TabularEditor*.

Each entry in the *columns* list can have one of two forms:

- string
- (string, id)

where *string* is the user interface name of the column (which will appear in the table column header) and *id* is any value that you want to use to identify that column to your adapter. Normally this value is either a trait name or an integer index value, but it can be any value you want. If only *string* is specified, then *id* is the index of the *string* within *columns*.

For example, say you want to display a table containing a list of tuples, each of which has three values: a name, an age, and a weight. You could then use the following value for the *columns* trait:

```
columns = ['Name', 'Age', 'Weight']
```

By default, the *id* values (also referred to in later sections as the *column ids*) for the columns will be the corresponding tuple index values.

Say instead that you have a list of *Person* objects, with *name*, *age* and *weight* traits that you want to display in the table. Then you could use the following *columns* value instead:

```
columns = [('Name', 'name'),
           ('Age', 'age'),
           ('Weight', 'weight')]
```

In this case, the *column ids* are the names of the traits you want to display in each column.

Note that it is possible to dynamically modify the contents of the *columns* trait while the *TabularEditor* is active. The *TabularEditor* will automatically modify the table to show the new set of defined columns.

The Core TabularAdapter Interface

In this section, we'll describe the core interface to the *TabularAdapter* class. This is the actual interface used by the *TabularEditor* to access your data and display attributes. In the most complex data representation cases, these are the methods that you must override in order to have the greatest control over what the editor sees and does.

However, the base *TabularAdapter* class provides default implementations for all of these methods. In subsequent sections, we'll look at how these default implementations provide simple means of customizing the adapter to your needs. But for now, let's start by covering the details of the core interface itself.

To reduce the amount of repetition, we'll use the following definitions in all of the method argument lists that follow in this section:

object The object whose trait is being edited by the *TabularEditor*.

trait The name of the trait the *TabularEditor* is editing.

row The row index (starting with 0) of a table item.

column The column index (starting with 0) of a table column.

The adapter interface consists of a number of methods which can be divided into two main categories: those which are sensitive to the type of a particular table item, and those which are not. We'll begin with the methods that are sensitive to an item's type:

get_alignment() Returns the alignment style to use for a specified column.

The possible values that can be returned are: 'left', 'center' or 'right'. All table items share the same alignment for a specified column.

get_width() Returns the width to use for a specified column.

If the value is ≤ 0 , the column will have a *default* width, which is the same as specifying a width of 0.1.

If the value is > 1.0 , it is converted to an integer and the result is the width of the column in pixels. This is referred to as a *fixed width* column.

If the value is a float such that $0.0 < \text{value} \leq 1.0$, it is treated as the *unnormalized fraction of the available space* that is to be assigned to the column. What this means requires a little explanation.

To arrive at the size in pixels of the column at any given time, the editor adds together all of the *unnormalized fraction* values returned for all columns in the table to arrive at a total value. Each *unnormalized fraction* is then divided by the total to create a *normalized fraction*. Each column is then assigned an amount of space in pixels equal to the maximum of 30 or its *normalized fraction* multiplied by the *available space*. The *available space* is defined as the actual width of the table minus the width of all *fixed width* columns. Note that this calculation is performed each time the table is resized in the user interface, thus allowing columns of this type to increase or decrease their width dynamically, while leaving *fixed width* columns unchanged.

get_can_edit() Returns whether the user can edit a specified row.

A *True* result indicates that the value can be edited, while a *False* result indicates that it cannot.

get_drag() Returns the value to be *dragged* for a specified row.

A result of *None* means that the item cannot be dragged. Note that the value returned does not have to be the actual row item. It can be any value that you want to drag in its place. In particular, if you want the drag target to receive a copy of the row item, you should return a copy or clone of the item in its place.

Also note that if multiple items are being dragged, and this method returns *None* for any item in the set, no drag operation is performed.

get_can_drop() Returns whether the specified value can be dropped on the specified row.

A value of *True* means the value can be dropped; and a value of *False* indicates that it cannot be dropped.

The result is used to provide the user positive or negative drag feedback while dragging items over the table. *value* will always be a single value, even if multiple items are being dragged. The editor handles multiple drag items by making a separate call to *get_can_drop()* for each item being dragged.

get_dropped() Returns how to handle a specified value being dropped on a specified row.

The possible return values are:

- 'before': Insert the specified value before the dropped on item.
- 'after': Insert the specified value after the dropped on item.

Note there is no result indicating *do not drop* since you will have already indicated that the `object` can be dropped by the result returned from a previous call to `get_can_drop()`.

`get_font()` Returns the font to use for displaying a specified row or cell.

A result of `None` means use the default font; otherwise a toolkit font object should be returned. Note that all columns for the specified table row will use the font value returned.

`get_text_color()` Returns the text color to use for a specified row or cell.

A result of `None` means use the default text color; otherwise a toolkit-compatible color should be returned. Note that all columns for the specified table row will use the text color value returned.

`get_bg_color()` Returns the background color to use for a specified row or cell.

A result of `None` means use the default background color; otherwise a toolkit-compatible color should be returned. Note that all columns for the specified table row will use the background color value returned.

`get_image()` Returns the image to display for a specified cell.

A result of `None` means no image will be displayed in the specified table cell. Otherwise the result should either be the name of the image, or an `ImageResource` object specifying the image to display.

A name is allowed in the case where the image is specified in the `TabularEditor` `images` trait. In that case, the name should be the same as the string specified in the `ImageResource` constructor.

`get_format()` Returns the Python formatting string to apply to the specified cell.

The resulting of formatting with this string will be used as the text to display it in the table.

The return can be any Python string containing exactly one old-style Python formatting sequence, such as `'%.4f'` or `'(%5.2f)'`.

`get_text()` Returns a string containing the text to display for a specified cell.

If the underlying data representation for a specified item is not a string, then it is your responsibility to convert it to one before returning it as the result.

`set_text()` Sets the value for the specified cell.

This method is called when the user completes an editing operation on a table cell.

The string specified by `text` is the value that the user has entered in the table cell. If the underlying data does not store the value as text, it is your responsibility to convert `text` to the correct representation used.

`get_tooltip()` Returns a string containing the tooltip to display for a specified cell.

You should return the empty string if you do not wish to display a tooltip.

The following are the remaining adapter methods, which are not sensitive to the type of item or column data:

`get_item()` Returns the specified row item.

The value returned should be the value that exists (or *logically* exists) at the specified `row` in your data. If your data is not really a list or array, then you can just use `row` as an integer *key* or *token* that can be used to retrieve a corresponding item. The value of `row` will always be in the range: `0 <= row < len(object, trait)` (i.e. the result returned by the adapter `len()` method).

`len()` Returns the number of row items in the specified `object.trait`.

The result should be an integer greater than or equal to 0.

`delete()` Deletes the specified row item.

This method is only called if the *delete* operation is specified in the `TabularEditor` `operation` trait, and the user requests that the item be deleted from the table.

The adapter can still choose not to delete the specified item if desired, although that may prove confusing to the user.

`insert()` Inserts `value` at the specified `object.trait[row]` index.

The specified `value` can be:

- An item being moved from one location in the data to another.
- A new item created by a previous call to `get_default_value()`.
- An item the adapter previously approved via a call to `get_can_drop()`.

The adapter can still choose not to insert the item into the data, although that may prove confusing to the user.

`get_default_value()` Returns a new default value for the specified `object.trait` list.

This method is called when `insert` or `append` operations are allowed and the user requests that a new item be added to the table. The result should be a new instance of whatever underlying representation is being used for table items.

Creating a Custom TabularAdapter

Having just taken a look at the core `TabularAdapter` interface, you might now be thinking that there are an awful lot of methods that need to be specified to get an adapter up and running. But as we mentioned earlier `TabularAdapter` is not an abstract base class. It is a concrete base class with implementations for each of the methods in its interface. And the implementations are written in such a way that you will hopefully hardly ever need to override them.

In this section, we'll explain the general implementation style used by these methods, and how you can take advantage of them in creating your own adapters.

One of the things you probably noticed as you read through the core adapter interface section is that most of the methods have names of the form: `get_xxx` or `set_xxx`, which is similar to the familiar *getter/setter* pattern used when defining trait properties. The adapter interface is purposely defined this way so that it can expose and leverage a simple set of design rules.

The design rules are followed consistently in the implementations of all of the adapter methods described in the first section of the core adapter interface, so that once you understand how they work, you can easily apply the design pattern to all items in that section. Then, only in the case where the design rules will not work for your application will you ever have to override any of those `TabularAdapter` base class method implementations.

So the first thing to understand is that if an adapter method name has the form: `get_xxx` or `set_xxx` it really is dealing with some kind of trait called `xxx`, or which contains `xxx` in its name. For example, the `py:meth`~TabularAdapter.get_alignment`` method retrieves the value of some `alignment` trait defined on the adapter. In the following discussion we'll simply refer to an attribute name generically as *attribute*, but you will need to replace it by an actual attribute name (e.g. `alignment`) in your adapter.

The next thing to keep in mind is that the adapter interface is designed to easily deal with items that are not all of the same type. As we just said, the design rules apply to all adapter methods in the first group, which were defined as methods which are sensitive to an item's type. Item type sensitivity plays an important part in the design rules, as we will see shortly.

With this in mind, we now describe the simple design rules used by the first group of methods in the `TabularAdapter` class:

- When getting or setting an adapter attribute, the method first retrieves the underlying item for the specified data row. The item, and type (i.e. class) of the item, are then used in the next rule.
- The method gets or sets the first trait it finds on the adapter that matches one of the following names:
 - `classname_columnid_attribute`

- *classname_attribute*
- *columnid_attribute*
- *attribute*

where:

- *classname* is the name of the class of the item found in the first step, or one of its base class names, searched in the order defined by the *mro* (**method resolution order**) for the item's class.
- *columnid* is the column id specified by the developer in the adapter's *column* trait for the specified table column.
- *attribute* is the attribute name as described previously (e.g. *alignment*).

Note that this last rule always finds a matching trait, since the *TabularAdapter* base class provides traits that match the simple *attribute* form for all attributes these rules apply to. Some of these are simple traits, while others are properties. We'll describe the behavior of all these *default* traits shortly.

The basic idea is that rather than override the first group of core adapter methods, you simply define one or more simple traits or trait properties on your *TabularAdapter* subclass that provide or accept the specified information.

All of the adapter methods in the first group provide a number of arguments, such as *object*, *trait*, *row* and *column*. In order to define a trait property, which cannot be passed this information directly, the adapter always stores the arguments and values it computes in the following adapter traits, where they can be easily accessed by a trait getter or setter method:

- *row*: The table row being accessed.
- *column*: The column id of the table column being accessed (not its index).
- *item*: The data item for the specified table row (i.e. the item determined in the first step described above).
- *value*: In the case of a *set_xxx* method, the value to be set; otherwise it is *None*.

As mentioned previously, the *TabularAdapter* class provides trait definitions for all of the attributes these rules apply to. You can either use the default values as they are, override the default, set a new value, or completely replace the trait definition in a subclass. A description of the default trait implementation for each attribute is as follows:

***default_value* = Any('')** The default value for a new row.

The default value is the empty string, but you will normally need to assign a different (default) value.

***format* = Str('%s')** The default Python formatting string for a column item.

The default value is '%s' which will simply convert the column item to a displayable string value.

***text* = Property** The text to display for the column item.

The implementation of the property checks the type of the column's *column id*:

- If it is an integer, it returns `format % item[column_id]`.
- Otherwise, it returns `format % item.column_id`.

Note that *format* refers to the value returned by a call to *get_format()* for the current column item.

***text_color* = Property** The text color for a row item.

The property implementation checks to see if the current table row is even or odd, and based on the result returns the value of the *even_text_color* or *odd_text_color* trait if the value is not *None*, and the value of the *default_text_color* trait if it is. The definition of these additional traits are as follows:

- *odd_text_color* = Color(*None*)
- *even_text_color* = Color(*None*)

- `default_text_color = Color (None)`

Remember that a `None` value means use the default text color.

`bg_color = Property` The background color for a row item.

The property implementation checks to see if the current table row is even or odd, and based on the result returns the value of the `even_bg_color` or `odd_bg_color` trait if the value is not `None`, and the value of the `default_bg_color` trait if it is. The definition of these additional traits are as follows:

- `odd_bg_color = Color (None)`
- `even_bg_color = Color (None)`
- `default_bg_color = Color (None)`

Remember that a `None` value means use the default background color.

`alignment = Enum('left', 'center', 'right')` The alignment to use for a specified column.

The default value is `'left'`.

`width = Float(-1)` The width of a specified column.

The default value is `-1`, which means a dynamically sized column with an *unnormalized fractional* value of `0.1`.

`can_edit = Bool(True)` Specifies whether the text value of the current item can be edited.

The default value is `True`, which means that the user can edit the value.

`drag = Property` A property which returns the value to be dragged for a specified row item.

The property implementation simply returns the current row item.

`can_drop = Bool(False)` Specifies whether the specified value be dropped on the current item.

The default value is `False`, meaning that the value cannot be dropped.

`dropped = Enum('after', 'before')` Specifies where a dropped item should be placed in the table relative to the item it is dropped on.

The default value is `'after'`.

`font = Font` The font to use for the current item.

The default value is the standard default Traits font value.

`image = Str(None)` The name of the default image to use for a column.

The default value is `None`, which means that no image will be displayed for the column.

`tooltip = Str` The tooltip information for a column item.

The default value is the empty string, which means no tooltip information will be displayed for the column.

The preceding discussion applies to all of the methods defined in the first group of `TabularAdapter` interface methods. However, the design rules do not apply to the remaining five adapter methods, although they all provide a useful default implementation:

`get_item()` The default implementation assumes the trait defined by `object.trait` is a *sequence* and attempts to return the value at index `row`. If an error occurs, it returns `None` instead. This definition should work correctly for lists, tuples and arrays, or any other object that is indexable, but will have to be overridden for all other cases.

Note that this method is the one called in the first design rule described previously to retrieve the item at the current table row.

`len()` Again, the default implementation assumes the trait defined by `object.trait` is a *sequence* and attempts to return the result of calling `len(object.trait)`. It will need to be overridden for any type of data which for which `len()` will not work.

`delete()` The default implementation assumes the trait defined by `object.trait` is a mutable sequence and attempts to perform a `del object.trait[row]` operation.

`insert()` The default implementation assumes the trait defined by `object.trait` is a mutable sequence and attempts to perform an `object.trait[row:row] = [value]` operation.

`get_default_value()` The default implementation simply returns the value of the adapter's `default_value` trait.

Examples

There are a number of examples of use of the *TabularAdapter* in the TraitsUI demos:

- *TabularEditor*
- *TabularEditor* (auto-update)
- NumPy array *TabularEditor*

1.11.3 The ListStrAdapter Class

Although the *ListStrEditor* editor is frequently used, as might be expected, with lists of strings, it also provides facilities to edit lists of other object types that can be adapted to produce strings for display and editing via *ListStrAdapter* subclasses

The design of the *ListStrAdapter* base class follows the same design as the *TabularAdapter*, simplified by the fact that there are only rows, no columns. However, the names and intents of the various methods and traits are the same as the *TabularAdapter*, and so the approaches discussed in the previous section work for the *ListStrAdapter* as well.

1.12 Tips, Tricks and Gotchas

1.12.1 Getting and Setting Model View Elements

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

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

- *View*
- *Group*
- *Item*

- *ViewElement*
- ViewSubElement

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

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

trait_view()

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

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

For example:

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

1.13 Appendix I: Glossary of Terms

attribute An element of data that is associated with all instances of a given class, and is named at the class level.¹⁹ 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”.

class attribute An element of data that is associated with a class, and is named at the class level. There is only one value for a class attribute, associated with the class itself. In contrast, for an instance *attribute*, there is a value associated with every instance of a class.

command button A button on a window that globally controls the window. Examples include *OK*, *Cancel*, *Apply*, *Revert*, and *Help*.

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.

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

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

¹⁹ This is not always the case in Python, where attributes can be added to individual objects.

- editor factory** An instance of the Traits class `EditorFactory`. Editor factories generate the actual widgets used in a user interface. You can use an editor factory without knowing what the underlying GUI toolkit is.
- factory** An object used to produce other objects at run time without necessarily assigning them to named variables or 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 TraitsUI View. Various display options can be specified by means of attributes of this class, including a border, a group label, and the orientation of elements within the Group. An instance of the TraitsUI class `Group`.
- Handler** A TraitsUI object that implements GUI logic (data manipulation and dynamic window behavior) for one or more user interface windows. A Handler instance fills the role of *controller* in the MVC design pattern. An instance of the TraitsUI class *Handler*.
- HasTraits** A class defined in the Traits package to specify objects whose attributes are typed. That is, any attribute of a HasTraits subclass can be a *trait attribute*.
- 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*.
- 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 TraitsUI class `Item`.
- live** A term used to describe a window that is linked directly to the underlying model data, so that changes to data in the interface are reflected immediately in the model. A window 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 window 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 window that causes the remainder of the application to be suspended, so that the user can interact only with the window 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 window that is neither *live* nor *modal*.
- object** Synonym for *instance*.
- panel** A user interface region similar to a window except that it is embedded in a larger window rather than existing independently.
- predefined trait type** Any trait type that is built into the Traits package.
- 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 explicit typing in Python.
- TraitsUI** A high-level user interface toolkit designed to be used with the Traits package.
- View** A template object for constructing a GUI window or panel for editing a set of traits. The structure of a View is defined by one or more Group or Item objects; a number of attributes are defined for specifying display options

including height and width, menu bar (if any), and the set of buttons (if any) that are displayed. A member of the TraitsUI 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 box* windows, 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`).

1.14 Appendix II: Editor Factories for Predefined Traits

Predefined traits that are not listed in this table use `TextEditor()` by default, and have no other appropriate editor factories.

| Trait | Default Editor Factory | Other Possible Editor Factories |
|---------------------|---|---|
| Any | <code>TextEditor</code> | <code>EnumEditor</code> , <code>ImageEnumEditor</code> , <code>ValueEditor</code> |
| Array | <code>ArrayEditor</code> (for 2-D arrays) | |
| Bool | <code>BooleanEditor</code> | <code>ThemedCheckboxEditor</code> |
| Button | <code>ButtonEditor</code> | |
| CArray | <code>ArrayEditor</code> (for 2-D arrays) | |
| CBool | <code>BooleanEditor</code> | |
| CComplex | <code>TextEditor</code> | |
| CFloat, CInt, CLong | <code>TextEditor</code> | <code>LEDEditor</code> |
| Code | <code>CodeEditor</code> | |
| Color | <code>ColorEditor</code> | |
| Complex | <code>TextEditor</code> | |
| CStr, CUnicode | <code>TextEditor</code> (<code>multi_line=True</code>) | <code>CodeEditor</code> , <code>HTMLEditor</code> |
| Dict | <code>TextEditor</code> | <code>ValueEditor</code> |
| Directory | <code>DirectoryEditor</code> | |
| Enum | <code>EnumEditor</code> | <code>ImageEnumEditor</code> |
| Event | (none) | <code>ButtonEditor</code> , <code>ToolbarButtonEditor</code> |
| File | <code>FileEditor</code> | <code>AnimatedGIFEditor</code> |
| Float | <code>TextEditor</code> | <code>LEDEditor</code> |
| Font | <code>FontEditor</code> | |
| HTML | <code>HTMLEditor</code> | |
| Instance | <code>InstanceEditor</code> | <code>TreeEditor</code> , <code>DropEditor</code> , <code>DNDEditor</code> , <code>ValueEditor</code> |
| List | <code>TableEditor</code> for lists of HasTraits objects; <code>ListEditor</code> for all other lists. | <code>CSVListEditor</code> , <code>CheckListEditor</code> , <code>SetEditor</code> |
| Long | <code>TextEditor</code> | <code>LEDEditor</code> |
| Password | <code>TextEditor</code> (<code>password=True</code>) | |
| PythonValue | <code>ShellEditor</code> | |
| Range | <code>RangeEditor</code> | <code>ThemedSliderEditor</code> |
| Regex | <code>TextEditor</code> | <code>CodeEditor</code> |
| RGBColor | <code>RGBColorEditor</code> | |
| Str | <code>TextEditor</code> (<code>multi_line=True</code>) | <code>CodeEditor</code> , <code>HTMLEditor</code> |

Table 1.1 – continued from previous page

| Trait | Default Editor Factory | Other Possible Editor Factories |
|---------------|---|---------------------------------|
| String | TextEditor | CodeEditor, ThemedTextEditor |
| This | InstanceEditor | |
| ToolBarButton | ButtonEditor | |
| Tuple | TupleEditor | |
| UIDebugger | ButtonEditor (button calls the UIDebugEditor factory) | |
| Unicode | TextEditor(multi_line=True) | HTMLEditor |
| WeakRef | InstanceEditor | |

This document contains the auto-generated API reference documentation for TraitsUI. For user documentation, please read the *TraitsUI User Manual*

2.1 traitsui package

2.1.1 Subpackages

traitsui.editors package

Submodules

traitsui.editors.api module

traitsui.editors.array_editor module

traitsui.editors.boolean_editor module

Defines the Boolean editor factory for all traits toolkit backends.

```
traitsui.editors.boolean_editor.BooleanEditor  
    alias of ToolkitEditorFactory
```

```
class traitsui.editors.boolean_editor.ToolkitEditorFactory(*args, **traits)  
    Bases: traitsui.editors.text_editor.ToolkitEditorFactory  
    Editor factory for Boolean editors.
```

traitsui.editors.button_editor module

Defines the button editor factory for all traits toolkit backends.

traitsui.editors.button_editor.**ButtonEditor**
 alias of *ToolkitEditorFactory*

class traitsui.editors.button_editor.**ToolkitEditorFactory** (**traits)
 Bases: *traitsui.editor_factory.EditorFactory*
 Editor factory for buttons.

traitsui.editors.check_list_editor module

Defines the editor factory for multi-selection enumerations, for all traits toolkit backends.

traitsui.editors.check_list_editor.**CheckListEditor**
 alias of *ToolkitEditorFactory*

class traitsui.editors.check_list_editor.**ToolkitEditorFactory** (*args, **traits)
 Bases: *traitsui.editor_factory.EditorWithListFactory*
 Editor factory for checklists.

traitsui.editors.code_editor module

Defines the code editor factory for all traits toolkit backends, useful for tools such as debuggers.

traitsui.editors.code_editor.**CodeEditor**
 alias of *ToolkitEditorFactory*

class traitsui.editors.code_editor.**ToolkitEditorFactory** (*args, **traits)
 Bases: *traitsui.editor_factory.EditorFactory*
 Editor factory for code editors.

traitsui.editors.color_editor module

Defines the color editor factory for the all traits toolkit backends.

traitsui.editors.color_editor.**ColorEditor** (*args, **traits)

Returns an instance of the toolkit-specific editor factory declared in traitsui.<toolkit>.color_editor. If such an editor factory cannot be located, an instance of the abstract ToolkitEditorFactory declared in traitsui.editors.color_editor is returned.

Parameters ****traits** (*args,) – arguments and keywords to be passed on to the editor factory's constructor.

class traitsui.editors.color_editor.**ToolkitEditorFactory** (*args, **traits)
 Bases: *traitsui.editor_factory.EditorFactory*
 Editor factory for color editors.

traitsui.editors.compound_editor module

Defines the compound editor factory for all traits toolkit backends.

traitsui.editors.compound_editor.**CompoundEditor**
alias of *ToolkitEditorFactory*

class traitsui.editors.compound_editor.**ToolkitEditorFactory** (*args, **traits)
Bases: *traitsui.editor_factory.EditorFactory*
Editor factory for compound editors.

traitsui.editors.csv_list_editor module

This modules defines CSVListEditor.

A CSVListEditor provides an editor for lists of simple data types. It allows the user to edit the list in a text field, using commas (or optionally some other character) to separate the elements.

class traitsui.editors.csv_list_editor.**CSVListEditor** (*args, **traits)
Bases: *traitsui.editors.text_editor.ToolkitEditorFactory*

A text editor for a List.

This editor provides a single line of input text of comma separated values. (Actually, the default separator is a comma, but this can be changed.) The editor can only be used with List traits whose inner trait is one of Int, Float, Str, Enum, or Range.

The ‘simple’, ‘text’, ‘custom’ and readonly styles are based on TextEditor. The ‘readonly’ style provides the same formatting in the text field as the other editors, but the user cannot change the value.

Like other Traits editors, the background of the text field will turn red if the user enters an incorrectly formatted list or if the values do not match the type of the inner trait. This validation only occurs while editing the text field. If, for example, the inner trait is Range(low=‘lower’, high=‘upper’), a change in ‘upper’ will not trigger the validation code of the editor.

The editor removes whitespace of entered items with strip(), so for Str types, the editor should not be used if whitespace at the beginning or end of the string must be preserved.

Parameters

- **sep** (*str* or *None*, *optional*) – The separator of the values in the list. If None, each contiguous sequence of whitespace is a separator. Default is ‘,’.
- **ignore_trailing_sep** (*bool*, *optional*) – If this is False, a line containing a trailing separator is invalid. Default is True.
- **auto_set** (*bool*) – If True, then every keystroke sets the value of the trait.
- **enter_set** (*bool*) – If True, the user input sets the value when the Enter key is pressed.

Example

The following will display a window containing a single input field. Entering, say, ‘0, .5, 1’ in this field will result in the list `x = [0.0, 0.5, 1.0]`.

custom_editor (*ui*, *object*, *name*, *description*, *parent*)
Generates an editor using the “custom” style.

readonly_editor (*ui, object, name, description, parent*)

Generates an “editor” that is read-only.

simple_editor (*ui, object, name, description, parent*)

Generates an editor using the “simple” style.

text_editor (*ui, object, name, description, parent*)

Generates an editor using the “text” style.

traitsui.editors.custom_editor module

Defines the editor factory used to wrap a non-Traits based custom control.

traitsui.editors.custom_editor.**CustomEditor**

alias of *ToolkitEditorFactory*

class traitsui.editors.custom_editor.**ToolkitEditorFactory** (**args, **traits*)

Bases: *traitsui.basic_editor_factory.BasicEditorFactory*

Editor factory for custom editors.

traitsui.editors.date_editor module

A Traits UI editor that wraps a WX calendar panel.

class traitsui.editors.date_editor.**DateEditor** (**args, **traits*)

Bases: *traitsui.editor_factory.EditorFactory*

Editor factory for date/time editors.

traitsui.editors.default_override module

Editor factory that overrides certain attributes of the default editor.

For example, the default editor for Range(low=0, high=1500) has ‘1500’ as the upper label. To change it to ‘Max’ instead, use

```
my_range = Range(low=0, high=1500, editor=DefaultOverride(high_label='Max'))
```

Alternatively, the override can also be specified in the view:

```
View(Item('my_range', editor=DefaultOverride(high_label='Max')))
```

class traitsui.editors.default_override.**DefaultOverride** (**args, **overrides*)

Bases: *traitsui.editor_factory.EditorFactory*

Editor factory for selectively overriding certain parameters of the default editor.

custom_editor (*ui, object, name, description, parent*)

readonly_editor (*ui, object, name, description, parent*)

simple_editor (*ui, object, name, description, parent*)

text_editor (*ui, object, name, description, parent*)

traitsui.editors.directory_editor module

Defines the directory editor factory for all traits toolkit backends.

traitsui.editors.directory_editor.**DirectoryEditor**
alias of *ToolkitEditorFactory*

class traitsui.editors.directory_editor.**ToolkitEditorFactory** (*args, **traits)
Bases: *traitsui.editors.file_editor.ToolkitEditorFactory*
Editor factory for directory editors.

traitsui.editors.dnd_editor module

Defines the editor factory for a drag-and-drop editor. A drag-and-drop editor represents its value as a simple image which, depending upon the editor style, can be a drag source only, a drop target only, or both a drag source and a drop target.

traitsui.editors.dnd_editor.**DNDEditor**
alias of *ToolkitEditorFactory*

class traitsui.editors.dnd_editor.**ToolkitEditorFactory** (*args, **traits)
Bases: *traitsui.editor_factory.EditorFactory*
Editor factory for drag-and-drop editors.

traitsui.editors.drop_editor module

Defines a drop editor factory for all traits toolkit backends. A drop target editor handles drag and drop operations as a drop target.

traitsui.editors.drop_editor.**DropEditor**
alias of *ToolkitEditorFactory*

class traitsui.editors.drop_editor.**ToolkitEditorFactory** (*args, **traits)
Bases: *traitsui.editors.text_editor.ToolkitEditorFactory*
Editor factory for drop editors.

traitsui.editors.enum_editor module

Defines the editor factory for single-selection enumerations, for all traits user interface toolkits.

traitsui.editors.enum_editor.**EnumEditor**
alias of *ToolkitEditorFactory*

class traitsui.editors.enum_editor.**ToolkitEditorFactory** (*args, **traits)
Bases: *traitsui.editor_factory.EditorWithListFactory*
Editor factory for enumeration editors.

traitsui.editors.file_editor module

Defines the file editor factory for all traits toolkit backends.

`traitsui.editors.file_editor.FileEditor`
 alias of *ToolkitEditorFactory*

class `traitsui.editors.file_editor.ToolkitEditorFactory(*args, **traits)`
 Bases: *traitsui.editors.text_editor.ToolkitEditorFactory*
 Editor factory for file editors.

traitsui.editors.font_editor module

Defines the font editor factory for all traits user interface toolkits.

`traitsui.editors.font_editor.FontEditor(*args, **traits)`
 Returns an instance of the toolkit-specific editor factory declared in `traitsui.<toolkit>.font_editor`. If such an editor factory cannot be located, an instance of the abstract `ToolkitEditorFactory` declared in `traitsui.editors.font_editor` is returned.

Parameters ****traits** (**args,*) – arguments and keywords to be passed on to the editor factory’s constructor.

class `traitsui.editors.font_editor.ToolkitEditorFactory(*args, **traits)`
 Bases: *traitsui.editor_factory.EditorFactory*
 Editor factory for font editors.

traitsui.editors.history_editor module

Defines a text editor which displays a text field and maintains a history of previously entered values.

class `traitsui.editors.history_editor.ToolkitEditorFactory(*args, **traits)`
 Bases: *traitsui.basic_editor_factory.BasicEditorFactory*
`traitsui.editors.history_editor.history_editor(*args, **traits)`

traitsui.editors.html_editor module

Defines the HTML editor factory. HTML editors interpret and display HTML-formatted text, but do not modify it.

class `traitsui.editors.html_editor.ToolkitEditorFactory(*args, **traits)`
 Bases: *traitsui.basic_editor_factory.BasicEditorFactory*

Editor factory for HTML editors.

indent (*line*)

Calculates the amount of white space at the beginning of a line.

parse_block (*lines, i*)

Parses a code block.

parse_list (*lines, i*)

Parses a list.

parse_text (*text*)

Parses the contents of a formatted text string into the corresponding HTML.

`traitsui.editors.html_editor.html_editor(*args, **traits)`

traitsui.editors.image_editor module

Traits UI ‘display only’ image editor.

```
class traitsui.editors.image_editor.ImageEditor (*args, **traits)
    Bases: traitsui.basic_editor_factory.BasicEditorFactory
```

traitsui.editors.image_enum_editor module

Defines the image enumeration editor factory for all traits user interface toolkits.

```
traitsui.editors.image_enum_editor.ImageEnumEditor
    alias of ToolkitEditorFactory

class traitsui.editors.image_enum_editor.ToolkitEditorFactory (*args, **traits)
    Bases: traitsui.editors.enum_editor.ToolkitEditorFactory

    Editor factory for image enumeration editors.

    init ()
        Performs any initialization needed after all constructor traits have been set.
```

traitsui.editors.instance_editor module

Defines the instance editor factory for all traits user interface toolkits.

```
traitsui.editors.instance_editor.InstanceEditor
    alias of ToolkitEditorFactory

class traitsui.editors.instance_editor.ToolkitEditorFactory (*args, **traits)
    Bases: traitsui.editor_factory.EditorFactory

    Editor factory for instance editors.
```

traitsui.editors.key_binding_editor module

Defines the key binding editor for use with the KeyBinding class. This is a specialized editor used to associate a particular key with a control (i.e., the key binding editor).

```
traitsui.editors.key_binding_editor.key_binding_editor (*args, **traits)
```

traitsui.editors.list_editor module

Defines the list editor factory for the traits user interface toolkits..

```
traitsui.editors.list_editor.ListEditor
    alias of ToolkitEditorFactory

class traitsui.editors.list_editor.ListItemProxy (object, name, index, trait, value)
    Bases: traits.has_traits.HasTraits

class traitsui.editors.list_editor.ToolkitEditorFactory (*args, **traits)
    Bases: traitsui.editor_factory.EditorFactory

    Editor factory for list editors.
```

traitsui.editors.list_str_editor module

Traits UI editor factory for editing lists of strings.

```
class traitsui.editors.list_str_editor.ListStrEditor(*args, **traits)
    Bases: traitsui.basic_editor_factory.BasicEditorFactory
    Editor factory for list of string editors.
```

traitsui.editors.null_editor module

Defines a completely empty editor, intended to be used as a spacer.

```
traitsui.editors.null_editor.null_editor(*args, **traits)
```

traitsui.editors.popup_editor module

```
class traitsui.editors.popup_editor.PopupEditor(*args, **traits)
    Bases: traitsui.basic_editor_factory.BasicEditorFactory
```

traitsui.editors.progress_editor module

Defines the progress editor factory for all traits toolkit backends,

```
traitsui.editors.progress_editor.ProgressEditor
    alias of ToolkitEditorFactory
```

```
class traitsui.editors.progress_editor.ToolkitEditorFactory(*args, **traits)
    Bases: traitsui.editor_factory.EditorFactory
    Editor factory for code editors.
```

traitsui.editors.range_editor module

Defines the range editor factory for all traits user interface toolkits.

```
traitsui.editors.range_editor.RangeEditor
    alias of ToolkitEditorFactory
```

```
class traitsui.editors.range_editor.ToolkitEditorFactory(*args, **traits)
    Bases: traitsui.editor_factory.EditorFactory
    Editor factory for range editors.
```

```
custom_editor(ui, object, name, description, parent)
```

Generates an editor using the “custom” style. Overridden to set the values of the `_low_value`, `_high_value` and `is_float` traits.

```
init(handler=None)
```

Performs any initialization needed after all constructor traits have been set.

```
simple_editor(ui, object, name, description, parent)
```

Generates an editor using the “simple” style. Overridden to set the values of the `_low_value`, `_high_value` and `is_float` traits.

traitsui.editors.rgb_color_editor module

Defines a subclass of the base color editor factory, for colors that are represented as tuples of the form (*red*, *green*, *blue*), where *red*, *green* and *blue* are floats in the range from 0.0 to 1.0.

`traitsui.editors.rgb_color_editor.RGBColorEditor(*args, **traits)`

Returns an instance of the toolkit-specific editor factory declared in `traitsui.<toolkit>.rgb_color_editor`. If such an editor factory cannot be located, an instance of the abstract `ToolkitEditorFactory` declared in `traitsui.editors.rgb_color_editor` is returned.

Parameters ****traits** (**args,*) – arguments and keywords to be passed on to the editor factory's constructor.

class `traitsui.editors.rgb_color_editor.ToolkitEditorFactory(*args, **traits)`

Bases: `traitsui.editors.color_editor.ToolkitEditorFactory`

Factory for editors for RGB colors.

traitsui.editors.scrubber_editor module

Editor factory for scrubber-based integer or float value editors.

class `traitsui.editors.scrubber_editor.ScrubberEditor(*args, **traits)`

Bases: `traitsui.basic_editor_factory.BasicEditorFactory`

traitsui.editors.search_editor module

A single line text widget that supports functionality common to native search widgets.

class `traitsui.editors.search_editor.SearchEditor(*args, **traits)`

Bases: `traitsui.basic_editor_factory.BasicEditorFactory`

A single line text widget that supports functionality common to native search widgets.

traitsui.editors.set_editor module

Defines the set editor factory for all traits user interface toolkits.

`traitsui.editors.set_editor.SetEditor`

alias of `ToolkitEditorFactory`

class `traitsui.editors.set_editor.ToolkitEditorFactory(*args, **traits)`

Bases: `traitsui.editor_factory.EditorWithListFactory`

Editor factory for editors for sets.

traitsui.editors.shell_editor module

Editor that displays an interactive Python shell.

`traitsui.editors.shell_editor.ShellEditor`

alias of `ToolkitEditorFactory`

class `traitsui.editors.shell_editor.ToolkitEditorFactory(*args, **traits)`

Bases: `traitsui.basic_editor_factory.BasicEditorFactory`

traitsui.editors.styled_date_editor module

```
class traitsui.editors.styled_date_editor.CellFormat (**args)
    Bases: object

    Encapsulates some common visual attributes to set on the cells of a calendar widget. All attributes default to
    None, which means that they will not override the existing values of the calendar widget.

    bgcolor = None
    bold = None
    fgcolor = None
    italics = None
    underline = None

traitsui.editors.styled_date_editor.StyledDateEditor
    alias of ToolkitEditorFactory

class traitsui.editors.styled_date_editor.ToolkitEditorFactory (*args, **traits)
    Bases: traitsui.editors.date_editor.DateEditor

    A DateEditor that can show sets of dates in different styles.
```

traitsui.editors.table_editor module

Defines the table editor factory for all traits user interface toolkits.

```
class traitsui.editors.table_editor.BaseTableEditor
    Bases: object

    Base class for toolkit-specific editors.

    add_to_menu (menu_item)
        Adds a menu item to the menu bar being constructed.

    add_to_toolbar (toolbar_item)
        Adds a toolbar item to the toolbar being constructed.

    can_add_to_menu (action)
        Returns whether the action should be defined in the user interface.

    can_add_to_toolbar (action)
        Returns whether the toolbar action should be defined in the user interface.

    eval_when (condition, object, trait)
        Evaluates a condition within a defined context and sets a specified object trait based on the result, which is
        assumed to be a Boolean.

    perform (action, action_event=None)
        Performs the action described by a specified Action object.

    set_menu_context (selection, object, column)
        Call before creating a context menu for a cell, then set self as the controller for the menu.

class traitsui.editors.table_editor.ReversedList (list)
    Bases: object

    A list whose order is the reverse of its input.
```

index (*value*)

Returns the index of the first occurrence of the specified value in the list.

insert (*index*, *value*)

Inserts a value at a specified index in the list.

`traitsui.editors.table_editor.TableEditor`

alias of *ToolkitEditorFactory*

class `traitsui.editors.table_editor.ToolkitEditorFactory` (**args*, ***traits*)

Bases: *traitsui.editor_factory.EditorFactory*

Editor factory for table editors.

readonly_editor (*ui*, *object*, *name*, *description*, *parent*)

Generates an “editor” that is read-only. Overridden to set the value of the editable trait to False before generating the editor.

traitsui.editors.tabular_editor module

A traits UI editor for editing tabular data (arrays, list of tuples, lists of objects, etc).

class `traitsui.editors.tabular_editor.TabularEditor` (**args*, ***traits*)

Bases: *traitsui.basic_editor_factory.BasicEditorFactory*

Editor factory for tabular editors.

traitsui.editors.text_editor module

Defines the text editor factory for all traits toolkit backends.

`traitsui.editors.text_editor.TextEditor`

alias of *ToolkitEditorFactory*

class `traitsui.editors.text_editor.ToolkitEditorFactory` (**args*, ***traits*)

Bases: *traitsui.editor_factory.EditorFactory*

Editor factory for text editors.

traitsui.editors.time_editor module

A Traits UI editor that wraps a WX timer control.

class `traitsui.editors.time_editor.TimeEditor` (**args*, ***traits*)

Bases: *traitsui.editor_factory.EditorFactory*

Editor factory for time editors. Generates `_TimeEditor()`s.

traitsui.editors.title_editor module

Defines the title editor factory for all traits toolkit backends.

`traitsui.editors.title_editor.TitleEditor`

alias of *ToolkitEditorFactory*

```
class traitsui.editors.title_editor.ToolkitEditorFactory (*args, **traits)
    Bases: traitsui.editor_factory.EditorFactory

    Editor factory for Title editors.
```

traitsui.editors.tree_editor module

Defines the tree editor factory for all traits user interface toolkits.

```
class traitsui.editors.tree_editor.ToolkitEditorFactory (*args, **traits)
    Bases: traitsui.editor_factory.EditorFactory

    Editor factory for tree editors.

activated = Str
    The optional extended trait name of the trait that should be assigned a node object when a tree node is
    activated, by double-clicking or pressing the Enter key when a node has focus (Note: if you want to
    receive repeated activated events on the same node, make sure the trait is defined as an Event):

alternating_row_colors = Bool(False)
    Whether to alternate row colors or not.

auto_open = Int
    Number of tree levels (down from the root) that should be automatically opened

click = Str
    The optional extended trait name of the trait that should be assigned a node object when a tree node is
    clicked on (Note: If you want to receive repeated clicks on the same node, make sure the trait is defined as
    an Event):

column_headers = List(Str)
    The column header labels if any.

dclick = Str
    The optional extended trait name of the trait that should be assigned a node object when a tree node is
    double-clicked on (Note: if you want to receive repeated double-clicks on the same node, make sure the
    trait is defined as an Event):

dock_theme = Instance(DockWindowTheme)
    The DockWindow graphical theme

editable = Bool(True)
    Are the individual nodes editable?

editor = Instance(EditorFactory)
    Reference to a shared object editor

expands_on_dclick = Bool(True)
    Whether or not to expand on a double-click.

hide_root = Bool(False)
    Hide the tree root node?

icon_size = IconSize
    Size of the tree node icons

lines_mode = Enum('appearance', 'on', 'off')
    Mode for lines connecting tree nodes
    • 'appearance': Show lines only when they look good.
    • 'on': Always show lines.
```

- 'off': Don't show lines.

multi_nodes = Dict

Mapping from `TreeNode` tuples to `MultiTreeNode`s

nodes = List(TreeNode)

Supported `TreeNode` objects

on_activated = Any

Called when a node is activated

on_click = Any

Called when a node is clicked

on_dclick = Any

Called when a node is double-clicked

on_hover = Any

Call when the mouse hovers over a node

on_select = Any

Called when a node is selected

orientation = Orientation

Layout orientation of the tree and the editor

refresh = Str

The optional extended trait name of the trait event that is fired when the application wishes the currently visible portion of the tree widget to repaint itself.

selected = Str

The optional extended trait name of the trait to synchronize with the editor's current selection:

selection_mode = Enum('single', 'extended')

Selection mode.

shared_editor = Bool(False)

Is the editor shared across trees?

show_icons = Bool(True)

Show icons for tree nodes?

vertical_padding = Int(0)

Any extra vertical padding to add.

veto = Str

The optional extended trait name of the trait event that is fired whenever the application wishes to veto a tree action in progress (e.g. double-clicking a non-leaf tree node normally opens or closes the node, but if you are handling the double-click event in your program, you may wish to veto the open or close operation). Be sure to fire the veto event in the event handler triggered by the operation (e.g. the 'dclick' event handler).

word_wrap = Bool(False)

Whether the labels should be wrapped around, if not an ellipsis is shown This works only in the qt backend and if there is only one column in tree

`traitsui.editors.tree_editor.TreeEditor`

Define the `TreeEditor` class.

alias of *ToolkitEditorFactory*

traitsui.editors.tuple_editor module

Defines the tuple editor factory for all traits user interface toolkits.

```
class traitsui.editors.tuple_editor.SimpleEditor (parent, **traits)
    Bases: traitsui.editor.Editor

    Simple style of editor for tuples.

    The editor displays an editor for each of the fields in the tuple, based on the type of each field.

    get_error_control ()
        Returns the editor's control for indicating error status.

    init (parent)
        Finishes initializing the editor by creating the underlying toolkit widget.

    update_editor ()
        Updates the editor when the object trait changes external to the editor.

class traitsui.editors.tuple_editor.ToolkitEditorFactory (*args, **traits)
    Bases: traitsui.editor_factory.EditorFactory

    Editor factory for tuple editors.

traitsui.editors.tuple_editor.TupleEditor
    alias of ToolkitEditorFactory

class traitsui.editors.tuple_editor.TupleStructure (editor)
    Bases: traits.has_traits.HasTraits

    Creates a view containing items for each field in a tuple.
```

traitsui.editors.value_editor module

Defines the tree-based Python value editor and the value editor factory.

```
class traitsui.editors.value_editor.ToolkitEditorFactory (*args, **traits)
    Bases: traitsui.editor_factory.EditorFactory

    Editor factory for tree-based value editors.

traitsui.editors.value_editor.ValueEditor
    alias of ToolkitEditorFactory
```

Module contents

traitsui.extras package

Submodules

traitsui.extras.api module

traitsui.extras.checkbox_column module

traitsui.extras.demo module

traitsui.extras.edit_column module

Defines the table column descriptor used for editing the object represented by the row

```
class traitsui.extras.edit_column.EditColumn (**traits)
```

Bases: *traitsui.table_column.ObjectColumn*

```
get_cell_color (object)
```

Returns the cell background color for the column for a specified object.

```
is_editable (object)
```

Returns whether the column is editable for a specified object.

traitsui.extras.progress_column module

traitsui.extras.saving module

Provides a lightweight framework that removes some of the drudge work involved with implementing user-friendly saving behavior in a Traits UI application.

```
class traitsui.extras.saving.CanSaveMixin
```

Bases: *traits.has_traits.HasTraits*

A mixin-class for objects that wish to support GUI saving via a *SaveHandler*. It is the responsibility of the child class to manage its dirty flag, which describes whether its information has changed since its last save.

```
save ()
```

Saves the object to the path specified by its 'filepath' trait. This method should also reset the dirty flag on this object.

```
validate ()
```

Returns whether the information in the object is valid to be saved in tuple form. The first item is the validation state (boolean) and the second item is the message to display if the object did not validate.

By default, an object always validates.

```
class traitsui.extras.saving.SaveHandler
```

Bases: *traitsui.handler.Handler*

A Handler that facilitates adding saving to a Traits UI application.

```
close (info, is_ok)
```

Called when the user requests to close the interface. Returns a boolean indicating whether the window should be allowed to close.

closed (*info, is_ok*)

Called after the window is destroyed. Makes sure that the autosave timer is stopped.

exit (*info*)

Closes the UI unless a save prompt is cancelled. Provided for convenience to be used with a Menu action.

init (*info*)

Set the default save object (the object being handled). Also, perform a questionable hack by which we remove the handled object from the keybinding's controllers. This means that a keybinding to 'save' only calls this object, not the object being edited as well. (For reasons unclear, the KeyBinding handler API is radically different from the Action API, which is the reason that this problem exists. Keybindings are a UI concept—they should *not* call the model by default.)

promptForSave (*info, cancel=True*)

Prompts the user to save the object, if appropriate. Returns whether the user canceled the action that invoked this prompt.

save (*info*)

Saves the object to its current filepath. If this is not specified, opens a dialog to select this path. Returns whether the save actually occurred.

saveAs (*info*)

Saves the object to a new path, and sets this as the 'filepath' on the object. Returns whether the save actually occurred.

Module contents

traitsui.image package

Submodules

traitsui.image.image module

Module contents

traitsui.null package

Submodules

traitsui.null.color_trait module

Trait definition for a null-based (i.e., no UI) color.

`traitsui.null.color_trait.convert_to_color` (*object, name, value*)

Converts a number into a wxColour object.

`traitsui.null.color_trait.get_color_editor` (**args, **traits*)

traitsui.null.font_trait module

Trait definition for a null-based (i.e., no UI) font.

class `traitsui.null.font_trait.TraitFont`

Bases: `traits.trait_handlers.TraitHandler`

Ensures that values assigned to a trait attribute are valid font descriptor strings; the value actually assigned is the corresponding canonical font descriptor string.

info()

validate(*object, name, value*)

Validates that the value is a valid font descriptor string.

`traitsui.null.font_trait.get_font_editor(*args, **traits)`

traitsui.null.rgb_color_trait module

Trait definitions for an RGB-based color, which is a tuple of the form (*red, green, blue*), where *red*, *green* and *blue* are floats in the range from 0.0 to 1.0.

`traitsui.null.rgb_color_trait.convert_to_color(object, name, value)`

Converts a tuple or an integer to an RGB color value, or raises a `TraitError` if that is not possible.

`traitsui.null.rgb_color_trait.get_rgb_color_editor(*args, **traits)`

`traitsui.null.rgb_color_trait.range_check(value)`

Checks that *value* can be converted to a value in the range 0.0 to 1.0.

If so, it returns the floating point value; otherwise, it raises a `TraitError`.

traitsui.null.toolkit module

Defines the concrete implementations of the traits Toolkit interface for the ‘null’ (do nothing) user interface toolkit.

class `traitsui.null.toolkit.GUIToolkit`(*package, toolkit, *packages, **traits*)

Bases: `traitsui.toolkit.Toolkit`

color_trait(**args, **traits*)

constants(**args, **traits*)

font_trait(**args, **traits*)

kiva_font_trait(**args, **traits*)

rgb_color_trait(**args, **traits*)

Module contents

Define the concrete implementations of the traits Toolkit interface for the ‘null’ (do nothing) user interface toolkit. This toolkit is provided to handle situations where no recognized traits-compatible UI toolkit is installed, but users still want to use traits for non-UI related tasks.

traitsui.tests package

Subpackages

traitsui.tests.editors package

Submodules

traitsui.tests.editors.test_button_editor module

traitsui.tests.editors.test_code_editor module

traitsui.tests.editors.test_csv_editor module

traitsui.tests.editors.test_default_override module

traitsui.tests.editors.test_instance_editor module

traitsui.tests.editors.test_liststr_editor module

Test case for ListStrEditor and ListStrAdapter

```
class traitsui.tests.editors.test_liststr_editor.TraitObject
    Bases: traits.has_traits.HasTraits
traitsui.tests.editors.test_liststr_editor.test_list_str_adapter_length()
    Test the ListStringAdapter len method
```

`traitsui.tests.editors.test_liststr_editor_selection` module

`traitsui.tests.editors.test_range_editor_spinner` module

`traitsui.tests.editors.test_range_editor_text` module

`traitsui.tests.editors.test_table_editor` module

`traitsui.tests.editors.test_tabular_editor` module

`traitsui.tests.editors.test_tree_editor` module

`traitsui.tests.editors.test_tuple_editor` module

Module contents

`traitsui.tests.null_backend` package

Submodules

`traitsui.tests.null_backend.test_font_trait` module

`traitsui.tests.null_backend.test_null_toolkit` module

Module contents

`traitsui.tests.ui_editors` package

Submodules

`traitsui.tests.ui_editors.test_data_frame_editor` module

Module contents

Submodules

`traitsui.tests.test_actions` module

`traitsui.tests.test_color_column` module

`traitsui.tests.test_controller` module

`traitsui.tests.test_handler` module

```
class traitsui.tests.test_handler.PyfaceAction
    Bases: traitsui.menu.Action
```

```

    perform(event)
class traitsui.tests.test_handler.SampleHandler
    Bases: traitsui.handler.Handler
    action_handler()
    apply(info)
    info_action_handler(info)
    revert(info)
    show_help(info, control=None)
class traitsui.tests.test_handler.SampleObject
    Bases: traits.has_traits.HasTraits
    action_handler()
    info_action_handler(info)
    object_action_handler()
class traitsui.tests.test_handler.TestHandler(methodName='runTest')
    Bases: unittest.case.TestCase
    test_close_handler()
    test_help_handler()
    test_perform_action_handler()
    test_perform_click_handler()
    test_perform_info_action_handler()
    test_perform_object_handler()
    test_perform_pyface_action()
    test_perform_traitsui_action()
    test_redo_handler()
    test_revert_handler()
    test_undo_handler()
class traitsui.tests.test_handler.TraitsUIAction
    Bases: traitsui.menu.Action
    perform()

```

traitsui.tests.test_labels module

traitsui.tests.test_layout module

traitsui.tests.test_regression module

General regression tests for various fixed bugs.

```

class traitsui.tests.test_regression.Child
    Bases: traits.has_traits.HasTraits

```

```
class traitsui.tests.test_regression.Parent
    Bases: traits.has_traits.HasTraits

class traitsui.tests.test_regression.TestRegression (methodName='runTest')
    Bases: unittest.case.TestCase

    test_attribute_error ()
        Make sure genuine AttributeErrors raise on Editor creation.

    test_editor_on_delegates_to_event ()
        Make sure that DelegatesTo on Events passes Editor creation.
```

traitsui.tests.test_shadow_group module

Tests for the ShadowGroup class.

```
class traitsui.tests.test_shadow_group.TestShadowGroup (methodName='runTest')
    Bases: unittest.case.TestCase

    test_creation_sets_shadow_first ()
```

traitsui.tests.test_splitter_prefs_restored module

traitsui.tests.test_toolkit module

```
class traitsui.tests.test_toolkit.TestToolkit (methodName='runTest')
    Bases: unittest.case.TestCase

    test_default_toolkit ()

    test_nonexistent_toolkit ()

    test_nonstandard_toolkit ()
```

```
traitsui.tests.test_toolkit.clear_toolkit (*args, **kwds)
    If a toolkit has been selected, clear it, resetting on exit
```

traitsui.tests.test_tuple_editor module

traitsui.tests.test_ui module

traitsui.tests.test_visible_when_layout module

Module contents

traitsui.ui_editors package

Submodules

traitsui.ui_editors.array_view_editor module

Defines an ArrayViewEditor for displaying 1-d or 2-d arrays of values.

```

class traitsui.ui_editors.array_view_editor.ArrayViewAdapter
    Bases: traitsui.tabular_adapter.TabularAdapter

    get_item(object, trait, row)
        Returns the value of the object.trait[row] item.

    len(object, trait)
        Returns the number of items in the specified object.trait list.

class traitsui.ui_editors.array_view_editor.ArrayViewEditor(*args, **traits)
    Bases: traitsui.basic_editor_factory.BasicEditorFactory

```

traitsui.ui_editors.data_frame_editor module

```

class traitsui.ui_editors.data_frame_editor.DataFrameAdapter
    Bases: traitsui.tabular_adapter.TabularAdapter

    Generic tabular adapter for data frames

    alignment = Property(Enum('left', 'center', 'right'))
        The alignment for each cell

    delete(object, trait, row)
        Override the base implementation to work with DataFrames

        Unavoidably does a copy of the data, setting the trait with the new value.

    font = Property
        The font to use for each column

    format = Property
        The format to use for each column

    get_item(object, trait, row)
        Override the base implementation to work with DataFrames

        This returns a dataframe with one row, rather than a series, since using a dataframe preserves dtypes.

    index_alignment = Property
        The alignment to use for a row index.

    index_text = Property
        The text to use for a row index.

    insert(object, trait, row, value)
        Override the base implementation to work with DataFrames

        Unavoidably does a copy of the data, setting the trait with the new value.

    text = Property
        The text to use for a generic entry.

class traitsui.ui_editors.data_frame_editor.DataFrameEditor(*args, **traits)
    Bases: traitsui.basic_editor_factory.BasicEditorFactory

    Editor factory for basic data frame editor

    columns = List()
        Optional list of either column ID or pairs of (column title, column ID).

    editable = Bool(False)
        Whether or not the entries can be edited.

```

```
fonts = Either(Font, Dict, default='Courier 10')
```

The font for each element, or a mapping column ID to font.

```
formats = Either(Str, Dict, default='%s')
```

The format for each element, or a mapping column ID to format.

```
klass = Property
```

The editor implementation class.

```
show_index = Bool(True)
```

Should an index column be displayed.

```
show_titles = Bool(True)
```

Should column headers be displayed.

Module contents

2.1.2 Submodules

traitsui.api module

Exports the symbols defined by the traits.ui package.

```
traitsui.api.raise_to_debug()
```

When we would otherwise silently swallow an exception, call this instead to allow people to set the TRAITS_DEBUG environment variable and get the exception.

traitsui.base_panel module

```
class traitsui.base_panel.BasePanel
```

Bases: `pyface.action.action_controller.ActionController`

Base class for Traits UI panels and dialog boxes.

Concrete subclasses of BasePanel are the Python-side owners of the top-level toolkit control for a UI. They also implement the Pyface ActionController API for menu and toolbar action handling.

```
add_to_menu (menu_item)
```

Adds a menu item to the menu bar being constructed.

The bulk of the back-end work is done in Pyface. This code is simply responsible for hooking up radio groups, checkboxes, and enabled status.

This routine is also used to add items to the toolbar, as logic and APIs are identical.

Parameters `menu_item` (*toolkit MenuItem*) – The Pyface toolkit-level item to add to the menu.

```
add_to_toolbar (toolbar_item)
```

Adds a menu item to the menu bar being constructed.

The bulk of the back-end work is done in Pyface. This code is simply responsible for hooking up radio groups, checkboxes, and enabled status.

This simply calls the analogous menu as logic and APIs are identical.

Parameters `toolbar_item` (*toolkit Tool*) – The Pyface toolkit-level item to add to the toolbar.

can_add_to_menu (*action*)

Should the toolbar action be defined in the user interface.

This simply calls the analogous menu as logic and APIs are identical.

Parameters *action* (*Action*) – The Action to add to the toolbar.

Returns *defined* – Whether or not the action should be added to the menu.

Return type *bool*

can_add_to_toolbar (*action*)

Should the toolbar action be defined in the user interface.

This simply calls the analogous menu as logic and APIs are identical.

Parameters *action* (*Action*) – The Action to add to the toolbar.

Returns *defined* – Whether or not the action should be added to the toolbar.

Return type *bool*

check_button (*buttons, action*)

Adds *action* to the system buttons list for this dialog, if it is not already in the list.

coerce_button (*action*)

Coerces a string to an Action if necessary.

control = *Any*

The top-level toolkit control of the UI.

default_icon ()

Return a default icon for a TraitsUI dialog.

is_button (*action, name*)

Returns whether a specified action button is a system button.

perform (*action, event*)

Dispatches the action to be handled by the handler.

Parameters

- **action** (*Action instance*) – The action to perform.
- **event** (*ActionEvent instance*) – The event that triggered the action.

Returns *result* – The result of the action's perform method (usually None).

Return type *any*

ui = *Instance('traitsui.ui.UI')*

The UI instance for the view.

traitsui.basic_editor_factory module

Defines the BasicEditorFactory class, which allows creating editor factories that use the same class for creating all editor styles.

class traitsui.basic_editor_factory.**BasicEditorFactory** (**args, **traits*)

Bases: *traitsui.editor_factory.EditorFactory*

Base class for editor factories that use the same class for creating all editor styles.

traitsui.color_column module

Table column object for RGBColor traits.

```
class traitsui.color_column.ColorColumn
    Bases: traitsui.table_column.ObjectColumn

    Table column object for RGBColor traits.

    get_cell_color (object)
        Returns the cell background color for the column for a specified object.

    get_value (object)
        Gets the value of the column for a specified object.

    style = 'readonly'
        For display by default.
```

traitsui.context_value module

Defines some helper classes and traits used to define ‘bindable’ editor values.

```
traitsui.context_value.CV
    alias of ContextValue

traitsui.context_value.CVType (type)

class traitsui.context_value.ContextValue (name)
    Bases: traits.has_traits.HasPrivateTraits

    Defines the name of a context value that can be bound to some editor value.
```

traitsui.default_dock_window_theme module

traitsui.delegating_handler module

A handler that delegates the handling of events to a set of sub-handlers.

This is typically used as the handler for dynamic views. See the **traits.has_dynamic_view** module.

```
class traitsui.delegating_handler.DelegatingHandler
    Bases: traitsui.handler.Handler

    A handler that delegates the handling of events to a set of sub-handlers.

    closed (info, is_ok)
        Handles the user interface being closed by the user.

        This method is overridden here to unregister any dispatchers that were set up in the init() method.

    init (info)
        Initializes the controls of a user interface.

        This method is called after all user interface elements have been created, but before the user interface is displayed. Use this method to further customize the user interface before it is displayed.

        This method is overridden here to delegate to sub-handlers.

        Parameters info (UIInfo object) – The UIInfo object associated with the view
```

Returns initialized – A boolean, indicating whether the user interface was successfully initialized. A True value indicates that the UI can be displayed; a False value indicates that the display operation should be cancelled.

Return type bool

traitsui.dock_window_theme module

Defines the theme style information for a DockWindow and its components.

class traitsui.dock_window_theme.DockWindowTheme

Bases: traits.has_traits.HasPrivateTraits

Defines the theme style information for a DockWindow and its components.

traitsui.dock_window_theme.dock_window_theme (theme=None)

traitsui.dockable_view_element module

traitsui.editor module

Defines the abstract Editor class, which represents an editing control for an object trait in a Traits-based user interface.

class traitsui.editor.Editor (parent, **traits)

Bases: traits.has_traits.HasPrivateTraits

Represents an editing control for an object trait in a Traits-based user interface.

dispose ()

Disposes of the contents of an editor.

error (excp)

Handles an error that occurs while setting the object's trait value.

get_undo_item (object, name, old_value, new_value)

Creates an undo history entry.

init (parent)

Finishes initializing the editor by creating the underlying toolkit widget.

log_change (undo_factory, *undo_args)

Logs a change made in the editor.

parse_extended_name (name)

Returns a tuple of the form (context_object, 'name[name...]', callable) for a specified extended name of the form: 'name' or 'context_object_name.name[name...]'.

prepare (parent)

Finishes setting up the editor.

restore_prefs (prefs)

Restores any saved user preference information associated with the editor.

save_prefs ()

Returns any user preference information associated with the editor.

set_focus ()

Assigns focus to the editor's underlying toolkit widget.

string_value (*value*, *format_func*=None)

Returns the text representation of a specified object trait value.

If the **format_func** attribute is set on the editor factory, then this method calls that function to do the formatting. If the **format_str** attribute is set on the editor factory, then this method uses that string for formatting. If neither attribute is set, then this method just calls the built-in `unicode()` function.

sync_value (*user_name*, *editor_name*, *mode*='both', *is_list*=False, *is_event*=False)

Set up synchronization between an editor trait and a user object trait.

Also sets the initial value of the editor trait from the user object trait (for modes 'from' and 'both'), and the initial value of the user object trait from the editor trait (for mode 'to').

Parameters

- **user_name** (*string*) – The name of the trait to be used on the user object. If empty, no synchronization will be set up.
- **editor_name** (*string*) – The name of the relevant editor trait.
- **mode** (*string*, *optional*; one of 'to', 'from' or 'both') – The direction of synchronization. 'from' means that trait changes in the user object should be propagated to the editor. 'to' means that trait changes in the editor should be propagated to the user object. 'both' means changes should be propagated in both directions. The default is 'both'.
- **is_list** (*bool*, *optional*) – If true, synchronization for item events will be set up in addition to the synchronization for the object itself. The default is False.
- **is_event** (*bool*, *optional*) – If true, this method won't attempt to initialize the user object or editor trait values. The default is False.

update_editor ()

Updates the editor when the object trait changes externally to the editor.

traitsui.editor_factory module

Defines the abstract `EditorFactory` class, which represents a factory for creating the `Editor` objects used in a Traits-based user interface.

class `traitsui.editor_factory.EditorFactory` (*args, **traits)

Bases: `traits.has_traits.HasPrivateTraits`

Represents a factory for creating the `Editor` objects in a Traits-based user interface.

custom_editor (*ui*, *object*, *name*, *description*, *parent*)

Generates an editor using the "custom" style.

init ()

Performs any initialization needed after all constructor traits have been set.

named_value (*name*, *ui*)

Returns the value of a specified extended name of the form: `name` or `context_object_name.name[name...]`:

readonly_editor (*ui*, *object*, *name*, *description*, *parent*)

Generates an "editor" that is read-only.

simple_editor (*ui*, *object*, *name*, *description*, *parent*)

Generates an editor using the "simple" style.

text_editor (*ui, object, name, description, parent*)
 Generates an editor using the “text” style.

class traitsui.editor_factory.**EditorWithListFactory** (*args, **traits)
 Bases: *traitsui.editor_factory.EditorFactory*
 Base class for factories of editors for objects that contain lists.

traitsui.editors_gen module

Generates a file containing definitions for editors defined in the various backends.

traitsui.editors_gen.**gen_editor_definitions** (*target_filename='editors.py'*)

Generates a file containing definitions for editors defined in the various backends.

The idea is that if a new editor has been declared in any of the backends, the author needs to create a file called ‘<myeditor>_definition’ in the Traits package (in traitsui). This function will be run each time the user runs the setup.py file, and the new editor’s definition will be appended to the editors.py file.

The structure of the <myeditor>_definition file should be as follows:

```
myeditor_definition = '<file name in the backend package>:
    <name of the Editor or the EditorFactory class'
```

traitsui.file_dialog module

traitsui.group module

Defines the Group class used to represent a group of items used in a Traits-based user interface.

class traitsui.group.**Group** (*values, **traits)
 Bases: *traitsui.view_element.ViewSubElement*

Represents a grouping of items in a user interface view.

get_label (*ui*)
 Gets the label to use this group.

get_shadow (*ui*)
 Returns a ShadowGroup object for the current Group object, which recursively resolves all embedded Include objects and which replaces each embedded Group object with a corresponding ShadowGroup.

is_includable ()
 Returns a Boolean value indicating whether the object is replacable by an Include object.

replace_include (*view_elements*)
 Replaces any items that have an **id** attribute with an Include object with the same ID value, and puts the object with the ID into the specified ViewElements object.

Parameters **view_elements** (*ViewElements object*) – A set of Group, Item, and Include objects

set_container ()
 Sets the correct container for the content.

class traitsui.group.**HFlow** (*values, **traits)
 Bases: *traitsui.group.HGroup*

A group in which items are laid out horizontally, and “wrap” when they exceed the available horizontal space..

```
class traitsui.group.HGroup(*values, **traits)
    Bases: traitsui.group.Group
```

A group whose items are laid out horizontally.

```
class traitsui.group.HSplit(*values, **traits)
    Bases: traitsui.group.Group
```

A horizontal group with splitter bars to separate it from other groups.

```
class traitsui.group.ShadowGroup(shadow, **traits)
    Bases: traitsui.group.Group
```

Corresponds to a Group object, but with all embedded Include objects resolved, and with all embedded Group objects replaced by corresponding ShadowGroup objects.

```
get_content (allow_groups=True)
```

Returns the contents of the Group within a specified context for building a user interface.

This method makes sure that all Group types are of the same type (i.e., Group or Item) and that all Include objects have been replaced by their substituted values.

```
get_id ()
```

Returns an ID for the group.

```
set_container ()
```

Sets the correct container for the content.

```
class traitsui.group.Tabbed(*values, **traits)
    Bases: traitsui.group.Group
```

A group that is shown as a tabbed notebook.

```
class traitsui.group.VFlow(*values, **traits)
    Bases: traitsui.group.VGroup
```

A group in which items are laid out vertically, and “wrap” when they exceed the available vertical space.

```
class traitsui.group.VFold(*values, **traits)
    Bases: traitsui.group.VGroup
```

A group in which items are laid out vertically and can be collapsed (i.e. ‘folded’) by clicking their title.

```
class traitsui.group.VGrid(*values, **traits)
    Bases: traitsui.group.VGroup
```

A group whose items are laid out in 2 columns.

```
class traitsui.group.VGroup(*values, **traits)
    Bases: traitsui.group.Group
```

A group whose items are laid out vertically.

```
class traitsui.group.VSplit(*values, **traits)
    Bases: traitsui.group.Group
```

A vertical group with splitter bars to separate it from other groups.

traitsui.handler module

Defines the Handler class used to manage and control the editing process in a Traits-based user interface.

class traitsui.handler.**Controller** (*model=None, **metadata*)

Bases: *traitsui.handler.Handler*

Defines a handler class which provides a view and controller for a specified model.

This class is used when implementing a standard MVC-based design. The **model** trait contains most, if not all, of the data being viewed, and can be referenced in a Controller instance's View definition using unadorned trait names. (e.g., `Item('name')`).

get_perform_handlers (*info*)

Return a list of objects which can handle actions.

By default this returns the Controller instance and the model.

Parameters *info* (*UIInfo instance or None*) – The UIInfo associated with the view, or None.

Returns *handlers* – A list of objects that may potentially have action methods on them.

Return type list

init_info (*info*)

Informs the handler what the UIInfo object for a View will be.

trait_context ()

Returns the default context to use for editing or configuring traits.

class traitsui.handler.**Handler**

Bases: *traits.has_traits.HasPrivateTraits*

Provides access to and control over the run-time workings of a Traits-based user interface.

apply (*info*)

Handles the **Apply** button being clicked.

can_drop (*info, object*)

Can the specified object be inserted into the view?

can_import (*info, category*)

close (*info, is_ok*)

Handles the user attempting to close a dialog-based user interface.

This method is called when the user attempts to close a window, by clicking an **OK** or **Cancel** button, or clicking a Close control on the window). It is called before the window is actually destroyed. Override this method to perform any checks before closing a window.

While Traits UI handles “OK” and “Cancel” events automatically, you can use the value of the *is_ok* parameter to implement additional behavior.

Parameters

- **info** (*UIInfo object*) – The UIInfo object associated with the view
- **is_ok** (*Boolean*) – Indicates whether the user confirmed the changes (such as by clicking **OK**.)

Returns *allow_close* – A Boolean, indicating whether the window should be allowed to close.

Return type bool

closed (*info, is_ok*)

Handles a dialog-based user interface being closed by the user.

This method is called *after* the window is destroyed. Override this method to perform any clean-up tasks needed by the application.

Parameters

- **info** (*UIInfo object*) – The UIInfo object associated with the view
- **is_ok** (*Boolean*) – Indicates whether the user confirmed the changes (such as by clicking **OK**.)

configure_traits (*filename=None, view=None, kind=None, edit=True, context=None, handler=None, id="", scrollable=None, **args*)
Configures the object's traits.

dock_control_for (*info, parent, object*)
Returns the DockControl object for a specified object.

dock_window_empty (*dock_window*)
Handles a DockWindow becoming empty.

edit_traits (*view=None, parent=None, kind=None, context=None, handler=None, id="", scrollable=None, **args*)
Edits the object's traits.

get_perform_handlers (*info*)
Return a list of objects which can handle actions.

This method may be overridden by sub-classes to return a more relevant set of objects.

Parameters info (*UIInfo instance or None*) – The UIInfo associated with the view, or None.

Returns handlers – A list of objects that may potentially have action methods on them.

Return type list

init (*info*)
Initializes the controls of a user interface.

This method is called after all user interface elements have been created, but before the user interface is displayed. Override this method to customize the user interface before it is displayed.

Parameters info (*UIInfo object*) – The UIInfo object associated with the view

Returns initialized – A Boolean, indicating whether the user interface was successfully initialized. A True value indicates that the UI can be displayed; a False value indicates that the display operation should be cancelled. The default implementation returns True without taking any other action.

Return type bool

init_info (*info*)
Informs the handler what the UIInfo object for a View will be.

This method is called before the UI for the View has been constructed. It is provided so that the handler can save the reference to the UIInfo object in case it exposes viewable traits whose values are properties that depend upon items in the context being edited.

open_view_for (*control, use_mouse=True*)
Creates a new view of a specified control.

perform (*info, action, event*)
Perform computation for an action.

The default method looks for a method matching `action.action` and calls it (sniffing the signature to determine how to call it for historical reasons). If this is not found, then it calls the `perform()` method of the action.

Parameters

- **info** (*UIInfo instance*) – The UIInfo associated with the view, if available.
- **action** (*Action instance*) – The Action that the user invoked.
- **event** (*ActionEvent instance*) – The ActionEvent associated with the user action.

Notes

If overriding in a subclass, the method needs to ensure that any standard menu action items that are needed (eg. “Close”, “Undo”, “Redo”, “Help”, etc.) get dispatched correctly.

position (*info*)

Positions a dialog-based user interface on the display.

This method is called after the user interface is initialized (by calling `init()`), but before the user interface is displayed. Override this method to position the window on the display device. The default implementation calls the `position()` method of the current toolkit.

Usually, you do not need to override this method, because you can control the window’s placement using the **x** and **y** attributes of the View object.

Parameters **info** (*UIInfo object*) – The UIInfo object associated with the window

revert (*info*)

Handles the **Revert** button being clicked.

setattr (*info, object, name, value*)

Handles the user setting a specified object trait’s value.

This method is called when an editor attempts to set a new value for a specified object trait attribute. Use this method to control what happens when a trait editor tries to set an attribute value. For example, you can use this method to record a history of changes, in order to implement an “undo” mechanism. No result is returned. The default implementation simply calls the built-in `setattr()` function. If you override this method, make sure that it actually sets the attribute, either by calling the parent method or by setting the attribute directly

Parameters

- **info** (*UIInfo instance*) – The UIInfo for the current UI
- **object** (*object*) – The object whose attribute is being set
- **name** (*string*) – The name of the attribute being set
- **value** – The value to which the attribute is being set

show_help (*info, control=None*)

Shows the help associated with the view.

This method is called when the user clicks a **Help** button in a Traits user interface. The method calls the global help handler, which might be the default help handler, or might be a custom help handler. See `traitsui.help` for details about the setting the global help handler.

Parameters

- **info** (*UIInfo object*) – The UIInfo object associated with the view
- **control** (*UI control*) – The control that invokes the help dialog box

trait_view_for (*info, view, object, object_name, trait_name*)
 Gets a specified View object.

class traitsui.handler.**ModelView** (*model=None, **metadata*)
 Bases: *traitsui.handler.Controller*

Defines a handler class which provides a view and controller for a specified model.

This class is useful when creating a variant of the standard MVC-based design. A subclass of ModelView reformulates a number of traits on its **model** object as properties on the ModelView subclass itself, usually in order to convert them into a more user-friendly format. In this design, the ModelView subclass supplies not only the view and the controller, but also, in effect, the model (as a set of properties wrapped around the original model). Because of this, the ModelView context dictionary specifies the ModelView instance itself as the special *object* value, and assigns the original model object as the *model* value. Thus, the traits of the ModelView object can be referenced in its View definition using unadorned trait names.

trait_context ()
 Returns the default context to use for editing or configuring traits.

class traitsui.handler.**ViewHandler**
 Bases: *traitsui.handler.Handler*

traitsui.handler.**close_dock_control** (*dock_control*)
 Closes a DockControl (if allowed by the associated Traits UI Handler).

traitsui.handler.**default_handler** (*handler=None*)
 Returns the global default handler.

If *handler* is an instance of Handler, this function sets it as the global default handler.

traitsui.help module

Defines the help interface for displaying the help associated with a Traits UI View object.

traitsui.help.**default_show_help** (*info, control*)
 Default handler for showing the help associated with a view.

traitsui.help.**on_help_call** (*new_show_help=None*)
 Sets a new global help provider function.

The help provider function must have a signature of *function*(*info, control)*, where *info* is a UIInfo object for the current view, and *control* is the UI control that invokes the function (typically, a **Help** button). It is provided in case the help provider needs to position the help window relative to the **Help** button.

To retrieve the current help provider function, call this function with no arguments.

Parameters *new_show_help* (*function*) – The function to set as the new global help provider

Returns *previous* – The previous global help provider function

Return type callable

traitsui.help.**show_help** (*info, control*)
 Default handler for showing the help associated with a view.

traitsui.help_template module

Defines the HTML help templates used for formatting Traits UI help pages.

```
class traitsui.help_template.HelpTemplate
    Bases: traits.has_traits.HasStrictTraits

    Contains HTML templates for displaying help.

traitsui.help_template.help_template (template=None)
    Gets or sets the current HelpTemplate in use.
```

traitsui.helper module

Defines various helper functions that are useful for creating Traits-based user interfaces.

```
traitsui.helper.commatize (value)
    Formats a specified value as an integer string with embedded commas. For example: commatize( 12345 ) returns
    "12,345".

traitsui.helper.enum_values_changed (values, strfunc=<type 'unicode'>)
    Recomputes the mappings for a new set of enumeration values.

traitsui.helper.user_name_for (name)
    Returns a "user-friendly" name for a specified trait.
```

traitsui.include module

Defines the Include class, which is used to represent a substitutable element within a user interface View.

```
class traitsui.include.Include (id, **traits)
    Bases: traitsui.view_element.ViewSubElement
```

A substitutable user interface element, i.e., a placeholder in a view definition.

When a view object constructs an attribute-editing window, any Include objects within the view definition are replaced with a group or item defined elsewhere in the object's inheritance tree, based on matching of the name of the element. If no matching element is found, the Include object is ignored.

An Include object can reference a group or item attribute on a parent class or on a subclass. For example, the following class contains a view definition that provides for the possibility that a subclass might add "extra" attributes in the middle of the view:

```
class Person (HasTraits):
    name = Str
    age = Int
    person_view = View('name', Include('extra'), 'age', kind='modal')
```

If you directly create an instance of Person, and edit its attributes, the Include object is ignored.

The following class extends Person, and defines a group of "extra" attributes to add to the view defined on Person:

```
class LocatedPerson (Person):
    street = Str
    city = Str
    state = Str
    zip = Int
    extra = Group('street', 'city', 'state', 'zip')
```

The attribute-editing window for an instance of LocatedPerson displays editors for these extra attributes.

traitsui.instance_choice module

Defines the various instance descriptors used by the instance editor and instance editor factory classes.

```
class traitsui.instance_choice.InstanceChoice
    Bases: traitsui.instance_choice.InstanceChoiceItem

    get_name (object=None)
        Returns the name of the item.

    get_object ()
        Returns the object associated with the item.

    is_compatible (object)
        Indicates whether a specified object is compatible with the item.

class traitsui.instance_choice.InstanceChoiceItem
    Bases: traits.has_traits.HasPrivateTraits

    get_name (object=None)
        Returns the name of the item.

    get_object ()
        Returns the object associated with the item.

    get_view ()
        Returns the view associated with the object.

    is_compatible (object)
        Indicates whether a specified object is compatible with the item.

    is_droppable ()
        Indicates whether the item supports drag and drop.

    is_selectable ()
        Indicates whether the item can be selected by the user.

class traitsui.instance_choice.InstanceDropChoice
    Bases: traitsui.instance_choice.InstanceFactoryChoice

class traitsui.instance_choice.InstanceFactoryChoice
    Bases: traitsui.instance_choice.InstanceChoiceItem

    get_name (object=None)
        Returns the name of the item.

    get_object ()
        Returns the object associated with the item.

    is_compatible (object)
        Indicates whether a specified object is compatible with the item.

    is_droppable ()
        Indicates whether the item supports drag and drop.

    is_selectable ()
        Indicates whether the item can be selected by the user.
```

traitsui.item module

Defines the Item class, which is used to represent a single item within a Traits-based user interface.

class traitsui.item.**Custom**(value=None, **traits)
 Bases: *traitsui.item.Item*

An Item using a ‘custom’ style.

class traitsui.item.**Heading**(label, **traits)
 Bases: *traitsui.item.Label*

An item that is a fancy label.

class traitsui.item.**Item**(value=None, **traits)
 Bases: *traitsui.view_element.ViewSubElement*

An element in a Traits-based user interface.

Magic:

- Items are rendered as layout elements if `name` is set to special values:
 - `name=' '`, the item is rendered as a static label
 - `name='_ '`, the item is rendered as a separator
 - `name=' '`, the item is rendered as a 5 pixel spacer
 - `name='23 '` (any number), the item is rendered as a spacer of the size specified (number of pixels)

get_help(ui)
 Gets the help text associated with the Item in a specified UI.

get_id()
 Returns an ID used to identify the item.

get_label(ui)
 Gets the label to use for a specified Item.

If not specified, the label is set as the name of the corresponding trait, replacing ‘_’ with ‘ ’, and capitalizing the first letter (see `user_name_for()`). This is called the *user name*.

Magic:

- if `attr:item.label` is specified, and it begins with ‘...’, the final label is the user name followed by the item label
- if `attr:item.label` is specified, and it ends with ‘...’, the final label is the item label followed by the user name

is_includable()
 Returns a Boolean indicating whether the object is replaceable by an Include object.

is_spacer()
 Returns True if the item represents a spacer or separator.

class traitsui.item.**Label**(label, **traits)
 Bases: *traitsui.item.Item*
 An item that is a label.

class traitsui.item.**Readonly**(value=None, **traits)
 Bases: *traitsui.item.Item*
 An Item using a ‘readonly’ style.

class traitsui.item.**Spring**(value=None, **traits)
 Bases: *traitsui.item.Item*
 An item that is a layout “spring”.

```
class traitsui.item.UCustom (value=None, **traits)
    Bases: traitsui.item.Custom
```

An Item using a ‘custom’ style with no label.

```
class traitsui.item.UItem (value=None, **traits)
    Bases: traitsui.item.Item
```

An Item that has no label.

```
class traitsui.item.UReadonly (value=None, **traits)
    Bases: traitsui.item.Readonly
```

An Item using a ‘readonly’ style with no label.

traitsui.key_bindings module

Defines KeyBinding and KeyBindings classes, which manage the mapping of keystroke events into method calls on controller objects that are supplied by the application.

```
class traitsui.key_bindings.KeyBinding
    Bases: traits.has_traits.HasStrictTraits
```

Binds one or two keystrokes to a method.

```
class traitsui.key_bindings.KeyBindings (*bindings, **traits)
    Bases: traits.has_traits.HasPrivateTraits
```

A set of key bindings.

```
clone ( **traits)
    Returns a clone of the KeyBindings object.
```

```
dispose ()
    Dispose of the object.
```

```
do (event, controllers=[], *args, **kw)
    Processes a keyboard event.
```

```
edit ()
    Edits a possibly hierarchical set of KeyBindings.
```

```
key_binding_for (binding, key_name)
    Returns the current binding for a specified key (if any).
```

```
merge (key_bindings)
    Merges another set of key bindings into this set.
```

traitsui.list_str_adapter module

Defines adapter interfaces for use with the ListStrEditor.

```
class traitsui.list_str_adapter.AnIListStrAdapter
    Bases: traits.has_traits.HasPrivateTraits
```

```
accepts = Bool(True)
    Does the adapter know how to handle the current item or not?
```

```
index = Int
    The index of the current item being adapted.
```

```

is_cacheable = Bool(True)
    Does the value of accepts depend only upon the type of item?

item = Any
    Current item being adapted.

value = Any
    The current value (if any).

class traitsui.list_str_adapter.IListStrAdapter
    Bases: traits.has_traits.Interface

    accepts = Bool
        Does the adapter know how to handle the current item or not?

    index = Int
        The index of the current item being adapted.

    is_cacheable = Bool
        Does the value of accepts depend only upon the type of item?

    item = Any
        Current item being adapted.

    value = Any
        The current value (if any).

class traitsui.list_str_adapter.ListStrAdapter
    Bases: traits.has_traits.HasPrivateTraits

    The base class for adapting list items to values that can be edited by a ListStrEditor.

    adapters = List(IListStrAdapter, update=True)
        List of optional delegated adapters.

    bg_color = Color(None, update=True)
        The default background color for list items.

    cache = Any({})
        Cache of attribute handlers.

    cache_flushed = Event(update=True)
        Event fired when the cache is flushed.

    can_edit = Bool(True)
        Can the text value of each list item be edited.

    default_text = Str
        Specifies the default text for a new list item.

    default_value = Any('')
        Specifies the default value for a new list item.

    delete (object, trait, index)
        Deletes the specified object.trait[index] list item.

    dropped = Enum('after', 'before')
        Specifies where a dropped item should be placed in the list relative to the item it is dropped on.

    even_bg_color = Color(None, update=True)
        The default background color for even list items.

    even_text_color = Color(None, update=True)
        The default text color for even list items.

```

get_bg_color (*object, trait, index*)

Returns the background color for a specified *object.trait[index]* list item. A result of *None* means use the default list item background color.

get_can_drop (*object, trait, index, value*)

Returns whether the specified *value* can be dropped on the specified *object.trait[index]* list item. A value of **True** means the *value* can be dropped; and a value of **False** indicates that it cannot be dropped.

get_can_edit (*object, trait, index*)

Returns whether the user can edit a specified *object.trait[index]* list item. A **True** result indicates the value can be edited, while a **False** result indicates that it cannot be edited.

get_default_bg_color (*object, trait*)

Returns the default background color for the specified *object.trait* list.

get_default_image (*object, trait*)

Returns the default image for the specified *object.trait* list.

get_default_text (*object, trait*)

Returns the default text for the specified *object.trait* list.

get_default_text_color (*object, trait*)

Returns the default text color for the specified *object.trait* list.

get_default_value (*object, trait*)

Returns a new default value for the specified *object.trait* list.

get_drag (*object, trait, index*)

Returns the 'drag' value for a specified *object.trait[index]* list item. A result of *None* means that the item cannot be dragged.

get_dropped (*object, trait, index, value*)

Returns how to handle a specified *value* being dropped on a specified *object.trait[index]* list item. The possible return values are:

'before' Insert the specified *value* before the dropped on item.

'after' Insert the specified *value* after the dropped on item.

get_image (*object, trait, index*)

Returns the name of the image to use for a specified *object.trait[index]* list item. A result of *None* means no image should be used. Otherwise, the result should either be the name of the image, or an *ImageResource* item specifying the image to use.

get_item (*object, trait, index*)

Returns the value of the *object.trait[index]* list item.

get_text (*object, trait, index*)

Returns the text to display for a specified *object.trait[index]* list item.

get_text_color (*object, trait, index*)

Returns the text color for a specified *object.trait[index]* list item. A result of *None* means use the default list item text color.

image = Str(None, update=True)

The name of the default image to use for list items.

index = Int

The index of the current item being adapter.

insert (*object, trait, index, value*)

Inserts a new value at the specified *object.trait[index]* list index.

item = Any
 The current item being adapted.

len (object, trait)
 Returns the number of items in the specified *object.trait* list.

odd_bg_color = Color(None, update=True)
 The default background color for odd list items.

odd_text_color = Color(None, update=True)
 The default text color for odd list items.

set_text (object, trait, index, text)
 Sets the text for a specified *object.trait[index]* list item to *text*.

text_color = Color(None, update=True)
 The default text color for list items.

value = Any
 The current value (if any).

traitsui.menu module

Defines the standard menu bar for use with Traits UI windows and panels, and standard actions and buttons.

class traitsui.menu.Action

Bases: `pyface.action.action.Action`

An action on a menu bar in a Traits UI window or panel.

action = Str

The method to call to perform the action, on the Handler for the window. The method must accept a single parameter, which is a UIInfo object. Because Actions are associated with Views rather than Handlers, you must ensure that the Handler object for a particular window has a method with the correct name, for each Action defined on the View for that window.

checked_when = Str

Boolean expression indicating when the action is displayed with a check mark beside it. This attribute applies only to actions that are included in menus.

defined_when = Str

Pre-condition for including the action in the menu bar or toolbar. If the expression evaluates to False, the action is not defined in the display. Conditions for **defined_when** are evaluated only once, when the display is first constructed.

enabled_when = Str

Pre-condition for enabling the action. If the expression evaluates to False, the action is disabled, that is, it cannot be selected. All **enabled_when** conditions are checked each time that any trait value is edited in the display. Therefore, you can use **enabled_when** conditions to enable or disable actions in response to user input.

visible_when = Str

Pre-condition for showing the action. If the expression evaluates to False, the action is not visible (and disappears if it was previously visible). If the value evaluates to True, the action becomes visible. All **visible_when** conditions are checked each time that any trait value is edited in the display. Therefore, you can use **visible_when** conditions to hide or show actions in response to user input.

traitsui.menu.ApplyButton = <traitsui.menu.Action object>

When the user clicks the **Apply** button, all changes made in the window are applied to the model. This option is meaningful only for modal windows.

`traitsui.menu.CancelButton = <traitsui.menu.Action object>`

When the user clicks the **Cancel** button, all changes made in the window are discarded; if the window is live, the model is restored to the values it held before the window was opened. The window is then closed.

`traitsui.menu.CloseAction = <traitsui.menu.Action object>`

The standard “close window” action

`traitsui.menu.HelpAction = <traitsui.menu.Action object>`

The standard “show help” action

`traitsui.menu.HelpButton = <traitsui.menu.Action object>`

When the user clicks the **Help** button, the current help handler is invoked. If the default help handler is used, a pop-up window is displayed, which contains the **help** text for the top-level Group (if any), and for the items in the view. If the default help handler has been overridden, the action is determined by the custom help handler. See `traitsui.help`.

`traitsui.menu.NoButtons = [<traitsui.menu.Action object>]`

The window has no command buttons

`traitsui.menu.OKButton = <traitsui.menu.Action object>`

When the user clicks the **OK** button, all changes made in the window are applied to the model, and the window is closed.

`traitsui.menu.RedoAction = <traitsui.menu.Action object>`

The standard “redo last undo” action

`traitsui.menu.RevertAction = <traitsui.menu.Action object>`

The standard “revert all changes” action

`traitsui.menu.RevertButton = <traitsui.menu.Action object>`

When the user clicks the **Revert** button, all changes made in the window are cancelled and the original values are restored. If the changes have been applied to the model (because the user clicked **Apply** or because the window is live), the model data is restored as well. The window remains open.

`traitsui.menu.Separator`

Menu separator

alias of Group

`traitsui.menu.StandardMenuBar = <pyface.ui.null.action.menu_bar_manager.MenuBarManager object>`

The standard Traits UI menu bar

`traitsui.menu.UndoAction = <traitsui.menu.Action object>`

The standard “undo last change” action

`traitsui.menu.UndoButton = <traitsui.menu.Action object>`

*Appears as two buttons – **Undo** and **Redo**.* When **Undo** is clicked, the most recent change to the data is cancelled, restoring the previous value. **Redo** cancels the most recent “undo” operation.

traitsui.message module

Displays a message to the user as a modal window.

class `traitsui.message.AutoCloseMessage`

Bases: `traits.has_traits.HasPrivateTraits`

show (*parent=None, title=""*)

Display the wait message for a limited duration.

class `traitsui.message.Message`

Bases: `traits.has_traits.HasPrivateTraits`

`traitsui.message.auto_close_message` (*message='Please wait', time=2.0, title='Please wait', parent=None*)

Displays a message to the user as a modal window with no buttons. The window closes automatically after a specified time interval (specified in seconds).

`traitsui.message.error` (*message="", title='Message', buttons=['OK', 'Cancel'], parent=None*)

Displays a message to the user as a modal window with the specified title and buttons.

If *buttons* is not specified, **OK** and **Cancel** buttons are used, which is appropriate for confirmations, where the user must decide whether to proceed. Be sure to word the message so that it is clear that clicking **OK** continues the operation.

`traitsui.message.message` (*message="", title='Message', buttons=['OK'], parent=None*)

Displays a message to the user as a model window with the specified title and buttons.

If *buttons* is not specified, a single **OK** button is used, which is appropriate for notifications, where no further action or decision on the user's part is required.

traitsui.mimedata module

traitsui.table_column module

Defines the table column descriptor used by the editor and editor factory classes for numeric and table editors.

class `traitsui.table_column.ExpressionColumn`

Bases: `traitsui.table_column.ObjectColumn`

A column for displaying computed values.

get_raw_value (*object*)

Gets the unformatted value of the column for a specified object.

class `traitsui.table_column.ListColumn`

Bases: `traitsui.table_column.TableColumn`

A column for editing lists.

get_editor (*object*)

Gets the editor for the column of a specified object.

get_value (*object*)

Gets the value of the column for a specified object.

key (*object*)

Returns the value to use for sorting.

set_value (*object, value*)

Sets the value of the column for a specified object.

class `traitsui.table_column.NumericColumn`

Bases: `traitsui.table_column.ObjectColumn`

A column for editing Numeric arrays.

get_cell_color (*object*)

Returns the cell background color for the column for a specified object row.

get_data_column (*object*)

Gets the entire contents of the specified object column.

get_editor (*object*)

Gets the editor for the column of a specified object row.

get_horizontal_alignment (*object*)

Returns the horizontal alignment for the column for a specified object row.

get_menu (*object*, *row*)

Returns the context menu to display when the user right-clicks on the column for a specified object row.

get_text_color (*object*)

Returns the text color for the column for a specified object row.

get_text_font (*object*)

Returns the text font for the column for a specified object row.

get_type (*object*)

Gets the type of data for the column for a specified object row.

get_value (*object*)

Gets the value of the column for a specified object row.

get_vertical_alignment (*object*)

Returns the vertical alignment for the column for a specified object row.

is_droppable (*object*, *row*, *value*)

Returns whether a specified value is valid for dropping on the column for a specified object row.

is_editable (*object*)

Returns whether the column is editable for a specified object row.

set_value (*object*, *row*, *value*)

Sets the value of the column for a specified object row.

class traitsui.table_column.ObjectColumn

Bases: `traitsui.table_column.TableColumn`

A column for editing objects.

get_drag_value (*object*)

Returns the drag value for the column.

get_editor (*object*)

Gets the editor for the column of a specified object.

get_raw_value (*object*)

Gets the unformatted value of the column for a specified object.

get_style (*object*)

Gets the editor style for the column of a specified object.

get_value (*object*)

Gets the formatted value of the column for a specified object.

is_droppable (*object*, *value*)

Returns whether a specified value is valid for dropping on the column for a specified object.

key (*object*)

Returns the value to use for sorting.

set_value (*object*, *value*)

Sets the value of the column for a specified object.

target_name (*object*)

Returns the target object and name for the column.

class traitsui.table_column.TableColumn

Bases: `traits.has_traits.HasPrivateTraits`

Represents a column in a table editor.

cmp (*object1*, *object2*)

Returns the result of comparing the column of two different objects.

This is deprecated.

get_cell_color (*object*)

Returns the cell background color for the column for a specified object.

get_edit_height (*object*)

Returns the height of the column cell's row while it is being edited.

get_edit_width (*object*)

Returns the edit width of the column.

get_graph_color (*object*)

Returns the cell background graph color for the column for a specified object.

get_horizontal_alignment (*object*)

Returns the horizontal alignment for the column for a specified object.

get_image (*object*)

Returns the image to display for the column for a specified object.

get_label ()

Gets the label of the column.

get_maximum (*object*)

Returns the maximum value a numeric column can have.

get_menu (*object*)

Returns the context menu to display when the user right-clicks on the column for a specified object.

get_object (*object*)

Returns the actual object being edited.

get_renderer (*object*)

Returns the renderer for the column of a specified object.

get_text_color (*object*)

Returns the text color for the column for a specified object.

get_text_font (*object*)

Returns the text font for the column for a specified object.

get_tooltip (*object*)

Returns the tooltip to display when the user mouses over the column for a specified object.

get_type (*object*)

Gets the type of data for the column for a specified object.

get_vertical_alignment (*object*)

Returns the vertical alignment for the column for a specified object.

get_view (*object*)

Returns the view to display when clicking a non-editable cell.

get_width ()

Returns the width of the column.

is_auto_editable (*object*)

Returns whether the column is automatically edited/viewed for a specified object.

is_droppable (*object*, *value*)

Returns whether a specified value is valid for dropping on the column for a specified object.

is_editable (*object*)

Returns whether the column is editable for a specified object.

on_click (*object*)

Called when the user clicks on the column.

on_dclick (*object*)

Called when the user clicks on the column.

traitsui.table_filter module

Defines the filter object used to filter items displayed in a table editor.

class traitsui.table_filter.**EvalTableFilter**

Bases: *traitsui.table_filter.TableFilter*

A table filter based on evaluating an expression.

description ()

Returns a user readable description of what kind of object satisfies the filter.

filter (*object*)

Returns whether a specified object meets the filter or search criteria.

class traitsui.table_filter.**GenericTableFilterRule** (***traits*)

Bases: *traits.has_traits.HasPrivateTraits*

A general rule used by a table filter.

clone_traits (*traits=None*, *memo=None*, *copy=None*, ***metadata*)

Clones a new object from this one, optionally copying only a specified set of traits.

contains (*value1*, *value2*)

description ()

Returns a description of the filter.

ends_with (*value1*, *value2*)

eq (*value1*, *value2*)

ge (*value1*, *value2*)

gt (*value1*, *value2*)

ignored_traits = ['filter', 'name_editor', 'value_editor']

is_true (*object*)

Returns whether the rule is true for a specified object.

le (*value1*, *value2*)

lt (*value1*, *value2*)

ne (*value1*, *value2*)

starts_with (*value1*, *value2*)

class traitsui.table_filter.**GenericTableFilterRuleAndOrColumn**

Bases: *traitsui.table_column.ObjectColumn*

Table column that displays whether a filter rule is conjoining ('and') or disjoining ('or').

get_value (*object*)

Returns the traits editor of the column for a specified object.

class traitsui.table_filter.GenericTableFilterRuleEnabledColumn

Bases: *traitsui.table_column.ObjectColumn*

Table column that indicates whether a filter rule is enabled.

get_value (*object*)

Returns the traits editor of the column for a specified object.

class traitsui.table_filter.GenericTableFilterRuleNameColumn

Bases: *traitsui.table_column.ObjectColumn*

Table column for the name of an object trait.

get_editor (*object*)

Returns the traits editor of the column for a specified object.

class traitsui.table_filter.GenericTableFilterRuleValueColumn

Bases: *traitsui.table_column.ObjectColumn*

Table column for the value of an object trait.

get_editor (*object*)

Returns the traits editor of the column for a specified object.

class traitsui.table_filter.MenuTableFilter

Bases: *traitsui.table_filter.RuleTableFilter*

A table filter based on a menu of rules.

description ()

Returns a user-readable description of what kind of object satisfies the filter.

filter (*object*)

Returns whether a specified object meets the filter or search criteria.

class traitsui.table_filter.RuleTableFilter

Bases: *traitsui.table_filter.TableFilter*

A table filter based on rules.

description ()

Returns a user-readable description of the kind of object that satisfies the filter.

edit_view (*object*)

Return a view to use for editing the filter.

The “object” parameter is a sample object for the table that the filter will be applied to. It is supplied in case the filter needs to extract data or metadata from the object. If the table is empty, the “object” argument is None.

filter (*object*)

Returns whether a specified object meets the filter or search criteria.

class traitsui.table_filter.TableFilter

Bases: *traits.has_traits.HasPrivateTraits*

Filter for items displayed in a table.

description ()

Returns a user-readable description of what kind of object satisfies the filter.

edit (*object*)

Edits the contents of the filter.

edit_view (*object*)

Return a view to use for editing the filter.

The “object” parameter is a sample object for the table that the filter will be applied to. It is supplied in case the filter needs to extract data or metadata from the object. If the table is empty, the “object” argument is None.

filter (*object*)

Returns whether a specified object meets the filter or search criteria.

ignored_traits = ['_name', 'template', 'desc']

traitsui.tabular_adapter module

Defines the adapter classes associated with the Traits UI TabularEditor.

class traitsui.tabular_adapter.**AnITabularAdapter**

Bases: traits.has_traits.HasPrivateTraits

accepts = Bool(True)

Does the adapter know how to handle the current *item* or not:

column = Any

The current column id being adapted (if any):

columns = List(Str)

The list of columns the adapter supports. The items in the list have the same format as the *columns* trait in the *TabularAdapter* class, with the additional requirement that the string values must correspond to a string value in the associated *TabularAdapter* class.

is_cacheable = Bool(True)

Does the value of *accepts* depend only upon the type of *item*?

item = Any

Current item being adapted:

row = Int

The row index of the current item being adapted:

value = Any

The current value (if any):

class traitsui.tabular_adapter.**ITabularAdapter**

Bases: traits.has_traits.Interface

accepts = Bool

Does the adapter know how to handle the current *item* or not:

column = Any

The current column id being adapted (if any):

columns = List(Str)

The list of columns the adapter supports. The items in the list have the same format as the *columns* trait in the *TabularAdapter* class, with the additional requirement that the string values must correspond to a string value in the associated *TabularAdapter* class.

is_cacheable = Bool

Does the value of *accepts* depend only upon the type of *item*?

item = Any
Current item being adapted:

row = Int
The row index of the current item being adapted:

value = Any
The current value (if any):

class traitsui.tabular_adapter.**TabularAdapter**

Bases: `traits.has_traits.HasPrivateTraits`

The base class for adapting list items to values that can be edited by a TabularEditor.

adapter_column_indices = Property(depends_on='adapters, columns')
For each adapter, specifies the column indices the adapter handles.

adapter_column_map = Property(depends_on='adapters, columns')
For each adapter, specifies the mapping from column index to column id.

adapters = List(ITabularAdapter, update=True)
List of optional delegated adapters.

alignment = Enum('left', 'center', 'right')
Horizontal alignment to use for a specified column.

bg_color = Property
The background color for a row item.

cache = Any({})
Cache of attribute handlers.

cache_flushed = Event(update=True)
Event fired when the cache is flushed.

can_drop = Bool(False)
Can any arbitrary value be dropped onto the tabular view.

can_edit = Bool(True)
Can the text value of each item be edited?

cleanup()
Clean up the adapter to remove references to objects.

column = Int
The column index of the current item being adapted.

column_dict = Property()
Maps UI name of column to value identifying column to the adapter, if different.

column_id = Any
The current column id being adapted (if any).

column_map = Property(depends_on='columns')
The mapping from column indices to column identifiers (defined by the `columns` trait).

column_menu = Any
The context menu for column header.

columns = List()
A list of columns that should appear in the table. Each entry can have one of two forms: `string` or `(string, id)`, where `string` is the UI name of the column, and `id` is a value that identifies that column to the adapter. Normally this value is either a trait name or an index, but it can be any value that the adapter wants. If only `string` is specified, then `id` is the index of the `string` within `columns`.

content = Property

The content of a row/column item (may be any Python value).

default_bg_color = Color(None, update=True)

The default background color for table rows.

default_text_color = Color(None, update=True)

The default text color for table rows.

default_value = Any('')

Specifies the default value for a new row. This will usually need to be overridden.

delete (object, trait, row)

Deletes the specified row item.

This method is only called if the *delete* operation is specified in the *TabularEditor* operation trait, and the user requests that the item be deleted from the table.

The adapter can still choose not to delete the specified item if desired, although that may prove confusing to the user.

The default implementation assumes the trait defined by `object.trait` is a mutable sequence and attempts to perform a `del object.trait[row]` operation.

drag = Property

The value to be dragged for a specified row item.

dropped = Enum('after', 'before')

Specifies where a dropped item should be placed in the table relative to the item it is dropped on.

even_bg_color = Color(None, update=True)

The default background color for even table rows.

even_text_color = Color(None, update=True)

The default text color for even table rows.

font = Font(None)

The font for a row item.

format = Str('%s')

The Python format string to use for a specified column.

get_alignment (object, trait, column)

Returns the alignment style to use for a specified column.

The possible values that can be returned are: 'left', 'center' or 'right'. All table items share the same alignment for a specified column.

get_bg_color (object, trait, row, column=0)

Returns the background color to use for a specified row or cell.

A result of `None` means use the default background color; otherwise a toolkit-compatible color should be returned. Note that all columns for the specified table row will use the background color value returned.

get_can_drop (object, trait, row, value)

Returns whether the specified `value` can be dropped on the specified row.

A value of `True` means the `value` can be dropped; and a value of `False` indicates that it cannot be dropped.

The result is used to provide the user positive or negative drag feedback while dragging items over the table. `value` will always be a single value, even if multiple items are being dragged. The editor handles multiple drag items by making a separate call to `get_can_drop()` for each item being dragged.

get_can_edit (*object, trait, row*)

Returns whether the user can edit a specified row.

A `True` result indicates that the value can be edited, while a `False` result indicates that it cannot.

get_column (*object, trait, index*)

Returns the column id corresponding to a specified column index.

get_column_menu (*object, trait, row, column*)

Returns the context menu for a specified column.

get_content (*object, trait, row, column*)

Returns the content to display for a specified cell.

get_default_value (*object, trait*)

Returns a new default value for the specified `object.trait` list.

This method is called when *insert* or *append* operations are allowed and the user requests that a new item be added to the table. The result should be a new instance of whatever underlying representation is being used for table items.

The default implementation simply returns the value of the adapter's `default_value` trait.

get_drag (*object, trait, row*)

Returns the value to be *dragged* for a specified row.

A result of `None` means that the item cannot be dragged. Note that the value returned does not have to be the actual row item. It can be any value that you want to drag in its place. In particular, if you want the drag target to receive a copy of the row item, you should return a copy or clone of the item in its place.

Also note that if multiple items are being dragged, and this method returns `None` for any item in the set, no drag operation is performed.

get_dropped (*object, trait, row, value*)

Returns how to handle a specified `value` being dropped on a specified row.

The possible return values are:

- `'before'`: Insert the specified value before the dropped on item.
- `'after'`: Insert the specified value after the dropped on item.

Note there is no result indicating *do not drop* since you will have already indicated that the `object` can be dropped by the result returned from a previous call to `get_can_drop()`.

get_font (*object, trait, row, column=0*)

Returns the font to use for displaying a specified row or cell.

A result of `None` means use the default font; otherwise a toolkit font object should be returned. Note that all columns for the specified table row will use the font value returned.

get_format (*object, trait, row, column*)

Returns the Python formatting string to apply to the specified cell.

The resulting of formatting with this string will be used as the text to display it in the table.

The return can be any Python string containing exactly one old-style Python formatting sequence, such as `'%.4f'` or `'(%5.2f)'`.

get_image (*object, trait, row, column*)

Returns the image to display for a specified cell.

A result of `None` means no image will be displayed in the specified table cell. Otherwise the result should either be the name of the image, or an `ImageResource` object specifying the image to display.

A name is allowed in the case where the image is specified in the `TabularEditor` `images` trait. In that case, the name should be the same as the string specified in the `ImageResource` constructor.

get_item (*object, trait, row*)

Returns the specified row item.

The value returned should be the value that exists (or *logically* exists) at the specified `row` in your data. If your data is not really a list or array, then you can just use `row` as an integer *key* or *token* that can be used to retrieve a corresponding item. The value of `row` will always be in the range: $0 \leq \text{row} < \text{len}(\text{object}, \text{trait})$ (i.e. the result returned by the adapter `len()` method).

The default implementation assumes the trait defined by `object.trait` is a *sequence* and attempts to return the value at index `row`. If an error occurs, it returns `None` instead. This definition should work correctly for lists, tuples and arrays, or any other object that is indexable, but will have to be overridden for all other cases.

get_label (*section, obj=None*)

Override this method if labels will vary from object to object.

get_menu (*object, trait, row, column*)

Returns the context menu for a specified cell.

get_row_label (*section, obj=None*)

get_text (*object, trait, row, column*)

Returns a string containing the text to display for a specified cell.

If the underlying data representation for a specified item is not a string, then it is your responsibility to convert it to one before returning it as the result.

get_text_color (*object, trait, row, column=0*)

Returns the text color to use for a specified row or cell.

A result of `None` means use the default text color; otherwise a toolkit-compatible color should be returned. Note that all columns for the specified table row will use the text color value returned.

get_tooltip (*object, trait, row, column*)

Returns a string containing the tooltip to display for a specified cell.

You should return the empty string if you do not wish to display a tooltip.

get_width (*object, trait, column*)

Returns the width to use for a specified column.

If the value is ≤ 0 , the column will have a *default* width, which is the same as specifying a width of 0.1.

If the value is > 1.0 , it is converted to an integer and the result is the width of the column in pixels. This is referred to as a *fixed width* column.

If the value is a float such that $0.0 < \text{value} \leq 1.0$, it is treated as the *unnormalized fraction of the available space* that is to be assigned to the column. What this means requires a little explanation.

To arrive at the size in pixels of the column at any given time, the editor adds together all of the *unnormalized fraction* values returned for all columns in the table to arrive at a total value. Each *unnormalized fraction* is then divided by the total to create a *normalized fraction*. Each column is then assigned an amount of space in pixels equal to the maximum of 30 or its *normalized fraction* multiplied by the *available space*. The *available space* is defined as the actual width of the table minus the width of all *fixed width* columns. Note that this calculation is performed each time the table is resized in the user interface, thus allowing columns of this type to increase or decrease their width dynamically, while leaving *fixed width* columns unchanged.

image = Str(None, update=True)

The name of the default image to use for column items.

insert (*object*, *trait*, *row*, *value*)

Inserts *value* at the specified `object.trait[row]` index.

The specified *value* can be:

- An item being moved from one location in the data to another.
- A new item created by a previous call to `get_default_value()`.
- An item the adapter previously approved via a call to `get_can_drop()`.

The adapter can still choose not to insert the item into the data, although that may prove confusing to the user.

The default implementation assumes the trait defined by `object.trait` is a mutable sequence and attempts to perform an `object.trait[row:row] = [value]` operation.

item = **Any**

Current item being adapted.

label_map = **Property**(depends_on='columns')

The mapping from column indices to column labels (defined by the `columns` trait).

len (*object*, *trait*)

Returns the number of row items in the specified `object.trait`.

The result should be an integer greater than or equal to 0.

The default implementation assumes the trait defined by `object.trait` is a *sequence* and attempts to return the result of calling `len(object.trait)`. It will need to be overridden for any type of data which for which `len()` will not work.

menu = **Any**

The context menu for a row/column item.

name = **Str**

The name of the trait being edited.

object = **Instance**(HasTraits)

The object whose trait is being edited.

odd_bg_color = **Color**(None, update=True)

The default background color for odd table rows.

odd_text_color = **Color**(None, update=True)

The default text color for odd table rows.

row = **Int**

The row index of the current item being adapted.

row_label_name = **Either**(None, Str)

The name of the trait on a row item containing the value to use as a row label. If `None`, the label will be the empty string.

set_text (*object*, *trait*, *row*, *column*, *text*)

Sets the value for the specified cell.

This method is called when the user completes an editing operation on a table cell.

The string specified by *text* is the value that the user has entered in the table cell. If the underlying data does not store the value as text, it is your responsibility to convert *text* to the correct representation used.

text = **Property**

The text of a row/column item.

```

text_color = Property
    The text color for a row item.

tooltip = Str
    The tooltip information for a row/column item.

value = Any
    The current value (if any).

width = Float(-1)
    Width of a specified column.

```

traitsui.theme module

Defines ‘theme’ related classes.

```

class traitsui.theme.Theme (image=None, **traits)
    Bases: traits.has_traits.HasPrivateTraits

```

traitsui.toolkit module

Defines the stub functions used for creating concrete implementations of the standard EditorFactory subclasses supplied with the Traits package.

Most of the logic for determining which backend toolkit to use can now be found in `pyface.base_toolkit`.

```

class traitsui.toolkit.Toolkit (package, toolkit, *packages, **traits)
    Bases: pyface.base_toolkit.Toolkit

```

Abstract base class for GUI toolkits.

```

array_editor (*args, **traits)
boolean_editor (*args, **traits)
button_editor (*args, **traits)
check_list_editor (*args, **traits)
code_editor (*args, **traits)
color_editor (*args, **traits)
color_trait (*args, **traits)
compound_editor (*args, **traits)

```

```

constants ()
    Returns a dictionary of useful constants.

```

Currently, the dictionary should have the following key/value pairs:

- `WindowColor`: the standard window background color in the toolkit specific color format.

```

custom_editor (*args, **traits)
destroy_children (control)
    Destroys all of the child controls of a specified GUI toolkit control.

destroy_control (control)
    Destroys a specified GUI toolkit control.

directory_editor (*args, **traits)

```

dnd_editor (*args, **traits)

drop_editor (*args, **traits)

enum_editor (*args, **traits)

file_editor (*args, **traits)

font_editor (*args, **traits)

font_trait (*args, **traits)

history_editor (*args, **traits)

hook_events (ui, control, events=None, handler=None)
 Hooks all specified events for all controls in a UI so that they can be routed to the correct event handler.

html_editor (*args, **traits)

image_editor (*args, **traits)

image_enum_editor (*args, **traits)

image_size (image)
 Returns a (width, height) tuple containing the size of a specified toolkit image.

instance_editor (*args, **traits)

key_binding_editor (*args, **traits)

key_event_to_name (event)
 Converts a keystroke event into a corresponding key name.

kiva_font_trait (*args, **traits)

list_editor (*args, **traits)

list_str_editor (*args, **traits)

null_editor (*args, **traits)

ordered_set_editor (*args, **traits)

plot_editor (*args, **traits)

position (ui)
 Positions the associated dialog window on the display.

range_editor (*args, **traits)

rebuild_ui (ui)
 Rebuilds a UI after a change to the content of the UI.

rgb_color_editor (*args, **traits)

rgb_color_trait (*args, **traits)

rgba_color_editor (*args, **traits)

rgba_color_trait (*args, **traits)

route_event (ui, event)
 Routes a “hooked” event to the current handler method.

save_window (ui)
 Saves user preference information associated with a UI window.

set_icon (ui)
 Sets the icon for the UI window.

set_title (*ui*)
Sets the title for the UI window.

shell_editor (**args, **traits*)

show_help (*ui, control*)
Shows a Help window for a specified UI and control.

skip_event (*event*)
Indicates that an event should continue to be processed by the toolkit.

table_editor (**args, **traits*)

tabular_editor (**args, **traits*)

text_editor (**args, **traits*)

title_editor (**args, **traits*)

tree_editor (**args, **traits*)

tuple_editor (**args, **traits*)

ui_editor ()

ui_info (*ui, parent*)
Creates a GUI-toolkit-specific temporary “live update” popup dialog user interface using information from the specified UI object.

ui_live (*ui, parent*)
Creates a GUI-toolkit-specific non-modal “live update” window user interface using information from the specified UI object.

ui_livemodal (*ui, parent*)
Creates a GUI-toolkit-specific modal “live update” dialog user interface using information from the specified UI object.

ui_modal (*ui, parent*)
Creates a GUI-toolkit-specific modal dialog user interface using information from the specified UI object.

ui_nonmodal (*ui, parent*)
Creates a GUI-toolkit-specific non-modal dialog user interface using information from the specified UI object.

ui_panel (*ui, parent*)
Creates a GUI-toolkit-specific panel-based user interface using information from the specified UI object.

ui_popover (*ui, parent*)
Creates a GUI-toolkit-specific temporary “live update” popup dialog user interface using information from the specified UI object.

ui_popup (*ui, parent*)
Creates a GUI-toolkit-specific temporary “live update” popup dialog user interface using information from the specified UI object.

ui_subpanel (*ui, parent*)
Creates a GUI-toolkit-specific subpanel-based user interface using information from the specified UI object.

ui_wizard (*ui, parent*)
Creates a GUI-toolkit-specific wizard dialog user interface using information from the specified UI object.

value_editor (**args, **traits*)

view_application (*context, view, kind=None, handler=None, id="", scrollable=None, args=None*)

Creates a GUI-toolkit-specific modal dialog user interface that runs as a complete application using information from the specified View object.

Parameters

- **context** (*object or dictionary*) – A single object or a dictionary of string/object pairs, whose trait attributes are to be edited. If not specified, the current object is used.
- **view** (*view or string*) – A View object that defines a user interface for editing trait attribute values.
- **kind** (*string*) – The type of user interface window to create. See the **traitsui.view.kind_trait** trait for values and their meanings. If *kind* is unspecified or None, the **kind** attribute of the View object is used.
- **handler** (*Handler object*) – A handler object used for event handling in the dialog box. If None, the default handler for Traits UI is used.
- **id** (*string*) – A unique ID for persisting preferences about this user interface, such as size and position. If not specified, no user preferences are saved.
- **scrollable** (*Boolean*) – Indicates whether the dialog box should be scrollable. When set to True, scroll bars appear on the dialog box if it is not large enough to display all of the items in the view at one time.

`traitsui.toolkit.assert_toolkit_import (names)`

Raise an error if a toolkit with the given name should not be allowed to be imported.

`traitsui.toolkit.toolkit (*toolkits)`

Selects and returns a low-level GUI toolkit.

Use this function to get a reference to the current toolkit.

Parameters **toolkits (strings)* – Toolkit names to try if toolkit not already selected. If not supplied, will try all `traitsui.toolkits` entry points until a match is found.

Returns Appropriate concrete Toolkit subclass for selected toolkit.

Return type *toolkit*

Raises

- `TraitError` – If no working toolkit is found.
- `RuntimeError` – If no `ETSTConfig.toolkit` is set but the toolkit cannot be loaded for some reason.

`traitsui.toolkit.toolkit_object (name, raise_exceptions=False)`

Return the toolkit specific object with the given name.

Parameters

- **name** (*str*) – The relative module path and the object name separated by a colon.
- **raise_exceptions** (*bool*) – Whether or not to raise an exception if the name cannot be imported.

Raises

- `TraitError` – If no working toolkit is found.

- `RuntimeError` – If no `ETSTConfig.toolkit` is set but the toolkit cannot be loaded for some reason. This is also raised if `raise_exceptions` is `True` the backend does not implement the desired object.

traitsui.toolkit_traits module

```
traitsui.toolkit_traits.ColorTrait (*args, **traits)
traitsui.toolkit_traits.FontTrait (*args, **traits)
traitsui.toolkit_traits.RGBColorTrait (*args, **traits)
```

traitsui.tree_node module

Defines the tree node descriptor used by the tree editor and tree editor factory classes.

```
class traitsui.tree_node.ITreeNode
    Bases: traits.has_traits.Interface

    activated ()
        Handles an object being activated.

    allows_children ()
        Returns whether this object can have children.

    append_child (child)
        Appends a child to the object's children.

    can_add (add_object)
        Returns whether a given object is droppable on the node.

    can_auto_close ()
        Returns whether the object's children should be automatically closed.

    can_auto_open ()
        Returns whether the object's children should be automatically opened.

    can_copy ()
        Returns whether the object's children can be copied.

    can_delete ()
        Returns whether the object's children can be deleted.

    can_delete_me ()
        Returns whether the object can be deleted.

    can_insert ()
        Returns whether the object's children can be inserted (vs. appended).

    can_rename ()
        Returns whether the object's children can be renamed.

    can_rename_me ()
        Returns whether the object can be renamed.

    click ()
        Handles an object being clicked.

    confirm_delete ()
        Checks whether a specified object can be deleted.

        The following return values are possible:
```

- **True** if the object should be deleted with no further prompting.
- **False** if the object should not be deleted.
- Anything else: Caller should take its default action (which might include prompting the user to confirm deletion).

dclick()

Handles an object being double-clicked.

delete_child(index)

Deletes a child at a specified index from the object's children.

drop_object(dropped_object)

Returns a droppable version of a specified object.

get_add()

Returns the list of classes that can be added to the object.

get_children()

Gets the object's children.

get_children_id()

Gets the object's children identifier.

get_column_labels(object)

Get the labels for any columns that have been defined.

get_drag_object()

Returns a draggable version of a specified object.

get_icon(is_expanded)

Returns the icon for a specified object.

get_icon_path()

Returns the path used to locate an object's icon.

get_label()

Gets the label to display for a specified object.

get_menu()

Returns the right-click context menu for an object.

get_name()

Returns the name to use when adding a new object instance (displayed in the "New" submenu).

get_tooltip()

Gets the tooltip to display for a specified object.

get_view()

Gets the view to use when editing an object.

has_children()

Returns whether the object has children.

insert_child(index, child)

Inserts a child into the object's children.

select()

Handles an object being selected.

set_label(label)

Sets the label for a specified object.

when_children_changed (*listener, remove*)

Sets up or removes a listener for children being changed on a specified object.

when_children_replaced (*listener, remove*)

Sets up or removes a listener for children being replaced on a specified object.

when_column_labels_change (*object, listener, remove*)

Sets up or removes a listener for the column labels being changed on a specified object.

This will fire when either the list is reassigned or when it is modified. I.e., it listens both to the trait change event and the trait_items change event. Implement the listener appropriately to handle either case.

when_label_changed (*listener, remove*)

Sets up or removes a listener for the label being changed on a specified object.

class traitsui.tree_node.**ITreeNodeAdapter** (*adaptee, **traits*)

Bases: `traits.adaptation.adapter.Adapter`

Abstract base class for an adapter that implements the ITreeNode interface.

Usage:

1. Create a subclass of ITreeNodeAdapter.
2. Register the adapter to define what class (or classes) this is an ITreeNode adapter for (ie. `register_factory(<from class>, ITreeNode, ITreeNodeAdapter)`).
3. Override any of the following methods as necessary, using the `self.adaptee` trait to access the adapted object if needed.

Note: This base class implements all of the ITreeNode interface methods, but does not necessarily provide useful implementations for all of the methods. It allows you to get a new adapter class up and running quickly, but you should carefully review your final adapter implementation class to make sure it behaves correctly in your application.

activated ()

Handles an object being activated.

allows_children ()

Returns whether this object can have children.

append_child (*child*)

Appends a child to the object's children.

can_add (*add_object*)

Returns whether a given object is droppable on the node.

can_auto_close ()

Returns whether the object's children should be automatically closed.

can_auto_open ()

Returns whether the object's children should be automatically opened.

can_copy ()

Returns whether the object's children can be copied.

can_delete ()

Returns whether the object's children can be deleted.

can_delete_me ()

Returns whether the object can be deleted.

can_insert ()

Returns whether the object's children can be inserted (vs. appended).

can_rename ()

Returns whether the object's children can be renamed.

can_rename_me ()

Returns whether the object can be renamed.

click ()

Handles an object being clicked.

confirm_delete ()

Checks whether a specified object can be deleted.

The following return values are possible:

- **True** if the object should be deleted with no further prompting.
- **False** if the object should not be deleted.
- Anything else: Caller should take its default action (which might include prompting the user to confirm deletion).

dclick ()

Handles an object being double-clicked.

delete_child (index)

Deletes a child at a specified index from the object's children.

drop_object (dropped_object)

Returns a droppable version of a specified object.

get_add ()

Returns the list of classes that can be added to the object.

get_background ()

Returns the background for object

get_children ()

Gets the object's children.

get_children_id ()

Gets the object's children identifier.

get_column_labels ()

Get the labels for any columns that have been defined.

get_drag_object ()

Returns a draggable version of a specified object.

get_foreground ()

Returns the foreground for object

get_icon (is_expanded)

Returns the icon for a specified object.

get_icon_path ()

Returns the path used to locate an object's icon.

get_label ()

Gets the label to display for a specified object.

get_menu ()

Returns the right-click context menu for an object.

get_name()
Returns the name to use when adding a new object instance (displayed in the “New” submenu).

get_tooltip()
Gets the tooltip to display for a specified object.

get_view()
Gets the view to use when editing an object.

has_children()
Returns whether the object has children.

insert_child(index, child)
Inserts a child into the object’s children.

select()
Handles an object being selected.

set_label(label)
Sets the label for a specified object.

when_children_changed(listener, remove)
Sets up or removes a listener for children being changed on a specified object.

when_children_replaced(listener, remove)
Sets up or removes a listener for children being replaced on a specified object.

when_column_labels_change(listener, remove)
Sets up or removes a listener for the column labels being changed on a specified object.

This will fire when either the list is reassigned or when it is modified. I.e., it listens both to the trait change event and the trait_items change event. Implement the listener appropriately to handle either case.

when_label_changed(listener, remove)
Sets up or removes a listener for the label being changed on a specified object.

class traitsui.tree_node.ITreeNodeAdapterBridge
Bases: `traits.has_traits.HasPrivateTraits`

Private class for use by a toolkit-specific implementation of the TreeEditor to allow bridging the TreeNode interface used by the editor to the ITreeNode interface used by object adapters.

activated(object)
Handles an object being activated.

allows_children(object)
Returns whether this object can have children.

append_child(object, child)
Appends a child to the object’s children.

can_add(object, add_object)
Returns whether a given object is droppable on the node.

can_auto_close(object)
Returns whether the object’s children should be automatically closed.

can_auto_open(object)
Returns whether the object’s children should be automatically opened.

can_copy(object)
Returns whether the object’s children can be copied.

can_delete (*object*)

Returns whether the object's children can be deleted.

can_delete_me (*object*)

Returns whether the object can be deleted.

can_insert (*object*)

Returns whether the object's children can be inserted (vs. appended).

can_rename (*object*)

Returns whether the object's children can be renamed.

can_rename_me (*object*)

Returns whether the object can be renamed.

click (*object*)

Handles an object being clicked.

confirm_delete (*object*)

Checks whether a specified object can be deleted.

The following return values are possible:

- **True** if the object should be deleted with no further prompting.
- **False** if the object should not be deleted.
- Anything else: Caller should take its default action (which might include prompting the user to confirm deletion).

dclick (*object*)

Handles an object being double-clicked.

delete_child (*object, index*)

Deletes a child at a specified index from the object's children.

drop_object (*object, dropped_object*)

Returns a droppable version of a specified object.

get_add (*object*)

Returns the list of classes that can be added to the object.

get_background (*object*)

Returns the background for object

get_children (*object*)

Gets the object's children.

get_children_id (*object*)

Gets the object's children identifier.

get_column_labels (*object*)

Get the labels for any columns that have been defined.

get_drag_object (*object*)

Returns a draggable version of a specified object.

get_foreground (*object*)

Returns the foreground for object

get_icon (*object, is_expanded*)

Returns the icon for a specified object.

get_icon_path (*object*)

Returns the path used to locate an object's icon.

get_label (*object*)
Gets the label to display for a specified object.

get_menu (*object*)
Returns the right-click context menu for an object.

get_name (*object*)
Returns the name to use when adding a new object instance (displayed in the “New” submenu).

get_tooltip (*object*)
Gets the tooltip to display for a specified object.

get_view (*object*)
Gets the view to use when editing an object.

has_children (*object*)
Returns whether the object has children.

insert_child (*object, index, child*)
Inserts a child into the object’s children.

select (*object*)
Handles an object being selected.

set_label (*object, label*)
Sets the label for a specified object.

when_children_changed (*object, listener, remove*)
Sets up or removes a listener for children being changed on a specified object.

when_children_replaced (*object, listener, remove*)
Sets up or removes a listener for children being replaced on a specified object.

when_column_labels_change (*object, listener, remove*)
Sets up or removes a listener for the column labels being changed on a specified object.

This will fire when either the list is reassigned or when it is modified. I.e., it listens both to the trait change event and the trait_items change event. Implement the listener appropriately to handle either case.

when_label_changed (*object, listener, remove*)
Sets up or removes a listener for the label being changed on a specified object.

class traitsui.tree_node.**MultiTreeNode** (***traits*)
Bases: *traitsui.tree_node.TreeNode*

activated (*object*)
Handles an object being activated.

allows_children (*object*)
Returns whether this object can have children (True for this class).

can_add (*object, add_object*)
Returns whether a given object is droppable on the node (False for this class).

can_auto_close (*object*)
Returns whether the object’s children should be automatically closed.

can_auto_open (*object*)
Returns whether the object’s children should be automatically opened.

can_copy (*object*)
Returns whether the object’s children can be copied.

can_delete (*object*)
Returns whether the object's children can be deleted (False for this class).

can_delete_me (*object*)
Returns whether the object can be deleted (True for this class).

can_insert (*object*)
Returns whether the object's children can be inserted (False, meaning that children are appended, for this class).

can_rename (*object*)
Returns whether the object's children can be renamed (False for this class).

can_rename_me (*object*)
Returns whether the object can be renamed (False for this class).

click (*object*)
Handles an object being clicked.

dclick (*object*)
Handles an object being double-clicked.

drop_object (*object, dropped_object*)
Returns a droppable version of a specified object.

get_add (*object*)
Returns the list of classes that can be added to the object.

get_children (*object*)
Gets the object's children.

get_children_id (*object*)
Gets the object's children identifier.

get_drag_object (*object*)
Returns a draggable version of a specified object.

get_icon (*object, is_expanded*)
Returns the icon for a specified object.

get_icon_path (*object*)
Returns the path used to locate an object's icon.

get_label (*object*)
Gets the label to display for a specified object.

get_menu (*object*)
Returns the right-click context menu for an object.

get_name (*object*)
Returns the name to use when adding a new object instance (displayed in the "New" submenu).

get_view (*object*)
Gets the view to use when editing an object.

has_children (*object*)
Returns whether this object has children (True for this class).

select (*object*)
Handles an object being selected.

set_label (*object, label*)
Sets the label for a specified object.

when_children_changed (*object, listener, remove*)

Sets up or removes a listener for children being changed on a specified object.

when_children_replaced (*object, listener, remove*)

Sets up or removes a listener for children being replaced on a specified object.

when_label_changed (*object, listener, remove*)

Sets up or removes a listener for the label being changed on a specified object.

class traitsui.tree_node.**ObjectTreeNode** (***traits*)

Bases: *traitsui.tree_node.TreeNode*

activated (*object*)

Handles an object being activated.

allows_children (*object*)

Returns whether this object can have children.

append_child (*object, child*)

Appends a child to the object's children.

can_add (*object, add_object*)

Returns whether a given object is droppable on the node.

can_auto_close (*object*)

Returns whether the object's children should be automatically closed.

can_auto_open (*object*)

Returns whether the object's children should be automatically opened.

can_copy (*object*)

Returns whether the object's children can be copied.

can_delete (*object*)

Returns whether the object's children can be deleted.

can_delete_me (*object*)

Returns whether the object can be deleted.

can_insert (*object*)

Returns whether the object's children can be inserted (vs. appended).

can_rename (*object*)

Returns whether the object's children can be renamed.

can_rename_me (*object*)

Returns whether the object can be renamed.

click (*object*)

Handles an object being clicked.

confirm_delete (*object*)

Checks whether a specified object can be deleted.

The following return values are possible:

- **True** if the object should be deleted with no further prompting.
- **False** if the object should not be deleted.
- Anything else: Caller should take its default action (which might include prompting the user to confirm deletion).

dclick (*object*)

Handles an object being double-clicked.

delete_child (*object*, *index*)
 Deletes a child at a specified index from the object's children.

drop_object (*object*, *dropped_object*)
 Returns a droppable version of a specified object.

get_add (*object*)
 Returns the list of classes that can be added to the object.

get_children (*object*)
 Gets the object's children.

get_children_id (*object*)
 Gets the object's children identifier.

get_drag_object (*object*)
 Returns a draggable version of a specified object.

get_icon (*object*, *is_expanded*)
 Returns the icon for a specified object.

get_icon_path (*object*)
 Returns the path used to locate an object's icon.

get_label (*object*)
 Gets the label to display for a specified object.

get_menu (*object*)
 Returns the right-click context menu for an object.

get_name (*object*)
 Returns the name to use when adding a new object instance (displayed in the "New" submenu).

get_tooltip (*object*)
 Gets the tooltip to display for a specified object.

get_view (*object*)
 Gets the view to use when editing an object.

has_children (*object*)
 Returns whether the object has children.

insert_child (*object*, *index*, *child*)
 Inserts a child into the object's children.

is_node_for (*object*)
 Returns whether this is the node that should handle a specified object.

select (*object*)
 Handles an object being selected.

set_label (*object*, *label*)
 Sets the label for a specified object.

when_children_changed (*object*, *listener*, *remove*)
 Sets up or removes a listener for children being changed on a specified object.

when_children_replaced (*object*, *listener*, *remove*)
 Sets up or removes a listener for children being replaced on a specified object.

when_label_changed (*object*, *listener*, *remove*)
 Sets up or removes a listener for the label being changed on a specified object.

```
class traitsui.tree_node.TreeNode (**traits)
    Bases: traits.has_traits.HasPrivateTraits
```

Represents a tree node. Used by the tree editor and tree editor factory classes.

activated (*object*)

Handles an object being activated.

allows_children (*object*)

Returns whether this object can have children.

append_child (*object*, *child*)

Appends a child to the object's children.

can_add (*object*, *add_object*)

Returns whether a given object is droppable on the node.

can_auto_close (*object*)

Returns whether the object's children should be automatically closed.

can_auto_open (*object*)

Returns whether the object's children should be automatically opened.

can_copy (*object*)

Returns whether the object's children can be copied.

can_delete (*object*)

Returns whether the object's children can be deleted.

can_delete_me (*object*)

Returns whether the object can be deleted.

can_insert (*object*)

Returns whether the object's children can be inserted (vs. appended).

can_rename (*object*)

Returns whether the object's children can be renamed.

can_rename_me (*object*)

Returns whether the object can be renamed.

click (*object*)

Handles an object being clicked.

confirm_delete (*object*)

Checks whether a specified object can be deleted.

The following return values are possible:

- **True** if the object should be deleted with no further prompting.
- **False** if the object should not be deleted.
- Anything else: Caller should take its default action (which might include prompting the user to confirm deletion).

dclick (*object*)

Handles an object being double-clicked.

delete_child (*object*, *index*)

Deletes a child at a specified index from the object's children.

drop_object (*object*, *dropped_object*)

Returns a droppable version of a specified object.

get_add (*object*)
Returns the list of classes that can be added to the object.

get_background (*object*)

get_children (*object*)
Gets the object's children.

get_children_id (*object*)
Gets the object's children identifier.

get_column_labels (*object*)
Get the labels for any columns that have been defined.

get_drag_object (*object*)
Returns a draggable version of a specified object.

get_foreground (*object*)

get_icon (*object, is_expanded*)
Returns the icon for a specified object.

get_icon_path (*object*)
Returns the path used to locate an object's icon.

get_label (*object*)
Gets the label to display for a specified object.

get_menu (*object*)
Returns the right-click context menu for an object.

get_name (*object*)
Returns the name to use when adding a new object instance (displayed in the "New" submenu).

get_tooltip (*object*)
Gets the tooltip to display for a specified object.

get_view (*object*)
Gets the view to use when editing an object.

has_children (*object*)
Returns whether the object has children.

insert_child (*object, index, child*)
Inserts a child into the object's children.

is_addable (*klass*)
Returns whether a specified object class can be added to the node.

is_node_for (*object*)
Returns whether this is the node that handles a specified object.

select (*object*)
Handles an object being selected.

set_label (*object, label*)
Sets the label for a specified object.

when_children_changed (*object, listener, remove*)
Sets up or removes a listener for children being changed on a specified object.

when_children_replaced (*object, listener, remove*)
Sets up or removes a listener for children being replaced on a specified object.

when_column_labels_change (*object, listener, remove*)

Sets up or removes a listener for the column labels being changed on a specified object.

This will fire when either the list is reassigned or when it is modified. I.e., it listens both to the trait change event and the trait_items change event. Implement the listener appropriately to handle either case.

when_label_changed (*object, listener, remove*)

Sets up or removes a listener for the label being changed on a specified object.

class traitsui.tree_node.**TreeNodeObject**

Bases: `traits.has_traits.HasPrivateTraits`

Represents the object that corresponds to a tree node.

tno_activated (*node*)

Handles an object being activated.

tno_allows_children (*node*)

Returns whether this object allows children.

tno_append_child (*node, child*)

Appends a child to the object's children.

tno_can_add (*node, add_object*)

Returns whether a given object is droppable on the node.

tno_can_auto_close (*node*)

Returns whether the object's children should be automatically closed.

tno_can_auto_open (*node*)

Returns whether the object's children should be automatically opened.

tno_can_copy (*node*)

Returns whether the object's children can be copied.

tno_can_delete (*node*)

Returns whether the object's children can be deleted.

tno_can_delete_me (*node*)

Returns whether the object can be deleted.

tno_can_insert (*node*)

Returns whether the object's children can be inserted (vs. appended).

tno_can_rename (*node*)

Returns whether the object's children can be renamed.

tno_can_rename_me (*node*)

Returns whether the object can be renamed.

tno_click (*node*)

Handles an object being clicked.

tno_confirm_delete (*node*)

Checks whether a specified object can be deleted.

The following return values are possible:

- **True** if the object should be deleted with no further prompting.
- **False** if the object should not be deleted.
- Anything else: Caller should take its default action (which might include prompting the user to confirm deletion).

tno_dclick (*node*)
 Handles an object being double-clicked.

tno_delete_child (*node, index*)
 Deletes a child at a specified index from the object's children.

tno_drop_object (*node, dropped_object*)
 Returns a droppable version of a specified object.

tno_get_add (*node*)
 Returns the list of classes that can be added to the object.

tno_get_children (*node*)
 Gets the object's children.

tno_get_children_id (*node*)
 Gets the object's children identifier.

tno_get_drag_object (*node*)
 Returns a draggable version of a specified object.

tno_get_icon (*node, is_expanded*)
 Returns the icon for a specified object.

tno_get_icon_path (*node*)
 Returns the path used to locate an object's icon.

tno_get_label (*node*)
 Gets the label to display for a specified object.

tno_get_menu (*node*)
 Returns the right-click context menu for an object.

tno_get_name (*node*)
 Returns the name to use when adding a new object instance (displayed in the "New" submenu).

tno_get_tooltip (*node*)
 Gets the tooltip to display for a specified object.

tno_get_view (*node*)
 Gets the view to use when editing an object.

tno_has_children (*node*)
 Returns whether this object has children.

tno_insert_child (*node, index, child*)
 Inserts a child into the object's children.

tno_is_node_for (*node*)
 Returns whether this is the node that should handle a specified object.

tno_select (*node*)
 Handles an object being selected.

tno_set_label (*node, label*)
 Sets the label for a specified object.

tno_when_children_changed (*node, listener, remove*)
 Sets up or removes a listener for children being changed on a specified object.

tno_when_children_replaced (*node, listener, remove*)
 Sets up or removes a listener for children being replaced on a specified object.

tno_when_label_changed (*node, listener, remove*)

Sets up or removes a listener for the label being changed on a specified object.

traitsui.ui module

Defines the UI class used to represent an active traits-based user interface.

class traitsui.ui.**Dispatcher** (*method, info, object, method_name*)

Bases: object

dispatch ()

Dispatches the method.

remove ()

Removes the dispatcher.

class traitsui.ui.**UI**

Bases: traits.has_traits.HasPrivateTraits

Information about the user interface for a View.

add_checked (*checked_when, editor*)

Adds a conditionally enabled (menu) Editor object to the list of monitored ‘checked_when’ objects.

add_defined (*method*)

Adds a Handler method to the list of methods to be called once the user interface has been constructed.

add_enabled (*enabled_when, editor*)

Adds a conditionally enabled Editor object to the list of monitored ‘enabled_when’ objects.

add_visible (*visible_when, editor*)

Adds a conditionally enabled Editor object to the list of monitored ‘visible_when’ objects.

disposable_traits = ['view_elements', 'info', 'handler', 'context', 'view', 'history',

dispose (*result=None, abort=False*)

Disposes of the contents of a user interface.

do_undoable (*action, *args, **kw*)

Performs an action that can be undone.

eval_when (*when, result=True*)

Evaluates an expression in the UI’s **context** and returns the result.

evaluate (*function, *args, **kw_args*)

Evaluates a specified function in the UI’s **context**.

find (*include*)

Finds the definition of the specified Include object in the current user interface building context.

finish ()

Finishes disposing of a user interface.

get_editors (*name*)

Returns a list of editors for the given trait name.

get_error_controls ()

Returns the list of editor error controls contained by the user interface.

get_extended_value (*name*)

Gets the current value of a specified extended trait name.

get_prefs (*prefs=None*)
 Gets the preferences to be saved for the user interface.

get_ui_db (*mode='r'*)
 Returns a reference to the Traits UI preference database.

key_handler (*event, skip=True*)
 Handles key events.

pop_level (*level*)
 Restores a previously pushed search stack level.

prepare_ui ()
 Performs all processing that occurs after the user interface is created.

push_level ()
 Returns the current search stack level.

recyclable_traits = ['_context', '_revert', '_defined', '_visible', '_enabled', '_checked']

recycle ()
 Recycles the user interface prior to rebuilding it.

reset (*destroy=True*)
 Resets the contents of a user interface.

restore_prefs ()
 Retrieves and restores any saved user preference information associated with the UI.

route_event (*event*)
 Routes a “hooked” event to the correct handler method.

save_prefs (*prefs=None*)
 Saves any user preference information associated with the UI.

set_prefs (*prefs*)
 Sets the values of user preferences for the UI.

sync_view ()
 Synchronize context object traits with view editor traits.

traits_init ()
 Initializes the traits object.

ui (*parent, kind*)
 Creates a user interface from the associated View template object.

traitsui.ui_editor module

Defines the BasicUIEditor class, which allows creating editors that define their function by creating an embedded Traits UI.

```
class traitsui.ui_editor.UIEditor (parent, **traits)
    Bases: traitsui.editor.Editor

    An editor that creates an embedded Traits UI.

    dispose ()
        Disposes of the contents of an editor.

    get_error_control ()
        Returns the editor's control for indicating error status.
```


init (*parent*)
 Finishes initializing the editor by creating the underlying toolkit widget.

init_ui (*parent*)
 Creates the traits UI for the editor.

restore_prefs (*prefs*)
 Restores any saved user preference information associated with the editor.

save_prefs ()
 Returns any user preference information associated with the editor.

update_editor ()
 Updates the editor when the object trait changes external to the editor.

traitsui.ui_info module

Defines the UIInfo class used to represent the object and editor content of an active Traits-based user interface.

class traitsui.ui_info.**UIInfo**
 Bases: `traits.has_traits.HasPrivateTraits`
 Represents the object and editor content of an active Traits-based user interface

bind (*name, value, id=None*)
 Binds a name to a value if it is not already bound.

bind_context ()
 Binds all of the associated context objects as traits of the object.

traitsui.ui_traits module

Defines common traits used within the traits.ui package.

class traitsui.ui_traits.**ATheme** (*value=None, **metadata*)
 Bases: `traits.trait_handlers.TraitType`
 Defines a trait whose value must be a traits UI Theme or a string that can be converted to one.

default_value = **None**

info_text = 'a Theme or string that can be used to define one'

validate (*object, name, value*)
 Validates that a specified value is valid for this trait.

class traitsui.ui_traits.**StatusItem** (*value=None, **traits*)
 Bases: `traits.has_traits.HasStrictTraits`

class traitsui.ui_traits.**ViewStatus** (*default_value=<traits.trait_handlers.NoDefaultSpecified object>, **metadata*)
 Bases: `traits.trait_handlers.TraitType`
 Defines a trait whose value must be a single StatusItem instance or a list of StatusItem instances.

default_value = **None**

info_text = 'None, a string, a single StatusItem instance, or a list or tuple of strings'

validate (*object, name, value*)
 Validates that a specified value is valid for this trait.

`traitsui.ui_traits.convert_theme (value, level=3)`
Converts a specified value to a Theme if possible.

traitsui.undo module

Defines the manager for Undo and Redo history for Traits user interface support.

class traitsui.undo.AbstractUndoItem

Bases: `traits.has_traits.HasPrivateTraits`

Abstract base class for undo items.

merge_undo (undo_item)

Merges two undo items if possible.

redo ()

Re-does the change.

undo ()

Undoes the change.

class traitsui.undo.ListUndoItem

Bases: `traitsui.undo.AbstractUndoItem`

A change to a list, which can be undone.

merge_undo (undo_item)

Merges two undo items if possible.

redo ()

Re-does the change.

undo ()

Undoes the change.

class traitsui.undo.UndoHistory

Bases: `traits.has_traits.HasStrictTraits`

Manages a list of undoable changes.

add (undo_item, extend=False)

Adds an UndoItem to the history.

clear ()

Clears the undo history.

extend (undo_item)

Extends the undo history.

If possible the method merges the new UndoItem with the last item in the history; otherwise, it appends the new item.

redo ()

Redoes an operation.

revert ()

Reverts all changes made so far and clears the history.

undo ()

Undoes an operation.

```
class traitsui.undo.UndoHistoryUndoItem
    Bases: traitsui.undo.AbstractUndoItem
```

An undo item for the undo history.

```
redo ()
    Re-does the change.
```

```
undo ()
    Undoes the change.
```

```
class traitsui.undo.UndoItem
    Bases: traitsui.undo.AbstractUndoItem
```

A change to an object trait, which can be undone.

```
merge_undo (undo_item)
    Merges two undo items if possible.
```

```
redo ()
    Re-does the change.
```

```
undo ()
    Undoes the change.
```

traitsui.value_tree module

Defines tree node classes and editors for various types of values.

```
class traitsui.value_tree.ArrayNode
    Bases: traitsui.value_tree.TupleNode
```

A tree node for arrays.

```
format_value (value)
    Returns the formatted version of the value.
```

```
class traitsui.value_tree.BoolNode
    Bases: traitsui.value_tree.SingleValueTreeNodeObject
```

A tree node for Boolean values.

```
class traitsui.value_tree.ClassNode
    Bases: traitsui.value_tree.ObjectNode
```

A tree node for classes.

```
format_value (value)
    Returns the formatted version of the value.
```

```
class traitsui.value_tree.ComplexNode
    Bases: traitsui.value_tree.SingleValueTreeNodeObject
```

A tree node for complex number values.

```
class traitsui.value_tree.DictNode
    Bases: traitsui.value_tree.TupleNode
```

A tree node for dictionaries.

```
format_value (value)
    Returns the formatted version of the value.
```

tno_can_delete (*node*)
Returns whether the object's children can be deleted.

tno_get_children (*node*)
Gets the object's children.

class traitsui.value_tree.**FloatNode**
Bases: *traitsui.value_tree.SingleValueTreeNodeObject*
A tree node for floating point values.

class traitsui.value_tree.**FunctionNode**
Bases: *traitsui.value_tree.SingleValueTreeNodeObject*
A tree node for functions

format_value (*value*)
Returns the formatted version of the value.

class traitsui.value_tree.**IntNode**
Bases: *traitsui.value_tree.SingleValueTreeNodeObject*
A tree node for integer values.

class traitsui.value_tree.**ListNode**
Bases: *traitsui.value_tree.TupleNode*
A tree node for lists.
format_value (*value*)
Returns the formatted version of the value.
tno_can_delete (*node*)
Returns whether the object's children can be deleted.
tno_can_insert (*node*)
Returns whether the object's children can be inserted (vs. appended).

class traitsui.value_tree.**MethodNode**
Bases: *traitsui.value_tree.MultiValueTreeNodeObject*
format_value (*value*)
Returns the formatted version of the value.
tno_get_children (*node*)
Gets the object's children.
tno_has_children (*node*)
Returns whether the object has children.

class traitsui.value_tree.**MultiValueTreeNodeObject**
Bases: *traitsui.value_tree.SingleValueTreeNodeObject*
A tree node for objects of types that have multiple values.
tno_allows_children (*node*)
Returns whether this object can have children (True for this class).
tno_has_children (*node*)
Returns whether the object has children (True for this class).

class traitsui.value_tree.**NoneNode**
Bases: *traitsui.value_tree.SingleValueTreeNodeObject*
A tree node for None values.

class traitsui.value_tree.ObjectNode

Bases: *traitsui.value_tree.MultiValueTreeNodeObject*

A tree node for objects.

format_value (*value*)

Returns the formatted version of the value.

tno_get_children (*node*)

Gets the object's children.

tno_has_children (*node*)

Returns whether the object has children.

class traitsui.value_tree.OtherNode

Bases: *traitsui.value_tree.SingleValueTreeNodeObject*

A tree node for single-value types for which there is not another node type.

class traitsui.value_tree.RootNode

Bases: *traitsui.value_tree.MultiValueTreeNodeObject*

A root node.

format_value (*value*)

Returns the formatted version of the value.

tno_get_children (*node*)

Gets the object's children.

class traitsui.value_tree.SetNode

Bases: *traitsui.value_tree.ListNode*

A tree node for sets.

format_value (*value*)

Returns the formatted version of the value.

class traitsui.value_tree.SingleValueTreeNodeObject

Bases: *traitsui.tree_node.TreeNodeObject*

A tree node for objects of types that have a single value.

format_value (*value*)

Returns the formatted version of the value.

node_for (*name*, *value*)

Returns the correct node type for a specified value.

tno_allows_children (*node*)

Returns whether this object can have children (False for this class).

tno_can_copy (*node*)

Returns whether the object's children can be copied (True for this class).

tno_can_delete (*node*)

Returns whether the object's children can be deleted (False for this class).

tno_can_insert (*node*)

Returns whether the object's children can be inserted (False, meaning children are appended, for this class).

tno_can_rename (*node*)

Returns whether the object's children can be renamed (False for this class).

tno_get_icon (*node, is_expanded*)
Returns the icon for a specified object.

tno_get_label (*node*)
Gets the label to display for a specified object.

tno_has_children (*node*)
Returns whether the object has children (False for this class).

tno_set_label (*node, label*)
Sets the label for a specified object.

class traitsui.value_tree.**StringNode**
Bases: *traitsui.value_tree.SingleValueTreeNodeObject*
A tree node for strings.

format_value (*value*)
Returns the formatted version of the value.

class traitsui.value_tree.**TraitsNode**
Bases: *traitsui.value_tree.ObjectNode*
A tree node for traits.

tno_get_children (*node*)
Gets the object's children.

tno_has_children (*node*)
Returns whether the object has children.

tno_when_children_changed (*node, listener, remove*)
Sets up or removes a listener for children being changed on a specified object.

tno_when_children_replaced (*node, listener, remove*)
Sets up or removes a listener for children being replaced on a specified object.

class traitsui.value_tree.**TupleNode**
Bases: *traitsui.value_tree.MultiValueTreeNodeObject*
A tree node for tuples.

format_value (*value*)
Returns the formatted version of the value.

tno_get_children (*node*)
Gets the object's children.

tno_has_children (*node*)
Returns whether the object has children, based on the length of the tuple.

traitsui.value_tree.**basic_types** ()

traitsui.view module

Defines the View class used to represent the structural content of a Traits-based user interface.

class traitsui.view.**View** (**values, **traits*)
Bases: *traitsui.view_element.ViewElement*
A Traits-based user interface for one or more objects.

The attributes of the View object determine the contents and layout of an attribute-editing window. A View object contains a set of Group, Item, and Include objects. A View object can be an attribute of an object derived from HasTraits, or it can be a standalone object.

replace_include (*view_elements*)

Replaces any items that have an ID with an Include object with the same ID, and puts the object with the ID into the specified ViewElements object.

set_content (**values*)

Sets the content of a view.

ui (*context, parent=None, kind=None, view_elements=None, handler=None, id="", scrollable=None, args=None*)

Creates a **UI** object, which generates the actual GUI window or panel from a set of view elements.

Parameters

- **context** (*object or dictionary*) – A single object or a dictionary of string/object pairs, whose trait attributes are to be edited. If not specified, the current object is used.
- **parent** (*window component*) – The window parent of the View object's window
- **kind** (*string*) – The kind of window to create. See the **AKind** trait for details. If *kind* is unspecified or None, the **kind** attribute of the View object is used.
- **view_elements** (*ViewElements object*) – The set of Group, Item, and Include objects contained in the view. Do not use this parameter when calling this method directly.
- **handler** (*Handler object*) – A handler object used for event handling in the dialog box. If None, the default handler for Traits UI is used.
- **id** (*string*) – A unique ID for persisting preferences about this user interface, such as size and position. If not specified, no user preferences are saved.
- **scrollable** (*Boolean*) – Indicates whether the dialog box should be scrollable. When set to True, scroll bars appear on the dialog box if it is not large enough to display all of the items in the view at one time.

traitsui.view_element module

Defines the abstract ViewElement class that all trait view template items (i.e., View, Group, Item, Include) derive from.

class traitsui.view_element.DefaultViewElement

Bases: *traitsui.view_element.ViewElement*

A view element that can be used as a default value for traits whose value is a view element.

class traitsui.view_element.ViewElement

Bases: *traits.has_traits.HasPrivateTraits*

An element of a view.

is_includable ()

Returns whether the object is replacable by an Include object.

replace_include (*view_elements*)

Searches the current object's **content** attribute for objects that have an **id** attribute, and replaces each one with an Include object with the same **id** value, and puts the replaced object into the specified ViewElements object.

Parameters **view_elements** (*ViewElements object*) – Object containing Group, Item, and Include objects

class traitsui.view_element.**ViewSubElement**

Bases: *traitsui.view_element.ViewElement*

Abstract class representing elements that can be contained in a view.

traitsui.view_elements module

Define the ViewElements class, which is used to define a (typically class-based) hierarchical name space of related ViewElement objects.

Normally there is a ViewElements object associated with each Traits-based class, which contains all of the ViewElement objects associated with the class. The ViewElements object is also linked to the ViewElements objects of its associated class's parent classes.

class traitsui.view_elements.**SearchStackItem**

Bases: *traits.has_traits.HasStrictTraits*

class traitsui.view_elements.**ViewElements**

Bases: *traits.has_traits.HasStrictTraits*

Defines a hierarchical name space of related ViewElement objects.

filter_by (*klass=None*)

Returns a sorted list of all names accessible from the ViewElements object that are of a specified (ViewElement) type.

find (*name, stack=None*)

Finds a specified ViewElement within the specified (optional) search context.

2.1.3 Module contents

3.1 Writing a graphical application for scientific programming using TraitsUI 6.0

A step by step guide for a non-programmer

Author Gael Varoquaux

Date 2018-04-12

License BSD

Building interactive Graphical User Interfaces (GUIs) is a hard problem, especially for somebody who has not had training in IT. TraitsUI is a python module that provides a great answer to this problem. I have found that I am incredibly productive when creating graphical application using traitsUI. However I had to learn a few new concepts and would like to lay them down together in order to make it easier for others to follow my footsteps.

This document is intended to help a non-programmer to use traits and traitsUI to write an interactive graphical application. The reader is assumed to have some basic python scripting knowledge (see ref¹ for a basic introduction). Knowledge of numpy/scipy² helps understanding the data processing aspects of the examples, but may not be paramount. Some examples rely on matplotlib³. This document is **not** a replacement for user manuals and references of the different packages (traitsUI⁴, scipy, matplotlib). It provides a “cookbook” approach, and not a reference.

This tutorial provides step-by-step guide to building a medium-size application. The example chosen is an application used to do control of a camera, analysis of the retrieved data and display of the results. This tutorial focuses on building the general structure and flow-control of the application, and on the aspects specific to traitsUI programming. Interfacing with the hardware or processing the data is left aside. The tutorial progressively introduces the tools used, and in the end presents the skeleton of a real application that has been developed for real-time controlling of an experiment, monitoring through a camera, and processing the data. The tutorial goes into more and more intricate details that are necessary to build the final application. Each section is in itself independent of the following ones. The complete beginner trying to use this as an introduction should not expect to understand all the details in a first pass.

¹ python tutorial: <http://docs.python.org/tut/tut.html>

² The scipy website: <http://www.scipy.org>

³ The matplotlib website: <http://matplotlib.sourceforge.net>

⁴ The traits and traitsUI user guide: <http://code.enthought.com/traits>

The author's experience while working on several projects in various physics labs is that code tends to be created in an 'organic' way, by different people with various levels of qualification in computer development, and that it rapidly decays to a disorganized and hard-to-maintain code base. This tutorial tries to prevent this by building an application shaped for modularity and readability.

3.1.1 From objects to dialogs using traitsUI

Creating user interfaces directly through a toolkit is a time-consuming process. It is also a process that does not integrate well in the scientific-computing work-flow, as, during the elaboration of algorithms and data-flow, the objects that are represented in the GUI are likely to change often.

Visual computing, where the programmer creates first a graphical interface and then writes the callbacks of the graphical objects, gives rise to a slow development cycle, as the work-flow is centered on the GUI, and not on the code.

TraitsUI provides a beautiful answer to this problem by building graphical representations of an object. Traits and TraitsUI have their own manuals (<http://code.enthought.com/traits/>) and the reader is encouraged to refer to these for more information.

We will use TraitsUI for *all* our GUIs. This forces us to store all the data and parameters in objects, which is good programming style. The GUI thus reflects the structure of the code, which makes it easier to understand and extend.

In this section we will focus on creating dialogs that allow the user to input parameters graphically in the program.

Object-oriented programming

Software engineering is a difficult field. As programs grow they become harder and harder to grasp for the developer. This problem is not new and has sometimes been known as the "tar pit". Many attempts have been made to mitigate the difficulties. Most often they consist in finding useful abstractions that allow the developer to manipulate larger ideas, rather than their software implementation.

Code re-use is paramount for good software development. It reduces the number of code-lines required to read and understand and allows to identify large operations in the code. Functions and procedures have been invented to avoid copy-and-pasting code, and hide the low-level details of an operation.

Object-oriented programming allows yet more modularity and abstraction.

Objects, attributes and methods

Suppose you want your program to manipulate geometric objects. You can teach the computer that a point is a set of 3 numbers, you can teach it how to rotate that point along a given axis. Now you want to use spheres too. With a bit more work your program has functions to create points, spheres, etc. It knows how to rotate them, to mirror them, to scale them. So in pure procedural programming you will have procedures to rotate, scale, mirror, each one of your objects. If you want to rotate an object you will first have to find its type, then apply the right procedure to rotate it.

Object-oriented programming introduces a new abstraction: the *object*. It consists of both data (our 3 numbers, in the case of a point), and procedures that use and modify this data (e.g., rotations). The data entries are called "*attributes*" of the object and the procedures "*methods*". Thus with object oriented programming an object "knows" how to be rotated.

A point object could be implemented in python with:

code snippet #0

```
from numpy import cos, sin

class Point(object):
```

```

""" 3D Point objects """
x = 0.
y = 0.
z = 0.

def rotate_z(self, theta):
    """ rotate the point around the Z axis """
    xtemp = cos(theta) * self.x + sin(theta) * self.y
    ytemp = -sin(theta) * self.x + cos(theta) * self.y
    self.x = xtemp
    self.y = ytemp
    
```

This code creates a *Point* class. Points objects can be created as *instances* of the *Point* class:

```

>>> from numpy import pi
>>> p = Point()
>>> p.x = 1
>>> p.rotate_z(pi)
>>> p.x
-1.0
>>> p.y
1.2246467991473532e-16
    
```

When manipulating objects, the developer does not need to know the internal details of their procedures. As long as the object has a *rotate* method, the developer knows how to rotate it.

Note: Beginners often use objects as structures: entities with several data fields useful to pass data around in a program. Objects are much more than that: they have methods. They are ‘active’ data structures that know how to modify themselves. Part of the point of object-oriented programming is that the object is responsible for modifying itself through its methods. The object therefore takes care of its internal logic and the consistency between its attributes.

In python, dictionaries make great structures and are more suited for such a use than objects.

Classes and inheritance

Suppose you have already created a *Point* class that tells your program what a point is, but that you also want some points to have a color. Instead of copy-and-pasting the *Point* class and adding a color attribute, you can define a new class *ColoredPoint* that inherits all of the *Point* class’s methods and attributes:

```

class ColoredPoint(Point):
    """ Colored 3D point """
    color = "white"
    
```

You do not have to implement rotation for the *ColoredPoint* class as it has been inherited from the *Point* class. This is one of the huge gains of object-oriented programming: objects are organized in classes and sub-classes, and method to manipulate objects are derived from the objects parent-ship: a *ColoredPoint* is only a special case of *Point*. This proves very handy on large projects.

Note: To stress the differences between classes and their instances (objects), classes are usually named with capital letters, and objects only with lower case letters.

An object and its representation

Objects are code entities that can be easily pictured by the developer. The *TraitsUI* python module allows the user to edit objects attributes with dialogs that form a graphical representation of the object.

In our example application, each process or experimental device is represented in the code as an object. These objects all inherit from the *HasTraits*, class which supports creating graphical representations of attributes. To be able to build the dialog, the *HasTraits* class enforces that the types of all the attributes are specified in the class definition.

The *HasTraits* objects have a *configure_traits()* method that brings up a dialog to edit the objects' attributes specified in its class definition.

Here we define a camera object (which, in our real world example, is a camera interfaced to python through the ctypes⁵ module), and show how to open a dialog to edit its properties :

code snippet #1

```
from traits.api import *
from traitsui.api import *

class Camera(HasTraits):
    """ Camera object """

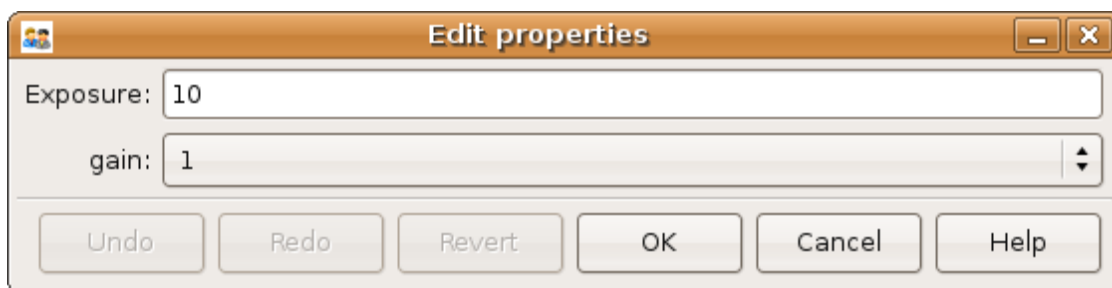
    gain = Enum(1, 2, 3,
               desc="the gain index of the camera",
               label="gain", )

    exposure = CInt(10,
                   desc="the exposure time, in ms",
                   label="Exposure", )

    def capture(self):
        """ Captures an image on the camera and returns it """
        print "capturing an image at %i ms exposure, gain: %i" % (
            self.exposure, self.gain )

if __name__ == "__main__":
    camera = Camera()
    camera.configure_traits()
    camera.capture()
```

The *camera.configure_traits()* call in the above example opens a dialog that allows the user to modify the camera object's attributes:



This dialog forms a graphical representation of our camera object. We will see that it can be embedded in GUI panels to build more complex GUIs that allow us to control many objects.

We will build our application around objects and their graphical representation, as this mapping of the code to the GUI helps the developer to understand the code.

⁵ ctypes: <http://starship.python.net/crew/theller/ctypes/>

Displaying several objects in the same panel

We now know how to build a dialog from objects. If we want to build a complex application we are likely to have several objects, for instance one corresponding to the camera we want to control, and one describing the experiment that the camera monitors. We do not want to have to open a new dialog per object: this would force us to describe the GUI in terms of graphical objects, and not structural objects. We want the GUI to be a natural representation of our objects, and we want the Traits module to take care of that.

The solution is to create a container object, that has as attributes the objects we want to represent. Playing with the *View* attribute of the object, we can control how the representation generated by Traits looks like (see the TraitsUI manual):

code snippet #2

```
from traits.api import *
from traitsui.api import *

class Camera(HasTraits):
    gain = Enum(1, 2, 3, )
    exposure = CInt(10, label="Exposure", )

class TextDisplay(HasTraits):
    string = String()

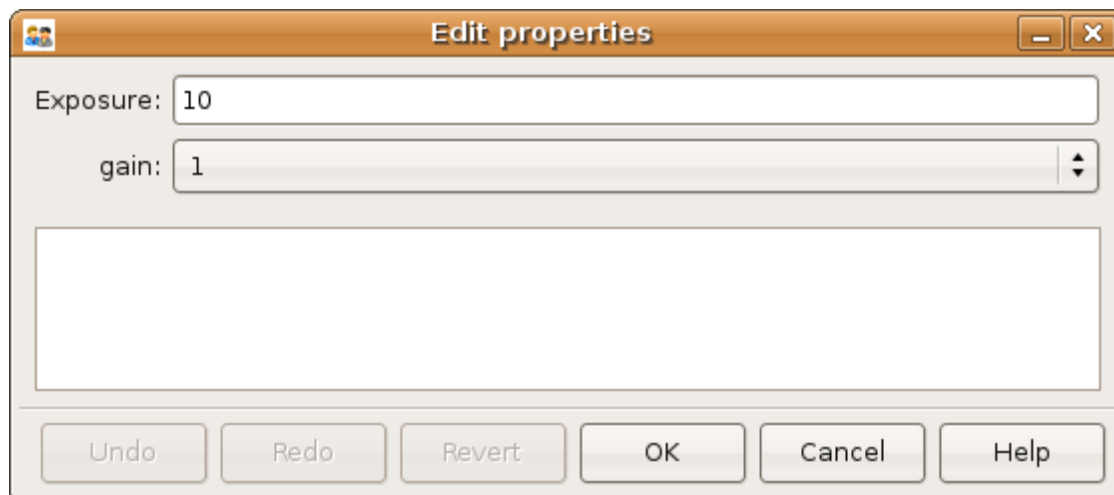
    view= View( Item('string', show_label=False, springy=True, style='custom
    ↪ ') )

class Container(HasTraits):
    camera = Instance(Camera)
    display = Instance(TextDisplay)

    view = View(
        Item('camera', style='custom', show_label=False, ),
        Item('display', style='custom', show_label=False, ),
    )

container = Container(camera=Camera(), display=TextDisplay())
container.configure_traits()
```

The call to *configure_traits()* creates the following dialog, with the representation of the *Camera* object created is the last example on top, and the *Display* object below it:



The *View* attribute of the *container* object has been tweaked to get the representation we are interested in: *traitsUI* is told to display the *camera* item with a *'custom'* style, which instructs it to display the representation of the object inside the current panel. The *'show_label'* argument is set to *False* as we do not want the name of the displayed object (*'camera'*, for instance) to appear in the dialog. See the *traitsUI* manual for more details on this powerful feature.

The *camera* and *display* objects are created during the call to the creator of the *container* object, and passed as its attributes immediately: “*container = Container(camera=Camera(), display=TextDisplay())*”

Writing a “graphical script”

If you want to create an application that has a very linear flow, popping up dialogs when user input is required, like a “setup wizard” often used to install programs, you already have all the tools to do it. You can use object oriented programming to write your program, and call the objects *configure_traits* method each time you need user input. This might be an easy way to modify an existing script to make it more user friendly.

The following section will focus on making interactive programs, where the user uses the graphical interface to interact with it in a continuous way.

3.1.2 From graphical to interactive

In an interactive application, the program responds to user interaction. This requires a slight paradigm shift in our programming methods.

Object-oriented GUIs and event loops

In a GUI application, the order in which the different parts of the program are executed is imposed by the user, unlike in a numerical algorithm, for instance, where the developer chooses the order of execution of his program. An event loop allows the programmer to develop an application in which each user action triggers an event, by stacking the user created events on a queue, and processing them in the order in which they appeared.

A complex GUI is made of a large numbers of graphical elements, called widgets (e.g., text boxes, check boxes, buttons, menus). Each of these widgets has specific behaviors associated with user interaction (modifying the content of a text box, clicking on a button, opening a menu). It is natural to use objects to represent the widgets, with their behavior being set in the object’s methods.

Dialogs populated with widgets are automatically created by *traitsUI* in the *configure_traits()* call. *traitsUI* allow the developer to not worry about widgets, but to deal only with objects and their attributes. This is a fabulous gain as the widgets no longer appear in the code, but only the attributes they are associated to.

A *HasTraits* object has an *edit_traits()* method that creates a graphical panel to edit its attributes. This method creates and returns the panel, but does not start its event loop. The panel is not yet “alive”, unlike with the *configure_traits()* method. Traits uses the *wxWidget* toolkit by default to create its widget. They can be turned live and displayed by starting a *wx* application, and its main loop (ie event loop in *wx* speech).

code snippet #3

```
from traits.api import *
import wx

class Counter(HasTraits):
    value = Int()

Counter().edit_traits()
wx.PySimpleApp().MainLoop()
```

The `Counter().edit_traits()` line creates a counter object and its representation, a dialog with one integer represented. However it does not display it until a wx application is created, and its main loop is started.

Usually it is not necessary to create the wx application yourself, and to start its main loop, traits will do all this for you when the `.configure_traits()` method is called.

Reactive programming

When the event loop is started, the program flow is no longer simply controlled by the code: the control is passed on to the event loop, and it processes events, until the user closes the GUI, and the event loop returns to the code.

Interactions with objects generate events, and these events can be associated to callbacks, ie functions or methods processing the event. In a GUI, callbacks created by user-generated events are placed on an “event stack”. The event loop processes each call on the event queue one after the other, thus emptying the event queue. The flow of the program is still sequential (two code blocks never run at the same time in an event loop), but the execution order is chosen by the user, and not by the developer.

Defining callbacks for the modification of an attribute *foo* of a *HasTraits* object can be done by creating a method called `_foo_changed()`. Here is an example of a dialog with two textboxes, *input* and *output*. Each time *input* is modified, its content is duplicated to output.

code snippet #4

```
from traits.api import *

class EchoBox(HasTraits):
    input = Str()
    output = Str()

    def _input_changed(self):
        self.output = self.input

EchoBox().configure_traits()
```

Events that do not correspond to a modification of an attribute can be generated with a *Button* traits. The callback is then called `_foo_fired()`. Here is an example of an interactive *traitsUI* application using a button:

code snippet #5

```
from traits.api import *
from traitsui.api import View, Item, ButtonEditor

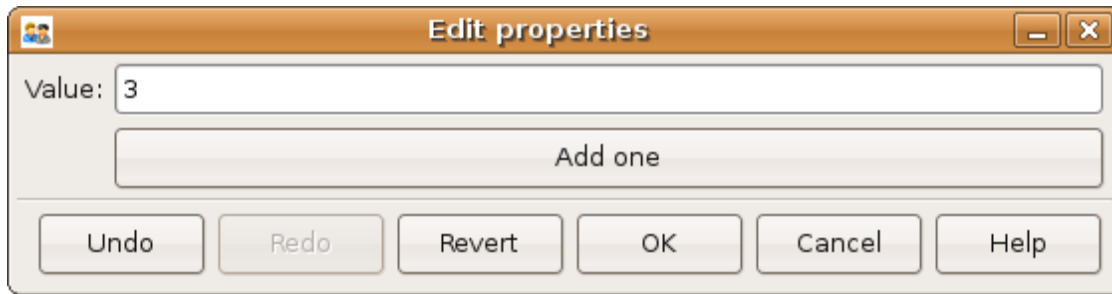
class Counter(HasTraits):
    value = Int()
    add_one = Button()

    def _add_one_fired(self):
        self.value += 1

    view = View('value', Item('add_one', show_label=False))

Counter().configure_traits()
```

Clicking on the button adds the `_add_one_fired()` method to the event queue, and this method gets executed as soon as the GUI is ready to handle it. Most of the time that is almost immediately.



This programming pattern is called *reactive programming*: the objects react to the changes made to their attributes. In complex programs where the order of execution is hard to figure out, and bound to change, like some interactive data processing application, this pattern is extremely efficient.

Using *Button* traits and a clever set of objects interacting with each others, complex interactive applications can be built. These applications are governed by the events generated by the user, in contrast to script-like applications (batch programming). Executing a long operation in the event loop blocks the reactions of the user-interface, as other events callbacks are not processed as long as the long operation is not finished. In the next section we will see how we can execute several operations in the same time.

3.1.3 Breaking the flow in multiple threads

What are threads ?

A standard python program executes in a sequential way. Consider the following code snippet :

```
do_a ()
do_b ()
do_c ()
```

do_b() is not called until *do_a()* is finished. Even in event loops everything is sequential. In some situation this can be very limiting. Suppose we want to capture an image from a camera and that it is a very lengthy operation. Suppose also that no other operation in our program requires the capture to be complete. We would like to have a different “timeline” in which the camera capture instructions can happen in a sequential way, while the rest of the program continues in parallel.

Threads are the solution to this problem: a thread is a portion of a program that can run concurrently with other portions of the program.

Programming with threads is difficult as instructions are no longer executed in the order they are specified and the output of a program can vary from a run to another, depending on subtle timing issues. These problems are known as “race conditions” and to minimize them you should avoid accessing the same objects in different threads. Indeed if two different threads are modifying the same object at the same time, unexpected things can happen.

Threads in python

In python a thread can be implemented with a *Thread* object, from the *threading*⁶ module. To create your own execution thread, subclass the *Thread* object and put the code that you want to run in a separate thread in its *run* method. You can start your thread using its *start* method:

code snippet #6

⁶ threading: <http://docs.python.org/lib/module-threading.html>


```

from threading import Thread
from time import sleep

class MyThread(Thread):
    def run(self):
        sleep(2)
        print "MyThread done"

my_thread = MyThread()

my_thread.start()
print "Main thread done"
    
```

The above code yields the following output:

```

Main thread done
MyThread done
    
```

Getting threads and the GUI event loop to play nice

Suppose you have a long-running job in a TraitsUI application. If you implement this job as an event placed on the event loop stack, it is going to freeze the event loop while running, and thus freeze the UI, as events will accumulate on the stack, but will not be processed as long as the long-running job is not done (remember, the event loop is sequential). To keep the UI responsive, a thread is the natural answer.

Most likely you will want to display the results of your long-running job on the GUI. However, as usual with threads, one has to be careful not to trigger race-conditions. Naively manipulating the GUI objects in your thread will lead to race conditions, and unpredictable crash: suppose the GUI was repainting itself (due to a window move, for instance) when you modify it.

In a wxPython application, if you start a thread, GUI event will still be processed by the GUI event loop. To avoid collisions between your thread and the event loop, the proper way of modifying a GUI object is to insert the modifications in the event loop, using the *GUI.invoke_later()* call. That way the GUI will apply your instructions when it has time.

Recent versions of the TraitsUI module (post October 2006) propagate the changes you make to a *HasTraits* object to its representation in a thread-safe way. However it is important to have in mind that modifying an object with a graphical representation is likely to trigger race-conditions as it might be modified by the graphical toolkit while you are accessing it. Here is an example of code inserting the modification to traits objects by hand in the event loop:

code snippet #7

```

from threading import Thread
from time import sleep
from traits.api import *
from traitsui.api import View, Item, ButtonEditor

class TextDisplay(HasTraits):
    string = String()

    view= View( Item('string',show_label=False, springy=True, style='custom'
↵) )

class CaptureThread(Thread):
    def run(self):
        self.display.string = 'Camera started\n' + self.display.string
    
```

```

n_img = 0
while not self.wants_abort:
    sleep(.5)
    n_img += 1
    self.display.string = '%d image captured\n' % n_img \
                        + self.display.string
self.display.string = 'Camera stopped\n' + self.display.string

class Camera(HasTraits):
    start_stop_capture = Button()
    display = Instance(TextDisplay)
    capture_thread = Instance(CaptureThread)

    view = View( Item('start_stop_capture', show_label=False ))

    def _start_stop_capture_fired(self):
        if self.capture_thread and self.capture_thread.isAlive():
            self.capture_thread.wants_abort = True
        else:
            self.capture_thread = CaptureThread()
            self.capture_thread.wants_abort = False
            self.capture_thread.display = self.display
            self.capture_thread.start()

class MainWindow(HasTraits):
    display = Instance(TextDisplay, ())

    camera = Instance(Camera)

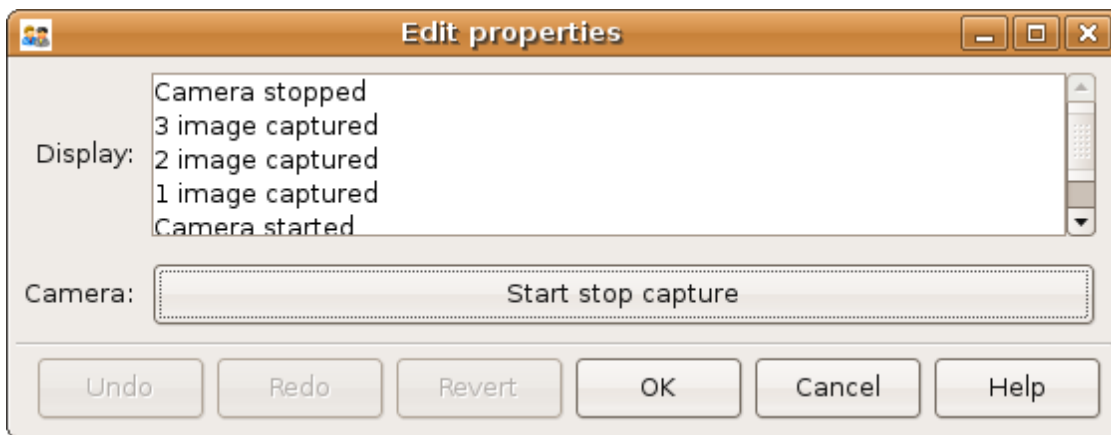
    def _camera_default(self):
        return Camera(display=self.display)

    view = View('display', 'camera', style="custom", resizable=True)

if __name__ == '__main__':
    MainWindow().configure_traits()

```

This creates an application with a button that starts or stop a continuous camera acquisition loop.



When the “Start stop capture” button is pressed the `_start_stop_capture_fired` method is called. It checks to see if a `CaptureThread` is running or not. If none is running, it starts a new one. If one is running, it sets its `wants_abort` attribute to true.

The thread checks every half a second to see if its attribute *wants_abort* has been set to true. If this is the case, it aborts. This is a simple way of ending the thread through a GUI event.

Using different threads lets the operations avoid blocking the user interface, while also staying responsive to other events. In the real-world application that serves as the basis of this tutorial, there are 2 threads and a GUI event loop.

The first thread is an acquisition loop, during which the program loops, waiting for a image to be captured on the camera (the camera is controlled by external signals). Once the image is captured and transferred to the computer, the acquisition thread saves it to the disk and spawns a thread to process the data, then returns to waiting for new data while the processing thread processes the data. Once the processing thread is done, it displays its results (by inserting the display events in the GUI event loop) and dies. The acquisition thread refuses to spawn a new processing thread if there still is one running. This makes sure that data is never lost, no matter how long the processing might be.

There are thus up to 3 set of instructions running concurrently: the GUI event loop, responding to user-generated events, the acquisition loop, responding to hardware-generated events, and the processing jobs, doing the numerical intensive work.

In the next section we are going to see how to add a home-made element to traits, in order to add new possibilities to our application.

3.1.4 Extending TraitsUI: Adding a matplotlib figure to our application

This section gives a few guidelines on how to build your own traits editor. A traits editor is the view associated with a trait that allows the user to graphically edit its value. We can twist a bit the notion and simply use it to graphically represent the attribute. This section involves a bit of *wxPython* code that may be hard to understand if you do not know *wxPython*, but it will bring a lot of power and flexibility to how you use traits. The reason it appears in this tutorial is that I wanted to insert a matplotlib in my *traitsUI* application. It is not necessary to fully understand the code of this section to be able to read on.

I should stress that there already exists a plotting module that provides traits editors for plotting, and that is very well integrated with traits: *chaco*⁷.

Making a *traits* editor from a Matplotlib plot

To use traits, the developer does not need to know its internals. However traits does not provide an editor for every need. If we want to insert a powerful tool for plotting we have to get our hands a bit dirty and create our own traits editor.

This involves some *wxPython* coding, as we need to translate a *wxPython* object to a traits editor by providing the corresponding API (i.e. the standard way of building a *traits* editor), so that the *traits* framework will know how to create the editor.

Traits editor are created by an editor factory that instantiates an editor class and passes it the object that the editor represents in its *value* attribute. It calls the editor *init()* method to create the *wx* widget. Here we create a *wx* figure canvas from a matplotlib figure using the matplotlib *wx* backend. Instead of displaying this widget, we set its control as the *control* attribute of the editor. TraitsUI takes care of displaying and positioning the editor.

code snippet #8

```
import wx

import matplotlib
# We want matplotlib to use a wxPython backend
matplotlib.use('WXAgg')
```

⁷ chaco: <http://code.enthought.com/chaco/>

```

from matplotlib.backends.backend_wxagg import FigureCanvasWxAgg as FigureCanvas
from matplotlib.figure import Figure
from matplotlib.backends.backend_wx import NavigationToolbar2Wx

from traits.api import Any, Instance
from traitsui.wx.editor import Editor
from traitsui.wx.basic_editor_factory import BasicEditorFactory

class _MPLFigureEditor(Editor):

    scrollable = True

    def init(self, parent):
        self.control = self._create_canvas(parent)
        self.set_tooltip()

    def update_editor(self):
        pass

    def _create_canvas(self, parent):
        """ Create the MPL canvas. """
        # The panel lets us add additional controls.
        panel = wx.Panel(parent, -1, style=wx.CLIP_CHILDREN)
        sizer = wx.BoxSizer(wx.VERTICAL)
        panel.SetSizer(sizer)
        # matplotlib commands to create a canvas
        mpl_control = FigureCanvas(panel, -1, self.value)
        sizer.Add(mpl_control, 1, wx.LEFT | wx.TOP | wx.GROW)
        toolbar = NavigationToolbar2Wx(mpl_control)
        sizer.Add(toolbar, 0, wx.EXPAND)
        self.value.canvas.SetMinSize((10,10))
        return panel

class MPLFigureEditor(BasicEditorFactory):

    klass = _MPLFigureEditor

if __name__ == "__main__":
    # Create a window to demo the editor
    from traits.api import HasTraits
    from traitsui.api import View, Item
    from numpy import sin, cos, linspace, pi

    class Test(HasTraits):

        figure = Instance(Figure, ())

        view = View(Item('figure', editor=MPLFigureEditor(),
                        show_label=False),
                    width=400,
                    height=300,
                    resizable=True)

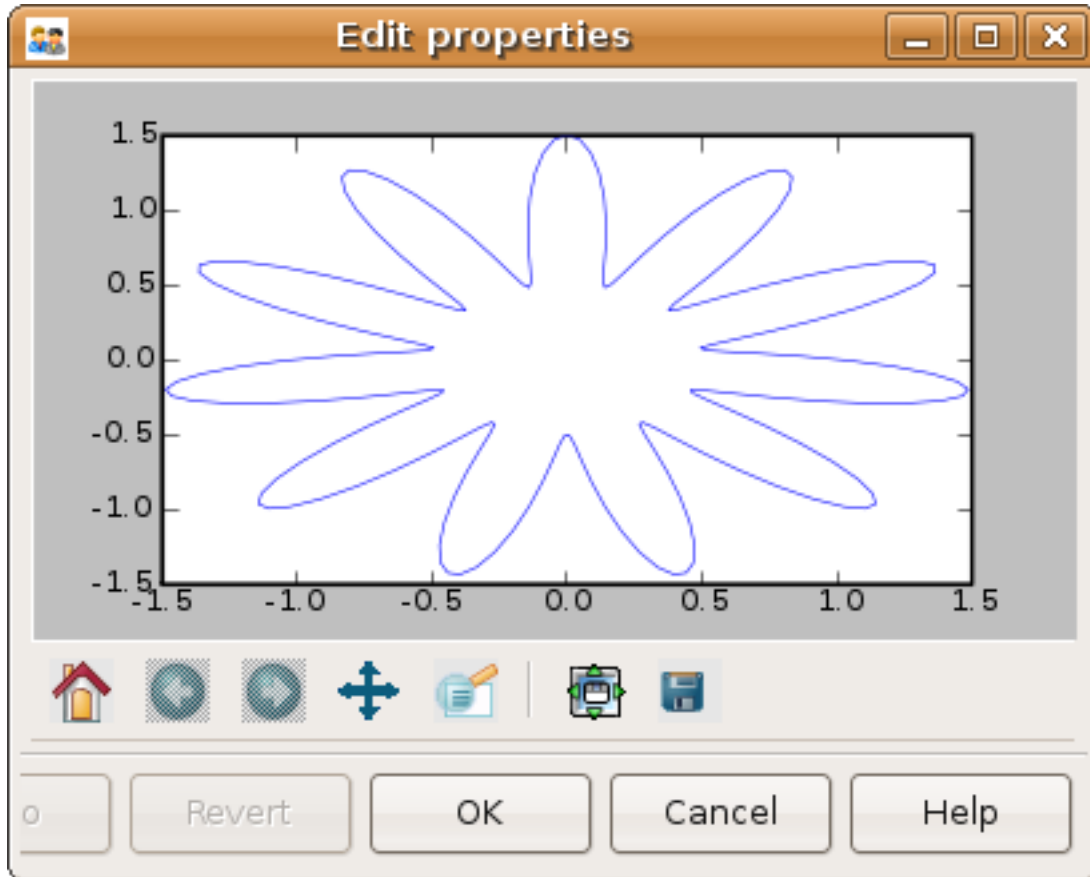
        def __init__(self):
            super(Test, self).__init__()
            axes = self.figure.add_subplot(111)

```

```
t = linspace(0, 2*pi, 200)
axes.plot(sin(t)*(1+0.5*cos(11*t)), cos(t)*(1+0.5*cos(11*t)))

Test().configure_traits()
```

This code first creates a traitsUI editor for a matplotlib figure, and then a small dialog to illustrate how it works:



The matplotlib figure traits editor created in the above example can be imported in a traitsUI application and combined with the power of traits. This editor allows to insert a matplotlib figure in a traitsUI dialog. It can be modified using reactive programming, as demonstrated in section 3 of this tutorial. However, once the dialog is up and running, you have to call `self.figure.canvas.draw()` to update the canvas if you made modifications to the figure. The matplotlib user guide³ details how this object can be used for plotting.

3.1.5 Putting it all together: a sample application

The real world problem that motivated the writing of this tutorial is an application that retrieves data from a camera, processes it and displays results and controls to the user. We now have all the tools to build such an application. This section gives the code of a skeleton of this application. This application actually controls a camera on a physics experiment (Bose-Einstein condensation), at the university of Toronto.

The reason I am providing this code is to give an example to study of how a full-blown application can be built. This code can be found in the [tutorial's zip file](#) (it is the file `application.py`).

- The camera will be built as an object. Its real attributes (exposure time, gain...) will be represented as the object's attributes, and exposed through traitsUI.

- The continuous acquisition/processing/user-interaction will be handled by appropriate threads, as discussed in section 2.3.
- The plotting of the results will be done through the MPLWidget object.

The imports

The MPLFigureEditor is imported from the last example.

```
from threading import Thread
from time import sleep
from traits.api import *
from traitsui.api import View, Item, Group, HSplit, Handler
from traitsui.menu import NoButtons
from mpl_figure_editor import MPLFigureEditor
from matplotlib.figure import Figure
from scipy import *
import wx
```

User interface objects

These objects store information for the program to interact with the user via traitsUI.

```
class Experiment(HasTraits):
    """ Object that contains the parameters that control the experiment,
    modified by the user.
    """
    width = Float(30, label="Width", desc="width of the cloud")
    x = Float(50, label="X", desc="X position of the center")
    y = Float(50, label="Y", desc="Y position of the center")

class Results(HasTraits):
    """ Object used to display the results.
    """
    width = Float(30, label="Width", desc="width of the cloud")
    x = Float(50, label="X", desc="X position of the center")
    y = Float(50, label="Y", desc="Y position of the center")

    view = View( Item('width', style='readonly'),
                  Item('x', style='readonly'),
                  Item('y', style='readonly'),
                  )
```

The camera object also is a real object, and not only a data structure: it has a method to acquire an image (or in our case simulate acquiring), using its attributes as parameters for the acquisition.

```
class Camera(HasTraits):
    """ Camera objects. Implements both the camera parameters controls, and
    the picture acquisition.
    """
    exposure = Float(1, label="Exposure", desc="exposure, in ms")
    gain = Enum(1, 2, 3, label="Gain", desc="gain")

    def acquire(self, experiment):
        X, Y = indices((100, 100))
        Z = exp(-((X-experiment.x)**2+(Y-experiment.y)**2)/experiment.
        ↪width**2)
```

```

Z += 1-2*rand(100,100)
Z *= self.exposure
Z[Z>2] = 2
Z = Z**self.gain
return(Z)

```

Threads and flow control

There are three threads in this application:

- The GUI event loop, the only thread running at the start of the program.
- The acquisition thread, started through the GUI. This thread is an infinite loop that waits for the camera to be triggered, retrieves the images, displays them, and spawns the processing thread for each image received.
- The processing thread, started by the acquisition thread. This thread is responsible for the numerical intensive work of the application. It processes the data and displays the results. It dies when it is done. One processing thread runs per shot acquired on the camera, but to avoid accumulation of threads in the case that the processing takes longer than the time lapse between two images, the acquisition thread checks that the processing thread is done before spawning a new one.

```

def process(image, results_obj):
    """ Function called to do the processing """
    X, Y = indices(image.shape)
    x = sum(X*image)/sum(image)
    y = sum(Y*image)/sum(image)
    width = sqrt(abs(sum(((X-x)**2+(Y-y)**2)*image)/sum(image)))
    results_obj.x = x
    results_obj.y = y
    results_obj.width = width

class AcquisitionThread(Thread):
    """ Acquisition loop. This is the worker thread that retrieves images
    from the camera, displays them, and spawns the processing job. """
    wants_abort = False

    def process(self, image):
        """ Spawns the processing job. """
        try:
            if self.processing_job.isAlive():
                self.display("Processing too slow")
                return
        except AttributeError:
            pass
        self.processing_job = Thread(target=process, args=(image,
                                                            self.results))
        self.processing_job.start()

    def run(self):
        """ Runs the acquisition loop. """
        self.display('Camera started')
        n_img = 0
        while not self.wants_abort:
            n_img += 1
            img = self.acquire(self.experiment)
            self.display('%d image captured' % n_img)

```

```

self.image_show(img)
self.process(img)
sleep(1)
self.display('Camera stopped')

```

The GUI elements

The GUI of this application is separated in two (and thus created by a sub-class of `SplitApplicationWindow`).

On the left a plotting area, made of an MPL figure and its editor, displays the images acquired by the camera.

On the right a panel hosts the TraitsUI representation of a `ControlPanel` object. This object is mainly a container for our other objects, but it also has an `Button` for starting or stopping the acquisition, and a string (represented by a textbox) to display information on the acquisition process. The view attribute is tweaked to produce a pleasant and usable dialog. Tabs are used to help the display to be light and clear.

```

class ControlPanel(HasTraits):
    """ This object is the core of the traitsUI interface. Its view is
    the right panel of the application, and it hosts the method for
    interaction between the objects and the GUI.
    """
    experiment = Instance(Experiment, ())
    camera = Instance(Camera, ())
    figure = Instance(Figure)
    results = Instance(Results, ())
    start_stop_acquisition = Button("Start/Stop acquisition")
    results_string = String()
    acquisition_thread = Instance(AcquisitionThread)
    view = View(Group(
        Group(
            Item('start_stop_acquisition', show_label=False),
            Item('results_string', show_label=False,
                springy=True, style='custom'),
            label="Control", dock='tab',),
        Group(
            Group(
                Group(
                    Item('experiment', style='custom', show_
→label=False),
                    label="Input",),
                Group(
                    Item('results', style='custom', show_
→label=False),
                    label="Results",),
                label='Experiment', dock="tab"),
            Item('camera', style='custom', show_label=False, dock="tab"),
            layout='tabbed'),
    ))

    def _start_stop_acquisition_fired(self):
        """ Callback of the "start stop acquisition" button. This starts
        the acquisition thread, or kills it.
        """
        if self.acquisition_thread and self.acquisition_thread.isAlive():
            self.acquisition_thread.wants_abort = True
        else:
            self.acquisition_thread = AcquisitionThread()
            self.acquisition_thread.display = self.add_line

```



```

        self.acquisition_thread.acquire = self.camera.acquire
        self.acquisition_thread.experiment = self.experiment
        self.acquisition_thread.image_show = self.image_show
        self.acquisition_thread.results = self.results
        self.acquisition_thread.start()

    def add_line(self, string):
        """ Adds a line to the textbox display.
        """
        self.results_string = (string + "\n" + self.results_string)[0:1000]

    def image_show(self, image):
        """ Plots an image on the canvas in a thread safe way.
        """
        self.figure.axes[0].images=[]
        self.figure.axes[0].imshow(image, aspect='auto')
        wx.CallAfter(self.figure.canvas.draw)

class MainWindowHandler(Handler):
    def close(self, info, is_OK):
        if ( info.object.panel.acquisition_thread
            and info.object.panel.acquisition_thread.isAlive() ):
            info.object.panel.acquisition_thread.wants_abort = True
            while info.object.panel.acquisition_thread.isAlive():
                sleep(0.1)
            wx.Yield()
        return True

class MainWindow(HasTraits):
    """ The main window, here go the instructions to create and destroy the
    ↪ application. """
    figure = Instance(Figure)

    panel = Instance(ControlPanel)

    def _figure_default(self):
        figure = Figure()
        figure.add_axes([0.05, 0.04, 0.9, 0.92])
        return figure

    def _panel_default(self):
        return ControlPanel(figure=self.figure)

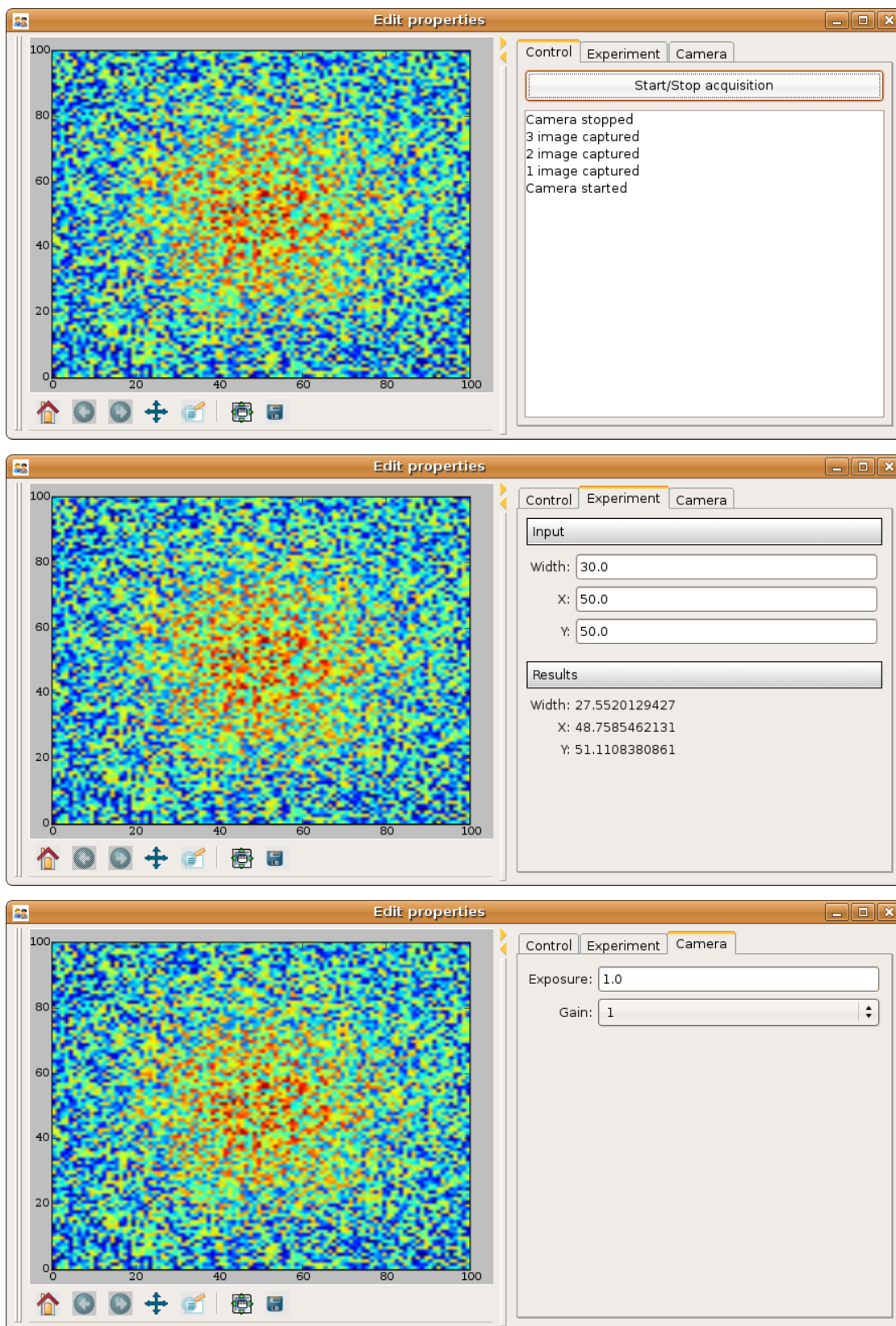
    view = View(HSplit(Item('figure', editor=MPLFigureEditor(),
                           dock='vertical'),
                     Item('panel', style="custom",
                           show_labels=False,
                           ),
                     resizable=True,
                     height=0.75, width=0.75,
                     handler=MainWindowHandler(),
                     buttons=NoButtons)

if __name__ == '__main__':
    MainWindow().configure_traits()

```

When the acquisition loop is created and running, the mock camera object produces noisy gaussian images, and the processing code estimates the parameters of the gaussian.

Here are screenshots of the three different tabs of the application:



Conclusion

I have summarized here all what most scientists need to learn in order to be able to start building applications with traitsUI. Using the traitsUI module to its full power requires you to move away from the procedural type of programming most scientists are used to, and think more in terms of objects and flow of information and control between them. I have found that this paradigm shift, although a bit hard, has been incredibly rewarding in terms of my own productivity and my ability to write compact and readable code.

Good luck!

Acknowledgments

I would like to thank the people on the enthought-dev mailing-list, especially Prabhu Ramachandran and David Morrill, for all the help they gave me, and Janet Swisher for reviewing this document. Big thanks go to enthought for developing the traits and traitsUI modules, and making them open-source. Finally the python, the numpy, and the matplotlib community deserve many thanks for both writing such great software, and being so helpful on the mailing lists.

References

This section contains links to a number of TraitsUI demos.

Warning: Some of the examples in this section may be out of date. We are in the process of updating them to the latest version of TraitsUI.

4.1 Standard Editors

- [BooleanEditor](#)
- [ButtonEditor](#)
- [CSVListEditor](#)
- [CheckListEditor](#)
- [CheckListEditor \(simple\)](#)
- [CodeEditor](#)
- [ColorEditor](#)
- [CompoundEditor](#)
- [DirectoryEditor](#)
- [EnumEditor](#)
- [FileEditor](#)
- [FontEditor](#)
- [HTMLEditor](#)
- [ImageEnumEditor](#)
- [InstanceEditor](#)

- ListEditor
- RGBColorEditor
- RangeEditor
- SetEditor
- TableEditor
- TextEditor
- TitleEditor
- TreeEditor
- TupleEditor

4.2 Advanced Demos

- Adapted TreeEditor
- Apply/Revert Handler
- Table (read-only, auto-edit table column)
- TabularEditor (auto-update)
- DateEditor
- Dynamic EnumEditor
- Dynamic Range Editor
- Dynamic Views
- HDF5 Tree
- History
- Invalid state handling
- ListStrEditor
- ListEditor
- ListEditor (notebook tabs)
- MVC
- Multi-selection string list
- Multithread
- Multithread 2
- NumPy array TabularEditor
- NumPy ArrayViewEditor
- Popup Dialog
- Property List
- ScrubberEditor
- Settable cached property

- Statusbar
- StringListEditor
- TableEditor (with checkbox column)
- TableEditor (with context menu)
- TableEditor (with live search and cell)
- TabularEditor
- TimeEditor

5.1 Release 6.0.0

This release introduces preliminary support for Qt5 via PyQt5, thanks to the work of Gregor Thalhammer which got the ball rolling. Qt5 support is not yet robustly tested in deployed applications, so there may yet be bugs to find. As part of this effort all occurrences of old-style signals and slots have been removed; and this has greatly improved stability under Qt.

This release also features a great deal of work at the API level to disentangle the two-way dependencies between Pyface and TraitsUI. This has involved moving a number of sub-packages between the two libraries, most notably the zipped image resource support and a number of trait definitions. We have endeavored to keep backwards compatibility via stub modules in the original locations, but we can't guarantee that there will be no issues with third party code caused by the change in locations. We haven't been able to remove all dependencies, but as of this release on the dock and workbench submodules have required dependencies on TraitsUI.

As part of the latter work, support for TraitsUI Themes have been removed. This was a feature that was only available under WxPython, was slow, was never used in production code, and was not supported for over a decade. Some of the codebase remains as it is still used by the PyFace Dock infrastructure and several editors, but their long-term intention is to remove this completely.

Another long-desired feature was the ability to write toolkit backends for Pyface and TraitsUI that are not part of the main codebase. This is now possible by contributing new toolkit backends to the "traitsui.toolkit" pkg_resources entry point in a setup.py. This work was accompanied by an overhaul of the toolkit discovery and loading infrastructure; in particular Pyface and TraitsUI now share the same code for performing these searches and loading the backends.

The entire TraitsUI codebase has been run through the AutoPEP8, assisted with some customized fixups and occasional drive-by cleanups of code, which means that the codebase is generally easier to read and follows modern Python conventions.

Finally, the testing infrastructure has been overhauled to provide a one-stop location to run tests in self-contained environments using Enthought's EDM package management tool. Tests can be run from any python environment with the "edm" command available and the "click" library installed with the "etstool.py" script at the top level of the repository. In particular:

```
python etstool.py test_all
```

will run all relevant tests for all available toolkits in all supported python versions. The TravisCI and Appveyor continuous integration tools have been updated to make use of these facilities as well.

Thanks to Matrin Bergtholdt, Alex Chabot, Kit Choi, Mark Dickinson, Robin Dunn, Pradyun Gedam, Robert Kern, Marika Murphy, Pankaj Pandey, Steve Peak, Prabhu Ramachandran, Jonathan Rocher, John Thacker, Gregor Thhammer, Senganal Thirunavukkarasu, John Tyree, Ioannis Tziakos, Alona Varshal, Corran Webster, John Wiggins

5.1.1 Enhancements

- Support for Qt5 (#347, #352, #350, #433)
- Remove TraitsUI Themes (#342)
- Improve Toolkit selection and handling (#425, #429)
- API Documentation (#438)
- Adapter documentation (#340)
- Support multi-selection in DataFrameEditor (#413)
- DataFrameEditor demo (#444)
- Common BasePanel class for toolkits (#392)
- Labels honor enable_when values (#401)
- Better error messages when toolkit doesn't implement methods (#391)
- Improve TraitsUI Action handling (#384)
- ListEditor UI improvements (#338, #396, #395)
- Remove old style signals and slots for Qt backend (#330, #346, #347, #403)
- Expose a “refresh” trait for the DataFrameEditor (#288)
- Use Enthought Deployment Manager to automate CI and testing (#321, #357)
- Continuous integration on OS X (#322)
- Reduce circular dependencies of PyFace on TraitsUI (#304)
- PEP8-compliant formatting of source (#290)
- Add progress bar column for TableEditor (#287)
- Add codecov coverage reports (#285, #328)

5.1.2 Fixes

- Fix some issues for newer WxPython (#418)
- Fix Wx simple FileEditor (#426)
- Fixes for DataFrameEditor (#415)
- Fixes for preferences state saving under Qt (#410, #447)
- Fix panel state after setting preferences (#253)
- Fix TableEditor ColorColumn (#399)

- Prevent loopback from slider in Qt RangeEditor (#400)
- Fix Action buttons under Qt (#393, #394)
- Fix ValueEditor icons (#386)
- Fix bug in update_object (#379)
- Avoid reading Event trait values in sync_value (#375)
- Fix raise_to_debug calls (#362, #372)
- Fix errors during garbage collection (#359)
- Remove unused argument in wx.hook_events (#360)
- Fix button label updates (#358)
- Fix TreeEditor label updates (#335)
- Proper InstanceEditor dialog lifecycle (#332)
- Don't explicitly destroy child widgets under Qt (#283)
- Test fixes and improvements (#329, #369, #371, #327)
- Fixes for demos and examples (#320, #445)
- Fix CheckListEditor string comparison (#318)
- Remove some spurious print statements (#305)
- Documentation fixes (#301, #326, #380, #438, #443)
- Fixes for Python 3 compatibility (#295, #300, #165, #311, #410)
- Fix error with Qt table model mimetype (#296)
- Fixes for continuous integration (#299, #345, #365, #397, #420, #427)
- Fix offset issue when dragging from Qt TreeEditor (#293)
- Fix Wx kill-focus event issues (#291)
- Fix readthedocs build (#281)

5.2 Release 5.2.0

5.2.1 Enhancements

- Add support for multi-select in the DataFrameEditor (#413).

5.3 Release 5.1.0

5.3.1 Enhancements

- Enthought Sphinx Theme Support (#219)
- Allow hiding groups in split layouts on Qt (#246)
- Allow subclass of Controller to set a default model (#255)

- Add toolbar in Qt UI panel (#263)

5.3.2 Fixes

- Fix Qt TableEditor segfault on editing close (#277)
- Update tree nodes when adding children to an empty tree (#251)
- Change default backend from Wx to Qt (#254, #256)
- Improve toolkit selection (#259)
- Fix capturing the mouse and click events on Wx (#265, #266)
- Remove duplicated traits in NotebookEditor (#268)
- Fix exception during disposal of ListStrEditor (#270)
- Version number in documentation (#273)

5.4 Release 5.0.0

This release features experimental support for Python 3 with the Qt toolkit!

This is based in large part on the work of Yves Delley and Pradyun Gedam, but also owes a lot to Ioannis Tziakos for implementing container-based continuous integration and Prabhu Ramachandran and Corran Webster for tracking down last-minute bugs. Python 3 support is probably not yet ready for production use, but feedback and bug reports are welcome, and all future pull requests will be expected to work with Python 3.4 and later. Python 3 support requires Traits 4.5 or greater, and Pyface 5.0 or greater.

In addition, this release includes fixes to support wxPython 3.0 and deprecates wxPython 2.6. Thanks to Robin Dunn for providing these improvements.

This release also introduces a DataFrameEditor which provides a tabular view of a Pandas DataFrame, similar to the existing ArrayViewEditor.

There are also a number of bug fixes and minor improvements detailed below.

Finally, this release changes the default GUI toolkit from Wx to Qt. This changes the behaviour of the library in the case where both WxPython and PyQt/PySide are installed and the user or code doesn't specify the toolkit to use explicitly.

5.4.1 New Features

- Experimental Python 3 support (#230)
- A DataFrameEditor for Pandas DataFrames, similar to the ArrayViewEditor (#196)

5.4.2 Enhancements

- Change the default backend from Wx to Qt (#212)
- Add a Qt version of the ProgressEditor (#217)
- Links to demos in the documentation (#159)
- Add minimal support for the dock_styles option to the Qt tabbed List Editor. (#143)

5.4.3 Fixes

- Fix failure to disconnect selection listeners for ListStrEditor in Qt (#223)
- Fix layout of TextEditors in some situations (#216)
- Fix removal of _popEventHandlers not owned by TraitsUI in Wx (#215)
- Remove some old (TraitsUI 3.0-era) documentation (#214)
- Help button now works on Qt (#160)

5.5 Release 4.5.1

5.5.1 Fixes

- Fix pypi installation problem (#206)

5.6 Release 4.5.0

5.6.1 Fixes

- Application-modal Traits UI dialogs are correctly styled as application-modal under Qt. On Macs, they will now appear as independent windows rather than drop-down sheets. (#164)
- Qt CodeEditor now honors ‘show_line_numbers’ and the ‘readonly’ style (#137)
- Deprecated *implements* declaration removed, use *provides* instead (#152)
- Fix TableEditor so that Qt.QSplitter honors ‘orientation’ trait (#171)
- Show row labels in Qt TableEditor when requested (#176)
- Fix TupleEditor so that multiple change events are not fired (#179)
- **Numpy dependency is now optional. *ArrayEditor* will not be available** if numpy cannot be imported (#181)
- Add development versioning (#200)

5.7 Release 4.4.0

The biggest change in this release is support for the new adaptation mechanism in Traits 4.4.0. Other than that, there are a number of other minor changes, improvements and bugfixes.

Corran Webster (corranwebster on GitHub) is now maintainer of TraitsUI.

Change summary since 4.3.0

5.7.1 New Features

- Changes for new Traits adaptation mechanism support (#113)

5.7.2 Enhancements

- Add Travis-CI support.
- Remove the use of the deprecated PySimpleApp under Wx and several other improvements. (#107)
- Improvements to Qt TabularEditor, TableEditor and TreeEditor drag and drop support. Should be roughly on par with Wx support. No API changes. (#124, #126, #129, #135)
- Improvements to PyMimeData coercion to better handle lists of items. (#127)

5.7.3 Fixes

- Fixes item selection issue #133 in ListStrEditor under Wx 2.9 (#137)
- Fixes to avoid asking for value of a Delegated Event (#123 and #136)
- Fix drag image location for Qt TreeEditor (#132)
- Qt TreeEditor supports bg and fg colors and column labels correctly. (#131)
- Fix ListEditor under PySide (#125)
- remove event handlers before window destruction in Wx. Required for Wx 2.9. (#108)

There are currently some other unlisted changes going back some time before this file was created.

5.8 Traits 3.5.0 (Oct 15, 2010)

5.8.1 Enhancements

- adding support for drop-down menu in Button traits, but only for qt backend
- adding 'show_notebook_menu' option to ListEditor so that the user can right-click and show or hide the context menu (Qt)
- added selection range traits to make it possible for users to replace selected text

5.8.2 Fixes

- fixed null color editor to work with tuples
- bug when opening a view with the ToolbarButton

5.9 Traits 3.4.0 (May 26, 2010)

5.9.1 Enhancements

- adding new example to make testing rgb color editor easier

5.9.2 Fixes

- fixed NumericColumn to not expect object to have model_selection attribute, and removed more dead theming code
- fixed API bugs with the NumericColumn where its function signatures differed from its base class, but the calling code expected them to all be the same
- fixed bug which was related to type name errors caused when running Sphinx
- when using File(exists=True), be sure to validate the type of the value first before using os.path.isfile()

5.10 Traits 3.3.0 (Feb 24, 2010)

5.10.1 Enhancements

The major enhancement this release is that the entire Traits package has been changed to use relative imports so that it can be installed as a sub-package inside another larger library or package. This was not previously possible, since the various modules inside Traits would import each other directly through “traits.[module]”. Many thanks to Darren Dale for the patch.

5.10.2 Fixes

There have been numerous minor bugfixes since the last release. The most notable ones are:

- Many fixes involve making Traits UI more robust if wxPython is not installed on a system. In the past, we have been able to use Qt if it was also installed, but removing Wx would lead to a variety of little bugs in various places. We’ve squashed a number of these. We’ve also added better checks to make sure we’re selecting the right toolkit at import and at runtime.
- A nasty cyclic reference was discovered and eliminated in DelegatesTo traits.
- The Undefined and Uninitialized Traits were made into true singletons.
- Much of the inconsistent formatting across the entire Traits source has been eliminated and normalized (tabs/spaces, line endings).

5.11 Traits 3.2.0 (July 15, 2009)

5.11.1 Enhancements

- Implemented editable_labels attribute in the TabularEditor for enabling editing of the labels (i.e. the first column)
- Saving/restoring window positions works with multiple displays of different sizes
- New ProgressEditor
- Changed default colors for TableEditor
- Added support for HTML editor for QT backend using QtWebKit
- Improved support for opening links in external browser from HTML editor
- Added support for TabularEditor for QT backend

- Added support for marking up the CodeEditor, including adding squiggles and dimming lines
- Added SearchEditor
- Improved unicode support
- Changed behavior of RangeEditor text box to not auto-set
- Added support in RangeEditor for specifying the method to evaluate new values.
- Add DefaultOverride editor factory courtesy Stéfan van der Walt
- Removed `sys.exit()` call from `SaveHandler.exit()`

5.11.2 Fixes

TraitsUI: Traits-capable windowing framework

The TraitsUI project contains a toolkit-independent GUI abstraction layer, which is used to support the “visualization” features of the [Traits](#) package. Thus, you can write model in terms of the Traits API and specify a GUI in terms of the primitives supplied by TraitsUI (views, items, editors, etc.), and let TraitsUI and your selected toolkit and back-end take care of the details of displaying them.

6.1 Example

Given a Traits model like the following:

```
from traits.api import HasTraits, Str, Range, Enum

class Person(HasTraits):
    name = Str('Jane Doe')
    age = Range(low=0)
    gender = Enum('female', 'male')

person = Person(age=30)
```

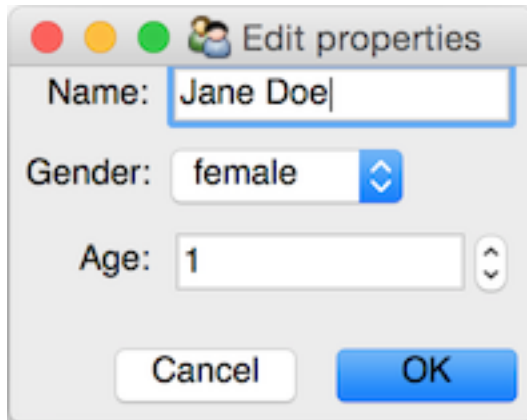
we can use TraitsUI to specify a and display a GUI view:

```
from traitsui.api import Item, RangeEditor, View

person_view = View(
    Item('name'),
    Item('gender'),
    Item('age', editor=RangeEditor(mode='spinner')),
    buttons=['OK', 'Cancel'],
    resizable=True,
)

person.configure_traits(view=person_view)
```

which creates a GUI which looks like this:



6.2 Important Links

- Website and Documentation: <http://docs.enthought.com/traitsui>
 - User Manual http://docs.enthought.com/traitsui/traitsui_user_manual
 - Tutorial <http://docs.enthought.com/traitsui/tutorial>
 - API Documentation <http://docs.enthought.com/traitsui/api>
- Source code repository: <https://github.com/enthought/traitsui>
 - Issue tracker: <https://github.com/enthought/traitsui/issues>
- Download releases: <https://pypi.python.org/pypi/traitsui>
- Mailing list: <https://groups.google.com/forum/#!forum/ets-users>

6.3 Installation

If you want to run traitsui, you must also install:

- Traits <https://github.com/enthought/traits>
- Pyface <https://github.com/enthought/pyface>

You will also need one of the following backends:

- PyQt
- wxPython
- PySide
- PyQt5

Backends have additional dependencies and there are optional dependencies on NumPy and Pandas for some editors.

TraitsUI along with all dependencies can be installed in a straightforward way using the [Enthought Deployment Manager](#), `pip` or other .

6.4 Running the Test Suite

To run the test suite, you will need to install Git and [EDM](#) as well as have a Python environment which has install [Click](#) available. You can then follow the instructions in `etstool.py`. In particular:

```
> python test_all
```

will run tests in all supported environments automatically.

CHAPTER 7

Indices and tables

- `genindex`
- `search`

t

traitsui, 180
traitsui.api, 123
traitsui.base_panel, 123
traitsui.basic_editor_factory, 124
traitsui.color_column, 125
traitsui.context_value, 125
traitsui.delegating_handler, 125
traitsui.dock_window_theme, 126
traitsui.editor, 126
traitsui.editor_factory, 127
traitsui.editors, 115
traitsui.editors.api, 101
traitsui.editors.boolean_editor, 101
traitsui.editors.button_editor, 102
traitsui.editors.check_list_editor, 102
traitsui.editors.code_editor, 102
traitsui.editors.color_editor, 102
traitsui.editors.compound_editor, 103
traitsui.editors.csv_list_editor, 103
traitsui.editors.custom_editor, 104
traitsui.editors.date_editor, 104
traitsui.editors.default_override, 104
traitsui.editors.directory_editor, 105
traitsui.editors.dnd_editor, 105
traitsui.editors.drop_editor, 105
traitsui.editors.enum_editor, 105
traitsui.editors.file_editor, 105
traitsui.editors.font_editor, 106
traitsui.editors.history_editor, 106
traitsui.editors.html_editor, 106
traitsui.editors.image_editor, 107
traitsui.editors.image_enum_editor, 107
traitsui.editors.instance_editor, 107
traitsui.editors.key_binding_editor, 107
traitsui.editors.list_editor, 107
traitsui.editors.list_str_editor, 108
traitsui.editors.null_editor, 108
traitsui.editors.popup_editor, 108
traitsui.editors.progress_editor, 108
traitsui.editors.range_editor, 108
traitsui.editors.rgb_color_editor, 109
traitsui.editors.scrubber_editor, 109
traitsui.editors.search_editor, 109
traitsui.editors.set_editor, 109
traitsui.editors.shell_editor, 109
traitsui.editors.styled_date_editor, 110
traitsui.editors.table_editor, 110
traitsui.editors.tabular_editor, 111
traitsui.editors.text_editor, 111
traitsui.editors.time_editor, 111
traitsui.editors.title_editor, 111
traitsui.editors.tree_editor, 112
traitsui.editors.tuple_editor, 114
traitsui.editors.value_editor, 114
traitsui.editors_gen, 128
traitsui.extras, 116
traitsui.extras.edit_column, 115
traitsui.extras.saving, 115
traitsui.group, 128
traitsui.handler, 129
traitsui.help, 133
traitsui.help_template, 133
traitsui.helper, 134
traitsui.image, 116
traitsui.include, 134
traitsui.instance_choice, 135
traitsui.item, 135
traitsui.key_bindings, 137
traitsui.list_str_adapter, 95
traitsui.menu, 140
traitsui.message, 141
traitsui.mimedata, 142
traitsui.null, 117
traitsui.null.color_trait, 116
traitsui.null.font_trait, 116
traitsui.null.rgb_color_trait, 117
traitsui.null.toolkit, 117
traitsui.table_column, 142

- traitsui.table_filter, [145](#)
- traitsui.tabular_adapter, [89](#)
- traitsui.tests, [121](#)
- traitsui.tests.editors, [119](#)
- traitsui.tests.editors.test_liststr_editor,
 [118](#)
- traitsui.tests.null_backend, [119](#)
- traitsui.tests.test_handler, [119](#)
- traitsui.tests.test_regression, [120](#)
- traitsui.tests.test_shadow_group, [121](#)
- traitsui.tests.test_toolkit, [121](#)
- traitsui.tests.ui_editors, [119](#)
- traitsui.theme, [153](#)
- traitsui.toolkit, [153](#)
- traitsui.toolkit_traits, [157](#)
- traitsui.tree_node, [85](#)
- traitsui.ui, [171](#)
- traitsui.ui_editor, [172](#)
- traitsui.ui_editors, [123](#)
- traitsui.ui_editors.array_view_editor,
 [121](#)
- traitsui.ui_editors.data_frame_editor,
 [122](#)
- traitsui.ui_info, [173](#)
- traitsui.ui_traits, [173](#)
- traitsui.undo, [174](#)
- traitsui.value_tree, [175](#)
- traitsui.view, [178](#)
- traitsui.view_element, [179](#)
- traitsui.view_elements, [180](#)

A

- AbstractUndoItem (class in traitsui.undo), 174
- accepts (traitsui.list_str_adapter.AnIListStrAdapter attribute), 137
- accepts (traitsui.list_str_adapter.IListStrAdapter attribute), 138
- accepts (traitsui.tabular_adapter.AnITabularAdapter attribute), 147
- accepts (traitsui.tabular_adapter.ITabularAdapter attribute), 147
- Action (class in traitsui.menu), 140
- action (traitsui.menu.Action attribute), 140
- action_handler() (traitsui.tests.test_handler.SampleHandler method), 120
- action_handler() (traitsui.tests.test_handler.SampleObject method), 120
- activated (traitsui.editors.tree_editor.ToolkitEditorFactory attribute), 112
- activated() (traitsui.tree_node.ITreeNode method), 157
- activated() (traitsui.tree_node.ITreeNodeAdapter method), 159
- activated() (traitsui.tree_node.ITreeNodeAdapterBridge method), 161
- activated() (traitsui.tree_node.MultiTreeNode method), 163
- activated() (traitsui.tree_node.ObjectTreeNode method), 165
- activated() (traitsui.tree_node.TreeNode method), 167
- adapter_column_indices (traitsui.tabular_adapter.TabularAdapter attribute), 148
- adapter_column_map (traitsui.tabular_adapter.TabularAdapter attribute), 148
- adapters (traitsui.list_str_adapter.ListStrAdapter attribute), 138
- adapters (traitsui.tabular_adapter.TabularAdapter attribute), 148
- add() (traitsui.undo.UndoHistory method), 174
- add_checked() (traitsui.ui.UI method), 171
- add_defined() (traitsui.ui.UI method), 171
- add_enabled() (traitsui.ui.UI method), 171
- add_to_menu() (traitsui.base_panel.BasePanel method), 123
- add_to_menu() (traitsui.editors.table_editor.BaseTableEditor method), 110
- add_to_toolbar() (traitsui.base_panel.BasePanel method), 123
- add_to_toolbar() (traitsui.editors.table_editor.BaseTableEditor method), 110
- add_visible() (traitsui.ui.UI method), 171
- alignment (traitsui.tabular_adapter.TabularAdapter attribute), 148
- alignment (traitsui.ui_editors.data_frame_editor.DataFrameAdapter attribute), 122
- allows_children() (traitsui.tree_node.ITreeNode method), 157
- allows_children() (traitsui.tree_node.ITreeNodeAdapter method), 159
- allows_children() (traitsui.tree_node.ITreeNodeAdapterBridge method), 161
- allows_children() (traitsui.tree_node.MultiTreeNode method), 163
- allows_children() (traitsui.tree_node.ObjectTreeNode method), 165
- allows_children() (traitsui.tree_node.TreeNode method), 167
- alternating_row_colors (traitsui.editors.tree_editor.ToolkitEditorFactory attribute), 112
- AnIListStrAdapter (class in traitsui.list_str_adapter), 137
- AnITabularAdapter (class in traitsui.tabular_adapter), 147
- append_child() (traitsui.tree_node.ITreeNode method), 157
- append_child() (traitsui.tree_node.ITreeNodeAdapter method), 159
- append_child() (traitsui.tree_node.ITreeNodeAdapterBridge method), 161

append_child() (traitsui.tree_node.ObjectTreeNode method), 165

append_child() (traitsui.tree_node.TreeNode method), 167

apply() (traitsui.handler.Handler method), 130

apply() (traitsui.tests.test_handler.SampleHandler method), 120

ApplyButton, 13

ApplyButton (in module traitsui.menu), 140

array_editor() (traitsui.toolkit.Toolkit method), 153

ArrayNode (class in traitsui.value_tree), 175

ArrayViewAdapter (class in traitsui.ui_editors.array_view_editor), 121

ArrayViewEditor (class in traitsui.ui_editors.array_view_editor), 122

assert_toolkit_import() (in module traitsui.toolkit), 156

ATheme (class in traitsui.ui_traits), 173

attribute, 96

attributes

- Group, 9
- Item, 6
- View, 14

auto_close_message() (in module traitsui.message), 141

auto_open (traitsui.editors.tree_editor.ToolkitEditorFactory attribute), 112

AutoCloseMessage (class in traitsui.message), 141

B

BasePanel (class in traitsui.base_panel), 123

BaseTableEditor (class in traitsui.editors.table_editor), 110

basic_types() (in module traitsui.value_tree), 178

BasicEditorFactory (class in traitsui.basic_editor_factory), 124

bg_color (traitsui.list_str_adapter.ListStrAdapter attribute), 138

bg_color (traitsui.tabular_adapter.TabularAdapter attribute), 148

bgcolor (traitsui.editors.styled_date_editor.CellFormat attribute), 110

bind() (traitsui.ui_info.UIInfo method), 173

bind_context() (traitsui.ui_info.UIInfo method), 173

bold (traitsui.editors.styled_date_editor.CellFormat attribute), 110

boolean_editor() (traitsui.toolkit.Toolkit method), 153

BooleanEditor (in module traitsui.editors.boolean_editor), 101

BoolNode (class in traitsui.value_tree), 175

button_editor() (traitsui.toolkit.Toolkit method), 153

ButtonEditor (in module traitsui.editors.button_editor), 102

buttons

- attribute, 12
- examples, 13

standard, 13

C

cache (traitsui.list_str_adapter.ListStrAdapter attribute), 138

cache (traitsui.tabular_adapter.TabularAdapter attribute), 148

cache_flushed (traitsui.list_str_adapter.ListStrAdapter attribute), 138

cache_flushed (traitsui.tabular_adapter.TabularAdapter attribute), 148

can_add() (traitsui.tree_node.ITreeNode method), 157

can_add() (traitsui.tree_node.ITreeNodeAdapter method), 159

can_add() (traitsui.tree_node.ITreeNodeAdapterBridge method), 161

can_add() (traitsui.tree_node.MultiTreeNode method), 163

can_add() (traitsui.tree_node.ObjectTreeNode method), 165

can_add() (traitsui.tree_node.TreeNode method), 167

can_add_to_menu() (traitsui.base_panel.BasePanel method), 123

can_add_to_menu() (traitsui.editors.table_editor.BaseTableEditor method), 110

can_add_to_toolbar() (traitsui.base_panel.BasePanel method), 124

can_add_to_toolbar() (traitsui.editors.table_editor.BaseTableEditor method), 110

can_auto_close() (traitsui.tree_node.ITreeNode method), 157

can_auto_close() (traitsui.tree_node.ITreeNodeAdapter method), 159

can_auto_close() (traitsui.tree_node.ITreeNodeAdapterBridge method), 161

can_auto_close() (traitsui.tree_node.MultiTreeNode method), 163

can_auto_close() (traitsui.tree_node.ObjectTreeNode method), 165

can_auto_close() (traitsui.tree_node.TreeNode method), 167

can_auto_open() (traitsui.tree_node.ITreeNode method), 157

can_auto_open() (traitsui.tree_node.ITreeNodeAdapter method), 159

can_auto_open() (traitsui.tree_node.ITreeNodeAdapterBridge method), 161

can_auto_open() (traitsui.tree_node.MultiTreeNode method), 163

can_auto_open() (traitsui.tree_node.ObjectTreeNode method), 165

- `can_auto_open()` (traitsui.tree_node.TreeNode method), 167
- `can_copy()` (traitsui.tree_node.ITreeNode method), 157
- `can_copy()` (traitsui.tree_node.ITreeNodeAdapter method), 159
- `can_copy()` (traitsui.tree_node.ITreeNodeAdapterBridge method), 161
- `can_copy()` (traitsui.tree_node.MultiTreeNode method), 163
- `can_copy()` (traitsui.tree_node.ObjectTreeNode method), 165
- `can_copy()` (traitsui.tree_node.TreeNode method), 167
- `can_delete()` (traitsui.tree_node.ITreeNode method), 157
- `can_delete()` (traitsui.tree_node.ITreeNodeAdapter method), 159
- `can_delete()` (traitsui.tree_node.ITreeNodeAdapterBridge method), 161
- `can_delete()` (traitsui.tree_node.MultiTreeNode method), 163
- `can_delete()` (traitsui.tree_node.ObjectTreeNode method), 165
- `can_delete()` (traitsui.tree_node.TreeNode method), 167
- `can_delete_me()` (traitsui.tree_node.ITreeNode method), 157
- `can_delete_me()` (traitsui.tree_node.ITreeNodeAdapter method), 159
- `can_delete_me()` (traitsui.tree_node.ITreeNodeAdapterBridge method), 162
- `can_delete_me()` (traitsui.tree_node.MultiTreeNode method), 164
- `can_delete_me()` (traitsui.tree_node.ObjectTreeNode method), 165
- `can_delete_me()` (traitsui.tree_node.TreeNode method), 167
- `can_drop` (traitsui.tabular_adapter.TabularAdapter attribute), 148
- `can_drop()` (traitsui.handler.Handler method), 130
- `can_edit` (traitsui.list_str_adapter.ListStrAdapter attribute), 138
- `can_edit` (traitsui.tabular_adapter.TabularAdapter attribute), 148
- `can_import()` (traitsui.handler.Handler method), 130
- `can_insert()` (traitsui.tree_node.ITreeNode method), 157
- `can_insert()` (traitsui.tree_node.ITreeNodeAdapter method), 159
- `can_insert()` (traitsui.tree_node.ITreeNodeAdapterBridge method), 162
- `can_insert()` (traitsui.tree_node.MultiTreeNode method), 164
- `can_insert()` (traitsui.tree_node.ObjectTreeNode method), 165
- `can_insert()` (traitsui.tree_node.TreeNode method), 167
- `can_rename()` (traitsui.tree_node.ITreeNode method), 157
- `can_rename()` (traitsui.tree_node.ITreeNodeAdapter method), 160
- `can_rename()` (traitsui.tree_node.ITreeNodeAdapterBridge method), 162
- `can_rename()` (traitsui.tree_node.MultiTreeNode method), 164
- `can_rename()` (traitsui.tree_node.ObjectTreeNode method), 165
- `can_rename()` (traitsui.tree_node.TreeNode method), 167
- `can_rename_me()` (traitsui.tree_node.ITreeNode method), 157
- `can_rename_me()` (traitsui.tree_node.ITreeNodeAdapter method), 160
- `can_rename_me()` (traitsui.tree_node.ITreeNodeAdapterBridge method), 162
- `can_rename_me()` (traitsui.tree_node.MultiTreeNode method), 164
- `can_rename_me()` (traitsui.tree_node.ObjectTreeNode method), 165
- `can_rename_me()` (traitsui.tree_node.TreeNode method), 167
- CancelButton, 13
- CancelButton (in module traitsui.menu), 140
- CanSaveMixin (class in traitsui.extras.saving), 115
- CellFormat (class in traitsui.editors.styled_date_editor), 110
- `check_button()` (traitsui.base_panel.BasePanel method), 124
- `check_list_editor()` (traitsui.toolkit.Toolkit method), 153
- checked_when (traitsui.menu.Action attribute), 140
- CheckListEditor (in module traitsui.editors.check_list_editor), 102
- Child (class in traitsui.tests.test_regression), 120
- class attribute, 96
- ClassNode (class in traitsui.value_tree), 175
- `cleanup()` (traitsui.tabular_adapter.TabularAdapter method), 148
- `clear()` (traitsui.undo.UndoHistory method), 174
- `clear_toolkit()` (in module traitsui.tests.test_toolkit), 121
- `click` (traitsui.editors.tree_editor.ToolkitEditorFactory attribute), 112
- `click()` (traitsui.tree_node.ITreeNode method), 157
- `click()` (traitsui.tree_node.ITreeNodeAdapter method), 160
- `click()` (traitsui.tree_node.ITreeNodeAdapterBridge method), 162
- `click()` (traitsui.tree_node.MultiTreeNode method), 164
- `click()` (traitsui.tree_node.ObjectTreeNode method), 165
- `click()` (traitsui.tree_node.TreeNode method), 167
- `clone()` (traitsui.key_bindings.KeyBindings method), 137
- `clone_traits()` (traitsui.table_filter.GenericTableFilterRule method), 145
- `close()` (traitsui.extras.saving.SaveHandler method), 115

- ul style="list-style-type: none; padding-left: 0;">
- close() (traitsui.handler.Handler method), 130
- close_dock_control() (in module traitsui.handler), 133
- close_result attribute, 15
- CloseAction (in module traitsui.menu), 141
- closed() (traitsui.delegating_handler.DelegatingHandler method), 125
- closed() (traitsui.extras.saving.SaveHandler method), 115
- closed() (traitsui.handler.Handler method), 130
- cmp() (traitsui.table_column.TableColumn method), 144
- code_editor() (traitsui.toolkit.Toolkit method), 153
- CodeEditor (in module traitsui.editors.code_editor), 102
- coerce_button() (traitsui.base_panel.BasePanel method), 124
- color_editor() (traitsui.toolkit.Toolkit method), 153
- color_trait() (traitsui.null.toolkit.GUIToolkit method), 117
- color_trait() (traitsui.toolkit.Toolkit method), 153
- ColorColumn (class in traitsui.color_column), 125
- ColorEditor() (in module traitsui.editors.color_editor), 102
- ColorTrait() (in module traitsui.toolkit_traits), 157
- column (traitsui.tabular_adapter.AnITabularAdapter attribute), 147
- column (traitsui.tabular_adapter.ITabularAdapter attribute), 147
- column (traitsui.tabular_adapter.TabularAdapter attribute), 148
- column_dict (traitsui.tabular_adapter.TabularAdapter attribute), 148
- column_headers (traitsui.editors.tree_editor.ToolkitEditorFactory attribute), 112
- column_id (traitsui.tabular_adapter.TabularAdapter attribute), 148
- column_map (traitsui.tabular_adapter.TabularAdapter attribute), 148
- column_menu (traitsui.tabular_adapter.TabularAdapter attribute), 148
- columns (traitsui.tabular_adapter.AnITabularAdapter attribute), 147
- columns (traitsui.tabular_adapter.ITabularAdapter attribute), 147
- columns (traitsui.tabular_adapter.TabularAdapter attribute), 148
- columns (traitsui.ui_editors.data_frame_editor.DataFrameEditor attribute), 122
- columns attribute, 9
- command button, 96
- commatize() (in module traitsui.helper), 134
- ComplexNode (class in traitsui.value_tree), 175
- compound_editor() (traitsui.toolkit.Toolkit method), 153
- CompoundEditor (in module traitsui.editors.compound_editor), 103
- configure_traits()
 - default view example, 16
 - examples, 4, 8
 - method, 20
 - view parameter, 4, 19
- configure_traits() (traitsui.handler.Handler method), 131
- confirm_delete() (traitsui.tree_node.ITreeNode method), 157
- confirm_delete() (traitsui.tree_node.ITreeNodeAdapter method), 160
- confirm_delete() (traitsui.tree_node.ITreeNodeAdapterBridge method), 162
- confirm_delete() (traitsui.tree_node.ObjectTreeNode method), 165
- confirm_delete() (traitsui.tree_node.TreeNode method), 167
- constants() (traitsui.null.toolkit.GUIToolkit method), 117
- constants() (traitsui.toolkit.Toolkit method), 153
- contains() (traitsui.table_filter.GenericTableFilterRule method), 145
- content (traitsui.tabular_adapter.TabularAdapter attribute), 148
- content attribute
 - Group, 9
 - View, 15
- context, 20
 - examples, 21
 - View, 16
- ContextValue (class in traitsui.context_value), 125
- control, 6
- control (traitsui.base_panel.BasePanel attribute), 124
- controller, 2, 96
- Controller (built-in class), 25
- Controller (class in traitsui.handler), 129
- convert_theme() (in module traitsui.ui_traits), 173
- convert_to_color() (in module traitsui.null.color_trait), 116
- convert_to_color() (in module traitsui.null.rgb_color_trait), 117
- CSVListEditor (class in traitsui.editors.csv_list_editor), 103
- Custom (class in traitsui.item), 135
- Custom class, 7
- custom_editor() (traitsui.editor_factory.EditorFactory method), 127
- custom_editor() (traitsui.editors.csv_list_editor.CSVListEditor method), 103
- custom_editor() (traitsui.editors.default_override.DefaultOverride method), 104
- custom_editor() (traitsui.editors.range_editor.ToolkitEditorFactory method), 108
- custom_editor() (traitsui.toolkit.Toolkit method), 153
- CustomEditor (in module traitsui.editors.custom_editor), 104
- CV (in module traitsui.context_value), 125
- CVType() (in module traitsui.context_value), 125

D

- DataFrameAdapter (class in trait-sui.ui_editors.data_frame_editor), 122
- DataFrameEditor (class in trait-sui.ui_editors.data_frame_editor), 122
- DateEditor (class in traitsui.editors.date_editor), 104
- dclick (traitsui.editors.tree_editor.ToolkitEditorFactory attribute), 112
- dclick() (traitsui.tree_node.ITreeNode method), 158
- dclick() (traitsui.tree_node.ITreeNodeAdapter method), 160
- dclick() (traitsui.tree_node.ITreeNodeAdapterBridge method), 162
- dclick() (traitsui.tree_node.MultiTreeNode method), 164
- dclick() (traitsui.tree_node.ObjectTreeNode method), 165
- dclick() (traitsui.tree_node.TreeNode method), 167
- default view, 16, 17
 - example, 16
- default_bg_color (trait-sui.tabular_adapter.TabularAdapter attribute), 149
- default_handler() (in module traitsui.handler), 133
- default_icon() (traitsui.base_panel.BasePanel method), 124
- default_show_help() (in module traitsui.help), 133
- default_text (traitsui.list_str_adapter.ListStrAdapter attribute), 138
- default_text_color (trait-sui.tabular_adapter.TabularAdapter attribute), 149
- default_traits_view()
 - default view method, 17
- default_value (traitsui.list_str_adapter.ListStrAdapter attribute), 138
- default_value (traitsui.tabular_adapter.TabularAdapter attribute), 149
- default_value (traitsui.ui_traits.ATheme attribute), 173
- default_value (traitsui.ui_traits.ViewStatus attribute), 173
- DefaultOverride (class in trait-sui.editors.default_override), 104
- DefaultViewElement (class in traitsui.view_element), 179
- defined_when (traitsui.menu.Action attribute), 140
- defined_when attribute
 - Group, 10
 - Item, 6
- DelegatingHandler (class in traitsui.delegating_handler), 125
- delete() (traitsui.list_str_adapter.ListStrAdapter method), 138
- delete() (traitsui.tabular_adapter.TabularAdapter method), 149
- delete() (traitsui.ui_editors.data_frame_editor.DataFrameAdapter method), 122
- delete_child() (traitsui.tree_node.ITreeNode method), 158
- delete_child() (traitsui.tree_node.ITreeNodeAdapter method), 160
- delete_child() (traitsui.tree_node.ITreeNodeAdapterBridge method), 162
- delete_child() (traitsui.tree_node.ObjectTreeNode method), 165
- delete_child() (traitsui.tree_node.TreeNode method), 167
- description() (traitsui.table_filter.EvalTableFilter method), 145
- description() (traitsui.table_filter.GenericTableFilterRule method), 145
- description() (traitsui.table_filter.MenuTableFilter method), 146
- description() (traitsui.table_filter.RuleTableFilter method), 146
- description() (traitsui.table_filter.TableFilter method), 146
- destroy_children() (traitsui.toolkit.Toolkit method), 153
- destroy_control() (traitsui.toolkit.Toolkit method), 153
- dialog box, 96
- DictNode (class in traitsui.value_tree), 175
- directory_editor() (traitsui.toolkit.Toolkit method), 153
- DirectoryEditor (in module trait-sui.editors.directory_editor), 105
- dispatch() (traitsui.ui.Dispatcher method), 171
- Dispatcher (class in traitsui.ui), 171
- disposable_traits (traitsui.ui.UI attribute), 171
- dispose() (traitsui.editor.Editor method), 126
- dispose() (traitsui.key_bindings.KeyBindings method), 137
- dispose() (traitsui.ui.UI method), 171
- dispose() (traitsui.ui_editor.UIEditor method), 172
- dnd_editor() (traitsui.toolkit.Toolkit method), 153
- DNDEditor (in module traitsui.editors.dnd_editor), 105
- do() (traitsui.key_bindings.KeyBindings method), 137
- do_undoable() (traitsui.ui.UI method), 171
- dock attribute
 - Group, 9
 - Item, 6
 - View, 14
- dock_control_for() (traitsui.handler.Handler method), 131
- dock_theme (traitsui.editors.tree_editor.ToolkitEditorFactory attribute), 112
- dock_theme attribute, 9
- dock_window_empty() (traitsui.handler.Handler method), 131
- dock_window_theme() (in module trait-sui.dock_window_theme), 126
- DockWindowTheme (class in trait-sui.dock_window_theme), 126
- drag (traitsui.tabular_adapter.TabularAdapter attribute),

- 149
- drop_class attribute, 15
- drop_editor() (traitsui.toolkit.Toolkit method), 154
- drop_object() (traitsui.tree_node.ITreeNode method), 158
- drop_object() (traitsui.tree_node.ITreeNodeAdapter method), 160
- drop_object() (traitsui.tree_node.ITreeNodeAdapterBridge method), 162
- drop_object() (traitsui.tree_node.MultiTreeNode method), 164
- drop_object() (traitsui.tree_node.ObjectTreeNode method), 166
- drop_object() (traitsui.tree_node.TreeNode method), 167
- DropEditor (in module traitsui.editors.drop_editor), 105
- dropped (traitsui.list_str_adapter.ListStrAdapter attribute), 138
- dropped (traitsui.tabular_adapter.TabularAdapter attribute), 149
- E**
- edit() (traitsui.key_bindings.KeyBindings method), 137
- edit() (traitsui.table_filter.TableFilter method), 146
- edit_traits(), 20
- edit_traits() (traitsui.handler.Handler method), 131
- edit_view() (traitsui.table_filter.RuleTableFilter method), 146
- edit_view() (traitsui.table_filter.TableFilter method), 147
- editable (traitsui.editors.tree_editor.ToolkitEditorFactory attribute), 112
- editable (traitsui.ui_editors.data_frame_editor.DataFrameEditor attribute), 122
- EditColumn (class in traitsui.extras.edit_column), 115
- editor, 96
- Editor (class in traitsui.editor), 126
- editor (traitsui.editors.tree_editor.ToolkitEditorFactory attribute), 112
- editor attribute, 6
- editor factory, 97
- EditorFactory (class in traitsui.editor_factory), 127
- EditorWithListFactory (class in traitsui.editor_factory), 128
- emphasized attribute, 6
- enabled_when (traitsui.menu.Action attribute), 140
- enabled_when attribute
 - Group, 10
 - Item, 6
- ends_with() (traitsui.table_filter.GenericTableFilterRule method), 145
- enum_editor() (traitsui.toolkit.Toolkit method), 154
- enum_values_changed() (in module traitsui.helper), 134
- EnumEditor (in module traitsui.editors.enum_editor), 105
- environment variable
 - ETS_TOOLKIT, 3
- eq() (traitsui.table_filter.GenericTableFilterRule method), 145
- error() (in module traitsui.message), 142
- error() (traitsui.editor.Editor method), 126
- ETS_TOOLKIT, 3
- ETSToolkit, 3
- eval_when() (traitsui.editors.table_editor.BaseTableEditor method), 110
- eval_when() (traitsui.ui.UI method), 171
- EvalTableFilter (class in traitsui.table_filter), 145
- evaluate() (traitsui.ui.UI method), 171
- even_bg_color (traitsui.list_str_adapter.ListStrAdapter attribute), 138
- even_bg_color (traitsui.tabular_adapter.TabularAdapter attribute), 149
- even_text_color (traitsui.list_str_adapter.ListStrAdapter attribute), 138
- even_text_color (traitsui.tabular_adapter.TabularAdapter attribute), 149
- examples
 - buttons, 13
 - configure_traits(), 4, 8
 - context, 21
 - default view, 16
 - Include, 22
 - multi-object Views, 21
 - multiple Views, 18
 - View Group, 8
 - View object, 4
- exit() (traitsui.extras.saving.SaveHandler method), 116
- expands_on_dclick (traitsui.editors.tree_editor.ToolkitEditorFactory attribute), 112
- export attribute
 - Item, 6
 - View, 15
- ExpressionColumn (class in traitsui.table_column), 142
- extend() (traitsui.undo.UndoHistory method), 174
- extended trait names
 - Item name attribute, 22
- extended_traitname_changed(), 26
- F**
- factory, 97
- fgcolor (traitsui.editors.styled_date_editor.CellFormat attribute), 110
- file_editor() (traitsui.toolkit.Toolkit method), 154
- FileEditor (in module traitsui.editors.file_editor), 105
- filter() (traitsui.table_filter.EvalTableFilter method), 145
- filter() (traitsui.table_filter.MenuTableFilter method), 146
- filter() (traitsui.table_filter.RuleTableFilter method), 146
- filter() (traitsui.table_filter.TableFilter method), 147
- filter_by() (traitsui.view_elements.ViewElements method), 180

- ul style="list-style-type: none; padding-left: 0;">
- find() (traitsui.ui.UI method), 171
- find() (traitsui.view_elements.ViewElements method), 180
- finish() (traitsui.ui.UI method), 171
- FloatNode (class in traitsui.value_tree), 176
- font (traitsui.tabular_adapter.TabularAdapter attribute), 149
- font (traitsui.ui_editors.data_frame_editor.DataFrameEditor attribute), 122
- font_editor() (traitsui.toolkit.Toolkit method), 154
- font_trait() (traitsui.null.toolkit.GUIToolkit method), 117
- font_trait() (traitsui.toolkit.Toolkit method), 154
- FontEditor() (in module traitsui.editors.font_editor), 106
- fonts (traitsui.ui_editors.data_frame_editor.DataFrameEditor attribute), 122
- FontTrait() (in module traitsui.toolkit_traits), 157
- format (traitsui.tabular_adapter.TabularAdapter attribute), 149
- format (traitsui.ui_editors.data_frame_editor.DataFrameEditor attribute), 122
- format_func attribute, 6
- format_str attribute, 6
- format_value() (traitsui.value_tree.ArrayNode method), 175
- format_value() (traitsui.value_tree.ClassNode method), 175
- format_value() (traitsui.value_tree.DictNode method), 175
- format_value() (traitsui.value_tree.FunctionNode method), 176
- format_value() (traitsui.value_tree.ListNode method), 176
- format_value() (traitsui.value_tree.MethodNode method), 176
- format_value() (traitsui.value_tree.ObjectNode method), 177
- format_value() (traitsui.value_tree.RootNode method), 177
- format_value() (traitsui.value_tree.SetNode method), 177
- format_value() (traitsui.value_tree.SingleValueTreeNodeObject method), 177
- format_value() (traitsui.value_tree.StringNode method), 178
- format_value() (traitsui.value_tree.TupleNode method), 178
- formats (traitsui.ui_editors.data_frame_editor.DataFrameEditor attribute), 123
- FunctionNode (class in traitsui.value_tree), 176
- GenericTableFilterRule (class in traitsui.table_filter), 145
- GenericTableFilterRuleAndOrColumn (class in traitsui.table_filter), 145
- GenericTableFilterRuleEnabledColumn (class in traitsui.table_filter), 146
- GenericTableFilterRuleNameColumn (class in traitsui.table_filter), 146
- GenericTableFilterRuleValueColumn (class in traitsui.table_filter), 146
- get_add() (traitsui.tree_node.ITreeNode method), 158
- get_add() (traitsui.tree_node.ITreeNodeAdapter method), 160
- get_add() (traitsui.tree_node.ITreeNodeAdapterBridge method), 162
- get_add() (traitsui.tree_node.MultiTreeNode method), 164
- get_add() (traitsui.tree_node.ObjectTreeNode method), 166
- get_add() (traitsui.tree_node.TreeNode method), 167
- get_alignment() (traitsui.tabular_adapter.TabularAdapter method), 149
- get_background() (traitsui.tree_node.ITreeNodeAdapter method), 160
- get_background() (traitsui.tree_node.ITreeNodeAdapterBridge method), 162
- get_background() (traitsui.tree_node.TreeNode method), 168
- get_bg_color() (traitsui.list_str_adapter.ListStrAdapter method), 138
- get_bg_color() (traitsui.tabular_adapter.TabularAdapter method), 149
- get_can_drop() (traitsui.list_str_adapter.ListStrAdapter method), 139
- get_can_drop() (traitsui.tabular_adapter.TabularAdapter method), 149
- get_can_edit() (traitsui.list_str_adapter.ListStrAdapter method), 139
- get_can_edit() (traitsui.tabular_adapter.TabularAdapter method), 149
- get_cell_color() (traitsui.color_column.ColorColumn method), 125
- get_cell_color() (traitsui.extras.edit_column.EditColumn method), 115
- get_cell_color() (traitsui.table_column.NumericColumn method), 142
- get_cell_color() (traitsui.table_column.TableColumn method), 144
- get_children() (traitsui.tree_node.ITreeNode method), 158
- get_children() (traitsui.tree_node.ITreeNodeAdapter method), 160
- get_children() (traitsui.tree_node.ITreeNodeAdapterBridge method), 162

G

[get_children\(\)](#) (traitsui.tree_node.MultiTreeNode method), [164](#)
[get_children\(\)](#) (traitsui.tree_node.ObjectTreeNode method), [166](#)
[get_children\(\)](#) (traitsui.tree_node.TreeNode method), [168](#)
[get_children_id\(\)](#) (traitsui.tree_node.ITreeNode method), [158](#)
[get_children_id\(\)](#) (traitsui.tree_node.ITreeNodeAdapter method), [160](#)
[get_children_id\(\)](#) (traitsui.tree_node.ITreeNodeAdapterBridge method), [162](#)
[get_children_id\(\)](#) (traitsui.tree_node.MultiTreeNode method), [164](#)
[get_children_id\(\)](#) (traitsui.tree_node.ObjectTreeNode method), [166](#)
[get_children_id\(\)](#) (traitsui.tree_node.TreeNode method), [168](#)
[get_color_editor\(\)](#) (in module traitsui.null.color_trait), [116](#)
[get_column\(\)](#) (traitsui.tabular_adapter.TabularAdapter method), [150](#)
[get_column_labels\(\)](#) (traitsui.tree_node.ITreeNode method), [158](#)
[get_column_labels\(\)](#) (traitsui.tree_node.ITreeNodeAdapter method), [160](#)
[get_column_labels\(\)](#) (traitsui.tree_node.ITreeNodeAdapterBridge method), [162](#)
[get_column_labels\(\)](#) (traitsui.tree_node.TreeNode method), [168](#)
[get_column_menu\(\)](#) (traitsui.tabular_adapter.TabularAdapter method), [150](#)
[get_content\(\)](#) (traitsui.group.ShadowGroup method), [129](#)
[get_content\(\)](#) (traitsui.tabular_adapter.TabularAdapter method), [150](#)
[get_data_column\(\)](#) (traitsui.table_column.NumericColumn method), [142](#)
[get_default_bg_color\(\)](#) (traitsui.list_str_adapter.ListStrAdapter method), [139](#)
[get_default_image\(\)](#) (traitsui.list_str_adapter.ListStrAdapter method), [139](#)
[get_default_text\(\)](#) (traitsui.list_str_adapter.ListStrAdapter method), [139](#)
[get_default_text_color\(\)](#) (traitsui.list_str_adapter.ListStrAdapter method), [139](#)
[get_default_value\(\)](#) (traitsui.list_str_adapter.ListStrAdapter method), [139](#)
[get_default_value\(\)](#) (traitsui.tabular_adapter.TabularAdapter method), [150](#)
[get_drag\(\)](#) (traitsui.list_str_adapter.ListStrAdapter method), [139](#)
[get_drag\(\)](#) (traitsui.tabular_adapter.TabularAdapter method), [150](#)
[get_drag_object\(\)](#) (traitsui.tree_node.ITreeNode method), [158](#)
[get_drag_object\(\)](#) (traitsui.tree_node.ITreeNodeAdapter method), [160](#)
[get_drag_object\(\)](#) (traitsui.tree_node.ITreeNodeAdapterBridge method), [162](#)
[get_drag_object\(\)](#) (traitsui.tree_node.MultiTreeNode method), [164](#)
[get_drag_object\(\)](#) (traitsui.tree_node.ObjectTreeNode method), [166](#)
[get_drag_object\(\)](#) (traitsui.tree_node.TreeNode method), [168](#)
[get_drag_value\(\)](#) (traitsui.table_column.ObjectColumn method), [143](#)
[get_dropped\(\)](#) (traitsui.list_str_adapter.ListStrAdapter method), [139](#)
[get_dropped\(\)](#) (traitsui.tabular_adapter.TabularAdapter method), [150](#)
[get_edit_height\(\)](#) (traitsui.table_column.TableColumn method), [144](#)
[get_edit_width\(\)](#) (traitsui.table_column.TableColumn method), [144](#)
[get_editor\(\)](#) (traitsui.table_column.ListColumn method), [142](#)
[get_editor\(\)](#) (traitsui.table_column.NumericColumn method), [142](#)
[get_editor\(\)](#) (traitsui.table_column.ObjectColumn method), [143](#)
[get_editor\(\)](#) (traitsui.table_filter.GenericTableFilterRuleNameColumn method), [146](#)
[get_editor\(\)](#) (traitsui.table_filter.GenericTableFilterRuleValueColumn method), [146](#)
[get_editors\(\)](#) (traitsui.ui.UI method), [171](#)
[get_error_control\(\)](#) (traitsui.editors.tuple_editor.SimpleEditor method), [114](#)
[get_error_control\(\)](#) (traitsui.ui_editor.UEditor method), [172](#)
[get_error_controls\(\)](#) (traitsui.ui.UI method), [171](#)
[get_extended_value\(\)](#) (traitsui.ui.UI method), [171](#)
[get_font\(\)](#) (traitsui.tabular_adapter.TabularAdapter method), [150](#)
[get_font_editor\(\)](#) (in module traitsui.null.font_trait), [117](#)
[get_foreground\(\)](#) (traitsui.tree_node.ITreeNodeAdapter

method), 160
 get_foreground() (traitsui.tree_node.ITreeNodeAdapterBridge method), 162
 get_foreground() (traitsui.tree_node.TreeNode method), 168
 get_format() (traitsui.tabular_adapter.TabularAdapter method), 150
 get_graph_color() (traitsui.table_column.TableColumn method), 144
 get_help() (traitsui.item.Item method), 136
 get_horizontal_alignment() (traitsui.table_column.NumericColumn method), 142
 get_horizontal_alignment() (traitsui.table_column.TableColumn method), 144
 get_icon() (traitsui.tree_node.ITreeNode method), 158
 get_icon() (traitsui.tree_node.ITreeNodeAdapter method), 160
 get_icon() (traitsui.tree_node.ITreeNodeAdapterBridge method), 162
 get_icon() (traitsui.tree_node.MultiTreeNode method), 164
 get_icon() (traitsui.tree_node.ObjectTreeNode method), 166
 get_icon() (traitsui.tree_node.TreeNode method), 168
 get_icon_path() (traitsui.tree_node.ITreeNode method), 158
 get_icon_path() (traitsui.tree_node.ITreeNodeAdapter method), 160
 get_icon_path() (traitsui.tree_node.ITreeNodeAdapterBridge method), 162
 get_icon_path() (traitsui.tree_node.MultiTreeNode method), 164
 get_icon_path() (traitsui.tree_node.ObjectTreeNode method), 166
 get_icon_path() (traitsui.tree_node.TreeNode method), 168
 get_id() (traitsui.group.ShadowGroup method), 129
 get_id() (traitsui.item.Item method), 136
 get_image() (traitsui.list_str_adapter.ListStrAdapter method), 139
 get_image() (traitsui.table_column.TableColumn method), 144
 get_image() (traitsui.tabular_adapter.TabularAdapter method), 150
 get_item() (traitsui.list_str_adapter.ListStrAdapter method), 139
 get_item() (traitsui.tabular_adapter.TabularAdapter method), 151
 get_item() (traitsui.ui_editors.array_view_editor.ArrayViewAdapter method), 122
 get_item() (traitsui.ui_editors.data_frame_editor.DataFrameAdapter method), 122
 get_label() (traitsui.group.Group method), 128
 get_label() (traitsui.item.Item method), 136
 get_label() (traitsui.table_column.TableColumn method), 144
 get_label() (traitsui.tabular_adapter.TabularAdapter method), 151
 get_label() (traitsui.tree_node.ITreeNode method), 158
 get_label() (traitsui.tree_node.ITreeNodeAdapter method), 160
 get_label() (traitsui.tree_node.ITreeNodeAdapterBridge method), 162
 get_label() (traitsui.tree_node.MultiTreeNode method), 164
 get_label() (traitsui.tree_node.ObjectTreeNode method), 166
 get_label() (traitsui.tree_node.TreeNode method), 168
 get_maximum() (traitsui.table_column.TableColumn method), 144
 get_menu() (traitsui.table_column.NumericColumn method), 143
 get_menu() (traitsui.table_column.TableColumn method), 144
 get_menu() (traitsui.tabular_adapter.TabularAdapter method), 151
 get_menu() (traitsui.tree_node.ITreeNode method), 158
 get_menu() (traitsui.tree_node.ITreeNodeAdapter method), 160
 get_menu() (traitsui.tree_node.ITreeNodeAdapterBridge method), 163
 get_menu() (traitsui.tree_node.MultiTreeNode method), 164
 get_menu() (traitsui.tree_node.ObjectTreeNode method), 166
 get_menu() (traitsui.tree_node.TreeNode method), 168
 get_name() (traitsui.instance_choice.InstanceChoice method), 135
 get_name() (traitsui.instance_choice.InstanceChoiceItem method), 135
 get_name() (traitsui.instance_choice.InstanceFactoryChoice method), 135
 get_name() (traitsui.tree_node.ITreeNode method), 158
 get_name() (traitsui.tree_node.ITreeNodeAdapter method), 160
 get_name() (traitsui.tree_node.ITreeNodeAdapterBridge method), 163
 get_name() (traitsui.tree_node.MultiTreeNode method), 164
 get_name() (traitsui.tree_node.ObjectTreeNode method), 166
 get_name() (traitsui.tree_node.TreeNode method), 168
 get_object() (traitsui.instance_choice.InstanceChoice method), 135
 get_object() (traitsui.instance_choice.InstanceChoiceItem method), 135

- method), 135
 - get_object() (traitsui.instance_choice.InstanceFactoryChoice method), 135
 - get_object() (traitsui.table_column.TableColumn method), 144
 - get_perform_handlers() (traitsui.handler.Controller method), 130
 - get_perform_handlers() (traitsui.handler.Handler method), 131
 - get_prefs() (traitsui.ui.UI method), 171
 - get_raw_value() (traitsui.table_column.ExpressionColumn method), 142
 - get_raw_value() (traitsui.table_column.ObjectColumn method), 143
 - get_renderer() (traitsui.table_column.TableColumn method), 144
 - get_rgb_color_editor() (in module traitsui.null.rgb_color_trait), 117
 - get_row_label() (traitsui.tabular_adapter.TabularAdapter method), 151
 - get_shadow() (traitsui.group.Group method), 128
 - get_style() (traitsui.table_column.ObjectColumn method), 143
 - get_text() (traitsui.list_str_adapter.ListStrAdapter method), 139
 - get_text() (traitsui.tabular_adapter.TabularAdapter method), 151
 - get_text_color() (traitsui.list_str_adapter.ListStrAdapter method), 139
 - get_text_color() (traitsui.table_column.NumericColumn method), 143
 - get_text_color() (traitsui.table_column.TableColumn method), 144
 - get_text_color() (traitsui.tabular_adapter.TabularAdapter method), 151
 - get_text_font() (traitsui.table_column.NumericColumn method), 143
 - get_text_font() (traitsui.table_column.TableColumn method), 144
 - get_tooltip() (traitsui.table_column.TableColumn method), 144
 - get_tooltip() (traitsui.tabular_adapter.TabularAdapter method), 151
 - get_tooltip() (traitsui.tree_node.ITreeNode method), 158
 - get_tooltip() (traitsui.tree_node.ITreeNodeAdapter method), 161
 - get_tooltip() (traitsui.tree_node.ITreeNodeAdapterBridge method), 163
 - get_tooltip() (traitsui.tree_node.ObjectTreeNode method), 166
 - get_tooltip() (traitsui.tree_node.TreeNode method), 168
 - get_type() (traitsui.table_column.NumericColumn method), 143
 - get_type() (traitsui.table_column.TableColumn method), 144
 - get_ui_db() (traitsui.ui.UI method), 172
 - get_undo_item() (traitsui.editor.Editor method), 126
 - get_value() (traitsui.color_column.ColorColumn method), 125
 - get_value() (traitsui.table_column.ListColumn method), 142
 - get_value() (traitsui.table_column.NumericColumn method), 143
 - get_value() (traitsui.table_column.ObjectColumn method), 143
 - get_value() (traitsui.table_filter.GenericTableFilterRuleAndOrColumn method), 145
 - get_value() (traitsui.table_filter.GenericTableFilterRuleEnabledColumn method), 146
 - get_vertical_alignment() (traitsui.table_column.NumericColumn method), 143
 - get_vertical_alignment() (traitsui.table_column.TableColumn method), 144
 - get_view() (traitsui.instance_choice.InstanceChoiceItem method), 135
 - get_view() (traitsui.table_column.TableColumn method), 144
 - get_view() (traitsui.tree_node.ITreeNode method), 158
 - get_view() (traitsui.tree_node.ITreeNodeAdapter method), 161
 - get_view() (traitsui.tree_node.ITreeNodeAdapterBridge method), 163
 - get_view() (traitsui.tree_node.MultiTreeNode method), 164
 - get_view() (traitsui.tree_node.ObjectTreeNode method), 166
 - get_view() (traitsui.tree_node.TreeNode method), 168
 - get_width() (traitsui.table_column.TableColumn method), 144
 - get_width() (traitsui.tabular_adapter.TabularAdapter method), 151
 - Group, 9, 97
 - attributes, 9
 - examples, View, 8
 - subclasses, 10
 - Group (class in traitsui.group), 128
 - gt() (traitsui.table_filter.GenericTableFilterRule method), 145
 - GUIToolkit (class in traitsui.null.toolkit), 117
- ## H
- Handler, 97
 - Handler (class in traitsui.handler), 130
 - handler attribute, 15
 - Handler class
 - as MVC controller, 2

- has_children() (traitsui.tree_node.ITreeNode method), 158
- has_children() (traitsui.tree_node.ITreeNodeAdapter method), 161
- has_children() (traitsui.tree_node.ITreeNodeAdapterBridge method), 163
- has_children() (traitsui.tree_node.MultiTreeNode method), 164
- has_children() (traitsui.tree_node.ObjectTreeNode method), 166
- has_children() (traitsui.tree_node.TreeNode method), 168
- has_focus attribute, 6
- HasTraits, 97
- HasTraits class
 - as MVC model, 2
- Heading (class in traitsui.item), 136
- Heading class, 7
- height attribute
 - Item, 6
 - View, 14
- help attribute
 - Group, 10
 - Item, 6
 - View, 16
- help_id attribute
 - Group, 10
 - Item, 6
 - View, 16
- help_template() (in module traitsui.help_template), 134
- HelpAction (in module traitsui.menu), 141
- HelpButton (in module traitsui.menu), 141
- HelpTemplate (class in traitsui.help_template), 133
- HFlow, 10
- HFlow (class in traitsui.group), 128
- HGroup, 10
- HGroup (class in traitsui.group), 128
- hide_root (traitsui.editors.tree_editor.ToolkitEditorFactory attribute), 112
- history_editor() (in module traitsui.editors.history_editor), 106
- history_editor() (traitsui.toolkit.Toolkit method), 154
- hook_events() (traitsui.toolkit.Toolkit method), 154
- HSplit, 10
- HSplit (class in traitsui.group), 129
- html_editor() (in module traitsui.editors.html_editor), 106
- html_editor() (traitsui.toolkit.Toolkit method), 154
- I
- icon attribute, 14
- icon_size (traitsui.editors.tree_editor.ToolkitEditorFactory attribute), 112
- id attribute
 - Group, 10
 - Item, 6
 - View, 16
- ignored_traits (traitsui.table_filter.GenericTableFilterRule attribute), 145
- ignored_traits (traitsui.table_filter.TableFilter attribute), 147
- IListStrAdapter (class in traitsui.list_str_adapter), 138
- image (traitsui.list_str_adapter.ListStrAdapter attribute), 139
- image (traitsui.tabular_adapter.TabularAdapter attribute), 151
- image attribute, 6
 - Group export attribute
 - Group, 9
 - View, 14
- image_editor() (traitsui.toolkit.Toolkit method), 154
- image_enum_editor() (traitsui.toolkit.Toolkit method), 154
- image_size() (traitsui.toolkit.Toolkit method), 154
- ImageEditor (class in traitsui.editors.image_editor), 107
- ImageEnumEditor (in module traitsui.editors.image_enum_editor), 107
- imports attribute, 15
- Include
 - examples, 22
 - object, 22
- Include (class in traitsui.include), 134
- indent() (traitsui.editors.html_editor.ToolkitEditorFactory method), 106
- index (traitsui.list_str_adapter.AnIListStrAdapter attribute), 137
- index (traitsui.list_str_adapter.IListStrAdapter attribute), 138
- index (traitsui.list_str_adapter.ListStrAdapter attribute), 139
- index() (traitsui.editors.table_editor.ReversedList method), 110
- index_alignment (traitsui.ui_editors.data_frame_editor.DataFrameAdapter attribute), 122
- index_text (traitsui.ui_editors.data_frame_editor.DataFrameAdapter attribute), 122
- info() (traitsui.null.font_trait.TraitFont method), 117
- info_action_handler() (traitsui.tests.test_handler.SampleHandler method), 120
- info_action_handler() (traitsui.tests.test_handler.SampleObject method), 120
- info_text (traitsui.ui_traits.ATheme attribute), 173
- info_text (traitsui.ui_traits.ViewStatus attribute), 173
- init() (traitsui.delegating_handler.DelegatingHandler method), 125
- init() (traitsui.editor.Editor method), 126
- init() (traitsui.editor_factory.EditorFactory method), 127
- init() (traitsui.editors.image_enum_editor.ToolkitEditorFactory

- method), 107
- init() (traitsui.editors.range_editor.ToolkitEditorFactory method), 108
- init() (traitsui.editors.tuple_editor.SimpleEditor method), 114
- init() (traitsui.extras.saving.SaveHandler method), 116
- init() (traitsui.handler.Handler method), 131
- init() (traitsui.ui_editor.UIEditor method), 172
- init_info() (traitsui.handler.Controller method), 130
- init_info() (traitsui.handler.Handler method), 131
- init_ui() (traitsui.ui_editor.UIEditor method), 173
- insert() (traitsui.editors.table_editor.ReversedList method), 111
- insert() (traitsui.list_str_adapter.ListStrAdapter method), 139
- insert() (traitsui.tabular_adapter.TabularAdapter method), 151
- insert() (traitsui.ui_editors.data_frame_editor.DataFrameAdapter method), 122
- insert_child() (traitsui.tree_node.ITreeNode method), 158
- insert_child() (traitsui.tree_node.ITreeNodeAdapter method), 161
- insert_child() (traitsui.tree_node.ITreeNodeAdapterBridge method), 163
- insert_child() (traitsui.tree_node.ObjectTreeNode method), 166
- insert_child() (traitsui.tree_node.TreeNode method), 168
- instance, 97
- instance_editor() (traitsui.toolkit.Toolkit method), 154
- InstanceChoice (class in traitsui.instance_choice), 135
- InstanceChoiceItem (class in traitsui.instance_choice), 135
- InstanceDropChoice (class in traitsui.instance_choice), 135
- InstanceEditor (in module traitsui.editors.instance_editor), 107
- InstanceFactoryChoice (class in traitsui.instance_choice), 135
- IntNode (class in traitsui.value_tree), 176
- is_addable() (traitsui.tree_node.TreeNode method), 168
- is_auto_editable() (traitsui.table_column.TableColumn method), 144
- is_button() (traitsui.base_panel.BasePanel method), 124
- is_cacheable (traitsui.list_str_adapter.AnIListStrAdapter attribute), 137
- is_cacheable (traitsui.list_str_adapter.IListStrAdapter attribute), 138
- is_cacheable (traitsui.tabular_adapter.AnITabularAdapter attribute), 147
- is_cacheable (traitsui.tabular_adapter.ITabularAdapter attribute), 147
- is_compatible() (traitsui.instance_choice.InstanceChoice method), 135
- is_compatible() (traitsui.instance_choice.InstanceChoiceItem method), 135
- is_compatible() (traitsui.instance_choice.InstanceFactoryChoice method), 135
- is_droppable() (traitsui.instance_choice.InstanceChoiceItem method), 135
- is_droppable() (traitsui.instance_choice.InstanceFactoryChoice method), 135
- is_droppable() (traitsui.table_column.NumericColumn method), 143
- is_droppable() (traitsui.table_column.ObjectColumn method), 143
- is_droppable() (traitsui.table_column.TableColumn method), 144
- is_editable() (traitsui.extras.edit_column.EditColumn method), 115
- is_editable() (traitsui.table_column.NumericColumn method), 143
- is_editable() (traitsui.table_column.TableColumn method), 145
- is_includable() (traitsui.group.Group method), 128
- is_includable() (traitsui.item.Item method), 136
- is_includable() (traitsui.view_element.ViewElement method), 179
- is_node_for() (traitsui.tree_node.ObjectTreeNode method), 166
- is_node_for() (traitsui.tree_node.TreeNode method), 168
- is_selectable() (traitsui.instance_choice.InstanceChoiceItem method), 135
- is_selectable() (traitsui.instance_choice.InstanceFactoryChoice method), 135
- is Spacer() (traitsui.item.Item method), 136
- is_true() (traitsui.table_filter.GenericTableFilterRule method), 145
- ITabularAdapter (class in traitsui.tabular_adapter), 147
- italics (traitsui.editors.styled_date_editor.CellFormat attribute), 110
- Item, 6, 97
 - attributes, 6
 - object, 6
 - subclasses, 7
- Item (class in traitsui.item), 136
- item (traitsui.list_str_adapter.AnIListStrAdapter attribute), 138
- item (traitsui.list_str_adapter.IListStrAdapter attribute), 138
- item (traitsui.list_str_adapter.ListStrAdapter attribute), 139
- item (traitsui.tabular_adapter.AnITabularAdapter attribute), 147
- item (traitsui.tabular_adapter.ITabularAdapter attribute), 147
- item (traitsui.tabular_adapter.TabularAdapter attribute), 152
- ITreeNode (class in traitsui.tree_node), 157

ITreeNodeAdapter (class in traitsui.tree_node), 159

ITreeNodeAdapterBridge (class in traitsui.tree_node), 161

K

key() (traitsui.table_column.ListColumn method), 142

key() (traitsui.table_column.ObjectColumn method), 143

key_binding_editor() (in module traitsui.editors.key_binding_editor), 107

key_binding_editor() (traitsui.toolkit.Toolkit method), 154

key_binding_for() (traitsui.key_bindings.KeyBindings method), 137

key_bindings attribute, 15

key_event_to_name() (traitsui.toolkit.Toolkit method), 154

key_handler() (traitsui.ui.UI method), 172

KeyBinding (class in traitsui.key_bindings), 137

KeyBindings (class in traitsui.key_bindings), 137

kind attribute, 11

kiva_font_trait() (traitsui.null.toolkit.GUIToolkit method), 117

kiva_font_trait() (traitsui.toolkit.Toolkit method), 154

klass (traitsui.ui_editors.data_frame_editor.DataFrameEditor attribute), 123

L

Label (class in traitsui.item), 136

label attribute

Group, 9

Item, 6

Label class, 7

label_map (traitsui.tabular_adapter.TabularAdapter attribute), 152

layout attribute, 9

le() (traitsui.table_filter.GenericTableFilterRule method), 145

len() (traitsui.list_str_adapter.ListStrAdapter method), 140

len() (traitsui.tabular_adapter.TabularAdapter method), 152

len() (traitsui.ui_editors.array_view_editor.ArrayViewAdapter method), 122

lines_mode (traitsui.editors.tree_editor.ToolkitEditorFactory attribute), 112

list_editor() (traitsui.toolkit.Toolkit method), 154

list_str_editor() (traitsui.toolkit.Toolkit method), 154

ListColumn (class in traitsui.table_column), 142

ListEditor (in module traitsui.editors.list_editor), 107

ListItemProxy (class in traitsui.editors.list_editor), 107

ListNode (class in traitsui.value_tree), 176

ListStrAdapter (class in traitsui.list_str_adapter), 138

ListStrEditor (class in traitsui.editors.list_str_editor), 108

ListUndoItem (class in traitsui.undo), 174

live, 97

definition, 11

window kind, 11

LiveButtons, 14

livemodal, 97

livemodal window kind, 11

log_change() (traitsui.editor.Editor method), 126

lt() (traitsui.table_filter.GenericTableFilterRule method), 145

M

menu (traitsui.tabular_adapter.TabularAdapter attribute), 152

menubar attribute, 15

MenuTableFilter (class in traitsui.table_filter), 146

merge() (traitsui.key_bindings.KeyBindings method), 137

merge_undo() (traitsui.undo.AbstractUndoItem method), 174

merge_undo() (traitsui.undo.ListUndoItem method), 174

merge_undo() (traitsui.undo.UndoItem method), 175

Message (class in traitsui.message), 141

message() (in module traitsui.message), 142

MethodNode (class in traitsui.value_tree), 176

modal, 97

definition, 11

window kind, 11

ModalButtons, 14

model, 2, 97

Model-View-Controller, 2

model_view attribute, 15

ModelView (built-in class), 25

ModelView (class in traitsui.handler), 133

multi-object Views, 21

examples, 21

multi_nodes (traitsui.editors.tree_editor.ToolkitEditorFactory attribute), 113

multiple Views, 18

examples, 18

MultiTreeNode (class in traitsui.tree_node), 163

MultiValueTreeNodeObject (class in traitsui.value_tree), 176

MVC, 97

MVC design pattern, 2

N

name (traitsui.tabular_adapter.TabularAdapter attribute), 152

name attribute, 6

named_value() (traitsui.editor_factory.EditorFactory method), 127

ne() (traitsui.table_filter.GenericTableFilterRule method), 145

NoButtons, 14

NoButtons (in module traitsui.menu), 141
 node_for() (traitsui.value_tree.SingleValueTreeNodeObject method), 177
 nodes (traitsui.editors.tree_editor.ToolkitEditorFactory attribute), 113
 NoneNode (class in traitsui.value_tree), 176
 nonmodal, 97
 nonmodal window kind, 11
 null toolkit, 3
 null_editor() (in module traitsui.editors.null_editor), 108
 null_editor() (traitsui.toolkit.Toolkit method), 154
 NumericColumn (class in traitsui.table_column), 142

O

object, 97
 Include, 22
 Item, 6
 View, 4, 6
 object (traitsui.tabular_adapter.TabularAdapter attribute), 152
 object attribute
 Group, 9
 View, 15
 object_action_handler() (traitsui.tests.test_handler.SampleObject method), 120
 ObjectColumn (class in traitsui.table_column), 143
 ObjectNode (class in traitsui.value_tree), 176
 ObjectTreeNode (class in traitsui.tree_node), 165
 odd_bg_color (traitsui.list_str_adapter.ListStrAdapter attribute), 140
 odd_bg_color (traitsui.tabular_adapter.TabularAdapter attribute), 152
 odd_text_color (traitsui.list_str_adapter.ListStrAdapter attribute), 140
 odd_text_color (traitsui.tabular_adapter.TabularAdapter attribute), 152
 OKButton, 13
 OKButton (in module traitsui.menu), 141
 OKCancelsButtons, 14
 on_activated (traitsui.editors.tree_editor.ToolkitEditorFactory attribute), 113
 on_apply attribute, 15
 on_click (traitsui.editors.tree_editor.ToolkitEditorFactory attribute), 113
 on_click() (traitsui.table_column.TableColumn method), 145
 on_dclick (traitsui.editors.tree_editor.ToolkitEditorFactory attribute), 113
 on_dclick() (traitsui.table_column.TableColumn method), 145
 on_help_call() (in module traitsui.help), 133
 on_hover (traitsui.editors.tree_editor.ToolkitEditorFactory attribute), 113

on_select (traitsui.editors.tree_editor.ToolkitEditorFactory attribute), 113
 open_view_for() (traitsui.handler.Handler method), 131
 ordered_set_editor() (traitsui.toolkit.Toolkit method), 154
 orientation (traitsui.editors.tree_editor.ToolkitEditorFactory attribute), 113
 orientation attribute, 9
 OtherNode (class in traitsui.value_tree), 177

P

padding attribute
 Group, 9
 Item, 6
 panel, 12, 97
 window kind, 11
 Parent (class in traitsui.tests.test_regression), 120
 parse_block() (traitsui.editors.html_editor.ToolkitEditorFactory method), 106
 parse_extended_name() (traitsui.editor.Editor method), 126
 parse_list() (traitsui.editors.html_editor.ToolkitEditorFactory method), 106
 parse_text() (traitsui.editors.html_editor.ToolkitEditorFactory method), 106
 perform() (traitsui.base_panel.BasePanel method), 124
 perform() (traitsui.editors.table_editor.BaseTableEditor method), 110
 perform() (traitsui.handler.Handler method), 131
 perform() (traitsui.tests.test_handler.PyfaceAction method), 119
 perform() (traitsui.tests.test_handler.TraitsUIAction method), 120
 plot_editor() (traitsui.toolkit.Toolkit method), 154
 pop_level() (traitsui.ui.UI method), 172
 PopupEditor (class in traitsui.editors.popup_editor), 108
 position() (traitsui.handler.Handler method), 132
 position() (traitsui.toolkit.Toolkit method), 154
 predefined trait type, 97
 prepare() (traitsui.editor.Editor method), 126
 prepare_ui() (traitsui.ui.UI method), 172
 ProgressEditor (in module traitsui.editors.progress_editor), 108
 promptForSave() (traitsui.extras.saving.SaveHandler method), 116
 push_level() (traitsui.ui.UI method), 172
 PyfaceAction (class in traitsui.tests.test_handler), 119

Q

Qt toolkit, 3

R

raise_to_debug() (in module traitsui.api), 123
 range_check() (in module traitsui.null.rgb_color_trait), 117

- [range_editor\(\)](#) (traitsui.toolkit.Toolkit method), 154
[RangeEditor](#) (in module traitsui.editors.range_editor), 108
[Readonly](#) (class in traitsui.item), 136
[Readonly](#) class, 7
[readonly_editor\(\)](#) (traitsui.editor_factory.EditorFactory method), 127
[readonly_editor\(\)](#) (traitsui.editors.csv_list_editor.CSVListEditor method), 103
[readonly_editor\(\)](#) (traitsui.editors.default_override.DefaultOverride method), 104
[readonly_editor\(\)](#) (traitsui.editors.table_editor.ToolkitEditorFactory method), 111
[rebuild_ui\(\)](#) (traitsui.toolkit.Toolkit method), 154
[recyclable_traits](#) (traitsui.ui.UI attribute), 172
[recycle\(\)](#) (traitsui.ui.UI method), 172
[redo\(\)](#) (traitsui.undo.AbstractUndoItem method), 174
[redo\(\)](#) (traitsui.undo.ListUndoItem method), 174
[redo\(\)](#) (traitsui.undo.UndoHistory method), 174
[redo\(\)](#) (traitsui.undo.UndoHistoryUndoItem method), 175
[redo\(\)](#) (traitsui.undo.UndoItem method), 175
[RedoAction](#) (in module traitsui.menu), 141
[refresh](#) (traitsui.editors.tree_editor.ToolkitEditorFactory attribute), 113
[remove\(\)](#) (traitsui.ui.Dispatcher method), 171
[replace_include\(\)](#) (traitsui.group.Group method), 128
[replace_include\(\)](#) (traitsui.view.View method), 179
[replace_include\(\)](#) (traitsui.view_element.ViewElement method), 179
[reset\(\)](#) (traitsui.ui.UI method), 172
[resizable](#) attribute, 6
[View](#), 14
[restore_prefs\(\)](#) (traitsui.editor.Editor method), 126
[restore_prefs\(\)](#) (traitsui.ui.UI method), 172
[restore_prefs\(\)](#) (traitsui.ui_editor.UEditor method), 173
[ReversedList](#) (class in traitsui.editors.table_editor), 110
[revert\(\)](#) (traitsui.handler.Handler method), 132
[revert\(\)](#) (traitsui.tests.test_handler.SampleHandler method), 120
[revert\(\)](#) (traitsui.undo.UndoHistory method), 174
[RevertAction](#) (in module traitsui.menu), 141
[RevertButton](#), 13
[RevertButton](#) (in module traitsui.menu), 141
[rgb_color_editor\(\)](#) (traitsui.toolkit.Toolkit method), 154
[rgb_color_trait\(\)](#) (traitsui.null.toolkit.GUIToolkit method), 117
[rgb_color_trait\(\)](#) (traitsui.toolkit.Toolkit method), 154
[rgba_color_editor\(\)](#) (traitsui.toolkit.Toolkit method), 154
[rgba_color_trait\(\)](#) (traitsui.toolkit.Toolkit method), 154
[RGBColorEditor\(\)](#) (in module traitsui.editors.rgb_color_editor), 109
[RGBColorTrait\(\)](#) (in module traitsui.toolkit_traits), 157
[RootNode](#) (class in traitsui.value_tree), 177
[route_event\(\)](#) (traitsui.toolkit.Toolkit method), 154
[route_event\(\)](#) (traitsui.ui.UI method), 172
[row](#) (traitsui.tabular_adapter.AnITabularAdapter attribute), 147
[row](#) (traitsui.tabular_adapter.ITabularAdapter attribute), 148
[row](#) (traitsui.tabular_adapter.TabularAdapter attribute), 152
[row_label_name](#) (traitsui.tabular_adapter.TabularAdapter attribute), 152
[RuleTableFilter](#) (class in traitsui.table_filter), 146
- ## S
- [SampleHandler](#) (class in traitsui.tests.test_handler), 120
[SampleObject](#) (class in traitsui.tests.test_handler), 120
[save\(\)](#) (traitsui.extras.saving.CanSaveMixin method), 115
[save\(\)](#) (traitsui.extras.saving.SaveHandler method), 116
[save_prefs\(\)](#) (traitsui.editor.Editor method), 126
[save_prefs\(\)](#) (traitsui.ui.UI method), 172
[save_prefs\(\)](#) (traitsui.ui_editor.UEditor method), 173
[save_window\(\)](#) (traitsui.toolkit.Toolkit method), 154
[saveAs\(\)](#) (traitsui.extras.saving.SaveHandler method), 116
[SaveHandler](#) (class in traitsui.extras.saving), 115
[scrollable](#) attribute, 14
[ScrubberEditor](#) (class in traitsui.editors.scrubber_editor), 109
[SearchEditor](#) (class in traitsui.editors.search_editor), 109
[SearchStackItem](#) (class in traitsui.view_elements), 180
[select\(\)](#) (traitsui.tree_node.ITreeNode method), 158
[select\(\)](#) (traitsui.tree_node.ITreeNodeAdapter method), 161
[select\(\)](#) (traitsui.tree_node.ITreeNodeAdapterBridge method), 163
[select\(\)](#) (traitsui.tree_node.MultiTreeNode method), 164
[select\(\)](#) (traitsui.tree_node.ObjectTreeNode method), 166
[select\(\)](#) (traitsui.tree_node.TreeNode method), 168
[selected](#) (traitsui.editors.tree_editor.ToolkitEditorFactory attribute), 113
[selected](#) attribute, 9
[selection_mode](#) (traitsui.editors.tree_editor.ToolkitEditorFactory attribute), 113
[Separator](#) (in module traitsui.menu), 141
[set_container\(\)](#) (traitsui.group.Group method), 128
[set_container\(\)](#) (traitsui.group.ShadowGroup method), 129
[set_content\(\)](#) (traitsui.view.View method), 179
[set_focus\(\)](#) (traitsui.editor.Editor method), 126
[set_icon\(\)](#) (traitsui.toolkit.Toolkit method), 154
[set_label\(\)](#) (traitsui.tree_node.ITreeNode method), 158

- set_label() (traitsui.tree_node.ITreeNodeAdapter method), 161
 - set_label() (traitsui.tree_node.ITreeNodeAdapterBridge method), 163
 - set_label() (traitsui.tree_node.MultiTreeNode method), 164
 - set_label() (traitsui.tree_node.ObjectTreeNode method), 166
 - set_label() (traitsui.tree_node.TreeNode method), 168
 - set_menu_context() (traitsui.editors.table_editor.BaseTableEditor method), 110
 - set_prefs() (traitsui.ui.UI method), 172
 - set_text() (traitsui.list_str_adapter.ListStrAdapter method), 140
 - set_text() (traitsui.tabular_adapter.TabularAdapter method), 152
 - set_title() (traitsui.toolkit.Toolkit method), 154
 - set_value() (traitsui.table_column.ListColumn method), 142
 - set_value() (traitsui.table_column.NumericColumn method), 143
 - set_value() (traitsui.table_column.ObjectColumn method), 143
 - setattr() (traitsui.handler.Handler method), 132
 - SetEditor (in module traitsui.editors.set_editor), 109
 - SetNode (class in traitsui.value_tree), 177
 - ShadowGroup (class in traitsui.group), 129
 - shared_editor (traitsui.editors.tree_editor.ToolkitEditorFactory attribute), 113
 - shell_editor() (traitsui.toolkit.Toolkit method), 155
 - ShellEditor (in module traitsui.editors.shell_editor), 109
 - show() (traitsui.message.AutoCloseMessage method), 141
 - show_border attribute, 9
 - show_help() (in module traitsui.help), 133
 - show_help() (traitsui.handler.Handler method), 132
 - show_help() (traitsui.tests.test_handler.SampleHandler method), 120
 - show_help() (traitsui.toolkit.Toolkit method), 155
 - show_icons (traitsui.editors.tree_editor.ToolkitEditorFactory attribute), 113
 - show_index (traitsui.ui_editors.data_frame_editor.DataFrameEditor attribute), 123
 - show_label attribute, 6
 - show_labels attribute, 9
 - show_left attribute, 9
 - show_titles (traitsui.ui_editors.data_frame_editor.DataFrameEditor attribute), 123
 - simple_editor() (traitsui.editor_factory.EditorFactory method), 127
 - simple_editor() (traitsui.editors.csv_list_editor.CSVListEditor method), 104
 - simple_editor() (traitsui.editors.default_override.DefaultOverride method), 104
 - simple_editor() (traitsui.editors.range_editor.ToolkitEditorFactory method), 108
 - SimpleEditor (class in traitsui.editors.tuple_editor), 114
 - SingleValueTreeNodeObject (class in traitsui.value_tree), 177
 - skip_event() (traitsui.toolkit.Toolkit method), 155
 - Spring (class in traitsui.item), 136
 - Spring class, 7
 - springy attribute
 - Group, 9
 - Item, 6
 - StandardMenuBar (in module traitsui.menu), 141
 - starts_with() (traitsui.table_filter.GenericTableFilterRule method), 145
 - statusbar attribute, 14
 - StatusItem (class in traitsui.ui_traits), 173
 - string_value() (traitsui.editor.Editor method), 126
 - StringNode (class in traitsui.value_tree), 178
 - style (traitsui.color_column.ColorColumn attribute), 125
 - style attribute
 - Group, 9
 - Item, 6
 - View, 14
 - StyledDateEditor (in module traitsui.editors.styled_date_editor), 110
 - subclasses
 - Group, 10
 - Item, 7
 - subpanel, 12, 97
 - window kind, 11
 - sync_value() (traitsui.editor.Editor method), 127
 - sync_view() (traitsui.ui.UI method), 172
- ## T
- Tabbed, 10
 - Tabbed (class in traitsui.group), 129
 - table_editor() (traitsui.toolkit.Toolkit method), 155
 - TableColumn (class in traitsui.table_column), 143
 - TableEditor (in module traitsui.editors.table_editor), 111
 - TableFilter (class in traitsui.table_filter), 146
 - tabular_editor() (traitsui.toolkit.Toolkit method), 155
 - TabularAdapter (class in traitsui.tabular_adapter), 148
 - TabularEditor (class in traitsui.editors.tabular_editor), 111
 - target_name() (traitsui.table_column.ObjectColumn method), 143
 - test_attribute_error() (traitsui.tests.test_regression.TestRegression method), 121
 - test_close_handler() (traitsui.tests.test_handler.TestHandler method), 120

- test_creation_sets_shadow_first() (trait-sui.tests.test_shadow_group.TestShadowGroup method), 121
- test_default_toolkit() (trait-sui.tests.test_toolkit.TestToolkit method), 121
- test_editor_on_delegates_to_event() (trait-sui.tests.test_regression.TestRegression method), 121
- test_help_handler() (trait-sui.tests.test_handler.TestHandler method), 120
- test_list_str_adapter_length() (in module trait-sui.tests.editors.test_liststr_editor), 118
- test_nonexistent_toolkit() (trait-sui.tests.test_toolkit.TestToolkit method), 121
- test_nonstandard_toolkit() (trait-sui.tests.test_toolkit.TestToolkit method), 121
- test_perform_action_handler() (trait-sui.tests.test_handler.TestHandler method), 120
- test_perform_click_handler() (trait-sui.tests.test_handler.TestHandler method), 120
- test_perform_info_action_handler() (trait-sui.tests.test_handler.TestHandler method), 120
- test_perform_object_handler() (trait-sui.tests.test_handler.TestHandler method), 120
- test_perform_pyface_action() (trait-sui.tests.test_handler.TestHandler method), 120
- test_perform_traitsui_action() (trait-sui.tests.test_handler.TestHandler method), 120
- test_redo_handler() (trait-sui.tests.test_handler.TestHandler method), 120
- test_revert_handler() (trait-sui.tests.test_handler.TestHandler method), 120
- test_undo_handler() (trait-sui.tests.test_handler.TestHandler method), 120
- TestHandler (class in traitsui.tests.test_handler), 120
- TestRegression (class in traitsui.tests.test_regression), 121
- TestShadowGroup (class in trait-sui.tests.test_shadow_group), 121
- TestToolkit (class in traitsui.tests.test_toolkit), 121
- text (traitsui.tabular_adapter.TabularAdapter attribute), 152
- text (traitsui.ui_editors.data_frame_editor.DataFrameAdapter attribute), 122
- text_color (traitsui.list_str_adapter.ListStrAdapter attribute), 140
- text_color (traitsui.tabular_adapter.TabularAdapter attribute), 152
- text_editor() (traitsui.editor_factory.EditorFactory method), 127
- text_editor() (traitsui.editors.csv_list_editor.CSVListEditor method), 104
- text_editor() (traitsui.editors.default_override.DefaultOverride method), 104
- text_editor() (traitsui.toolkit.Toolkit method), 155
- TextEditor (in module traitsui.editors.text_editor), 111
- Theme (class in traitsui.theme), 153
- TimeEditor (class in traitsui.editors.time_editor), 111
- title attribute, 14
- title_editor() (traitsui.toolkit.Toolkit method), 155
- TitleEditor (in module traitsui.editors.title_editor), 111
- tno_activated() (traitsui.tree_node.TreeNodeObject method), 169
- tno_allows_children() (trait-sui.tree_node.TreeNodeObject method), 169
- tno_allows_children() (trait-sui.value_tree.MultiValueTreeNodeObject method), 176
- tno_allows_children() (trait-sui.value_tree.SingleValueTreeNodeObject method), 177
- tno_append_child() (traitsui.tree_node.TreeNodeObject method), 169
- tno_can_add() (traitsui.tree_node.TreeNodeObject method), 169
- tno_can_auto_close() (traitsui.tree_node.TreeNodeObject method), 169
- tno_can_auto_open() (traitsui.tree_node.TreeNodeObject method), 169
- tno_can_copy() (traitsui.tree_node.TreeNodeObject method), 169
- tno_can_copy() (traitsui.value_tree.SingleValueTreeNodeObject method), 177
- tno_can_delete() (traitsui.tree_node.TreeNodeObject method), 169
- tno_can_delete() (traitsui.value_tree.DictNode method), 175
- tno_can_delete() (traitsui.value_tree.ListNode method), 176
- tno_can_delete() (traitsui.value_tree.SingleValueTreeNodeObject method), 177
- tno_can_delete_me() (traitsui.tree_node.TreeNodeObject method), 169
- tno_can_insert() (traitsui.tree_node.TreeNodeObject

- method), 169
- tno_can_insert() (traitsui.value_tree.ListNode method), 176
- tno_can_insert() (traitsui.value_tree.SingleValueTreeNodeObject method), 177
- tno_can_rename() (traitsui.tree_node.TreeNodeObject method), 169
- tno_can_rename() (traitsui.value_tree.SingleValueTreeNodeObject method), 177
- tno_can_rename_me() (traitsui.tree_node.TreeNodeObject method), 169
- tno_click() (traitsui.tree_node.TreeNodeObject method), 169
- tno_confirm_delete() (traitsui.tree_node.TreeNodeObject method), 169
- tno_dclick() (traitsui.tree_node.TreeNodeObject method), 169
- tno_delete_child() (traitsui.tree_node.TreeNodeObject method), 170
- tno_drop_object() (traitsui.tree_node.TreeNodeObject method), 170
- tno_get_add() (traitsui.tree_node.TreeNodeObject method), 170
- tno_get_children() (traitsui.tree_node.TreeNodeObject method), 170
- tno_get_children() (traitsui.value_tree.DictNode method), 176
- tno_get_children() (traitsui.value_tree.MethodNode method), 176
- tno_get_children() (traitsui.value_tree.ObjectNode method), 177
- tno_get_children() (traitsui.value_tree.RootNode method), 177
- tno_get_children() (traitsui.value_tree.TraitsNode method), 178
- tno_get_children() (traitsui.value_tree.TupleNode method), 178
- tno_get_children_id() (traitsui.tree_node.TreeNodeObject method), 170
- tno_get_drag_object() (traitsui.tree_node.TreeNodeObject method), 170
- tno_get_icon() (traitsui.tree_node.TreeNodeObject method), 170
- tno_get_icon() (traitsui.value_tree.SingleValueTreeNodeObject method), 177
- tno_get_icon_path() (traitsui.tree_node.TreeNodeObject method), 170
- tno_get_label() (traitsui.tree_node.TreeNodeObject method), 170
- tno_get_label() (traitsui.value_tree.SingleValueTreeNodeObject method), 177
- method), 178
- tno_get_menu() (traitsui.tree_node.TreeNodeObject method), 170
- tno_get_name() (traitsui.tree_node.TreeNodeObject method), 170
- tno_get_tooltip() (traitsui.tree_node.TreeNodeObject method), 170
- tno_get_view() (traitsui.tree_node.TreeNodeObject method), 170
- tno_has_children() (traitsui.tree_node.TreeNodeObject method), 170
- tno_has_children() (traitsui.value_tree.MethodNode method), 176
- tno_has_children() (traitsui.value_tree.MultiValueTreeNodeObject method), 176
- tno_has_children() (traitsui.value_tree.ObjectNode method), 177
- tno_has_children() (traitsui.value_tree.SingleValueTreeNodeObject method), 178
- tno_has_children() (traitsui.value_tree.TraitsNode method), 178
- tno_has_children() (traitsui.value_tree.TupleNode method), 178
- tno_insert_child() (traitsui.tree_node.TreeNodeObject method), 170
- tno_is_node_for() (traitsui.tree_node.TreeNodeObject method), 170
- tno_select() (traitsui.tree_node.TreeNodeObject method), 170
- tno_set_label() (traitsui.tree_node.TreeNodeObject method), 170
- tno_set_label() (traitsui.value_tree.SingleValueTreeNodeObject method), 178
- tno_when_children_changed() (traitsui.tree_node.TreeNodeObject method), 170
- tno_when_children_changed() (traitsui.value_tree.TraitsNode method), 178
- tno_when_children_replaced() (traitsui.tree_node.TreeNodeObject method), 170
- tno_when_children_replaced() (traitsui.value_tree.TraitsNode method), 178
- tno_when_label_changed() (traitsui.tree_node.TreeNodeObject method), 170
- toolbar attribute, 15
- toolkit
 - environment variable, 3
 - flag, 3
 - selection, 3
- Toolkit (class in traitsui.toolkit), 153

- toolkit() (in module traitsui.toolkit), 156
 toolkit_object() (in module traitsui.toolkit), 156
 ToolkitEditorFactory (class in trait-sui.editors.boolean_editor), 101
 ToolkitEditorFactory (class in trait-sui.editors.button_editor), 102
 ToolkitEditorFactory (class in trait-sui.editors.check_list_editor), 102
 ToolkitEditorFactory (class in trait-sui.editors.code_editor), 102
 ToolkitEditorFactory (class in trait-sui.editors.color_editor), 102
 ToolkitEditorFactory (class in trait-sui.editors.compound_editor), 103
 ToolkitEditorFactory (class in trait-sui.editors.custom_editor), 104
 ToolkitEditorFactory (class in trait-sui.editors.directory_editor), 105
 ToolkitEditorFactory (class in traitsui.editors.dnd_editor), 105
 ToolkitEditorFactory (class in trait-sui.editors.drop_editor), 105
 ToolkitEditorFactory (class in trait-sui.editors.enum_editor), 105
 ToolkitEditorFactory (class in traitsui.editors.file_editor), 106
 ToolkitEditorFactory (class in trait-sui.editors.font_editor), 106
 ToolkitEditorFactory (class in trait-sui.editors.history_editor), 106
 ToolkitEditorFactory (class in trait-sui.editors.html_editor), 106
 ToolkitEditorFactory (class in trait-sui.editors.image_enum_editor), 107
 ToolkitEditorFactory (class in trait-sui.editors.instance_editor), 107
 ToolkitEditorFactory (class in traitsui.editors.list_editor), 107
 ToolkitEditorFactory (class in trait-sui.editors.progress_editor), 108
 ToolkitEditorFactory (class in trait-sui.editors.range_editor), 108
 ToolkitEditorFactory (class in trait-sui.editors.rgb_color_editor), 109
 ToolkitEditorFactory (class in traitsui.editors.set_editor), 109
 ToolkitEditorFactory (class in trait-sui.editors.shell_editor), 109
 ToolkitEditorFactory (class in trait-sui.editors.styled_date_editor), 110
 ToolkitEditorFactory (class in trait-sui.editors.table_editor), 111
 ToolkitEditorFactory (class in traitsui.editors.text_editor), 111
 ToolkitEditorFactory (class in trait-sui.editors.title_editor), 111
 ToolkitEditorFactory (class in traitsui.editors.tree_editor), 112
 ToolkitEditorFactory (class in trait-sui.editors.tuple_editor), 114
 ToolkitEditorFactory (class in trait-sui.editors.value_editor), 114
 tooltip (traitsui.tabular_adapter.TabularAdapter attribute), 153
 tooltip attribute, 6
 trait attribute, 97
 trait type, 97
 trait_context() (traitsui.handler.Controller method), 130
 trait_context() (traitsui.handler.ModelView method), 133
 trait_view_for() (traitsui.handler.Handler method), 132
 TraitFont (class in traitsui.null.font_trait), 116
 TraitObject (class in trait-sui.tests.editors.test_liststr_editor), 118
 Traits, 97
 traits_init() (traitsui.ui.UI method), 172
 traits_view attribute, 19
 TraitsNode (class in traitsui.value_tree), 178
 TraitsUI, 97
 traitsui (module), 180
 traitsui.api (module), 123
 traitsui.base_panel (module), 123
 traitsui.basic_editor_factory (module), 124
 traitsui.color_column (module), 125
 traitsui.context_value (module), 125
 traitsui.delegating_handler (module), 125
 traitsui.dock_window_theme (module), 126
 traitsui.editor (module), 126
 traitsui.editor_factory (module), 127
 traitsui.editors (module), 115
 traitsui.editors.api (module), 101
 traitsui.editors.boolean_editor (module), 101
 traitsui.editors.button_editor (module), 102
 traitsui.editors.check_list_editor (module), 102
 traitsui.editors.code_editor (module), 102
 traitsui.editors.color_editor (module), 102
 traitsui.editors.compound_editor (module), 103
 traitsui.editors.csv_list_editor (module), 103
 traitsui.editors.custom_editor (module), 104
 traitsui.editors.date_editor (module), 104
 traitsui.editors.default_override (module), 104
 traitsui.editors.directory_editor (module), 105
 traitsui.editors.dnd_editor (module), 105
 traitsui.editors.drop_editor (module), 105
 traitsui.editors.enum_editor (module), 105
 traitsui.editors.file_editor (module), 105
 traitsui.editors.font_editor (module), 106
 traitsui.editors.history_editor (module), 106

[traitsui.editors.html_editor \(module\)](#), 106
[traitsui.editors.image_editor \(module\)](#), 107
[traitsui.editors.image_enum_editor \(module\)](#), 107
[traitsui.editors.instance_editor \(module\)](#), 107
[traitsui.editors.key_binding_editor \(module\)](#), 107
[traitsui.editors.list_editor \(module\)](#), 107
[traitsui.editors.list_str_editor \(module\)](#), 108
[traitsui.editors.null_editor \(module\)](#), 108
[traitsui.editors.popup_editor \(module\)](#), 108
[traitsui.editors.progress_editor \(module\)](#), 108
[traitsui.editors.range_editor \(module\)](#), 108
[traitsui.editors.rgb_color_editor \(module\)](#), 109
[traitsui.editors.scrubber_editor \(module\)](#), 109
[traitsui.editors.search_editor \(module\)](#), 109
[traitsui.editors.set_editor \(module\)](#), 109
[traitsui.editors.shell_editor \(module\)](#), 109
[traitsui.editors.styled_date_editor \(module\)](#), 110
[traitsui.editors.table_editor \(module\)](#), 110
[traitsui.editors.tabular_editor \(module\)](#), 111
[traitsui.editors.text_editor \(module\)](#), 111
[traitsui.editors.time_editor \(module\)](#), 111
[traitsui.editors.title_editor \(module\)](#), 111
[traitsui.editors.tree_editor \(module\)](#), 112
[traitsui.editors.tuple_editor \(module\)](#), 114
[traitsui.editors.value_editor \(module\)](#), 114
[traitsui.editors_gen \(module\)](#), 128
[traitsui.extras \(module\)](#), 116
[traitsui.extras.edit_column \(module\)](#), 115
[traitsui.extras.saving \(module\)](#), 115
[traitsui.group \(module\)](#), 128
[traitsui.handler \(module\)](#), 129
[traitsui.help \(module\)](#), 133
[traitsui.help_template \(module\)](#), 133
[traitsui.helper \(module\)](#), 134
[traitsui.image \(module\)](#), 116
[traitsui.include \(module\)](#), 134
[traitsui.instance_choice \(module\)](#), 135
[traitsui.item \(module\)](#), 135
[traitsui.key_bindings \(module\)](#), 137
[traitsui.list_str_adapter \(module\)](#), 95, 137
[traitsui.menu \(module\)](#), 140
[traitsui.message \(module\)](#), 141
[traitsui.mimedata \(module\)](#), 142
[traitsui.null \(module\)](#), 117
[traitsui.null.color_trait \(module\)](#), 116
[traitsui.null.font_trait \(module\)](#), 116
[traitsui.null.rgb_color_trait \(module\)](#), 117
[traitsui.null.toolkit \(module\)](#), 117
[traitsui.table_column \(module\)](#), 142
[traitsui.table_filter \(module\)](#), 145
[traitsui.tabular_adapter \(module\)](#), 89, 147
[traitsui.tests \(module\)](#), 121
[traitsui.tests.editors \(module\)](#), 119
[traitsui.tests.editors.test_liststr_editor \(module\)](#), 118

[traitsui.tests.null_backend \(module\)](#), 119
[traitsui.tests.test_handler \(module\)](#), 119
[traitsui.tests.test_regression \(module\)](#), 120
[traitsui.tests.test_shadow_group \(module\)](#), 121
[traitsui.tests.test_toolkit \(module\)](#), 121
[traitsui.tests.ui_editors \(module\)](#), 119
[traitsui.theme \(module\)](#), 153
[traitsui.toolkit \(module\)](#), 153
[traitsui.toolkit_traits \(module\)](#), 157
[traitsui.tree_node \(module\)](#), 85, 157
[traitsui.ui \(module\)](#), 171
[traitsui.ui_editor \(module\)](#), 172
[traitsui.ui_editors \(module\)](#), 123
[traitsui.ui_editors.array_view_editor \(module\)](#), 121
[traitsui.ui_editors.data_frame_editor \(module\)](#), 122
[traitsui.ui_info \(module\)](#), 173
[traitsui.ui_traits \(module\)](#), 173
[traitsui.undo \(module\)](#), 174
[traitsui.value_tree \(module\)](#), 175
[traitsui.view \(module\)](#), 178
[traitsui.view_element \(module\)](#), 179
[traitsui.view_elements \(module\)](#), 180
[TraitsUIAction \(class in traitsui.tests.test_handler\)](#), 120
[tree_editor\(\) \(traitsui.toolkit.Toolkit method\)](#), 155
[TreeEditor \(in module traitsui.editors.tree_editor\)](#), 113
[TreeNode \(class in traitsui.tree_node\)](#), 166
[TreeNodeObject \(class in traitsui.tree_node\)](#), 169
[tuple_editor\(\) \(traitsui.toolkit.Toolkit method\)](#), 155
[TupleEditor \(in module traitsui.editors.tuple_editor\)](#), 114
[TupleNode \(class in traitsui.value_tree\)](#), 178
[TupleStructure \(class in traitsui.editors.tuple_editor\)](#), 114

U

[UCustom \(class in traitsui.item\)](#), 136
[UCustom class](#), 7
[UI \(class in traitsui.ui\)](#), 171
[ui \(traitsui.base_panel.BasePanel attribute\)](#), 124
[ui\(\), 20](#)
[ui\(\) \(traitsui.ui.UI method\)](#), 172
[ui\(\) \(traitsui.view.View method\)](#), 179
[ui_editor\(\) \(traitsui.toolkit.Toolkit method\)](#), 155
[ui_info\(\) \(traitsui.toolkit.Toolkit method\)](#), 155
[ui_live\(\) \(traitsui.toolkit.Toolkit method\)](#), 155
[ui_livemodal\(\) \(traitsui.toolkit.Toolkit method\)](#), 155
[ui_modal\(\) \(traitsui.toolkit.Toolkit method\)](#), 155
[ui_nonmodal\(\) \(traitsui.toolkit.Toolkit method\)](#), 155
[ui_panel\(\) \(traitsui.toolkit.Toolkit method\)](#), 155
[ui_popover\(\) \(traitsui.toolkit.Toolkit method\)](#), 155
[ui_popup\(\) \(traitsui.toolkit.Toolkit method\)](#), 155
[ui_subpanel\(\) \(traitsui.toolkit.Toolkit method\)](#), 155
[ui_wizard\(\) \(traitsui.toolkit.Toolkit method\)](#), 155
[UIEditor \(class in traitsui.ui_editor\)](#), 172
[UIInfo \(class in traitsui.ui_info\)](#), 173
[UIItem \(class in traitsui.item\)](#), 137

UItem class, 7
underline (traitsui.editors.styled_date_editor.CellFormat attribute), 110
undo() (traitsui.undo.AbstractUndoItem method), 174
undo() (traitsui.undo.ListUndoItem method), 174
undo() (traitsui.undo.UndoHistory method), 174
undo() (traitsui.undo.UndoHistoryUndoItem method), 175
undo() (traitsui.undo.UndoItem method), 175
UndoAction (in module traitsui.menu), 141
UndoButton, 13
UndoButton (in module traitsui.menu), 141
UndoHistory (class in traitsui.undo), 174
UndoHistoryUndoItem (class in traitsui.undo), 174
UndoItem (class in traitsui.undo), 175
update_editor() (traitsui.editor.Editor method), 127
update_editor() (traitsui.editors.tuple_editor.SimpleEditor method), 114
update_editor() (traitsui.ui_editor.UIEditor method), 173
updated attribute, 15
UReadonly (class in traitsui.item), 137
UReadonly class, 7
user_name_for() (in module traitsui.helper), 134

V

validate() (traitsui.extras.saving.CanSaveMixin method), 115
validate() (traitsui.null.font_trait.TraitFont method), 117
validate() (traitsui.ui_traits.ATheme method), 173
validate() (traitsui.ui_traits.ViewStatus method), 173
value (traitsui.list_str_adapter.AnIListStrAdapter attribute), 138
value (traitsui.list_str_adapter.IListStrAdapter attribute), 138
value (traitsui.list_str_adapter.ListStrAdapter attribute), 140
value (traitsui.tabular_adapter.AnITabularAdapter attribute), 147
value (traitsui.tabular_adapter.ITabularAdapter attribute), 148
value (traitsui.tabular_adapter.TabularAdapter attribute), 153
value_editor() (traitsui.toolkit.Toolkit method), 155
ValueEditor (in module traitsui.editors.value_editor), 114
vertical_padding (traitsui.editors.tree_editor.ToolkitEditorFactory attribute), 113
veto (traitsui.editors.tree_editor.ToolkitEditorFactory attribute), 113
VFlow, 10
VFlow (class in traitsui.group), 129
VFold, 10
VFold (class in traitsui.group), 129
VGrid, 10
VGrid (class in traitsui.group), 129
VGroup, 10
VGroup (class in traitsui.group), 129
View, 97
 as MVC view, 2
 attributes, 14
 contents, 6
 context, 16
 customizing, 11
 default, 16
 external, 16, 19
 Group examples, 8
 internal, 16
 methods for displaying, 19
 multi-object, 21
 multiple, 18
 object, 4, 6
 ways of displaying, 16
view, 98
View (class in traitsui.view), 178
view (in MVC), 2
view_application() (traitsui.toolkit.Toolkit method), 155
ViewElement, 98
ViewElement (class in traitsui.view_element), 179
ViewElements (class in traitsui.view_elements), 180
ViewHandler (class in traitsui.handler), 133
ViewStatus (class in traitsui.ui_traits), 173
ViewSubElement (class in traitsui.view_element), 180
visible_when (traitsui.menu.Action attribute), 140
visible_when attribute
 Group, 10
 Item, 6
VSplit, 10
VSplit (class in traitsui.group), 129

W

when_children_changed() (traitsui.tree_node.ITreeNode method), 158
when_children_changed() (traitsui.tree_node.ITreeNodeAdapter method), 161
when_children_changed() (traitsui.tree_node.ITreeNodeAdapterBridge method), 163
when_children_changed() (traitsui.tree_node.MultiTreeNode method), 164
when_children_changed() (traitsui.tree_node.ObjectTreeNode method), 166
when_children_changed() (traitsui.tree_node.TreeNode method), 168
when_children_replaced() (traitsui.tree_node.ITreeNode method), 159
when_children_replaced() (traitsui.tree_node.ITreeNodeAdapter method),

161

when_children_replaced() (traitsui.tree_node.ITreeNodeAdapterBridge method), 163

when_children_replaced() (traitsui.tree_node.MultiTreeNode method), 165

when_children_replaced() (traitsui.tree_node.ObjectTreeNode method), 166

when_children_replaced() (traitsui.tree_node.TreeNode method), 168

when_column_labels_change() (traitsui.tree_node.ITreeNode method), 159

when_column_labels_change() (traitsui.tree_node.ITreeNodeAdapter method), 161

when_column_labels_change() (traitsui.tree_node.ITreeNodeAdapterBridge method), 163

when_column_labels_change() (traitsui.tree_node.TreeNode method), 168

when_label_changed() (traitsui.tree_node.ITreeNode method), 159

when_label_changed() (traitsui.tree_node.ITreeNodeAdapter method), 161

when_label_changed() (traitsui.tree_node.ITreeNodeAdapterBridge method), 163

when_label_changed() (traitsui.tree_node.MultiTreeNode method), 165

when_label_changed() (traitsui.tree_node.ObjectTreeNode method), 166

when_label_changed() (traitsui.tree_node.TreeNode method), 169

widget, 6, 98

width (traitsui.tabular_adapter.TabularAdapter attribute), 153

width attribute

- Item, 6
- View, 14

windows

- panel, 12
- stand-alone, 11
- subpanel, 12
- wizard, 12

wizard, 12, 98

- window kind, 11

word_wrap (traitsui.editors.tree_editor.ToolkitEditorFactory attribute), 113

wx, 98

wxPython toolkit, 3

X

x attribute, 14

Y

y attribute, 14