

How-Tos

Wing IDE Personal

Wingware
www.wingware.com

Version 2.1.4
February 9, 2007

This is a collection of HOW-TOs designed to make it easier to get started using Wing with specific tools or in advanced development tasks.

Contents

Wing IDE Quick Start Guide

- Install Python and Wing IDE
- Set up a Project
- Key Features
- Related Documents

Using Wing IDE with wxPython

- Installation and Configuration
- Test Driving the Debugger
- Test Driving the Source Browser
- Using a GUI Builder
- Related Documents

Using Wing IDE with PyGTK

- Installation and Configuration
- Auto-completion and Source Assistant
- Using a GUI Builder
- Details and Notes
- Related Documents

Using Wing IDE with PyQt

- Installation and Configuration
- Test Driving the Debugger
- Test Driving the Source Browser
- Using a GUI Builder
- Tips for Keeping the Debug Process Responsive
- Related Documents

Using Wing IDE with Zope

- Quick Start on a Single Host

- Starting the Debugger
- Test Drive Wing IDE
- Setting Up Auto-Refresh
- Setting up Remote Debugging
- Trouble Shooting Guide
- Related Documents

Using Wing IDE with Plone

- Performance Hints
- Related Documents

Using Wing IDE with Webware

- Introduction
- Setting up a Project
- Starting Debug
- Related Documents

Using Wing IDE with mod_python

- Quick Start
- Example
- Related Documents

Debugging Web CGIs with Wing IDE

Wing IDE for OS X

- Usage Tips
- Changing Display Themes
- Finding WINGHOME
- Mouse Buttons
- Window Focus
- Known Problems
- Related Documents

Using Wing IDE with pygame

- Debugging pygame
- Related Documents

[Using Wing IDE with scons](#)

[Debugging scons](#)

[Related Documents](#)

[Handling Large Values and Strings in the Debugger](#)

[Debugging C/C++ and Python together](#)

[Debugging Extension Modules on Linux/Unix](#)

[Debugging Code with XGrab* Calls](#)

[Debugging Non-Python Mainloops](#)

[Supported Non-Python Mainloops](#)

[Working with Non-Python Mainloops](#)

[Non-Python Mainloop Internals](#)

[Supporting Non-Python Mainloops](#)

[Writing Non-Python Mainloop Support](#)

[Debugging Code Running Under Py2exe](#)

Wing IDE Quick Start Guide

This is a minimalist guide for those wanting to get started with Wing IDE as quickly as possible. For a more in-depth introduction, try the **Tutorial**.

Also available: Quick start guides specifically for **OS X installation**, **Zope**, **Plone**, **wxPython**, **PyGTK** **PyQt**, **Debugging Web CGIs**, **mod_python**, and **PyGame**.

We welcome feedback and bug reports, both of which can be submitted directly from Wing IDE using the Submit Feedback and Submit Bug Report items in the Help menu, or by emailing us at [support at wingware.com](mailto:support@wingware.com).

Install Python and Wing IDE

Both Python and Wing IDE must be installed. The Wing IDE executable is called `wing-personal12.1`. See **Installing, Running the IDE**, and **Installing your License** for details.

Set up a Project

Wing starts up initially with a blank Default Project. To get the most out of Wing, you must set up your project as follows:

- Use **Add Directory Tree** and other items in the **Project** menu to add source files to the project.
- Use **Project Properties** in the **Project** menu to specify `PYTHONPATH` and select the specific version of Python for use with your project. These two steps tell Wing how to resolve imports in your code, so it can discover and analyze your source base. This powers Wing's source browser, auto-completer, source assistant, and code navigation features.

- Save your project to disk.

Note: Depending on the size of the code base you have added to your project, Wing may consume considerable CPU time for up to several minutes analyzing your source code. Once this is done, Wing should run with a snappy and responsive interface even on slower machines.

See **Debug Properties** and **Source Code Analysis** for details.

Key Features

You are now ready to start coding and debugging. Most of Wing's features are readily evident from the user interface. The **Wing Tips** tool provides useful usage hints.

These are the features you should be sure to try while evaluating Wing IDE:

- *Customizable User Interface* -- Many options are available from **Preferences**, and you can split tools panels and move around the tools within them. Right click on the notebook tabs for options, or use the **Windows** menu to create tools in separate windows. Your configuration is remembered in your project.
- *Configurable Key Bindings* -- Wing can emulate Visual Studio, VI/Vim, Emacs, and Brief key bindings, selected with the editor **Personality** preference.
- *Auto-completion and Source Assistant* -- Wing's editor and the **Source Assistant** tool (in Wing Pro only) provide context-appropriate completion options and documentation as you edit your code.
- *Goto-definition* -- Available from the toolbar, **Source** menu, and by right-clicking on symbols in the editor.
- *Source Index* -- Quick access to other parts of a source file from the menus at the top of the source editor.
- *Mini-search* -- Wing's powerful keyboard-driven search and replace facility is available from the **Edit** menu. Using the keyboard equivalents given in the menu, bring up the search entry area at the bottom of the screen, type in a search string, then repeat the key bindings for repeated search forward/backward.
- *Search Managers* -- Provide single and multi-file, wild card, and regular expression search and replace. Available as **Search** and **Search in Files** in the tools area.
- *Python Shell* -- This **Python command prompt** lets you try out code in a sandbox process kept isolated from Wing IDE and your debug process.

- *Basic Debugging* -- Set a breakpoint and start debugging. You can use the **Stack Data** tools to inspect or change program data. Wing distinguishes between fatal and non-fatal exceptions at the time they are raised, allowing you to inspect live program state more often. Debug process I/O is shown in the **integrated I/O** tool (or optionally in an external console).

Explicit markup ends without a blank line; unexpected unindent.

Related Documents

Wing IDE provides many other options and tools. For more information:

- **Wing IDE Tutorial**, a detailed guided tour for Wing IDE.
- **Wing IDE Reference Manual**, which describes Wing IDE in detail.
- **OS X Quickstart**
- **Zope Quickstart**
- **Plone Quickstart**
- **wxPython Quickstart**
- **PyQt Quickstart**
- **mod_python Quickstart**

Using Wing IDE with wxPython

[Wing IDE](#) is an integrated development environment for the Python programming language. Wing can be used to speed up the process of writing and debugging code that is written for the powerful [wxPython](#) cross-platform GUI development toolkit.

wxPython is a good choice for GUI developers. It is currently available for MS Windows, Linux, Unix, and Mac OS X and provides native look and feel on each of these platforms.

While Wing IDE does not currently provide a GUI builder for wxPython, it does provide the most advanced debugger and source browser capabilities available for the Python programming language and it can be used with other available GUI builders, as described below.

Installation and Configuration

Take the following steps to set up and configure Wing IDE for use with wxPython:

- Install Python and Wing. You will need a specific version of Python depending on the version of wxPython you plan to use. Check the wxPython [Getting Started Wiki](#) when in doubt. See the generic **Wing IDE Quickstart Guide** for installation instructions.
- Install wxPython. See the wxPython's website [Getting Started Wiki](#) for installation instructions. Note that you need to install the version of wxPython to match your Python version, as indicated on the [download page](#).
- Start Wing from the Start menu on Windows or by typing "wing" on the command line on Linux, OS X, or other Posix systems.
- Select Show Analysis Stats from the Source menu and if the Python version reported there doesn't match the one you're using with wxPython, then select Project Properties from the Project menu and use the Python Executable field to select the correct Python version.

- Open the wxPython demo into Wing IDE. This is located within your Python installation at `Lib/site-packages/wx/demo/demo.py` or at `c:\Program Files\wxPython2.6 Docs and Demos\demo` or similar location. Then select Add Current File from the Project menu.
- Set `demo.py` as main entry point for debugging by selecting Set Current As Main Debug File from the Debug menu.
- Save your project to disk. Use a name ending in `.wpr`.

Test Driving the Debugger

Now you're ready to try out the debugger. To do this:

Start debugging with the Debug / Continue item in the Debug menu. Uncheck the **Show this dialog before each run** checkbox at the bottom of the dialog that appears and select OK.

The demo application will start up. If its main window doesn't come to front, bring it to front from your task bar or window manager. As you try out the various demos from the tree on the left of the wxPython demo app, you will sometimes see Wing IDE report exceptions in the debugger's **Exceptions** tool. These are not program errors or a malfunction of the IDE but are caused by Wing's proactive exception detection algorithm, which cannot see into the C and C++ code that is handling these exceptions outside of the debugger.

To get past them, select "Ignore this exception location" in the **Exceptions** tool and continue execution. You will see 3-4 of these the first time Wing IDE encounters them. After that, your ignore list is stored in the project so you will never see them again, even in future debug sessions. Subsequently, you will benefit from Wing's ability to stop immediately at the point of exception, rather than after the fact. This makes understanding the conditions leading to an error much easier and speeds up debug turn-around.

In wxPython 2.3.4.2 for Python 2.2, examples of these exceptions can be seen by bringing up More Dialogs / ImageBrowser, More Dialogs / wxMultipleChoiceDialog, and New since last release / Throbber.

Important: In wxPython 2.6 a change to the demo code breaks all debuggers by not setting the `co_filename` attribute on code objects correctly. To fix this, change the line that reads `description = self.modules[modID][2]` around line 804 in `demo\main.py` to instead read `description = self.modules[modID][3]` -- Wing will not stop at breakpoints until this is done.

Next open `Lib/site-packages/wx/demo/ImageBrowser.py` into Wing IDE. Set a breakpoint on the first line of `runTest()` by clicking on the dark grey left margin. Go into the running demo app and select `More Dialogs / ImageBrowser`. Wing will stop on your breakpoint.

Select **Stack Data** from the Tools menu. Look around the stack in the popup at the top of the window and the locals and globals shown below that for the selected stack frame. You may see some sluggishness (a few seconds) in displaying values because of the widespread use of `from wx import *` in wxPython code, which imports a huge number of symbols into the globals name space. This depends on the speed of your machine.

Select **Debug Probe** from the Tools menu. This is an interactive command prompt that lets you type expressions or even change values in the context of the stack frame that is selected on the Debugger window when your program is paused or stopped at an exception. It is a very powerful debugging tool.

Also take a look at these tools available from the Tools menu:

- **I/O** -- displays debug process output and processes keyboard input to the debug process, if any
- **Exceptions** -- displays exceptions that occur in the debug process
- **Modules** -- browses data for all modules in `sys.modules`
- **Watch** -- watches values selected from other value views (by right-clicking and selecting one of the **Watch** items) and allows entering expressions to evaluate in the current stack frame

Test Driving the Source Browser

Don't forget to check out Wing's powerful source browser:

- Add package `Lib/site-packages/wx` (inside your Python installation) to your project file with the **Add Package** item in the Project menu.
- After doing so, Wing will consume 100% of your CPU for 20 seconds or more, depending on the speed of your machine. As it does this, you can already bring up the Source Browser from the Tools menu. Just be patient if things are a bit sluggish at first; there is an awful lot of Python code that Wing needs to analyse. Once the initial analysis is done, Wing will return to being responsive since the results are cached (a similar but shorter effect is seen when Wing is restarted, as it reads the analysis disk cache).

- Select **All Classes** mode at the top of the source browser. This is generally the best view to use for wxPython. On slower machines, the By Module view can also be sluggish as a result of the large number of symbols found at the module level because of the use of `from wx import *` in wxPython source. If you do use the By Module view, it helps to uncheck the **Inherited** filter checkbox.
- Use the right-click menu to zoom to base classes. In general, right-clicking will bring up menus specific to the tool being clicked on.
- Related to the Source Browser is the auto-completion capability in Wing's source editor. Try typing in one of the wxPython source files and you will see the auto-completer appear. Tab completes the currently selected item, but you can set the **Auto-complete on Enter** preference to also complete when the Enter key is pressed. See the **Wing IDE Quickstart Guide** for information on other commonly used preferences. **Note:** Depending on the speed of your machine, the auto-completer can be sluggish at first, once again due to the large number of symbols imported into most wxPython files with `from wx import *`. However, this will only happen once per Wing IDE session.
- See also the **Source Assistant** tool in the Tools menu. This provides additional information about source constructs in the active source editor as the insertion cursor or selection is moved around.

Using a GUI Builder

Wing IDE doesn't currently include a GUI builder for wxPython but it can be used with other tools, such as [Boa Constructor](#), which does provide a GUI builder but doesn't have the raw power of Wing IDE's debugger and source browser.

To use an external GUI builder, **configure Wing to automatically reload files** that are altered by the GUI builder.

Then you can run Wing IDE and your GUI builder at the same time, working with both in an almost seamless manner.

A Caveat: Because Python lends itself so well to writing data-driven code, you may want to reconsider using a GUI builder for some tasks. In many cases, Python's introspection features make it possible to write generic GUI code that you can use to build user interfaces on the fly based on models of your data and your application. This can be much more efficient than using a GUI builder to craft individual menus and dialogs by hand. In general hand-coded GUIs also tend to be more maintainable.

Related Documents

Wing IDE provides many other options and tools. For more information:

- **Wing IDE Reference Manual**, which describes Wing IDE in detail.
- [wxPython Getting Started page](#), which contains much additional information for wxPython programmers.
- **Wing IDE Quickstart Guide** which contains additional basic information about getting started with Wing IDE.

Using Wing IDE with PyGTK

[Wing IDE](#) is an integrated development environment for the Python programming language. Wing can be used to speed up the process of writing and debugging code that is written for [PyGTK](#) and [GTK+](#), a mature open source GUI development toolkit.

PyGTK is currently available for Linux/Unix, MS Windows, and Mac OS X (requires X11 Server). Like **PyQt** and unlike **wxPython**, PyGTK runs on the same (GTK-provided) widget implementations on all platforms. Themes can be used to approximate the look and behavior of widgets on the native OS. It is also possible to display native dialogs like the Windows file and print dialogs along side GTK windows. While PyGTK does not offer perfect native look and feel, it provides excellent write-once-works-anywhere capability even in very complex GUIs. Wing IDE is itself written using PyGTK.

Other advantages of PyGTK include: (1) high quality anti-aliased text rendering, (2) powerful signal-based architecture that, among other things, allows subclassing C classes in Python, (3) multi-font text widget with embeddable sub-widgets, (4) model-view architecture for list and tree widgets, and (5) a rich collection of widgets and stock icons.

While Wing IDE does not currently provide a GUI builder for PyGTK, it does provide the most advanced debugger and source browser capabilities available for the Python programming language and it can be used with other available GUI builders, as described below.

Installation and Configuration

Take the following steps to set up and configure Wing IDE for use with PyGTK:

- Install Python and Wing. See the generic **Wing IDE Quickstart Guide** for installation instructions.
- Install GTK and PyGTK. If you are on Linux, you may already have one or both

installed, or you may be able to install them using your distribution's package manager. Otherwise, check out the [gtk website](#) and [pygtk website](#).

- Start Wing from the Start menu on Windows or by typing “wing” on the command line on Linux, OS X, or other Posix systems.
- Select Show Analysis Stats from the Source menu and if the Python version reported there doesn't match the one you're using with PyGTK, then select Project Properties from the Project menu and use the Python Executable field to select the correct Python version.
- Add some files to your project, and set the main entry point using the Set Main Debug File item in the Debug menu. Save the project file to disk.
- You should now be able to debug your PyGTK application from within Wing. If you see ImportError on the PyGTK modules, you will need to add Python Path information in the Debug tab of Project Properties, accessed from the Project menu.

Auto-completion and Source Assistant

To obtain auto-completion options and call signature information in Wing's Source Assistant, you will need to run a script that converts from PyGTK's defs files into Python interface files that Wing's source analyser can read.

- Download the [pygtk_to_pi.py](#) script and the [PyGTK sources](#) for your version of PyGTK if you don't already have them.
- Run as described within the script to produce a *.pi file for each *.so or *.pyd file in the PyGTK sources.
- Copy these *.pi files into the installed copy of PyGTK, so they sit next to the compiled *.so or *.pyd extension module file that they describe.
- Wing should now provide auto-completion and Source Assistant information when you `import gtk` and type `gtk.` in the editor.

Using a GUI Builder

Wing IDE doesn't currently include a GUI builder for PyGTK but it can be used with other tools, such as [glade](#).

To use an external GUI builder, **configure Wing to automatically reload files** that are altered by the GUI builder.

Then you can run Wing IDE and your GUI builder at the same time, working with both in an almost seamless manner.

A Caveat: Because Python lends itself so well to writing data-driven code, you may want to reconsider using a GUI builder for some tasks. In many cases, Python's introspection features make it possible to write generic GUI code that you can use to build user interfaces on the fly based on models of your data and your application. This can be much more efficient than using a GUI builder to craft individual menus and dialogs by hand. In general hand-coded GUIs also tend to be more maintainable.

Details and Notes

- Building GTK from sources can be a challenge. Wingware has developed some build support scripts which we can provide on request to support at wingware dot com (these are slated for release as open source soon). We also have patches that allow GTK to be relocated after building on Linux/Unix.
- Native look and feel on Windows is provided by the [gtk-wimp](#) theme. If you plan to deploy on Windows, you may wish to contact us to obtain our latest performance patches for GTK on Windows.

Related Documents

Wing IDE provides many other options and tools. For more information:

- **Wing IDE Reference Manual**, which describes Wing IDE in detail.
- **Wing IDE Quickstart Guide** which contains additional basic information about getting started with Wing IDE.

Using Wing IDE with PyQt

[Wing IDE](#) is an integrated development environment for the Python programming language. Wing can be used to speed up the process of writing and debugging code that is written for the [PyQt](#) cross-platform GUI development toolkit.

PyQt is a commercial GUI development environment that runs with native look and feel on Windows, Linux/Unix, Mac OS, and the Sharp Zaurus. It is also available for free for non-commercial users on Windows and for open source developers on Linux/Unix. Licensing is per developer, with no cost for each deployed product (even for commercial products).

While Wing IDE does not currently provide a GUI builder for PyQt, it does provide the most advanced debugger and source browser capabilities available for the Python programming language and it can be used with other available GUI builders, as described below.

Installation and Configuration

Take the following steps to set up and configure Wing IDE for use with PyQt:

- Install Python and Wing. Check the [PyQt download page](#) to make sure you install a version of Python that will work with the version of PyQt that you use. For example, PyQt 3.5 works with any version of Python between 1.5.2 and 2.2.x. The generic **Wing IDE Quickstart Guide** provides installation instructions for Wing.
- Install Qt from [Trolltech](#). You will either need to purchase a developer's licence or download the non-commercial package for Windows or Linux/Unix. The easiest way to find the downloads area is to search on "download qt" on their site.
- Install PyQt from the [Riverbank PyQt download area](#).
- Start Wing from the Start menu on Windows or by typing `wing` on the command line on Linux, OS X, or other Posix systems.

- Select Show Analysis Stats from the Source menu and if the Python version reported there doesn't match the one you're using with PyQt, then select Project Properties from the Project menu and use the Python Executable field to select the correct Python version.
- Open `PyQt/Examples/widgets.py` into Wing IDE (located within your Python installation) and select Add Current File from the Project menu.
- Set `widgets.py` as main entry point for debugging by selecting Set Current As Main Debug File from the Project menu.
- Save your project to disk. Use a name ending in `.wpr`.

Test Driving the Debugger

Now you're ready to try out the debugger. To do this:

- Start debugging with the Debug / Continue item in the Debug menu. Uncheck the Show this dialog before each run checkbox at the bottom of the dialog that appears and select OK.
- The demo application will start up. If its main window doesn't come to front, bring it to front from your task bar or window manager.
- Next open `PyQt/Examples/widgets.py` (within your Python installation) into Wing IDE. Set a breakpoint towards the end of `AnalogClock's paintEvent()` method. During the next clock update, which happens once a minute, Wing IDE will stop at that point. You can also cause the breakpoint to be reached sooner by obscuring and then unobscuring the clock with some other window.
- Use the Stack Data tool to look around the stack and the locals and globals for the selected stack frame.
- Select Debug Probe from the Tools menu. This is an interactive command prompt that lets you type expressions or even change values in the context of the stack frame that is selected on the Debugger window when your program is paused or stopped at an exception. It is a very powerful debugging tool.

Also take a look at these tools available from the Tools menu:

- I/O -- displays debug process output and processes keyboard input to the debug process, if any

- **Exceptions** -- displays exceptions that occur in the debug process
- **Modules** -- browses data for all modules in `sys.modules`
- **Watch** -- watches values selected from other value views (by right-clicking and selecting one of the **Watch** items) and allows entering expressions to evaluate in the current stack frame

As you try out the various demos for PyQt, you may sometimes see Wing IDE pause and report exceptions in the debugger's **Exceptions** tool. There are a few bugs in some versions of PyQt's demos, so Wing will catch those when they occur.

Also useful to know is that sometimes (but very rarely in PyQt apps) Wing IDE will report exceptions that you don't see outside of the debugger. These are not program errors but are caused by Wing's proactive exception detection algorithm, which cannot see into the C and C++ code that is handling these exceptions outside of the debugger.

To get past these types of exceptions, select **Ignore this exception location** in the debugger's **Exceptions** tool and continue execution. Your ignore list is stored in the project so you will never see them again, even in future debug sessions. Subsequently, you will benefit from Wing's ability to stop immediately at the point of exception, rather than after the fact. This makes understanding the conditions leading to an error much easier and speeds up debug turn-around.

Test Driving the Source Browser

Don't forget to check out Wing's powerful source browser:

- Add package **Lib/site-packages** (inside your Python installation) to your project with the **Add Package** item in the **Project** menu. Also add directory tree PyQt (also inside your Python installation) to your project file with the **Add Directory Tree** item in the **Project** menu. You should now see four items at the top level of the **Project** window.
- Next bring up the **Source Browser** from the **Tools** menu. You can select the view style at the top of the window: **By Modules** to browse by disk layout, **Class Hierarchy** to see only base classes at the top level, and **All Classes** to see a list of all classes by name. The **Options** menu on the right will filter what types of symbols are being displayed in the browser.
- Clicking on the browser will show the corresponding source code in the source editor area. Note that files are automatically closed when you browse elsewhere unless they were already open, edits are made, or you click on the stick pin icon

in the upper right of the editor area to specify that the editor should remain open until closed explicitly.

- Use the right-click menu to zoom to base classes. In general, right-clicking will bring up menus specific to the tool being clicked on.
- Related to the Source Browser is the auto-completion capability in Wing's source editor. Try typing in one of the PyQt source files and you will see the auto-completer appear. Tab completes the currently selected item, but you can set the **Auto-complete on Enter** preference to also complete when the Enter key is pressed. See the **Wing IDE Quickstart Guide** for information on this and other commonly used preferences.
- See also the **Source Assistant** tool in the Tools menu. This provides additional information about source constructs in the active source editor as the insertion cursor or selection is moved around.

Using a GUI Builder

Wing IDE doesn't currently include a GUI builder for PyQt but it can be used with other tools, such as [Black Adder](#), which does provide a GUI builder but doesn't have the raw power of Wing IDE's debugger and source browser. Another GUI builder for PyQt is [Qt Designer](#), which outputs language-independent UI files that can be converted into Python using PyQt's `pyuic` utility.<P>

To use an external GUI builder, **configure Wing to automatically reload files** that are altered by the GUI builder.

Then you can run Wing IDE and your GUI builder at the same time, working with both in an almost seamless manner.

A Caveat: Because Python lends itself so well to writing data-driven code, you may want to reconsider using a GUI builder for some tasks. In many cases, Python's introspection features make it possible to write generic GUI code that you can use to build user interfaces on the fly based on models of your data and your application. This can be much more efficient than using a GUI builder to craft individual menus and dialogs by hand. In general hand-coded GUIs also tend to be more maintainable, and the Qt widget set was designed specifically to make hand-coding easy.

Tips for Keeping the Debug Process Responsive

Because of bugs in some versions of PyQt, there is no code inside the debugger to ensure that PyQt debug processes remains responsive to the debugger while free-running. This means that you may not always be able to Pause PyQt debug processes, and the debugger may time out if you try to add breakpoints or execute certain other debugger operations while the GUI application is free-running and no Python code is being reached.

This problem occurs only when no Python code is reached at all, so it is easy to work around with the following after your `QApplication` has been created and before you call `exec_loop()`:

```
# Hack to burn some Python bytecode periodically so Wing's
# debugger can remain responsive while free-running
import os
if os.environ.has_key('WINGDB_ACTIVE'):
    timer = QTimer()
    def donothing(*args):
        for i in range(0, 100):
            x = i
    timer.connect(timer, SIGNAL("timeout()"), donothing)
    timer.start(500, 0)
```

Related Documents

Wing IDE provides many other options and tools. For more information:

- **Wing IDE Reference Manual**, which describes Wing IDE in detail.
- [PyQt home page](#), which provides links to documentation.
- **Wing IDE Quickstart Guide** which contains additional basic information about getting started with Wing IDE.

Using Wing IDE with Zope

“The best solution for debugging Zope and Plone” -- *Joel Burton, Member, Plone Team, Jul 2005*

[Wing IDE](#) can be used to develop and debug Python code running under Zope2 or Zope3, including Products, External Methods, file system-based Scripts and Zope itself. It is also useful for Zope-based frameworks like [Plone](#) (see **Plone Quickstart**).

Wing provides auto-completion, call tips, and other features that help to write, navigate, and understand Zope code. Wing’s debugger can work with Zope’s code reloading features to achieve a very short edit/debug cycle.

Note: This guide is for Zope2 users. If you are using Zope3, please try [z3wingdbg](#) by Martijn Pieters or refer to **Debugging Externally Launched Code** in the users manual to set up Zope3 debugging manually.

Limitations: Wing IDE cannot debug DTML, Page Templates, ZCML, or Python code that is not stored on the file system.

Security Warning: We advise against using the WingDBG product on production web servers. Any user connected to the Wing IDE debugger will (unavoidably) have extensive access to files and data on the system.

Quick Start on a Single Host

To use Wing IDE with Zope running on the same host as the IDE:

- **Install Zope** -- You can obtain Zope from [zope.org](#). Version 2.5.1 or newer will work with Wing.

- **Install Wing IDE** -- You will need [Wing IDE 2.1](#) or later. See **Installing** for details.
- **Configure Wing IDE** -- Start Wing, create or open the project you wish to use (from the Project menu). Then use the **Extensions** tab in **Project Properties** to enable **Zope2/Plone support** and to specify the **Zope2 Instance Home** to use with the project. Wing will find your Zope installation by reading the file `etc/zope.conf` in the provided Zope instance. Once you press **Apply** or **OK** in the Project Properties dialog, Wing will ask to install the WingDBG product and will offer to add files from your Zope installation to the project.
- **Configure the WingDBG Product** -- Start or restart Zope and log into <http://localhost:8080/manage> (assuming default Zope configuration). The Wing Debugging Service will be created automatically on startup; you can find it under the Control Panel of your server.

Starting the Debugger

Proceed to the Wing Debugger Service by navigating to the Control Panel, then selecting the 'Wing Debugging Service'. Click in the "Start" button. The Wing IDE status area should display "Debugger: Debug process running".

Note that you can configure WingDBG to start and connect to the IDE automatically when Zope is started from the Advanced configuration tab.

Problems? See the Trouble-Shooting Guide below.

Test Drive Wing IDE

Once you've started the debugger successfully, here are some things to try:

Run to a Breakpoint -- Open up your Zope code in Wing IDE and set a breakpoint on a line that will be reached as the result of a browser page load. Then load that page in your web browser using the port number displayed by the Zope Management Interface after you started the debugger. By default, this is 50080, so your URL would look something like this:

```
http://localhost:50080/Rest/Of/Usual/Url
```

Explore the Debugger Tools -- Take a look at these tools available from the Tools menu:

- **Stack Data** -- displays the stack, allows selecting current stack frame, and shows the locals and globals for that frame.
- **Debug Probe** -- lets you interact with your paused debug process using a Python shell prompt
- **Watch** -- watches values selected from other value views (by right-clicking and selecting one of the **Watch** items) and allows entering expressions to evaluate in the current stack frame
- **Modules** -- browses data for all modules in `sys.modules`
- **Exceptions** -- displays exceptions that occur in the debug process
- **Debug I/O** -- displays debug process output and processes keyboard input to the debug process, if any

Continue the Page Load -- When done, select **Debug / Continue** from the **Debug** menu or toolbar.

Try Pause -- From Wing, you can pause the Zope process by pressing the **Pause** icon in the toolbar or using **Pause** from the **Debug** menu. This is a good way to interrupt a lengthy computation to see what's going on. When done between page loads, it pauses Zope in its network service code.

Other Features -- Notice that Wing IDE's editor contains a source index and presents you with an auto-completer when you're editing source code. The **Source Assistant** will display context appropriate call tips and documentation. Control-click on a source symbol to jump to its point of definition (or use **Goto Selected Symbol** in the **Source** menu). Bring up the **Source Browser** from the **Tools** menu to look at the module and class structure of your code.

Setting Up Auto-Refresh

When you edit and save Zope External Methods or Scripts, your changes will automatically be loaded into Zope with each new browser page load.

By default, Zope Products are not automatically reloaded, but it is possible to configure them to do so. This can make debugging much faster and easier.

Take the following steps to take advantage of this feature:

- Place a file called `refresh.txt` in your Product's source directory (for example, `Products/MyProductName` inside your Zope installation). This file tells Zope to allow refresh for this product.

- Open the Zope Management Interface.
- Expand the Control Panel and Products tabs on the upper left.
- Click on your product.
- Select the Refresh tab.
- Check the “Auto refresh mode” check box and press “Change”.
- Make an edit to your product source, and you should see the changes you made take effect in the next browser page load.

Limitations: Zope may not refresh code if you use `import` statements within functions or methods. Also, code that manages to retain references to old code objects after a refresh (for example, by holding the references in a C/C++ extension module) will not perform as expected.

If you do run into a case where auto-reload causes problems, you will need to restart Zope from the Zope Management Interface’s Control Panel or from the command line. Note that pressing the Stop button in Wing only disconnects from the debug process and does not terminate Zope.

Setting up Remote Debugging

Configuring Wing for remote debugging can be complicated, so we recommend using X Windows (Linux/Unix) or Remote Desktop (Windows) to run Wing IDE on the same machine as Zope but display it remotely.

When this is not possible, you can set up Wing to debug Zope running on another machine, as described below:

- **Set up File Sharing** -- You will need some mechanism for sharing files between the Zope host and the Wing IDE host. Windows file sharing, Samba, NFS, and ftp or rsync mirroring are all options. For secure file sharing via SSH on Linux, try [sshfs](#).
- **Install Wing on Server** -- You will also need to install Wing on the host where Zope is running, if it is not already there. No license is needed for this installation, unless you plan to also run the IDE there. If there is no binary distribution of Wing available for the operating system where Zope is running, you can instead install only the debugger libraries as outlined in **Compiling the Wing IDE Debugger from Source**.

- **Basic Configuration** -- Follow the instructions for Single-Host Debugging above first if you have not already done so. Then return here for additional setup instructions.
- **Configure Allowed Hosts** -- You will need to add the IP address of the Zope host to the **Allowed Hosts** preference in Wing. Otherwise Wing will not accept your debug connections.
- **Configure File Mapping** -- Next, set up a mapping between the location of the Zope installation on your Zope host and the point where it is accessible on you Wing IDE host. For example, if your Zope host is 192.168.1.1 Zope is installed in /home/myuser/Zope on that machine, and /home/myuser is mounted on your Wing IDE host as e:, you would add a **Location Map** preference setting that maps 192.168.1.1 to a list containing /home/myuser/Zope and file:e:/Zope. For more information on this, see **File Location Maps** and **Location Map Examples** in the Wing IDE manual.
- **Modify WingDBG Configuration** -- When debugging remotely, the value given to WingDBG for the Wing Home Directory must be the location where Wing is installed on the Zope host (the default value will usually need to be changed).
- **Check Project Configuration** -- Similarly, the paths identified in Project Properties should be those on the host where Wing IDE is running, not the paths on the Zope host.

Trouble Shooting Guide

You can obtain additional verbose output from Wing IDE and the debug process as follows:

- Go into the Wing Debugging Service in the Zope Management Interface and set **Log file** under the **Configure** tab. Using <stdout> will cause logging information to be printed to the console from which Zope was started. Alternatively, set this to the full path of a log file. This file must already exist for logging to occur.
- Restart Zope and Wing and try to initiate debug.
- Inspect the contents of the log. If you are running Zope and Wing IDE on two separate hosts, you should also inspect the **error-log** file on the Wing IDE host (located in the **User Settings Directory**). It contains additional logging information from the Wing IDE process.

For additional help, send these errors logs to [support at wingware.com](mailto:support@wingware.com).

Related Documents

Wing IDE provides many other options and tools. For more information:

- **Wing IDE Reference Manual**, which describes Wing IDE in detail.
- [Zope home page](#), which contains much additional information for Zope programmers.
- **Quick Start Guide** and **Tutorial** which contain additional basic information about getting started with Wing IDE.

Using Wing IDE with Plone

“The best solution for debugging Zope and Plone” -- *Joel Burton, Member, Plone Team, Jul 2005*

[Wing IDE](#) is an integrated development environment for the Python programming language. Wing can be used to speed up the process of writing and debugging code that is written for [Plone](#), a powerful web content management system.

Since Plone is based on [Zope](#), setting up Wing IDE with Plone is very similar to setting up Wing IDE with Zope, which is described in detail in the **Zope quickstart guide**. The only difference is that you will download and install Plone instead of Zope. Also, instead of launching Zope only, you will be launching Plone (which on win32 is done with the Plone launcher tool from the Start menu).

Performance Hints

Plone and Zope together contain a very large Python code base. If you Add Directory Tree from the Project menu to include the entire Plone installation, you will see significant CPU intensive processing, which can be an issue on slower machines and may take several minutes to complete. This happens as Wing analyses the Python source code in order to build the datastructures needed for the source browser, auto-completer, source index menu, goto-definition, and other features of the IDE. Wing should remain responsive during this time but may be sluggish.

As long as the initial analysis is in process, the source browser will contain a subset of all the code constructs available and some features like goto-definition may not work until the entire source has been processed.

Most of the work happens in the time following the point when you first add the Python files to your project. After that, a disk cache is used to speed up access to analysis information, but you will see reprocessing of the disk cache whenever you start a new Wing IDE session; this can also be significant on some machines.

In all cases, processing will cease after a period of time and the rest of your Wing IDE session should run at near zero CPU usage, with a snappy and responsive GUI even on slower machines.

Related Documents

Wing IDE provides many other options and tools. For more information:

- **Using Wing IDE with Zope**, which describes how to set up Zope for use with Wing IDE.
- **Wing IDE Reference Manual**, which describes Wing IDE in detail.
- [Plone home page](#), which provides links to documentation.
- **Wing IDE Quickstart Guide** which contains additional basic information about getting started with Wing IDE.
- [Plone Bootcamps](#) offer comprehensive training on Plone using Wing IDE throughout the course. Students learn how to set up and use Wing IDE with Plone.

Using Wing IDE with Webware

[Wing IDE](#) is an integrated development environment for the Python programming language. Wing can be used to speed up the process of writing and debugging code that is written for [Webware](#), an open source web development framework.

To get started, refer to the tutorial in the **Help** menu in Wing and/or the **Wing IDE Quickstart Guide**.

Introduction

Wing IDE allows you to graphically debug a Webware application as you interact with it from your web browser. Breakpoints set in your code from the IDE will be reached, allowing inspection of your running code's local and global variables with Wing's various debugging tools. In addition, in Wing IDE Pro, the **Debug Probe** tab allows you to interactively execute methods on objects and get values of variables that are available in the context of the running web app.

There is more than one way to do this, but in this document we focus on an “in process” method where the Webware server is run from within Wing as opposed to attaching to a remote process. The technique described below was tested with **Webware 0.9** and **Python 2.4** on **CentOS Linux**. It should work with other versions and on other OSes as well. Your choice of browser should have no impact on this technique.

Setting up a Project

Though Wing supports the notion of “Projects” for organizing one's work for this debugging scenario you can use the **Default Project** and simply add your source code directory to it by using **Add Directory Tree** from the **Project** menu.

You will also need to specify a **Python Path** in your **Project Properties** with something like following (your actual paths depend on your installation of Webware and OS):

```
/usr/local/lib/Webware-0.9/WebKit:/usr/local/lib/Webware-0.9:/home/dev/mycodebase
```

The Webware `MakeAppDir.py` script creates a default directory structure and this example assumes that the source code is nested within this directory. Note that on Windows, the path separator should be `';` (semicolon) instead.

To debug your Webware app you'll actually be running the `DebugAppServer` and not the regular `AppServer`. So you'll need to bring in the `Debug AppServer` and a couple of other files with these steps:

- 1) Copy the `DebugAppServer.py`, `ThreadedAppServer.py`, and `Launch.py` from the `WebKit` directory and put them in the root of the directory that `MakeAppDir.py` created.
- 2) Right click on `Launch.py` in Wing's editor and select the menu choice `Properties`. A properties window will display with an `Environment` and `Debug` tab. Click the `Debug` tab and enter `DebugAppServer.py` in the `Run Arguments` field. If you're using the default project then leave the initial directory and build command settings as they are.
- 3) If you need to modify the version of Python you're running, you can change the `Python Executable` on the `Environment` tab of this debug properties window, or project-wide from the `Project Properties`.
- 4) Optionally, after adding `Launch.py` to the project, right-click on its name in the project view and select `Set as Main Debug File`. This will cause Wing to always launch this file, regardless of which file is current in the editor.

Starting Debug

To debug, press the green `Debug` icon in the toolbar. If you did not set a main debug file in the previous section, you must do this when `Launch.py` is the current file.

The file properties dialog will appear. Optionally, deselect `Show this dialog before each run`. If you do this you can access the dialog again later by right clicking on the file in Wing's editor and selecting `Properties`.

Click OK to start the debug process. The `Debug I/O` tool will show output from the Webware process as it starts up. What you will see there depends upon your Webware application and server settings, but you should see some log messages scroll by. If there is a path or other kind of problem as the debugging process proceeds errors will display

in the Debug I/O tool or in a pop-up error message in Wing if you have a missing library or run into another unhandled exception.

Once the process has started up, you will be able to access web pages from your browser according to your configuration of Webware, just as you would when running the server outside of Wing.

Now for the fun part -- fire up your browser and go to the home page of your application. Go into the source file for any Python servlet in Wing and set a breakpoint somewhere in the code path that you know will be executed when a given page is requested. Navigate to that page in your browser and you should see the Wing program icon in your OS task bar begin to flash. (You'll see that the web page won't finish loading -- this is because the debugger has control now; the page will finish loading when you continue running your app by pressing the **Debug** icon in the toolbar).

Now you can make use of all of the powerful debugging functionality available in Wing instead of sprinkling your code with print statements.

Related Documents

Wing IDE provides many other options and tools. For more information:

- **Wing IDE Reference Manual**, which describes Wing IDE in detail.
- **Wing IDE Quickstart Guide** which contains additional basic information about getting started with Wing IDE.

Using Wing IDE with mod_python

[Wing IDE](#) is an integrated development environment for the Python programming language. Wing can be used to debug code that is run by the [mod_python](#) module for the Apache web server. This document assumes mod_python is installed and Apache is configured to use it; please see the installation chapter of the mod_python manual for information on how to install it.

Since Wing's debugger support is currently single threaded, only one http request can be debugged at a time. A new debugging session is created for each request and the session is ended when the request processing ends. If a second request is made while one is being debugged, it will either block until the first request completes or it will be processed without the debugger. This is true of requests processed by a single Python module and it is true of requests processed by multiple Python modules in the same Apache process and its child processes. It is recommended that only one person debug mod_python based modules per Apache instance.

Quick Start

- Copy `wingdbstub.py` from the Wing IDE installation directory into either the directory the module is in or another directory in the Python path used by the module.
- Edit `wingdbstub.py` if needed so the settings match the settings in your preferences. Typically, nothing needs to be set unless Wing's debug preferences have been modified. If you do want to alter these settings, see the **Remote Debugging** section of the Wing IDE reference manual for more information.
- Copy `.wingdebugpw` from your **User Settings Directory** into the directory that contains the module you plan to debug. This step can be skipped if the module to be debugged is going to run on the same machine and under the same user as Wing IDE. The `.wingdebugpw` file must contain exactly one line.

- Insert `import wingdbstub` at the top of the module imported by the `mod_python` core.
- Insert `if wingdbstub.debugger != None: wingdbstub.debugger.StartDebug()` at the top of each function that is called by the `mod_python` core.
- Enable passive listening in Wing by setting the **Enable Passive Listen** preference to true.
- Restart Apache and load a URL to trigger the module's execution.

Example

To debug the `hello.py` example from the Publisher chapter of the `mod_python` tutorial, modify the `hello.py` file so it contains the following code:

```
import wingdbstub

def say(req, what="NOTHING"):
    if wingdbstub.debugger != None:
        wingdbstub.debugger.StartDebug()
    return "I am saying %s" % what
```

And set up the `mod_python` configuration directives for the directory that `hello.py` is in as follows:

```
AddHandler python-program .py
PythonHandler mod_python.publisher
```

Then set a breakpoint on the `return "I am saying %s" % what` line, make sure Wing is listening for a debug connection, and load `http://[server]/[path]/hello.py` in a web browser (substitute appropriate values for `[server]` and `[path]`). Wing should then stop at the breakpoint.

Related Documents

- **Wing IDE Reference Manual**, which describes Wing IDE in detail.
- [Mod_python Manual](#), which describes how to install, configure, and use `mod_python`.

Debugging Web CGIs with Wing IDE

Wing IDE can be used to debug CGI scripts written in Python, even as they are running as the result of a page load from a web browser. To set up your CGIs for debugging with Wing IDE, refer to the **Debugging Externally Launched Code** section of the manual. Pay careful attention to the permissions on files, especially if your web server is running as a different user than the process that is running Wing IDE. You will also need to make sure that the `.wingdebugpw` file is referenced correctly as described in the instructions.

The rest of this guide provides some tips specific to the task of debugging CGIs:

(1) If Wing is failing to stop on breakpoints, check whether you are loading a web page that loads multiple parts with separate http requests -- in that case, Wing may still be busy processing an earlier CGI request when a new one comes in and will fail to stop on breakpoints because only one debug process is serviced at a time. This is a limitation in Wing; in the future we plan to support multiple oncurrent debug sessions. The work-around is to load specific parts of the page in the browser by entering URL you wish to debug.

(2) Any content from your CGI script that isn't understood by the web server will be written to the server's error log. Since this can be annoying to search through, it is much easier to ensure that all output, including output made in error, is displayed in your web browser.

To do this, insert the following at the very start of your code, before importing `wingdbstub` or calling the debugger API:

```
print "Content-type: text/html\n\n\n<html>\n"
```

This will cause all subsequent data to be included in the browser window, even if your normal Content-type specifier code is not being reached.

(3) Place a catch-all exception handler at the top level of your CGI code and print

exception information to the browser. The following function is useful for inspecting the state of the CGI environment when an exception occurs:

```
import sys
import cgi
import traceback
import string

#-----
def DisplayError():
    """ Output an error page with traceback, etc """

    print "<H2>An Internal Error Occurred!</H2>"
    print "<I>Runtime Failure Details:</I><P>"

    t, val, tb = sys.exc_info()
    print "<P>Exception = ", t, "<br>"
    print "Value = ", val, "\n", "<p>"

    print "<I>Traceback:</I><P>"
    tbf = traceback.format_tb(tb)
    print "<pre>"
    for item in tbf:
        outstr = string.replace(item, '<', '&lt;')
        outstr = string.replace(outstr, '>', '&gt;')
        print string.replace(outstr, '\n', '\n'), "<BR>"
    print "</pre>"
    print "<P>"

    cgi.print_environ()
    print "<BR><BR>"
```

(4) If you are using `wingdbstub.py`, you can set `kSilent=0` to receive extra information from the debug server, in order to debug problems connecting back to Wing IDE. This information is written to `stderr` and thus will be found in the web server's error log file.

(5) If you are using the full debugger API, you can set your `CErrStream` object to send output either to `stdout`, `stderr`, or any other file stream. Use this to send errors to the browser, web server error log, or to a file, respectively.

(6) If you are unable to see script output that may be relevant to trouble-shooting, try invoking your CGI script from the command line. The script may fail but you will be able to see messages from the debug server, when those are enabled.

(7) If all else fails, read your web browser documentation to locate and read its error log file. On Linux with Apache, this is often in `/var/log/httpd/error_log`. Any errors not seen on the browser are appended there.

(8) Once you have the debugger working for one CGI script, you will have to set up the `wingdbstub` import in each and every other top-level CGI in the same way. Because this can be somewhat tedious, and because the import needs to happen at the start of each file (in the `__main__` scope), it makes sense to develop your code so that all page loads for a site are through a single entry point CGI and page-specific behavior is obtained via dispatch within that CGI to other modules. With Python's flexible import and invocation features, this is relatively easy to do.

Wing IDE for OS X

Wing IDE uses X windows on OS X, but support for X is not by default part of all versions of OS X. Thus you also need to obtain and install an X server and X Window manager. There are a number of options for this:

- (1) [Apple's X11 Server for OS X](#) is among the fastest and best integrated options. It includes both the X Server and a native Aqua window manager, although you can replace the default window manager with your favorite if you wish. Apple X11 Server comes with OS X 10.3 and later, but is not installed by default and must be installed separately from the installation CDs. X11 is part of optional installs package on the installation disk; often it's hidden by default so you'll need to scroll down in the finder window for the installation disk to find it. For OS X 10.3, X11 is also available as downloadable package from Apple's website, but this version will not work with OS X 10.4.
- (2) [XDarwin](#) (1.1 or later) can be used together with the window manager of your choice. [Window Maker](#) is one that users have reported as working well. [OroborOSX](#) (0.75a4r2 or later) also works but can be quite slow in comparison with other options (as of 0.8 preview 2). Note that for some versions of OroborOSX, you need to unpack both the top-level OroborOSX tar file and the XDarwin.tar file located inside the installation.

Once this is set up, you're ready to install Wing IDE. Just download [Wing IDE](#), unpack it and move it into place.

Double clicking on the Wing IDE tar archive will expand its contents into a new folder on disk in the same location as the archive. Subsequently, the tar archive can be removed and the expanded form of the application can be moved on disk as desired.

<p>Note: On some systems, tar will truncate file names leading to a broken installation. In that case, unpack the archive with gnutar instead.</p>

Next make sure your X Windows server chosen above is running and set up to allow connections from X clients on `:0.0`. Wing will automatically start Apple X11 Server if it is present and not yet running.

Then double click on the Wing IDE app folder created by unpacking the `tar` archive. The first time you start Wing, it will ask you to accept the license agreement and will ask for a license. Use the first (default) option in the dialog that appears to obtain a 10 day trial license (this can be renewed twice when the trial period expires).

At this time, Wing will create a **User Settings Directory** in `~/.wingide2`, which is used to store preferences and other settings.

The location of Wing's internal application contents folder is referred to as `WINGHOME`. For example if you unpacked Wing into `/Applications/Wing` then `WINGHOME` will be `/Applications/Wing/WingIDE.app/Contents/MacOS`.

To start Wing from the command line, execute `wing-personal` located in `WINGHOME` (`/Applications/Wing/WingIDE.app/Contents/MacOS/wing` in the above example). When you do this, you may need to set your `DISPLAY` environment variable to point to your X Server (for example `setenv DISPLAY :0.0`).

See the **Tutorial** and **Wing IDE Quickstart Guide** for additional information on getting starting with Wing IDE.

Usage Tips

Wing starts with a keyboard mapping that emulates the most commonly used key standards on the Macintosh. This mapping will not work properly unless you uncheck the "Enable Key Equivalents under X11" preference in Apple X11 Server configuration. Wing will warn when this option is checked, but may fail to do so under other X11 server software. Wing also tries to detect if an Apple keyboard is in use, but you may need to set the **Apple Keyboard** preference to `yes` or `no` if the detection fails.

You can alter keyboard mapping (for example, to use Emacs bindings instead) with the **Personality** preference, or change individual key mappings with the **Custom Key Bindings** preference. If you do use a keyboard personality other than the OS X personality, you may want to map the option or command key to the Alt modifier using the **Global X11 Alt Key** preference. Note that this preference affects all X11 applications, not just Wing.

Under OS X 10.4 (Tiger), the option/compose key used to enter accented and other foreign characters will not work because of changes in the Apple X11 applications. To fix this, enable the **Fix Option key bug in Tiger (OS X 10.4)** preference. This will affect all X11 applications, not just Wing.

If you are running other X11 applications and want to workaroud this bug yourself, you'll want to either enable the XKEYBOARD extension on the X11 server or use xmodmap to assign the Mode_Switch key to a modifier other than mod1. The xmodmap script that Wing runs when the **Fix Option key bug in Tiger (OS X 10.4)** preference is enabled is `${WINGHOME}/resources/osx/fix-tiger-mode-switch.xmodmaprc`. The script removes Mode_Switch from mod1 and adds it to mod5 and is run only if the preference is enabled and xmodmap reports that Mode_Switch is assigned to mod1.

If the X11 Application "Use the system keyboard layout" preference is enabled, then the X11 server may modify its keyboard mapping when the system keyboard changes. You may need to disable this preference or restart Wing after the keyboard layout changes because Wing will not re-apply the fix after the X11 keyboard changes. This should only be an issue if you change keyboard layouts while Wing is running.

Changing Display Themes

Although Wing is not a native OS X application, it starts up with a display theme that tries to match the OS X look and feel (font size often needs to be altered from the **Display Font/Size** preference). Additional display themes can be selected from the **Display Theme** preference.

It is also possible to download [other themes for GTK2](#) and place them into `Contents/MacOS/bin/gtk-bin/share/themes` inside your Wing IDE installation. Once this is done and Wing is restarted, they will show up in the **Display Theme** preference.

One nice OS X like theme is [AquaX](#), currently not included with Wing because we cannot redistribute it under its licensing.

Note that only themes that do not use a theme engine or use one of Redmond, Smooth, or Pixmap will work with Wing IDE. We cannot make any guarantees for performance or results when using themes not included with Wing IDE, although Aqua X is known to work well.

Finding WINGHOME

When using the **Zope Support Module** or following instructions that refer to WINGHOME note that WINGHOME is defined as the location of the wing executable, which on Mac OS X is inside the Contents/MacOS folder of the Wing IDE app folder. E.g., if you unpacked Wing into `/Applications/Wing` then WINGHOME will be `/Applications/Wing/WingIDE.app/Contents/MacOS`.

Mouse Buttons

Right-click for menus by holding down the Option/Alt key while clicking. Middle-click by holding down the Control key while clicking. These defaults can be changed in your X11 server's preferences. For example, under Apple X11 Server, change so Option/Alt is button two and Control is button three with this command:

```
defaults write com.apple.x11 fake_button2 option
defaults write com.apple.x11 fake_button3 control
```

Or change so that Option/Alt is button two and Apple/Command is button three:

```
defaults write com.apple.x11 fake_button2 option
defaults write com.apple.x11 fake_button3 command
```

Then restart the X11 Server.

Window Focus

You can configure Apple X11 Server to automatically transfer focus to the window the mouse pointer is over, or to pass through the click that is used to bring focus to the window so it is also processed by the application.

To move focus with the mouse pointer:

```
defaults write com.apple.x11 wm_ffm true
```

To pass through the focus click:

```
defaults write com.apple.x11 wm_click_through -bool true
```

You must restart Apple X11 changing either of these configurations before they take effect.

Other configuration options like this can be obtained by looking in the manual pages for quartz-wm and Xquartz:

```
man quartz-wm
man Xquartz
```

Known Problems

There are some known problems resulting from platform-specific behaviors on OS X:

There is no way to control-click, so control-left-click for the goto-definition feature doesn't work. Instead, use Goto Selected Symbol from the Source menu instead; this works relative to position of the insertion cursor in the current editor and can be accessed by the key binding shown in the Source menu.

Please send bug reports to [bugs at wingware.com](mailto:bugs@wingware.com).

Related Documents

- **Wing IDE Quickstart Guide**, which contains additional information about getting started with Wing IDE.
- **Other How-Tos**, for getting started with Wing IDE and specific tools.
- **Wing IDE Reference Manual**, which describes Wing IDE in detail.

Using Wing IDE with pygame

[Wing IDE](#) is an integrated development environment for the Python programming language. Wing can be used to speed up the process of writing and debugging code that is written for [pygame](#), an open source framework for game development with Python..

To get started, refer to the tutorial in the **Help** menu in Wing and/or the **Wing IDE Quickstart Guide**.

Debugging pygame

You should be able to debug pygame code with Wing just by starting debug from the IDE. However, pygame running in full screen mode will not work properly and may crash Wing. Use window mode instead.

This problem exists with other Python IDEs as well; we have not yet determined what the cause is.

Related Documents

Wing IDE provides many other options and tools. For more information:

- **Wing IDE Reference Manual**, which describes Wing IDE in detail.
- **Wing IDE Quickstart Guide** which contains additional basic information about getting started with Wing IDE.

Using Wing IDE with scons

[Wing IDE](#) is an integrated development environment for the Python programming language. Wing can be used to speed up the process of writing and debugging code that is written for [scons](#), an open source software construction or build control framework that uses Python.

To get started, refer to the tutorial in the **Help** menu in Wing and/or the **Wing IDE Quickstart Guide**.

Debugging scons

As of version 0.96.1 of `scons`, the way that `scons` executes build control scripts does not work properly with Wing's (or probably *any*) debugger because the scripts are executed in an environment that effectively sets the wrong file name for the script. Wing will bring up the wrong file on exceptions and will fail to stop on breakpoints.

The solution for this is to patch `scons` by replacing the `exec _file_` call with one that unsets the incorrect file name, so that Wing's debugger looks into the correctly set `co_filename` in the code objects instead.

The code to replace is in `engine/SCons/Script/SConscript.py` (around line 239 in `scons` version 0.96.1):

```
exec _file_ in stack[-1].globals
```

Here is the replacement code to use:

```
old_file = call_stack[-1].globals.pop('__file__', None)
try:
    exec _file_ in call_stack[-1].globals
finally:
    if old_file is not None:
        call_stack[-1].globals.update({'__file__':old_file})
```

Once this is done, Wing should show the correct file on exceptions and stop on breakpoints set within the IDE.

Note that if you launch scones from the command line (likely the preferred method) rather than from within Wing IDE, you will need to use `wingdbstub` as described in **Debugging Externally Launched Code**.

Related Documents

Wing IDE provides many other options and tools. For more information:

- **Wing IDE Reference Manual**, which describes Wing IDE in detail.
- **Wing IDE Quickstart Guide** which contains additional basic information about getting started with Wing IDE.

Handling Large Values and Strings in the Debugger

To avoid hanging up on large values during stepping and other debugger actions, the debugger limits the size of constructs that it will display.

You can alter the size limit on large compound values with the **Huge List Threshold** preference. If you do this, you may also need to increase the debugger's patience in waiting for these large lists to transfer with the **Network Timeout** preference. This value is purposely set low now to prevent lengthy hanging of the IDE when the debug program crashes.

Long strings are also truncated by default when they are sent to the IDE from the debug process. To expand a truncated string, click on it and view its full form in the Textual View area at the bottom of the debugger window. If you are working from a zoom-ed out view, right click on the string instead and use the Display in Textual View item in the popup menu that appears.

The maximum displayable length of strings is controlled with the **Huge String Threshold** preference.

One way of viewing large values without increasing preference thresholds is to enter expression in the Watch or Debug Probe tools, accessible from the Tools menu. For example, for a large array, you can enter a value like `a[2:5] [7]` to arrive at a manageable value size.

Despite per-value size limits, it is still possible to transfer large amounts of data from the debug server. For example, we recommend some caution when using Expand More in the Textual display view.

Debugging C/C++ and Python together

The Wing debugger is for Python code only at this point and doesn't itself handle stepping into C or C++. However, you can set up VC++ or the gdb debugger in conjunction with the Wing IDE debugger to debug errors in either C or Python at the same time.

This is done by running your Python code under the VC++ or gdb debugger as you would anyway for C/C++ debugging, while using the Wing debugger at the same time by importing `wingdbstub` into your code.

To debug the C/C++ code you need to be running with a copy of Python compiled from sources with debug symbols. To configure `wingdbstub`, see the manual section on **Debugging Externally Launched Code**.

See also this additional information on **using gdb and Wing together**. Using Wing and VC++ is prone to fewer problems so doesn't currently have its own How-To.

Debugging Extension Modules on Linux/Unix

Gdb can be used as a tool to aid in debugging C/C++ extension modules written for Python, although doing so can be a bit tricky and prone to problems. The following text contains hints to make this easier.

Note that this section assumes you are already familiar with gdb; for more information on gdb commands, please refer to the gdb documentation.

The first step in debugging C/C++ modules with gdb is to make sure that you are using a version of Python that was compiled with debug symbols. To do this, you need a source distribution of Python and you need to configure the distribution as described in the accompanying README file.

In most cases, this can be done as follows: (1) Type `./configure`, (2) type `make OPT=-g` or edit the Makefile so `OPT=-g`, (3) type `make`, and (4) once the build is complete, install it with `make install` (but see the README first if you don't want to install into `/usr/local/lib/python`).

If you are building an extension module that you are compiling into the Python interpreter, you can now just run Python within gdb, set a breakpoint at the desired location in your extension module, and execute your Python test program.

In most cases, however, the extension module is not compiled into Python but is instead loaded dynamically at runtime. In order to get your extension module to load, it must be on the `PYTHONPATH` or within the same directory where the module is `import`-ed into Python source.

Gdb additionally requires setting `LD_LIBRARY_PATH` to include the directory where the dynamically loaded module is located. A common problem in doing this is that gdb will reread `.cshrc` each time that it runs, so setting `LD_LIBRARY_PATH` before invoking gdb has no effect if you also set `LD_LIBRARY_PATH` in `.cshrc`. To work around this, set `LD_LIBRARY_PATH` in `.profile` instead. This file is read only once at login time.

Next you may want to set up your `~/gdbinit` file by copying the contents of the file

Misc/gdbinit from the Python source distribution. This contains some useful macros for inspecting Python code from gdb. You may also want to add the following, which lets you print the Python stack:

```
define ppystack
  while $pc < Py_Main || $pc > Py_GetArgcArgv
    if $pc > eval_frame && $pc < PyEval_EvalCodeEx
      set $__fn = PyString_AsString(co->co_filename)
      set $__n = PyString_AsString(co->co_name)
      printf "%s (%d): %s\n", $__fn, f->f_lineno, $__n
    end
    up-silently 1
  end
  select-frame 0
end
```

Then start Python as follows:

```
myhost> gdb
(gdb) file python
(gdb) run yourprogram.py yourargs
```

Note that breakpoints in a shared library cannot be set until after the shared library is loaded. If running your program triggers loading of your extension module library, you can use `^C^C` to interrupt the debug program, set breakpoints, and then continue.

Otherwise, you must continue running your program until the extension module is loaded. When in doubt, add a `print` statement at point of import, or you can set a breakpoint at `PyImport_AddModule` (this can be set after `file python` and before running since this call is not in a shared library).

Unfortunately, even if you take all of the above steps, some versions of gdb will often get confused if you load and unload shared libraries repeatedly during a single debug session. You can usually re-run Python 5-10 times but subsequently may see crashing, failure to stop at breakpoints, or other odd behaviors. When this occurs, there is no alternative but to exit and restart gdb.

Finally a hint for viewing Python data from the C/C++ side when using gdb. The following gdb command will print out the contents of a `PyObject *` called `obj` as if you had issued the command `print obj` from within the Python language:

```
(gdb) p PyObject_Print (obj, stderr, 0)
```

Debugging Code with XGrab* Calls

Under X11 (Linux/Unix), Wing does not attempt to break XGrabPointer or XGrabKey and similar resource grabs when your debug process pauses. This means that X may be unresponsive to the keyboard or mouse or both in some debugging cases.

Fixing the debugger to detect and ungrab resources in this case is quite difficult and prone to confusing and unwanted behaviors. Here are some tips for working around this problem:

(1) Some toolkits have an option to disable resource grabs specifically to avoid this problem during debugging. For example, PyQt has a command line option `-nogradb` that prevents it from ever grabbing the keyboard or pointer. Adding this to the debug process command line solves the problem.

If you are writing your own calls to XGrab* or similar functions, consider adding a mode where these calls are skipped.

One trick that often helps is to move processing from the callback where the pointer or keyboard grab is still in effect into an asynchronous call-back that occurs at idle time. For example, under GTK use `gtk_idle_add()` and in wxPython try a `wxTimer`.

(2) Under XFree 4.2 and later, there is a configuration option you can set so that `Ctrl-Alt-Keypad-/` will break through any active pointer and keyboard grabs:

```
# Let user break server grabs with ctrl-alt-keypad-/  
Option "AllowDeactivateGrabs" "true"
```

This goes into your XF86Config file in the "ServerOptions" section. Check `man XF86Config` for the search path that X uses to find the config file and find the one that's used on your system. For example, on Mandrake 8.2 the config file is `/etc/X11/XF86Config-4`.

You will need to restart your X server for the config changes to take effect (for example, log out and back in again). Be aware that altering your XF86Config file can cause X to fail to start up. If this happens, you will need to fix it in text mode. If you get into this situation, typing `startx` after each edit is a good way to check whether your fix works.

If you need to check what version of XFree you're running, typing `rpm -q XFree86` usually works or `man XFree86` shows the version number at the very end of the man page.

(3) On Linux, if all else fails, you can use `Ctrl-Alt-F1` through `Ctrl-Alt-F6` on most distributions to get at six text-only virtual console. From there you can `ps` to find the debug process and kill it with `kill -TERM` or `kill -9` if necessary. This will unlock your X windows display, which you can return to with `Ctrl-Alt-F7`.

(4) Displaying your debug process to another screen avoids tying up Wing in this way. Most servers will unlock the screen once you kill the debug process from Wing.

Debugging Non-Python Mainloops

Because of the way the Python interpreter supports debugging, the debug process may become unresponsive if your debug process is free-running for long periods of time in non-Python code, such as C or C++. Whenever the Python interpreter is not called for long periods of time, messages from Wing IDE may be entirely ignored and the IDE may disconnect from the debug process as if it had crashed. This primarily affects pausing a free-running program or setting, deleting, or editing breakpoints while free-running.

Examples of environments that can spend significant amounts of time outside of the Python interpreter include GUI kits such as Gtk, Qt, Tkinter, wxPython, and some web development tools like Zope. For the purposes of this section, we call these “non-Python mainloops”.

Supported Non-Python Mainloops

Wing already supports Gtk, Tkinter, wxPython, and Zope. If you are using one of these, or you aren't using a non-Python mainloop at all, then you do not need to read further in this section.

Working with Non-Python Mainloops

If you are using an unsupported non-Python mainloop that normally doesn't call Python code for longer periods of time, you can work around the problem by adding code to your application that causes Python code to be called periodically. For example, under PyQt, add the following code after you have created your `QApplication` and before your call to `exec_loop()`:

```
# Hack to burn some Python bytecode periodically so Wing's
# debugger can remain responsive while free-running
timer = QTimer()
```

```
def donothing(*args):
    for i in range(0, 100):
        x = i
    timer.connect(timer, SIGNAL("timeout()", donothing))
    timer.start(500, 0)
```

Similar code can be crafted in most non-Python mainloop environments.

The alternative to altering your code is to write special plug-in support for the Wing debugger that causes the debug server sockets to be serviced even when your debug program is free-running in non-Python code. The rest of this section describes what you need to know in order to do this.

Non-Python Mainloop Internals

Wing uses a network connection between the debug server (the debug process) and the debug client (Wing IDE) to control the debug process from the IDE and to inform the IDE when events (such as reaching a breakpoint or exception) occur in the debug process.

As long as the debug program is paused or stopped at a breakpoint or exception, the debugger remains in control and it can respond to requests from the IDE. Once the debug program is running, however, the debugger itself is only called as long as Python code is being executed by the interpreter.

This is usually not a problem because most running Python programs are executing a lot of Python code! However, in a non-Python mainloop, the program may remain entirely in C, C++, or another language and not call the Python interpreter at all for long periods of time. As a result, the debugger does not get a chance to service requests from the IDE. Pause or attach requests and new breakpoints may be completely ignored in this case, and the IDE may detach from the debug process because it is unresponsive.

Wing deals with this by installing its network sockets into each of the supported non-Python mainloops, when they are detected as present in the debug program. Once the sockets are registered, the non-Python mainloop will call back into Python code whenever there are network requests pending.

Supporting Non-Python Mainloops

For those using an unsupported non-Python mainloop, Wing provides an API for adding the hooks necessary to ensure that the debugger's network sockets are serviced at all times.

If you wish to write support for a non-Python mainloop, you first need to check whether there is any hope of registering the debugger's socket in that environment. Any mainloop that already calls UNIX/BSD sockets `select()` and is designed for extensible socket registration will work and is easy to support. Gtk and Zope both fell into this category.

In other cases, it may be necessary to write your own `select()` call and to trick the mainloop into calling that periodically. This is how the Tkinter and wxPython hooks work. Some environments may additionally require writing some non-Python glue code if the environment is not already set up to call back into Python code.

Mainloop hooks are written as separate modules that are placed into `src/debug/server` within `WINGHOME`. The module `_extensions.py` also found there includes a generic class that defines the API functions required of each module, and is the place where new modules must be registered (in the constant `kSupportedMainloops`).

Writing Non-Python Mainloop Support

To add your own non-Python mainloop support, you need to:

- 1) Copy one of the source examples (such as `_gtkhooks.py`) found in `src/debug/server`, as a framework for writing your hooks. Name your module something like `_xxxxhooks.py` where `xxxx` is the name of your non-Python mainloop environment.
- 2) Implement the `_Setup()`, `RegisterSocket()`, and `UnregisterSocket()` methods. Do not alter any code from the examples except the code within the methods. The name of the classes and constants at the top level of the file must remain the same.
- 3) Add the name of your module, minus the `.py` to the list `kSupportedMainloops` in `_extensions.py`

Examples of existing support hooks for non-Python mainloops can be found in `src/debug/server` within `WINGHOME`.

If you have difficulties writing your non-Python mainloop hooks, please contact Technical Support via <http://wingware.com/support>. We will be happy to assist you, and welcome the contribution of any hooks you may write.

Debugging Code Running Under Py2exe

Sometimes it is useful to debug Python code launched by an application produced by `py2exe` -- for example, to solve a problem only seen when the code has been packaged by "py2exe", or so that users of the packaged application can debug Python scripts that they write for the app.

When `py2exe` produces the `*.exe`, it strips out all but the modules it thinks will be needed by the application and may miss any required by scripts added after the fact. Also, `py2exe` runs in a slightly modified environment (for example the `PYTHONPATH` environment is ignored). Both of these can cause problems for Wing's debugger, but can be worked around with some modifications to the packaged code, as illustrated in the following example:

```
# Add extra environment needed by Wing's debugger
import sys
import os
extra = os.environ.get('EXTRA_PYTHONPATH')
if extra:
    sys.path.extend(extra.split(os.pathsep))
print sys.path

# Start debugging
import wingdbstub

# Just some test code
print "Hello from py2exe"
print "frozen", repr(getattr(sys, "frozen", None))
print "sys.path", sys.path
print "sys.executable", sys.executable
print "sys.prefix", sys.prefix
print "sys.argv", sys.argv
```

You will need to set the following environment variables before launching the packaged application:

```
EXTRA_PYTHONPATH=\Python25\Lib\site-
packages\py2exe\samples\simple\dist;\Python25\lib;\Python25\dlls
WINGDB_EXITONFAILURE=1
WINGHOME=\Program Files\Wing IDE 2.1
```

To debug, an installation of Python matching the one used by `py2exe` must be present and referenced by the `EXTRA_PYTHONPATH` environment variable. This example assumes the installation of Python 2.5 at `\Python25` was used by `py2exe`.

The directory `\Python25\Lib\site-packages\py2exe\samples\simple\dist` is where `wingdbstub.py` was placed; this can be altered as desired. Also, `WINGHOME` should be altered to match the location where Wing is installed and isn't needed at all if the value set in `wingdbstub.py` is correct (which it usually will be if copied out of the Wing installation).

When trying this out, be sure to **Enable Passive Listen** in Wing IDE by clicking on the bug icon in the lower left of the main window. For more information on using `wingdbstub` to debug, see **Debugging Externally Launched Code**

Enabling End Users to Debug

The above example is geared at the primary developers trying to find bugs in packaged code. If the packaged application is one that allows the end user to write add-on scripts and they want to debug these in Wing's debugger, then the `import wingdbstub` in the above example should be replaced with the following imports:

```
import socket
import select
import traceback
import struct
import cPickle
import site
import string
```

This forces `py2exe` to bundle the modules needed by Wing's debugger with the `.exe`, so that the end user can place `include wingdbstub` in their scripts instead.

Of course it's also possible to conditionally include the `import wingdbstub` in the main code, based on an environment variable or checking user settings in some other way. For example:

```
import os
if os.environ.has_key('USE_WING_DEBUGGER'):
    import wingdbstub
```

A combination of the above techniques can be used to craft debugging support appropriate to your particular `py2exe` packaged application.

The above was tested with `py2exe` run with `-q` and `-b2` options.