37.6. `FrameWork` — Interactive application framework¶

Platforms: Mac

The `FrameWork` module contains classes that together provide a framework for an interactive Macintosh application. The programmer builds an application by creating subclasses that override various methods of the bases classes, thereby implementing the functionality wanted. Overriding functionality can often be done on various different levels, i.e. to handle clicks in a single dialog window in a non-standard way it is not necessary to override the complete event handling.

Note

This module has been removed in Python 3.x.

Work on the `FrameWork` has pretty much stopped, now that `PyObjC` is available for full Cocoa access from Python, and the documentation describes only the most important functionality, and not in the most logical manner at that. Examine the source or the examples for more details. The following are some comments posted on the MacPython newsgroup about the strengths and limitations of `FrameWork`:

> The strong point of `FrameWork` is that it allows you to break into the control-flow at many different places. `W`, for instance, uses a different way to enable/disable menus and that plugs right in leaving the rest intact. The weak points of `FrameWork` are that it has no abstract command interface (but that shouldn't be difficult), that its dialog support is minimal and that its control/toolbar support is non-existent.

The `FrameWork` module defines the following functions:

`FrameWork.Application()`¶
An object representing the complete application. See below for a description of the methods. The default `__init__()` routine creates an empty window dictionary and a menu bar with an apple menu.

`FrameWork.MenuBar()`¶
An object representing the menubar. This object is usually not created by the user.

`FrameWork.Menu`(*bar*, *title*[, *after*])¶
An object representing a menu. Upon creation you pass the `MenuBar` the menu appears in, the *title* string and a position (1-based) *after* where the menu should appear (default: at the end).

`FrameWork.MenuItem`(*menu*, *title*[, *shortcut*, *callback*])¶

Create a menu item object. The arguments are the menu to create, the item title string and optionally the keyboard shortcut and a callback routine. The callback is called with the arguments menu-id, item number within menu (1-based), current front window and the event record.

Instead of a callable object the callback can also be a string. In this case menu selection causes the lookup of a method in the topmost window and the application. The method name is the callback string with `'domenu_'` prepended.

Calling the `MenuBar` `fixmenudimstate()` method sets the correct dimming for all menu items based on the current front window.

`FrameWork.Separator`(*menu*)¶
Add a separator to the end of a menu.

`FrameWork.SubMenu`(*menu*, *label*)¶
Create a submenu named *label* under menu *menu*. The menu object is returned.

`FrameWork.Window`(*parent*)¶
Creates a (modeless) window. *Parent* is the application object to which the window belongs. The window is not displayed until later.

`FrameWork.DialogWindow`(*parent*)¶
Creates a modeless dialog window.

`FrameWork.windowbounds`(*width*, *height*)¶
Return a `(left, top, right, bottom)` tuple suitable for creation of a window of given width and height. The window will be staggered with respect to previous windows, and an attempt is made to keep the whole window on-screen. However, the window will however always be the exact size given, so parts may be offscreen.

`FrameWork.setwatchcursor()`¶
Set the mouse cursor to a watch.

`FrameWork.setarrowcursor()`¶
Set the mouse cursor to an arrow.

37.6.1. Application Objects¶

Application objects have the following methods, among others:

`Application.makeusermenus()`¶
Override this method if you need menus in your application. Append the menus to the attribute `menubar`.

`Application.getabouttext()`¶
Override this method to return a text string describing your application. Alternatively, override the `do_about()` method for more elaborate “about” messages.

`Application.mainloop`([*mask*[, *wait*]])¶

This routine is the main event loop, call it to set your application rolling. *Mask* is the mask of events you want to handle, *wait* is the number of ticks you want to leave to other concurrent application (default 0, which is probably not a good idea). While raising *self* to exit the mainloop is still supported it is not recommended: call `self._quit()` instead.

The event loop is split into many small parts, each of which can be overridden. The default methods take care of dispatching events to windows and dialogs, handling drags and resizes, Apple Events, events for non-FrameWork windows, etc.

In general, all event handlers should return `1` if the event is fully handled and `0` otherwise (because the front window was not a FrameWork window, for instance). This is needed so that update events and such can be passed on to other windows like the Sioux console window. Calling `MacOS.HandleEvent()` is not allowed within *our_dispatch* or its callees, since this may result in an infinite loop if the code is called through the Python inner-loop event handler.

`Application.asyncevents`(*onoff*)¶

Call this method with a nonzero parameter to enable asynchronous event handling. This will tell the inner interpreter loop to call the application event handler *async_dispatch* whenever events are available. This will cause FrameWork window updates and the user interface to remain working during long computations, but will slow the interpreter down and may cause surprising results in non-reentrant code (such as FrameWork itself). By default *async_dispatch* will immediately call *our_dispatch* but you may override this to handle only certain events asynchronously. Events you do not handle will be passed to Sioux and such.

The old on/off value is returned.

`Application._quit()`¶
Terminate the running `mainloop()` call at the next convenient moment.

`Application.do_char`(*c*, *event*)¶
The user typed character *c*. The complete details of the event can be found in the *event* structure. This method can also be provided in a `Window` object, which overrides the application-wide handler if the window is frontmost.

`Application.do_dialogevent`(*event*)¶
Called early in the event loop to handle modeless dialog events. The default method simply dispatches the event to the relevant dialog (not through the `DialogWindow` object involved). Override if you need special handling of dialog events (keyboard shortcuts, etc).

`Application.idle`(*event*)¶
Called by the main event loop when no events are available. The null-event is passed (so you can look at mouse position, etc).

37.6.2. Window Objects¶

Window objects have the following methods, among others:

`Window.open()`¶
Override this method to open a window. Store the Mac OS window-id in `self.wid` and call the `do_postopen()` method to register the window with the parent application.

`Window.close()`¶
Override this method to do any special processing on window close. Call the `do_postclose()` method to cleanup the parent state.

`Window.do_postresize`(*width*, *height*, *macoswindowid*)¶
Called after the window is resized. Override if more needs to be done than calling `InvalRect`.

`Window.do_contentclick`(*local*, *modifiers*, *event*)¶
The user clicked in the content part of a window. The arguments are the coordinates (window-relative), the key modifiers and the raw event.

`Window.do_update`(*macoswindowid*, *event*)¶
An update event for the window was received. Redraw the window.

`Window.do_activate`(*activate*, *event*)¶
The window was activated (`activate == 1`) or deactivated (`activate == 0`). Handle things like focus highlighting, etc.

37.6.3. ControlsWindow Object¶

ControlsWindow objects have the following methods besides those of `Window` objects:

`ControlsWindow.do_controlhit`(*window*, *control*, *pcode*, *event*)¶
Part *pcode* of control *control* was hit by the user. Tracking and such has already been taken care of.

37.6.4. ScrolledWindow Object¶

ScrolledWindow objects are ControlsWindow objects with the following extra methods:

`ScrolledWindow.scrollbars`([*wantx*[, *wanty*]])¶
Create (or destroy) horizontal and vertical scrollbars. The arguments specify which you want (default: both). The scrollbars always have minimum `0` and maximum `32767`.

`ScrolledWindow.getscrollbarvalues`()¶
You must supply this method. It should return a tuple `(x, y)` giving the current position of the scrollbars (between `0` and `32767`). You can return `None` for either to indicate the whole document is visible in that direction.

`ScrolledWindow.updatescrollbars`()¶
Call this method when the document has changed. It will call `getscrollbarvalues()` and update the scrollbars.

`ScrolledWindow.scrollbar_callback`(*which*, *what*, *value*)¶
Supplied by you and called after user interaction. *which* will be `'x'` or `'y'`, *what* will be `'-'`, `'--'`, `'set'`, `'++'` or `'+'`. For `'set'`, *value* will contain the new scrollbar position.

`ScrolledWindow.scalebarvalues`(*absmin*, *absmax*, *curmin*, *curmax*)¶
Auxiliary method to help you calculate values to return from `getscrollbarvalues()`. You pass document minimum and maximum value and topmost (leftmost) and bottommost (rightmost) visible values and it returns the correct number or `None`.

`ScrolledWindow.do_activate`(*onoff*, *event*)¶
Takes care of dimming/highlighting scrollbars when a window becomes frontmost. If you override this method, call this one at the end of your method.

`ScrolledWindow.do_postresize`(*width*, *height*, *window*)¶
Moves scrollbars to the correct position. Call this method initially if you override it.

`ScrolledWindow.do_controlhit`(*window*, *control*, *pcode*, *event*)¶
Handles scrollbar interaction. If you override it call this method first, a nonzero return value indicates the hit was in the scrollbars and has been handled.

37.6.5. DialogWindow Objects¶

DialogWindow objects have the following methods besides those of `Window` objects:

`DialogWindow.open`(*resid*)¶
Create the dialog window, from the DLOG resource with id *resid*. The dialog object is stored in `self.wid`.

`DialogWindow.do_itemhit`(*item*, *event*)¶
Item number *item* was hit. You are responsible for redrawing toggle buttons, etc.

Table Of Contents

Previous topic

37.5. `EasyDialogs` — Basic Macintosh dialogs

Next topic

37.7. `autoGIL` — Global Interpreter Lock handling in event loops

This Page

- Show Source

Navigation

- index
- modules |
- next |
- previous |
- Python v2.6.4 documentation »
- The Python Standard Library »
- 37. Mac OS X specific services »