The DockWindowFeature
Feature of DockWindows

David C. Morrill, Enthought, Inc.

Version 1, 21-Dec-06

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

Table of Contents

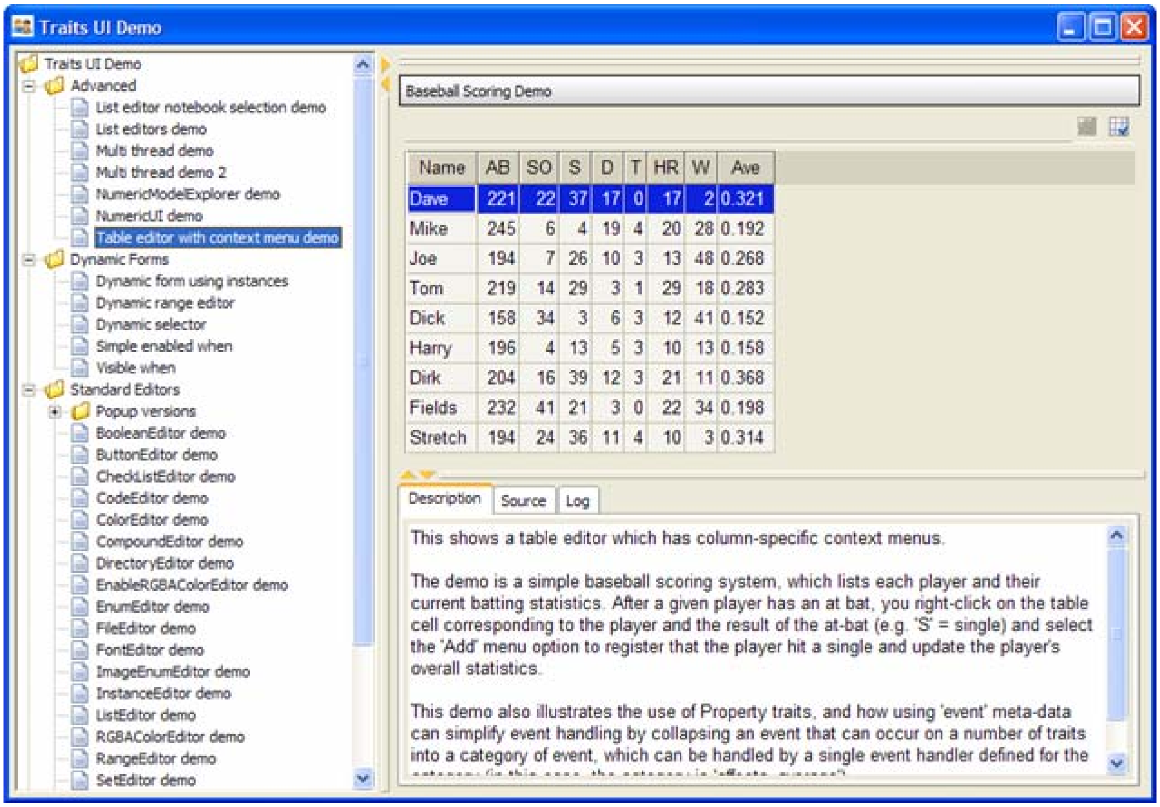
1 What is a DockWindow?

Before we describe what a **DockWindowFeature** is, let's begin with what a **DockWindow** is. A DockWindow is a combination of a user interface window and a layout manager that allows a user to arrange and rearrange the contents of the DockWindow using a combination of:

- Splitter bars
- Horizontal and vertical drag handles
- Draggable notebook tabs.

Users can change the size of individual DockWindow components by dragging or clicking on the splitter bars that separate the components, and reorganize the contents of the window by dragging notebook tabs or drag handles to other parts of the DockWindow.

The following two screen shots illustrate a DockWindow in use; they show different versions of the same Traits UI demo, which employs a DockWindow to allow users to organize the various informational sections of the demo in whatever way is most convenient for them.

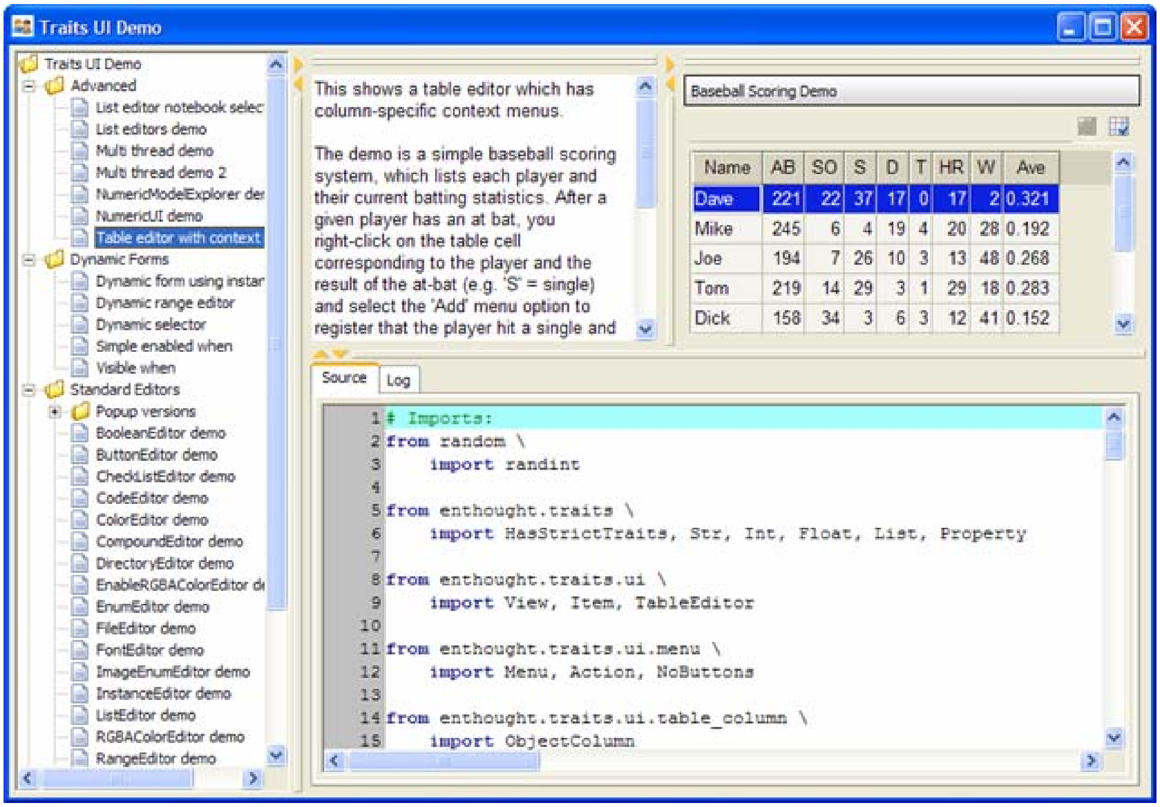


Name	AB	SO	S	D	T	HR	W	Ave
Dave	221	22	37	17	0	17	2	0.321
Mike	245	6	4	19	4	20	28	0.192
Joe	194	7	26	10	3	13	48	0.268
Tom	219	14	29	3	1	29	18	0.283
Dick	158	34	3	6	3	12	41	0.152
Harry	196	4	13	5	3	10	13	0.158
Dirk	204	16	39	12	3	21	11	0.368
Fields	232	41	21	3	0	22	34	0.198
Stretch	194	24	36	11	4	10	3	0.314

This shows a table editor which has column-specific context menus.

The demo is a simple baseball scoring system, which lists each player and their current batting statistics. After a given player has an at bat, you right-click on the table cell corresponding to the player and the result of the at-bat (e.g. 'S' = single) and select the 'Add' menu option to register that the player hit a single and update the player's overall statistics.

This demo also illustrates the use of Property traits, and how using 'event' meta-data can simplify event handling by collapsing an event that can occur on a number of traits into a category of event, which can be handled by a single event handler defined for the



This shows a table editor which has column-specific context menus.

The demo is a simple baseball scoring system, which lists each player and their current batting statistics. After a given player has an at bat, you right-click on the table cell corresponding to the player and the result of the at-bat (e.g. 'S' = single) and select the 'Add' menu option to register that the player hit a single and

Name	AB	SO	S	D	T	HR	W	Ave
Dave	221	22	37	17	0	17	2	0.321
Mike	245	6	4	19	4	20	28	0.192
Joe	194	7	26	10	3	13	48	0.268
Tom	219	14	29	3	1	29	18	0.283
Dick	158	34	3	6	3	12	41	0.152

```
# Imports:
from random \
    import randint

from enthought.traits \
    import HasStrictTraits, Str, Int, Float, List, Property

from enthought.traits.ui \
    import View, Item, TableEditor

from enthought.traits.ui.menu \
    import Menu, Action, NoButtons

from enthought.traits.ui.table_column \
    import ObjectColumn
```

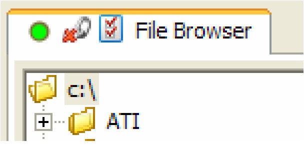
Developers can directly create DockWindows themselves or can have them automatically created using either the Traits UI toolkit (**enthought.traits.ui**) or the Envisage Workbench framework (**enthought.envisage.workbench**), two of the major user interface components of the Enthought Tool Suite (ETS). In particular, using either of these packages greatly simplifies the process of creating DockWindow-based applications, which in turn greatly increases the user configurability of applications, while requiring very little effort on the part of the developer.

The DockWindow package is a sub-package of PyFace (**enthought.pyface**), which provides access to the wxWidgets library in a way that supports the Model-View-Controller (MVC) design pattern.

2 What is a DockWindowFeature?

A **DockWindowFeature** (or *feature*, for short) is an architectural extension to the core **DockWindow** functionality that allows new capabilities and services to be dynamically added to DockWindows and to the application components embedded in them. Any number of unique features can be added to the DockWindows of an application, and the set of available features can vary from one application to the next. In addition, an application user is free to enable or disable any or all of the available features using the **Features** sub-menu of the DockWindow shortcut menu.

Visually, a feature typically appears as an icon on a DockWindow tab or drag bar. The following figure shows a DockWindow tab containing three icons, corresponding to three features associated with the DockWindow:



Each icon provides the user interface for its corresponding feature and, depending upon how the feature is implemented, might support any or all of the following user actions:

- Dragging: To act as a source of information.
- Dropping: To act as a receiver of information.
- Clicking: To trigger an action.
- Right-clicking: To display a context menu.

A feature does not add any new capabilities to the DockWindow component itself. It instead adds new capabilities to the application components whose views are contained in a DockWindow. These new capabilities facilitate the implementation and operation of the application components, as well as providing interesting and powerful ways for users to create temporary or permanent information flows and connections between independent application components. The DockWindows simply act as intermediaries, providing the framework for embedding features and application components, as well as the mechanism for connecting features to the application components they apply to.

The following list describes the stock features that are included with the DockWindow package. This list indicates some of the possible capabilities that features can implement, though they are definitely not limited to this list. The stock features are described in greater detail in the “Stock Features” section.

- **Connect**: Enables connecting two application objects by transferring data from a trait attribute on one object to a trait attribute on the other object.
- **Drag and Drop**: Enables dragging and dropping of application objects onto other application objects, with the receiver object taking appropriate action.
- **Drop File**: Enables dragging and dropping of specific file types.
- **Options**: Enables the user to customize preferences on an application object.
- **Popup Menu**: Enables displaying a context menu for an application object.
- **Save State**: Enables automatic saving and restoring of the state of an application object.
- **Save**: Provides a visual indication of when an application object’s state is not saved, as well as a mechanism for saving it.
- **Debug**: Enables a user (typically the application developer) to visualize and debug the internal state of an application object.
- **DockControl**: Enables an application object to access the **DockControl** object in which its view is displayed.
- **Custom Feature**: Enables creating a feature that is specific to a particular application object.

3 Using DockWindowFeatures

You can use **DockWindowFeature** capabilities in various ways:

- Add stock features to your application.
- Create your own feature, and then add it to your application.

3.1 Adding a Feature

There are two different ways to integrate a **DockWindowFeature** subclass into a DockWindows-based application, depending upon whether or not the application is using the Envisage Workbench plug-in.

3.1.1 Adding a Feature using the DockWindow API

If the application does not use the Envisage Workbench plug-in, then the feature can be added directly by calling the **enthought.pyface.dock.api.add_feature** function, which has the signature:

```
add_feature( feature_class )
```

The parameter *feature_class* is the **DockWindowFeature** subclass to be added to DockWindows in the application. This function returns **True** if the class is installed successfully and **False** if it was previously installed.

In the case where the class is installed successfully, and the class's **state** variable is initially 0 (uninstalled), the value is changed to 1 (active) to indicate that the feature is now active and can be applied to application components.

If the initial value of the class's **state** variable is 2 (disabled), then the value is not changed, and the class is left in the disabled state. In this case, the user must explicitly enable the feature using the DockWindow **Feature** shortcut sub-menu.

3.1.2 Adding a Feature as an Envisage Plug-in

If the application uses the Envisage Workbench plug-in, then the feature can be added by including an **enthought.envisage.workbench.workbench_plugin_definition.Feature** extension point in one of the application's plug-in definition files. The format of the **Feature** extension point is:

```
Feature( class_name = class_name )
```

The *class_name* parameter is the name of the **DockWindowFeature** subclass to be added.

3.1.3 Adding Stock Features to an Envisage Application

All of the features in the **enthought.pyface.dock.features** package can be added to any Envisage-based application by including the `enthought.developer.plugin_definition.py` file in your application's plug-in definitions file. Adding this file includes also all of the developer tool plug-ins contained in the **enthought.developer.tools** package.

However, if you want to include only a subset of the available features in your application, you can do so by cutting and pasting one or more of the following Envisage extension point definitions into one of your plug-in definition files:

```
# Necessary imports:
from \
enthought.envisage.workbench.workbench_plugin_definition \
    import Feature

# Add the features you need from the following list to the
# 'extension_points' section of an Envisage
plugin_definition.py
# file:

# Connect feature:
Feature( class_name =
    'enthought.pyface.dock.features.connect_feature.'
    'ConnectFeature' )
```

```
# Drag and Drop feature:
Feature( class_name =
    'enthought.pyface.dock.features.drag_drop_feature.'
    'DragDropFeature' )

# Drop File feature:
Feature( class_name =
    'enthought.pyface.dock.features.drop_file_feature.'
    'DropFileFeature' )

# Options feature:
Feature( class_name =
    'enthought.pyface.dock.features.options_feature.'
    'OptionsFeature' )

# Popup menu feature:
Feature( class_name =
    'enthought.pyface.dock.features.popup_menu_feature.'
    'PopupMenuFeature' )

# Save State feature:
Feature( class_name =
    'enthought.pyface.dock.features.save_state_feature.'
    'SaveStateFeature' )

# Save feature:
Feature( class_name =
    'enthought.pyface.dock.features.save_feature.'
    'SaveFeature' )

# Debug feature:
Feature( class_name =
    'enthought.pyface.dock.features.debug_feature.'
    'DebugFeature' )

# DockControl feature:
Feature( class_name =
    'enthought.pyface.dock.features.dock_control_feature.'
    'DockControlFeature' )

# Custom Feature feature:
Feature( class_name =
    'enthought.pyface.dock.features.custom_feature.'
    'ACustomFeature' )
```

3.2 Creating a Feature

Creating a new feature is a matter of creating a subclass of **DockWindowFeature**. The **DockWindowFeature** class defines the core methods and trait attributes that DockWindows use to determine which features apply to which application components, and to notify features when key application and user interface events occur. These items are described in detail in the API documentation for **enthought.pyface.dock.feature**. The following items are designed to be overridden:

- **feature_name** class variable
- **tab_image** trait attribute
- **bar_image** trait attribute
- **tooltip** trait attribute
- **left_click** method
- **right_click** method
- **drag** method
- **control_drag** method
- **shift_drag** method
- **alt_drag** method
- **can_drop** method
- **drop** method
- **dispose** method
- **feature_for** class method
- **is_feature_for** class method
- **new_feature** class method

There is no single correct way to create a new feature. The **DockWindowFeature** API is open-ended, which allows for creating many new and interesting interfaces. However, two introspection techniques have proven useful in creating a number of feature and application object classes: metadata and interfaces.

3.2.1 Using Object Metadata in a Feature

The **feature_for()** and **is_feature_for()** class methods can be the key to writing a useful feature. They provide an opportunity to examine an application object to determine whether the feature applies to it. The standard Python **isinstance()** function and the Traits metadata query methods (such as **trait_names()**) can be very useful for this task. In fact, many of the stock features are defined in terms of specific Traits metadata patterns that application object classes must follow in order to use a particular feature.

Using metadata has the following advantages:

- It does not require the application object class to inherit from a special base class (other than **HasTraits**) or to implement particular interfaces or protocols.
- The supplied metadata is often orthogonal to the class implementation. That is, the metadata is usually descriptive information attached to certain traits in the application object class that allows the feature to discover its applicability to the class and to retrieve information about how the feature should interact with the class. In general, removing the metadata does not alter the operation of the class in any way.

Related to the use of metadata is the importance of defining appropriate traits to attach the metadata to. Defining a trait with metadata is a way to implement, in essence, a *protocol* for interacting with the object that can be easily discovered by a feature. The feature can use the metadata to determine when and how it should read data from or write data to the application object's trait attribute. The application object class in turn can define a trait notification handler to process data assigned to the trait attribute, either by the feature or another application object attached to the trait attribute by the feature or some other means.

This pattern of defining feature-specific metadata and attaching it to the s of various application object classes is one that is used repeatedly in the stock features. You are encouraged to browse the source code for the various features defined in the **enthought.pyface.dock.features** package to see some complete examples of features that use these patterns. Likewise, you might also look at some of the application object classes in the **enthought.developer.tools** package for examples of tools that enable various features by following the traits-with-metadata design pattern.

3.2.2 Using “Interfaces” in a Feature

Another useful approach that can be used when defining new feature classes is to look for application object classes that implement specific interfaces supported by the feature. The term *interface* is used somewhat loosely here to refer to either a specific class the application object inherits from, or a specific set of methods that the application object class implements. This approach can easily be combined with metadata, for example to allow the feature to discover application object trait attributes that contain instances of objects which implement a feature-specific interface.

4 Stock Features

In addition to the core **DockWindowFeature** base class, there are a number of features that have already been implemented and are included in the **enthought.pyface.dock.features** package. These features can be used freely in any DockWindows-based application, and also provide useful working examples of how to write feature classes.

These features are also used heavily by the developer oriented plug-ins provided in the **enthought.developer.tools** package. Refer to the *Enthought Developer Tools Suite* documentation for a description of these plug-ins, and to the source files contained in the **enthought.developer.tools** package for examples of how to integrate feature functionality into tools and application components.

The following sections provide in-depth descriptions of the function provided by each of these **DockWindowFeature** classes.

4.1 The Connect Feature

The Connect feature allows a user to create an information flow between two application objects by connecting data supplied by one object to the other. The connections made can be either temporary or permanent. In a temporary connection, the data is transferred just once, while in a permanent connection, once the connection is made, the application objects stay connected until the user explicitly disconnects them. The feature applies to any application object which derives from **HasTraits** and that has one or more traits with **connect** metadata.

The **connect** metadata has the format: `connect =` *value*, where *value* is a string with one of the following forms:

```
{to|from|both}[:name]
{to|from|both}[::type]
```

In both these forms, one of the strings `to`, `from` or `both` must be specified and may optionally be followed by colon and a name or a double colon and a type.

The `to`, `from` and `both` strings specify the allowed direction of data transfer when another application object is connected to the object's trait attribute:

- `to`: Data may be copied from the other application object to this trait attribute.
- `from`: Data may be copied from this trait attribute to the other application object.
- `both`: Data may be copied from either this trait attribute or the other application object to the other.

If specified, the *name* string provides text that will appear in the Connect feature's shortcut menu to describe the trait being connected (e.g., `current file`). If not specified, a generic term to describe the connection is used instead.

The *type* string, if specified, performs the same function but, in addition, requires that both ends of the connection provide the same type string in order for a connection to be made between them (e.g., `employee information`). Note that the *type* string does not need to correspond to an actual Python type. It ensures that both ends of a connection use the data for the same logical purpose.

The following are all legal values for **connect** metadata:

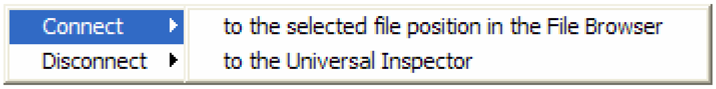
```
to
from
both
to:file name
from::employee information
```

The user interface for the Connect feature is one of the following icons, which appear on the tab associated with application object:

- ✗🔗: The application object is not connected to any other object.
- 🔗: The application object is connected to at least one other object.

The user can interact with the Connect feature as follows:

- **Drag**: Drag the Connect feature icon from one application object and drop it on the Connect feature icon for another application object. If the connection can be made, the user sees the valid drop target cursor while holding the pointer over the target icon. If no connection can be made, or there is more than one possible connection, the invalid drop target cursor is displayed. If the connection is valid, the feature automatically connects the two application objects when the user releases the mouse button over the drop target.
- **Left click**: Left clicking the Connect feature icon displays a shortcut menu showing the valid connections that can be made to other application objects, and the current connections, if any, that can be disconnected. Selecting a menu option will make or break the specified connection. An example of a typical shortcut menu is shown below:


Connect ▸
Disconnect ▸
to the selected file position in the File Browser
to the Universal Inspector

Internally, when a connection is made between two application object trait attributes, the feature creates a trait change notification listener that is called when the *from* object trait attribute changes value. When called, the listener attempts to assign the new *from* trait value to the *to* trait attribute. If an exception occurs, the error is ignored.

Note that before a connection is made, the Connect feature always tries to ensure that the *to* trait is type-compatible with the *from* trait by validating that the current value of the *from* trait is a valid value for the *to* trait. This does not guarantee that all future *from* trait values will be valid for the *to* trait, but it does prevent many incompatible assignments from being made or listed in the shortcut menu.

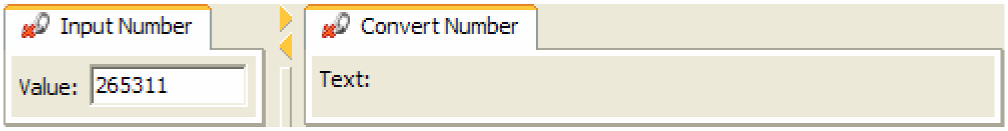
Any connection made by the user is immediately persisted so that in future application sessions, the connection can be automatically restored as soon as both ends of the connection have been loaded. Similarly, when the user disconnects an existing connection, the connection is immediately removed from set of persisted connections, so that the connection is not automatically restored in future application sessions.

Connecting two application objects also causes them to be treated as if they form a new compound tool or application. Specifically, this means that if each object is contained in a separated notebook within the containing DockWindow, clicking the tab of one application object to activate it automatically activates the tab of the other object.

The following code provides a simple example of two application object classes that can be connected together. The first class, **InputNumber**, contains a **value** trait attribute that the user can type an integer value into. The **value** trait attribute allows connections as a *from* trait. The second class, **ConvertNumber**, displays the textual representation of an integer value. Its **number** trait attribute contains the integer to be converted and allows connections as a *to* trait:

```
from enthought.traits.api \
    import HasTraits, Int, Str, View, Item

# Define the InputNumber class:
class InputNumber ( HasTraits ):

    # Define the input value:
    value = Int( connect = 'from:the integer value' )

    # Define the application object view:
    traits_view = View( 'value' )


# Define the ConvertNumber class:
class ConvertNumber ( HasTraits ):

    # Define the number and converted value traits:
    number = Int( connect = 'to:the integer value' )
    text   = Str

    # Define the application object view:
    traits_view = View( Item( 'text', style='readonly' ) )

    # Handle the 'number' trait being changed:
    def _number_changed ( self, number ):
        self.text = self._convert( number )

    # Convert an integer to its text representation:
    def _convert ( self, n ):
        if n == 0:
            return 'zero'

        result = ''
        if n < 0:
            result = 'minus '
            n = -n
        if n >= 1000000000:
            result += \
                self._convert( n/1000000000 ) + ' billion '
            n %= 1000000000
        if n >= 1000000:
            result += \
                self._convert( n / 1000000 ) + ' million '
            n %= 1000000
        if n >= 1000:
            result += \
                self._convert( n / 1000 ) + ' thousand '
            n %= 1000
        if n >= 100:
            result += \
                self._convert( n / 100 ) + ' hundred '
            n %= 100
        if n >= 20:
            result += tens[ (n / 10) - 2 ]
            n %= 10
            if n > 0:
                result += '-'
        if n > 0:
            result += digits[ n ]
        return result.strip()

# List of multiples of ten names:
tens = [ 'twenty', 'thirty', 'forty', 'fifty',
         'sixty', 'seventy', 'eighty', 'ninety' ]

# List of digit names:
digits = [ 'zero', 'one', 'two', 'three', 'four', 'five',
           'six', 'seven', 'eight', 'nine', 'ten',
           'eleven', 'twelve', 'thirteen', 'fourteen',
           'fifteeen', 'sixteen', 'seventeen', 'eighteen',
           'nineteen' ]

# Create importable application objects:
input_number   = InputNumber()
convert_number = ConvertNumber()
```

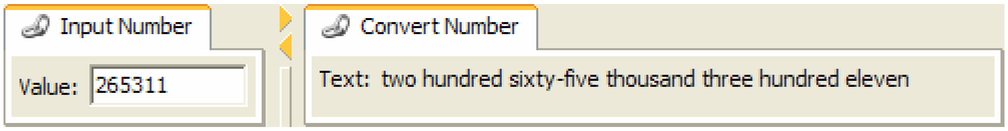
The following screen shot shows the two application object views before being connected together:



Value: 265311

Text:

The next screen shot shows the same two views after connecting them by dragging the icon from the **Input Number** application object to the same icon on the **Convert Number** object:



Value: 265311

Text: two hundred sixty-five thousand three hundred eleven

4.2 The Drag and Drop Feature

The Drag and Drop feature provides a mechanism that allows a user to drag and drop objects to or from an associated application object. The receiver of the dragged object can then take whatever action is appropriate both for the receiver and the object received. The feature applies to any application object that derives from **HasTraits** and which has a least one trait with **draggable** or **droppable** metadata.

4.2.1 Dragging Traits

An application object trait is draggable if it has `draggable =` *value* metadata set on it. The *value* can be any value other than **None**, but normally it is a string that is used as a tooltip to describe the object that will be dragged (e.g., `draggable = 'Drag this document'`). The dragged object is the current value of the trait attribute at the time the user initiates the drag operation by clicking and dragging the Drag and Drop feature icon.

Note that it is valid for an object to have more than one trait with **draggable** metadata. In this case, when the user begins a drag operation, the feature makes a list of the values of all of the traits whose **draggable** metadata is not **None**, and determines the final dragged value based on the length of the resulting list:

- If the length is 0, the application object is not draggable.
- If the length is 1, the single value in the list becomes the dragged value.
- If the length is greater than 1, the dragged value is an **enthought.pyface.dock.features.api.MultiDragDrop** instance that contains each of the draggable items in the list. The **MultiDragDrop** object is discussed further in the section on dropping objects.

4.2.2 Dropping Traits

Similarly, an application object trait is droppable if it has `droppable =` *value* metadata set on it. Again, the *value* can be anything other than **None** and normally it is a string describing what type of object can be dropped (e.g., `droppable = 'Drop a document here'`).

The action taken when an object is dropped on the application object's Drag and Drop feature icon depends upon the type of the object dropped:

- If the object is an **enthought.pyface.dock.features.api.MultiDragDrop** instance, then the feature attempts to assign each draggable object contained in the **MultiDragDrop** object to each droppable trait in the application object.
- If the object is not a **MultiDragDrop** instance, the feature attempts to assign the object to each droppable trait in the application object.

Any error caused by attempting to assign an incompatible value to a trait attribute is ignored. Note that the feature allow a drop operation to occur only if at least one of the draggable values is compatible with at least one of the application object's droppable traits.

It is perfectly valid to set both **draggable** and **droppable** metadata on the same trait. In this case, obviously, the trait is both draggable and droppable.

4.2.3 Feature Icon

The Drag and Drop feature icon that is displayed depend on the combination of **draggable** and **droppable** metadata present in the application object:

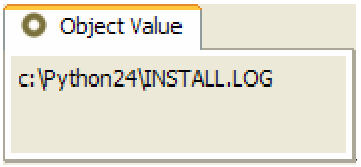
- ◉: The object contains both **draggable** and **droppable** metadata.
- ●: The object only contains **draggable** metadata.
- ○: The object only contains **droppable** metadata.

4.2.4 Example

The following is a simple example of an application object that has a droppable trait. The trait accepts any type of data and the application object simply displays the string value of the object that was dropped on it most recently:

```
from enthought.traits.api \
    import HasTraits, Any, Str, View, Item

class DroppedObjectValue ( HasTraits ):

    # Holds the most recently dropped object:
    object = Any( droppable='Drop any object here to see '
                            'its value' )

    # Contains string value of the most recently dropped
    # object:
    value=Str('Drop an object on me to display its value')

    traits_view = View(
        Item( 'value',
              style      = 'readonly',
              show_label = False
        )
    )

    # Handle a new object being dropped:
    def _object_changed ( self, object ):
        self.value = str( object )

# Create an importable application object:
dropped_object_value = DroppedObjectValue()
```

The following is a screen shot of the application object after having dragged a file name from a file explorer-type view and dropping it on the Drag and Drop feature icon:



c:\Python24\INSTALL.LOG

4.3 The Drop File Feature

The Drop File feature is similar to the Drag and Drop feature, but it is specialized to handle the case of dragging and dropping files and file-related information. The feature applies to any application object that derives from **HasTraits** and has exactly one trait with **drop_file** metadata.

The format of the **drop_file** metadata is: `drop_file =` *value*, where *value* can be:

- **True**: Indicates that the trait accepts any type of file.
- A string: Indicates that the trait accepts any type of file whose file extension is the specified string (e.g., `'py'`).
- A list or tuple of strings: Indicates that the trait accepts any type of file whose file extension matches one of the strings in the specified list or tuple (e.g., `['py', 'pyc', 'pyo' ]`).
- An **enthought.pyface.dock.features.api.DropFile** object: Indicates that the trait accepts any type of file which matches the files described by the **DropFile** object. See the “DropFile Objects” section for details.

The trait containing the **drop_file** metadata can be declared to accept any of the following types of data:

- **Str** or **File**: Accepts a single string file name.
- **List(Str)** or **List(File)**: Accepts a list of string file names.
- **Instance(FilePosition)**: Accepts a single **FilePosition** object, which is an extended file description including optional line number, column, and range information. See the “FilePosition Objects” section for details.
- **List(FilePosition)**: Accepts a list of **FilePosition** objects.

The Drop File feature automatically converts any type of file name or **FilePosition** data dropped on it to a format acceptable to the associated application object trait as long as all dropped files match the file extension filter, if any, specified in the **drop_file** metadata.

For example, if the trait is declared to be a **Str** that accepts only a single file name, and the user drops a list of **FilePosition** objects on the Drop File feature icon, the feature extracts the file names from each **FilePosition** object and assigns them, one by one, to the application object trait. While this might not seem intuitively correct, in practice it often works quite well, because the application object usually has a trait change event handler associated with the trait that fires each time a new file name is assigned to it, thus allowing it to process each assigned file.

Similarly, if the trait is defined to be a list of **FilePosition** objects, and the user drops a single file name on the application object's Drop File feature icon, the feature automatically creates a list containing a single **FilePosition** object referencing the dropped file name, and assigns the list to the application object trait.

4.3.1 DropFile Objects

A **DropFile** object can be used as the value of the **drop_file** metadata for an application object trait. The **DropFile** object provides an extended description of the types of files that the feature should accept, and has the following traits:

```
# List of valid droppable file extensions:
extensions = List( Str )

# Is the trait also draggable?
draggable = false

# The tooltip to use for the feature:
tooltip = Str
```

Setting the **DropFile draggable** trait to **True** allows the current value of the application object trait to be used as a drag source if the user attempts to drag the Drop File feature icon. If it is set to **False**, or the **drop_file** metadata is not a **DropFile** object, file dragging is not supported. The **draggable** trait also affects how the Drop File feature icon is displayed:

- 🗎➔: `draggable == True`
- 🗎: `draggable == False`, or no **DropFile** metadata.

The **DropFile tooltip** trait lets you specify a tooltip string describing the files that can be dragged or dropped. If left empty, or if the **drop_file** metadata is not a **DropFile** object, a default tooltip is created automatically by the feature and is used instead.

4.3.2 FilePosition Objects

A **FilePosition** object is an extended form of file description that includes more than just a file name. The complete set of traits defined on a **FilePosition** object is as follows:

```
# The name of the file:
file_name = File

# The logical name of the file fragment:
name = Property

# The line number within the file:
line = Int

# The number of lines within the file (starting at 'line').
# A value of -1 means the entire file:
lines = Int( -1 )

# The column number within the line:
column = Int

# An object associated with this file:
object = Any
```

The **name** trait is a property that returns either a logical name that has been assigned to it, or the *base name* (i.e., file name minus path) of the **file_name** trait if no value has been explicitly assigned to **name**.

The **line**, **lines,** and **column** traits define a range or position in the file, depending upon how they are used. For example, to define a range of lines, the **line** and **lines** trait are usually set. To definition a position, the **line** and **column**, or just the **line** trait, are set.

Finally, an arbitrary object can be associated with a **FilePosition**.

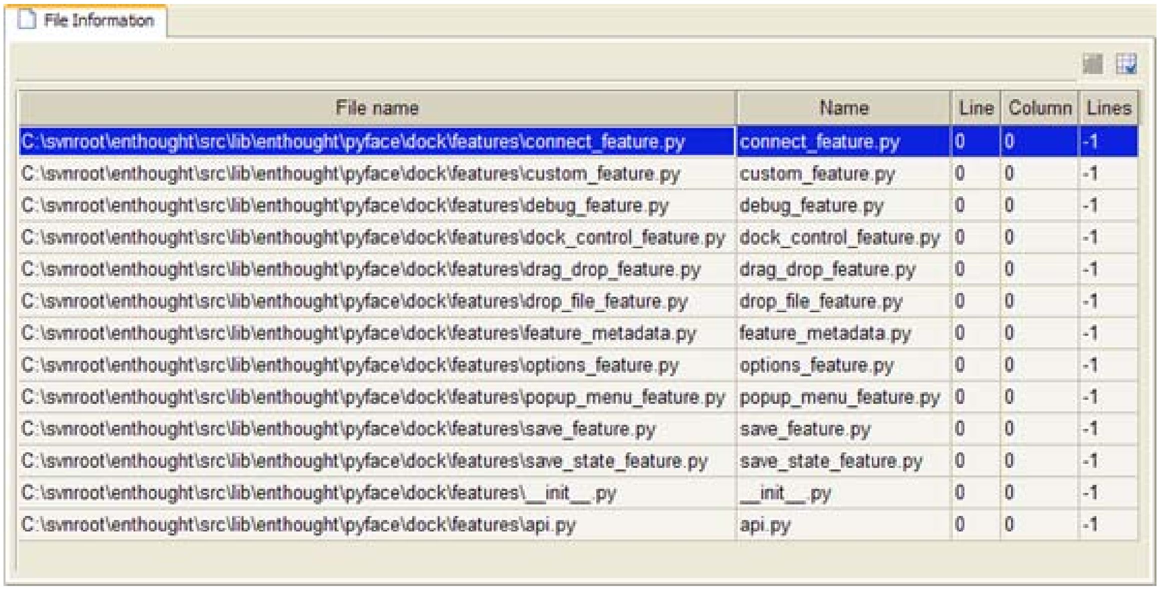
The **FilePosition** object is used by a number of the feature-based plug-ins defined in the **enthought.developer.tools** package. Refer to the source code for these plug-ins for examples of how **FilePosition** objects can be used.

4.3.3 Example

The following example shows a simple application object that uses the Drop File feature to display a tabular view of the **FilePosition** information dropped on it:

```
from enthought.traits.api \
    import HasTraits, List, View, Item, TableEditor

from enthought.traits.ui.table_column \
    import ObjectColumn

from enthought.developer.api \
    import FilePosition

# Define a read-only table column:
class ReadOnlyColumn ( ObjectColumn ):
    editable = False

# The table editor for displaying the FilePosition columns:
files_table_editor = TableEditor(
    columns = [ ReadOnlyColumn( name = 'file_name' ),
                ReadOnlyColumn( name = 'name' ),
                ReadOnlyColumn( name = 'line' ),
                ReadOnlyColumn( name = 'column' ),
                ReadOnlyColumn( name = 'lines' ) ]
)

# Define the application object class:
class FileDropper ( HasTraits ):

    # The list of files being displayed:
    files = List( FilePosition, drop_file = True )

    # The table view for displaying them:
    traits_view = View(
        Item( 'files',
              show_label = False,
              editor     = files_table_editor
        )
    )

# Create an importable application object:
file_dropper = FileDropper()
```

The following is a screen shot of the application object after selecting all of the Python source files in the **enthought.pyface.dock.features** package from Windows Explorer and dragging them onto the Drop File feature icon:



File name	Name	Line	Column	Lines
C:\svnroot\enthought\src\lib\enthought\pyface\dock\features\connect_feature.py	connect_feature.py	0	0	-1
C:\svnroot\enthought\src\lib\enthought\pyface\dock\features\custom_feature.py	custom_feature.py	0	0	-1
C:\svnroot\enthought\src\lib\enthought\pyface\dock\features\debug_feature.py	debug_feature.py	0	0	-1
C:\svnroot\enthought\src\lib\enthought\pyface\dock\features\dock_control_feature.py	dock_control_feature.py	0	0	-1
C:\svnroot\enthought\src\lib\enthought\pyface\dock\features\drag_drop_feature.py	drag_drop_feature.py	0	0	-1
C:\svnroot\enthought\src\lib\enthought\pyface\dock\features\drop_file_feature.py	drop_file_feature.py	0	0	-1
C:\svnroot\enthought\src\lib\enthought\pyface\dock\features\feature_metadata.py	feature_metadata.py	0	0	-1
C:\svnroot\enthought\src\lib\enthought\pyface\dock\features\options_feature.py	options_feature.py	0	0	-1
C:\svnroot\enthought\src\lib\enthought\pyface\dock\features\popup_menu_feature.py	popup_menu_feature.py	0	0	-1
C:\svnroot\enthought\src\lib\enthought\pyface\dock\features\save_feature.py	save_feature.py	0	0	-1
C:\svnroot\enthought\src\lib\enthought\pyface\dock\features\save_state_feature.py	save_state_feature.py	0	0	-1
C:\svnroot\enthought\src\lib\enthought\pyface\dock\features__init__.py	__init__.py	0	0	-1
C:\svnroot\enthought\src\lib\enthought\pyface\dock\features\api.py	api.py	0	0	-1

4.4 The Options Feature

The Options feature provides a simple mechanism for an application object to allow a user to set the object's options (i.e., user preferences). The feature applies to any application object that derives from **HasTraits** and has a traits view called `options`.

If the feature applies to an application object, the Options feature icon (☑) is displayed on the application object's tab or drag bar. When the user clicks the icon, the application object's `options` view is displayed so that the user can set the desired user preference options.

This feature can be combined with the Save State feature to automatically save and restore the user preference traits of an application object.

4.5 The Popup Menu Feature

The Popup Menu feature provides a simple mechanism for an application object to display a shortcut menu of actions specific to the object. The feature applies to any application object that has an attribute called **popup_menu**.

If the feature applies to an application object, the Popup Menu feature icon (☰) is displayed on the application object's tab or drag bar. When the user clicks the icon, the shortcut menu defined by the application object's **popup_menu** attribute is displayed. The value of **popup_menu** can be either an **enthought.traits.ui.menu.Menu** object or a *callable* that, when invoked with no arguments, returns a **Menu** object. The popup menu is normally displayed so that the topmost menu item lies under the position that the mouse pointer had at the time the user clicked on the Popup Menu feature icon.

4.6 The Save State Feature

The Save State feature automatically manages saving and restoring the state of its associated application object. This feature does not interact directly with the user, so it has no visible icon. The feature applies to any application object which has at least one trait with `save_state = True` metadata.

Any application object trait with `save_state = True` metadata is automatically restored (if saved data exists for it) when the feature is first applied, and is immediately saved by the feature each time the trait value is changed. All of an application object's saved trait values are stored under the ID specified by the application object trait with metadata `save_state_id = True`. This trait must be a (usually constant) string value (e.g., `enthought.developer.tools.syntax_checker.state`). If no such trait exists, a default ID is used instead that has the form: `unknown.plugins.`*dock_control*`.state`, where *dock_control* is the name of the feature's associated DockControl object.

Although it is not mandatory, specifying a **save_state_id** of the form: **package_name.module_name.***class_name***.state** is recommended, to help avoid the possibility of one application object overwriting a different application object's saved state data. Specifying the class name is necessary only if the specified module contains more than one application object class with `save_state = True` metadata.

The Save State feature can be used in conjunction with the Options feature to automatically save and restore an application object's user preferences.

4.7 The Save Feature

The Save feature provides a simple mechanism for an application object to notify the user when it has modifications that have not yet been saved, as well as providing a way for the user to request that all current changes be saved. The feature applies to any application object that inherits from the **enthought.developer.api.Saveable** interface.

If the feature applies to an application object, the feature displays its Save feature icon (💾), but only when the application object's **needs_save** trait is set to **True**. When the user clicks the icon, the Save feature calls the application object's **save()** method to save the application object's unsaved data.

It is the application object's responsibility to set the **needs_save** trait to **True** whenever the user modifies any application data that needs to be saved. It is also the application object's responsibility to set the **needs_save** trait to **False** once its **save()** method has successfully saved the modified user data.

4.8 The Debug Feature

The Debug feature is intended to help developers debug application components by providing dynamic access to various application objects while the application is running. The feature applies to any **DockWindow** component that has an associated application object (i.e., whose **dock_control.object** trait is not **None**).

This feature is never enabled by default (i.e., its **state** class variable is initialized to 2, for "disabled"). In order to use the Debug feature, the user (i.e., developer) must explicitly enable the feature.

1. Right-click any DockWindow tab or drag bar.

2. On the shortcut menu that appears, point to **Feature** and click **Debug**.

This default behavior prevents the Debug icon from cluttering the user interface until the developer needs it. After the Debug feature is enabled, the Debug icon (🐞) appears on every application component tab or drag bar that has an associated application object.

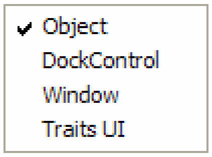
NOTE:

> If you see a tab or drag bar without the debug icon, it probably does not use **envisage.workbench.api. TraitsUIView**. You might consider to sending an e-mail message to the developer of the component and gently suggesting converting the plug-in.

By default, the Debug feature provides access to the application object associated with the application component. But it can also provide access to several other useful objects as well. The complete set of available objects is as follows:

- 🐞:Application object
- 🐞: DockControl object
- 🐞: Window object
- 🐞: Traits UI object

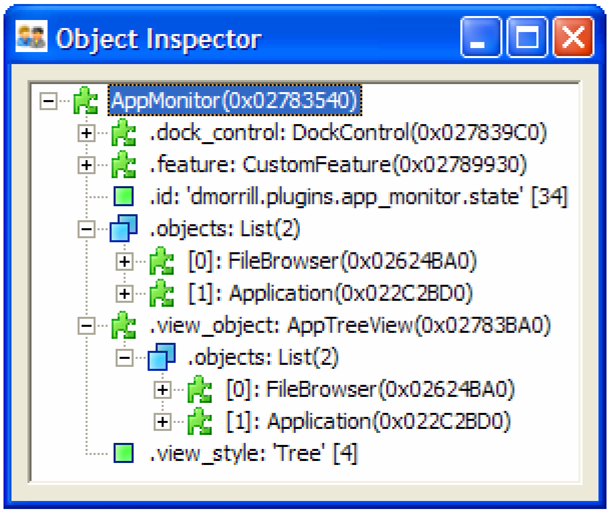
To choose the current object, right-click the Debug icon and selecting the appropriate object type on the shortcut menu:



The Debug icon immediately changes color to reflect the current object it is connected to.

You can view or interact with the selected object in several ways:

- **Left click**: Displays a window containing a tree-view of the currently selected object, like the one shown in the following figure:



```
AppMonitor(0x02783540)
    .dock_control: DockControl(0x027839C0)
    .feature: CustomFeature(0x02789930)
    .id: 'dmorrill.plugins.app_monitor.state' [34]
    .objects: List(2)
        [0]: FileBrowser(0x02624BA0)
        [1]: Application(0x022C2BD0)
    .view_object: AppTreeView(0x02783BA0)
        .objects: List(2)
            [0]: FileBrowser(0x02624BA0)
            [1]: Application(0x022C2BD0)
    .view_style: 'Tree' [4]
```

- **Right-click**: Displays the object selection menu described above.
- **Drag**: Dragging the Debug icon is the same as dragging the currently selected object. The action taken depends on the target it is dropped on.
- **Control-drag**: This is a shortcut for dragging the associated application object without first selecting it from the context menu.
- **Shift-drag**: This is a shortcut for dragging the associated **DockControl** object without first selecting it from the context menu.
- **Alt-drag**: This is a shortcut for dragging the associated window object without first selecting it from the context menu.

4.9 The DockControl Feature

The DockControl feature provides a mechanism for an application object to access to its associated **DockControl** object. The feature applies to any application object that derives from **HasTraits** and has at least one trait that has `dock_control = True` metadata.

If the feature applies to an application object, the feature attempts to set all application object trait attributes containing `dock_control = True` metadata to the **DockControl** object associated with the application object. Declare such traits to be of type **Instance(DockControl)** or **Any**. Any exception raised by attempting to assign the **DockControl** object to a trait attribute is ignored.

Because the feature does not require any user interaction, it does not have an icon associated with it.

4.10 The Custom Feature Feature

All of the stock features provide functionality that can be useful in a wide variety of application objects. But sometimes you might need to create a feature that is very specific to an application and applies to a single application component or object.

In these cases, the Custom Feature feature is very convenient to use. It allows you to embed the complete implementation of a feature's functionality directly in an application object's class definition, without having to write a separate **DockWindowFeature** subclass.

The Custom Feature feature applies to any application object that derives from **HasTraits**, that contains one or more traits with **custom_feature** metadata, and whose corresponding trait value is an **enthought.pyface.dock.features.api.CustomFeature** object, or a list of such objects. The actual value of the **custom_feature** metadata is unimportant, and can be any value as long as it is not **None** (e.g., `custom_feature = True`).

The Custom Feature feature works as follows:

1. It verifies that a new application component has an associated application object that derives from **HasTraits** and contains at least one trait containing the appropriate **custom_feature** metadata, as described above.

2. For each trait with the **custom_feature** metadata, it retrieves the corresponding trait value and verifies that it is an instance of **CustomFeature** or a list of such instances.

3. For each such **CustomFeature** object found, it creates an **ACustomFeature** object (derived from **DockWindowFeature**) that references the application object's **CustomFeature** object and delegates all of the feature protocol operations through it. This is what allows each **ACustomFeature** object to have totally unique and custom behavior.

Note that the **CustomFeature** objects that the application object provides are completely descriptive in nature. In fact, the complete definition of the **CustomFeature** class is as follows:

```
class CustomFeature ( HasPrivateTraits ):

    # The current image to display on the DockControl tab:
    tab_image = Instance( ImageResource )

    # The current (optional) image to display on the
    # DockControl drag bar:
    bar_image = Instance( ImageResource )

    # The tooltip to display when the mouse is hovering
    # over the  image:
    tooltip = Str

    # Is the feature currently enabled?
    enabled = true

    # Name of the method to invoke on a left click:
    left_click = Str

    # Name of the method to invoke on a right click:
    right_click = Str

    # Name of the method to invoke when the user starts to
    # drag:
    drag = Str

    # Name of the method to invoke when the user starts to
    # ctrl-drag:
    control_drag = Str

    # Name of the method to invoke when the user starts to
    # shift-drag:
    shift_drag = Str

    # Name of the method to invoke when the user starts to
    # alt-drag:
    alt_drag = Str

    # Name of the method to invoke when the user drops an
    # object:
    drop = Str
    # Name of the method to invoke to see if the user can
    # drop an object:
    can_drop = Str
```

The **CustomFeature** class definition closely parallels the **DockWindowFeature** class, but it substitutes method name strings for actual methods. The method names must be the names of methods defined on the application object class containing the **CustomFeature**.

The **ACustomFeature** object that is created to reference the **CustomFeature** object automatically delegates all aspects of the **DockWindowFeature** protocol to the corresponding item in its **CustomFeature** object.

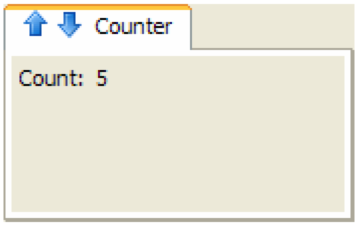
Note that it is not necessary to initialize all of the **CustomFeature** traits. For example, letting the **drag** trait default to the empty string means that the **CustomFeature** does not support dragging its feature icon.

Any of a **CustomFeature** object's traits can be modified dynamically. For instance, assigning a new **ImageResource** object to a **CustomFeature** object's **tab_image** trait automatically results in the corresponding feature icon being updated on the application object's tab.

The following code sample provides a simple example that shows how an application object class can use the Custom Feature feature to implement application specific features. In this example the **Counter** application object class defines two custom features, called **IncrementFeature** and **DecrementFeature**. Clicking the **IncrementFeature** icon (⬆) increments the object's **count** trait attribute, while clicking the **DecrementFeature** icon (⬇) decrements the count. The application object itself simply displays the current value of the **count** trait attribute.

```
from enthought.traits.api \
    import HasTraits, Int, List, View, Item

from enthought.pyface.image_resource \
    import ImageResource

from enthought.pyface.dock.features.api \
    import CustomFeature

IncrementFeature = CustomFeature(
    tab_image  = ImageResource( 'increment' ),
    tooltip    = 'Click to increment count',
    left_click = 'increment'
)

DecrementFeature = CustomFeature(
    tab_image  = ImageResource( 'decrement' ),
    tooltip    = 'Click to decrement count',
    left_click = 'decrement'
)

class Counter ( HasTraits ):

    # The current count value:
    count = Int

    # The feature definitions:
    features = List( [IncrementFeature, DecrementFeature],
                     custom_feature = True )

    # The application object view:
    traits_view = View(
        Item( 'count', style = 'readonly' )
    )

    # Methods to handle 'feature' left click actions:
    def increment ( self ):
        self.count += 1

    def decrement ( self ):
        self.count -= 1

# Create an importable instance:
counter = Counter()
```

The following screen shot shows the **Counter** application object after clicking on the **IncrementFeature** icon a few times:



Count: 5