How-Tos
Wing IDE Professional

Wingware

www.wingware.com

Version 4.1.13

May 3, 2013

This is a collection of HOW-TOs designed to make it easier to get started using Wing on certain platforms with special requirements and with specific tools and libraries for GUI, web, and other types of development.

Contents

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**, **Turbogears**, **Django**, **wxPython**, **PyGTK PyQt**, **Debugging Web CGIs**, **mod_python**, **PyGame**, and **many others**

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.

Install Python and Wing IDE

Both Python and Wing IDE must be installed. The Wing IDE executable is called `wing4.1`. 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` and other items in the `Project` menu to add source files to the project.

- Use `Project Properties` in the `Project` menu to specify `Python Path` and in some cases 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. It is also used for the debugger.

- 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. If not, please let us know.

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:

- *Configurable Key Bindings* -- Wing can emulate Visual Studio, VI/Vim, Emacs, and Brief key bindings, selected with the editor **Personality** preference.

- *Configurable Tab Key* -- Wing maps the tab key to Indent To Match by default for Python and Increase Indent for other files. This can be changed from the **Tab Key Action** preference.

- *Auto-completion and Source Assistant* -- Wing's editor and the **Source Assistant** tool provide context-appropriate completion options and documentation as you edit your code. The auto-completer uses Tab by default for completion, but this can be changed in the **Completion Keys** preference

- *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.

- *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.

- *Source Browser* -- Module or class oriented display of the structure of your source code. Show both the **Source Browser** and **Source Assistant** for detailed information about selected symbols.

- *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).

- *Conditional breakpoints* -- A powerful tool in isolating complex bugs by stopping before they occur.

- *Debug Probe* -- This tool provides an interactive Python prompt that executes code in the current stack frame of a paused debug process. The **Debug Probe** is particularly useful in isolating and understanding complex bugs, and designing code that fixes the problem.

- *Watching Values* -- Enter expressions to watch in the **Watch** tool, or right-click on any of the data views to track value over time by symbolic name or object reference.

- *Other Tools* -- Templating / code snippets, bookmarks, macros, unit testing tool, and many other powerful features are available from Wing's menus.

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**

- **How-tos for using Wing with common packages**

Wing IDE for OS X

Wing IDE runs as an X11 application on OS X. If you don't already have one, you will need to obtain and install an X11 server:

(1) **For OS X 10.5 and later**, we strongly recommend XQuartz. The X11 provided on the OS X installation disk contains bugs that affect key bindings, the clipboard, and integration with Spaces.

(2) **For older OS X versions**, the X11 server provided on the OS X installation disk is the best option. This is not installed by default and must be installed separately, usually from Install Disk #2 (but location varies by version). Sometimes the installer is hidden by default and you'll need to scroll down in the Finder window to see it.

Once this is set up, you're ready to install Wing IDE. Just download Wing IDE, double click on the disk image, and drag it to your disk (or, you can just run Wing directly from the disk image).

To start Wing, double click on the Wing IDE app folder.

Starting Wing from the Command Line

The location of Wing's internal application contents folder is referred to as `WINGHOME`. On OS X this is a sub-directory of the app folder visible from the Finder as an application. For example if you unpacked Wing into `/Applications` then `WINGHOME` will be `/Applications/WingIDE.app/Contents/MacOS`.

To start Wing from the xterm command line, execute `wing` located in `WINGHOME` (`/Applications/WingIDE.app/Contents/MacOS/wing` in the above example). If you are not launching from an xterm, you may need to set your `DISPLAY` environment variable to point to your X Server (for example `export DISPLAY=:0.0`).

See the **Tutorial** and **Wing IDE Quickstart Guide** for additional information on getting starting with Wing IDE. To set up Wing for one of the commonly used web and GUI frameworks, see our collection of **How-Tos**.

Keyboard Setup 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 Keyboard Shortcuts under X11" preference in Apple X11 Server configuration (this was called "Enable Key Equivalents under X11" in older versions). 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. The option key should be used only if you don't need the option key to enter individual characters. Note that this preference affects all X11 applications, not just Wing. It may also affect whether the `Custom Key Bindings` preference will accept `Command` as a key modifier.

If OS X intercepts `F` keys (such as `F1`, and `F9`), you may also need to change the settings in the OS X System Settings, under Keyboard/Mouse.

Configuration and Trouble-shooting

For most users, Wing should work out of the box on OS X. The remainder of this document provides some details for further configuration when required, as may be the case primarily OS X versions 10.4 and earlier.

Keyboard Mapping Problems

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 work around 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.

Depending on the version of Apple X11 Server and possibly other factors, any of the keyboard preference changes made in X11 Server configuration or those Wing preferences indicated above as affecting all X11 applications may require restarting both Wing and Apple X11 Server before they work properly.

Clipboard Problems

As noted above, clipboard (copy/paste) confusion can occur if the Apple X11 or XQuartz preference "Enable Keyboard Shortcuts under X11" (or "Enable Key Equivalents under X11" in older versions) is turned on.

On OS X 10.5 with older XQuartz versions you may also see the clipboard get overwritten with the OS X / system clipboard whenever Wing shows a popup window, such as an auto-completer. To prevent this, turn off the Focus on New Windows preference in the X11 Preferences, under the Windows tab.

Debugging 32-bit Python

On 64-bit OS X systems, you can set up a shell script with the following contents and set it as the Python Executable in Project Properties, in order to facilitate debugging Python in 32-bit mode:

```
#!/bin/bash
arch -i386 python "$@"
```

This should only be necessary if your code needs 32-bit libraries. Wing's debugger works in either 64-bit or 32-bit mode.

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.

Note that only themes that do not use a theme engine or use one of those in /bin/gtk-bin/lib/gtk-2.0/2.4.0/engines in your Wing installation will work with Wing IDE. We cannot make any guarantees for performance or results when using themes not included with Wing IDE.

Using other Fonts

The X11-provided set of fonts may not be adequate for all languages and may not be able to display all the needed glyphs. If none of the provided fonts works for you, make a directory `~/.fonts` (`.fonts` in your home directory) and then selectively copy `.ttf` fonts over from `/System/Library/Fonts` and/or `/Library/Fonts`. They should appear in the list of choices when you restart Wing. **Important**: Do this one or a few fonts at a time since some of the `.ttf` fonts will hang Wing at startup.

Finding WINGHOME for WingDBG (Zope)

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.

Configuring Focus

XQuartz provides focus options in X11 preferences under the Windows tab.

If you are using Apple X11 Server, you can configure it from the command line as follows.

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
```

Setting up SSH

If you plan to use revision control in Wing and you connect to your server via ssh and are using OS X 10.4 and earlier, you will either want to launch Wing from the command line after doing `ssh-agent bash` followed by `ssh-add` or make use of a utility like SSHKeyChain. For details see **Setting up SSH**.

Other Known Problems

At least some versions of the MacPorts packaging of Python are known not to work with Wing's debugger because it contains an _md5 module that won't load. To work around this, use a different distribution of Python instead.

The copy of Python in /Library/Python on OS X does not include source files for the standard libraries, so Wing's editor will not offer autocompletion values for those modules. To work around this, use Python from within /Library/Frameworks/Python.frameworks instead or copy of Python installed from the standard source distribution.

In some older versions of XQuartz, the default dots-per-inch (dpi) reported by the X11 server is 96 instead of the earlier default of 72. Depending on your display, this may lead to poor initial font size choices. You can change the font size in Wing's Preferences (in the Edit menu) or in extreme cases where the font is entirely unreadable, you can set the X11 server's reported dpi as follows from the xterm:

```
defaults write org.x.X11 dpi -int 75
```

For additional information on this and other issues, see the XQuartz X11 Users FAQ.

Please send bug reports to bugs at 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 development frameworks.

- **Wing IDE Reference Manual**, which describes Wing IDE in detail.

How-Tos for Web Development

The following How-Tos provide tips and short cuts for using a number of popular web development frameworks with Wing IDE.

3.1. Using Wing IDE with Django

> **"Having to debug Django issues is almost enjoyable with the new debugger support."** -- *Doug Napoleone, Oct 2007*

Wing IDE is an integrated development environment that can be used to write, test, and debug Python code that is written for Django, a powerful web development system. Wing provides auto-completion, call tips, a powerful debugger, and many other features that help you write, navigate, and understand Python code.

Wing IDE can also be used to step through and debug Django templates, and it includes Django-specific plugin functionality to make it easier to create Django projects and apps, set up Wing projects for use with Django, and manage routine tasks. The debugger can be configured to launch Django from the IDE and to reinitiate automatically when Django reload occurs.

For more information on Wing IDE see the product overview. If you do not already have Wing IDE installed, download a free trial now. To get started using Wing, refer to the tutorial in the `Help` menu in Wing and/or the **Wing IDE Quickstart Guide**.

Installing Django

The Django website provides complete instructions for installing Django.

Quick Start with Wing IDE Professional

If you have Wing IDE Professional, the fastest way to get started using Wing IDE with Django is to use the provided Django extensions. If you have Wing IDE Personal, skip ahead to the Manual Configuration section below.

Existing Django Project

To set up a Wing IDE Professional project for an *existing* Django project:

(1) Create a new project from the `Project` menu,

(2) Add the Django site directory to the Wing project (so that `manage.py` and `settings.py` are both in the project),

(3) Wait until the"Django" menu appears in the menu bar, and

(4) Select the `Configure Project for Django` item from that menu.

This sets the Python Executable in Project Properties (if it can be located), sets up `manage.py runserver --noreload 8080` as the main debug file, adds `DJANGO_SITENAME` and `DJANGO_SETTINGS_MODULE` to the environment in Project Properties, adds the site directory to the Python Path in the Wing project, ensures **Django Template Debugging** in Project Properties is enabled, turns on `TEMPLATE_DEBUG` in your site's `settings.py` file (debugging templates will not work without this), and sets the `Default Test Framework` in the `Testing` tab of `Project Properties` so that Wing's Testing tool will invoke `manage.py test`.

Now you should be able to start Django in Wing IDE's debugger, set breakpoints in Python code and Django templates, and reach those breakpoints in response to a browser page load.

New Django Project

If you are starting a new Django project at the same time as you are setting up your Wing IDE project:

(1) Select `Start Django Project` from the `Extensions` sub-menu of the `Project` menu.

(2) You will be prompted for the location of `django_admin.py`, location to place the new project, and the site name in the same entry area. Defaults for these values are based on the current project contents, if a Django project is already open.

(3) Press Enter and Wing will set up a new Django project and your Wing IDE project at the same time.

This runs `django_admin.py startproject <sitename>`, sets up `settings.py` to use `sqlite3` default database engine, adds `django.contrib.admin` to `INSTALLED_APPS`, runs `syncdb`, and copies the default admin template `base_site.html` from your Django installation into your site's `templates/admin` directory.

Note that on Windows you will see an error that the superuser account could not be set up. The error includes the command that needs to be run interactively. To complete project creation, copy/paste this into a command console.

When project setup is completed, the command offers to creates a new Wing IDE project, add the files, and configure the project for use with Django as described in the Existing Django Project sub-section above.

Notes on Debugger Configuration

(1) Debugging Exceptions

Django contains a catch-all handler that displays exception information to the browser. When debugging with Wing, it is useful to also propagate these exceptions to the IDE. This can be done with a monkey patch as follows (for example, in `local_settings.py` on your development systems):

```
import os
import sys

import django.views.debug

def wing_debug_hook(*args, **kwargs):
    if __debug__ and 'WINGDB_ACTIVE' in os.environ:
        exc_type, exc_value, traceback = sys.exc_info()
        sys.excepthook(exc_type, exc_value, traceback)
    return old_technical_500_response(*args, **kwargs)

old_technical_500_response = django.views.debug.technical_500_response
django.views.debug.technical_500_response = wing_debug_hook
```

The monkey patch only activates if Wing's debugger is active and assumes that the **Report Exceptions** preference is set to `When Printed`.

(2) Debugging with Reload Enabled

The default configuration of the debugger launches Django from Wing IDE and disables auto-reload, so the restart button in Wing must be pressed before changes take effect in the debug process. To launch from outside the IDE and to handle auto-reload automatically see the Manual Configuration section below.

Django-specific Actions

The `Django` menu that auto-activates for Django projects also contains special actions for starting a new Django app (this creates the app and adds it to `INSTALLED_APPS` in `settings.py`, running sync db, generating SQL for a selected app, and running validation checks. This menu also provides convenient access to this How-To.

This functionality is implemented as an open source plugin that can be found in `scripts/django.py` in your Wing IDE installation directory. This code can be usermodified by working from the existing functionality as examples. For detailed information on writing extensions for Wing IDE, see the **Scripting and Extending Wing IDE** chapter.

Debugging Django Templates

The above-described project setup scripts enable template debugging automatically. If you configured your project manually, you will need to take the following two steps to enable template debugging:

(1) Set `TEMPLATE_DEBUG` to `True` in your Django application's `settings.py` file,

(2) Be sure that Wing IDE's **Enable Django Template Debugging** setting in your project's properties is enabled. When you change this property, you will need to restart your Django debug process if one is already running.

Once this is done, you should be able to set breakpoints in any file that contains `{%%}` or `{{}}` tags, and the debugger will stop at them.

When debugging Django templates is enabled, Wing will replace the Python stack frames within the template invocation with frames for the template files, and the locals shown in the Stack Data tool will be extracted from the template's runtime context. When

working in a template stack frame, the Debug Probe, Watch, and other tools will operate in the environment that is displayed in the Stack Data tool.

Note that stepping is tag by tag and not line by line, but breakpoints are limited to being set for a particular line and thus match all tags on that line.

Stepping in the debugger while a template invocation is active will be limited to templates and any user code or code within the `contrib` area of your Django installation. If you need to step into Django internals during a template invocation, you will need to disable Django template debugging in your project properties, set a breakpoint at the relevant place in Django, and restart your debug process.

Notes on Auto-Completion

Wing provides auto-completion on Python code and Django templates. The completion information is based on static analysis of the files unless the debugger is active and paused and the template or Python code being edited are on the stack. In that case, Wing sources the information shown in the auto-completer and Source Assistant from live runtime state. As a result, it is often more informative to work with the debugger a breakpoint, particularly in Django templates where static analysis is not as effective as it is in Python code.

Running Unit Tests

In Wing IDE Professional, the `Default Testing Framework` under the `Testing` tab of `Project Properties` can be set to `Django Tests` to cause the `Testing` tool in Wing to run `manage.py test` and display the results. Particular tests can be debugged by selecting them and using `Debug` in the `Testing` menu (or right-clicking on them).

If unit tests need to be run with a different settings.py file, the environment variable `WING_TEST_DJANGO_SETTINGS_MODULE` can be set to replace `DJANGO_SETTINGS_MODULE` when unit tests are run.

Django with Buildout

When using Django with buildout, Wing won't auto-detect your project as a Django project because the `manage.py` file is instead named `bin/django`. To get it working, copy `bin/django` to `manage.py` in the same directory as `settings.py`.

Manual Configuration

This section describes manual configuration of Wing IDE projects for use with Django. If you are using Wing IDE Professional, see the above Quick Start for Wing IDE Professional.

Configuring the Project

To get started, create a new project from the `Project` menu, add your files, and determine if the correct Python is being found by displaying the `Python Shell` tool in Wing. If the wrong Python is being used, alter the `Python Executable` in Project Properties (in the `Project` menu) and restart the shell from its `Options` menu.

You may also want to set the `DJANGO_SITENAME` and `DJANGO_SETTINGS_MODULE` environment variables in Project Properties.

Configuring the Debugger

By default, Django runs in an environment that spawns and automatically re-launches a sub-process for servicing web requests. This is used to automatically restart the server if for some reason it crashes. However, this does not work with Wing's debugger if you launch Django from Wing, since the debugger has no way to cause the sub-process to be debugged when it is started by the main process.

There are two ways to debug Django code: Either configure Django so it can be launched by Wing's debugger, or cause Django to attach to Wing on the fly as code that you wish to debug is executed.

Launching from Wing

When Django is launched from Wing, it must be configured to avoid auto-reload and other options that break the debugger.

Newer versions of Django include a `--noreload` option that you can pass to the `manage.py` or `django-admin.py` scripts to turn off the auto restart feature of Django.

The typical way to do all this is to set the `manage.py` file as the primary debug file in Wing and give it the following `Run Arguments` via the Debug properties tab in the **File Properties** dialog:

```
runserver --noreload
```

Other options can be added here as necessary for your application.

Some older versions of Django may also require adding `--settings=devsettings` to the arguments for `runserver`, in order for debugging to work. If Wing is not be able to stop on any breakpoints, try adding this.

In much older versions of Django, the `--noreload` option does not exist. The only way to solve the problem there is to make a modification to the code to prevent launching of a sub-process, or to use the alternative method described below.

Debugging with Auto-reload

Another method of debugging Django is to use `wingdbstub.py` to initiate debugging when Django is started from outside of Wing IDE. This will allow Django to restart automatically after code changes and to automatically reconnect the debugger to Wing IDE as needed.

This is done by placing a copy of `wingdbstub.py`, which is located in your Wing IDE installation directory, in the top of the Django directory, where `manage.py` is located. Make sure that `WINGHOME` is set inside `wingdbstub.py`; if not, set it to the location of your Wing IDE installation (or on OS X, to the `Contents/MacOS` folder within Wing's `.app` folder). This allows the debug process to find the debugger implementation.

Next, place the following code into files you wish to debug:

```
import wingdbstub
```

Then make sure that the **Enable Passive Listen** preference is enabled in Wing and start Django. The Django process should connect to Wing IDE and stop at any breakpoints set after the `import wingdbstub`.

When code is changed, just save it and Django will restart. The debugger will reconnect to Wing IDE once you request a page load in your browser that leads to one of your `import wingdbstub` statements.

Related Documents

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

- **Wing IDE Reference Manual**, which describes Wing IDE in detail.

- Django home page, which provides links to documentation.

- **Wing IDE Quickstart Guide** which contains additional basic information about getting started with Wing IDE.

3.2. Using Wing IDE with the Google App Engine

Wing IDE is an integrated development environment that can be used to write, test, and debug Python code that is written for the Google App Engine. Wing provides auto-completion, call tips, a powerful debugger, and many other features that help you write, navigate, and understand Python code. Since Google App Engine will reload your code when you save it to disk, you can achieve a very fast edit/debug cycle without restarting the debug process.

For more information on Wing IDE see the product overview. If you do not already have Wing IDE installed, download a free trial now. To get started using Wing, refer to the tutorial in the `Help` menu in Wing and/or the **Wing IDE Quickstart Guide**.

Configuring the Environment

Before trying to configure a Wing IDE project, first install and set up Google App Engine and verify that it is working by starting it outside of Wing IDE and testing it with a web browser. It is also a good idea to install App Engine upgrades at this time, before doing anything else.

Next create a project in Wing using `New Project` in the `Project` menu and use `Add Directory` in the `Project` menu to add your source directories to the project. This must include `dev_appserver.py`, which is located in the top level of the Google SDK directory.

Then open up `dev_appserver.py` in Wing's editor and select `Set Current as Main Debug File` in the `Debug` menu. This tells Wing to use this file as the main entry point when you start debugging, and it also helps Wing infer your Python Path and other settings. The main debug file is highlighted in red in the Project tool. If a main debug file is already defined the `Debug` menu item will be `Clear Main Debug File` instead.

Finally, save your project with `Save Project` in the `Project` menu. Store the project at or near the top level of your source tree.

Using Django

If you are using the Django via the `app-engine-patch` project, you need to use `man-age.py` as your main debug file instead of `dev_appserver.py` and should follow the Manual Configuration instructions in the **Django How-To** (except that the `--noreload` option is not supported).

Debugging Your Application

Before trying to debug make sure you stop Google App Engine if it is running already outside of Wing IDE.

You can debug code running under the Google App Engine by selecting `Start / Continue` from the `Debug` menu (or using the green run icon in the toolbar). This will bring up a dialog that contains a `Run Arguments` field that must be altered to specify the application to run. For example, to run the guestbook demo that comes with the SDK, the run arguments would be `"${GOOGLE_APPENGINE_DIR}/demos/guestbook"` where `${GOOGLE_APPENGINE_DIR}` is replaced by the full pathname of the directory the SDK is installed in. The quotation marks are needed if the pathname contains a space. In other apps, this is the directory path to where the `app.yaml` file is located. If this path name is incorrect, you will get an `AppConfigNotFoundError` exception when you start debugging.

You can also leave the environment reference `${GOOGLE_APPENGINE_DIR}` in the path and define an environment variable under the `Environment` tab of the `Debug` dialog. Or use `${WING:PROJECT_DIR}` instead to base the path on the directory where the project file is located.

Add a `--port=8082` style argument if you wish to change the port number that Google App Engine is using when run from Wing's debugger. Otherwise the default of `8080` will be used.

Using a partial path for the application may also be possible if the `Initial Directory` is also set in under the `Debug` tab.

Next, click the `OK` button to start debugging. Once the debugger is started, the Debug I/O tool (accessed from the `Tools` menu) should display output from App Engine, and this should include a message indicating the hostname and port at which App Engine is taking requests. Requests may be made with a web browser using that URL. If Google App Engine asks to check for updates at startup, it will do so in the Debug I/O tool and you can press "y" or "n" and then Enter as you would on the command line.

To try out the debugger, set a break point in any Python code that is executed by a request and load the page in the browser. For example, to break when the main page

of the guestbook demo is generated, set a breakpoint in the method `Mainpage.get` in `guestbook.py`. When you reach the breakpoint, the browser will sit and wait while Wing displays a red run marker on code at the breakpoint and other lines as you step through code using the buttons in Wing IDE's toolbar.

Check out the `Stack Data` and `Watch` tools in the `Tools menu` to inspect debug data, or just use the `Debug Probe`, which is an interctive Python shell that works in the context of the current debug stack frame. When the debug process is paused, both the `Debug Probe` and editor show auto-completion and call tips based on live runtime state, making it quick and easy to write and try out new code.

Continuing with the green run button in the toolbar will complete the page load in the browser, unless some other breakpoints or an exception are reached first.

You may edit the Python code for an application while the App Engine is running, and then reload in your browser to see the result of any changes made. In most cases, there is no need to start the debug process after edits are made.

Configuration Details

Windows users will want to set the `TZ` environment variable to `UTC` via the environment field in Project Properties to work around problems with setting `os.environ['TZ']` while a process is running (see **Issues on Microsoft Windows** for details about the Windows runtime bug). One possible symptom of this is repeated 302 redirects that prevent logging in or other use of the site.

If you need to send command line arguments to app engine (for example, to change the port it listens on), these should be added to `Run Arguments` under the `Debug tab` of File Properties for `dev_appserver.py`. You can access this dialog by right clicking on the editor and selecting `Properties`.

The `Debugger > Exceptions > Report Exceptions` preference should be set to `When Printed` (the default) when working with Google App Engine or Wing will report some additional exceptions that are handled internally when running Google App Engine outside of the debugger.

Improving Auto-Completion and Goto-Definition

Wing can't parse the `sys.path` hackery in more recent versions of Google App Engine so it may fail to find some modules for auto-completion, goto-definition and other features. To work around this, set a breakpoint in `fix_sys_path` in `dev_appserver.py` and start debugging. Then, in the `Debug Probe` tool (in Wing Pro only) type the following:

```
os.pathsep.join(EXTRA_PATHS)
```

Copy this to the clipboard and open up the file properties for `dev_appserver.py` by right-clicking on the file. Then, under the `Environment` tab select `Custom` for the `Python Path`, click on the `View as Text` button and paste in the extra path.

You will need to redo this if you move the app engine installation, or you can use `${WING:PROJECT_DIR}` to convert those paths to base on the location of the project file.

Improving Responsiveness when Setting Breakpoints or Pausing

Out of the box, earlier versions of Google App Engine would sit in a `select` call for up to thirty seconds when it has nothing to do. This results in delays in setting breakpoints or pausing the debug process. To speed this up, go into `tools/dev_appserver.py` and change `DEFAULT_SELECT_DELAY` to a shorter value, such as `1.0`. This is longer needed in newer versions of Google App Engine.

Related Documents

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

- **Wing IDE Reference Manual**, which describes Wing IDE in detail.

- Google App Engine home page, which provides links to documentation.

- **Wing IDE Quickstart Guide** which contains additional basic information about getting started with Wing IDE.

3.3. Using Wing IDE with Pyramid

Wing IDE is an integrated development environment that can be used to write, test, and debug Python code that is written for Pyramid, a powerful web development system. Wing provides auto-completion, call tips, a powerful debugger, and many other features that help you write, navigate, and understand Python code.

For more information on Wing IDE see the product overview. If you do not already have Wing IDE installed, download a free trial now. To get started using Wing, refer to the tutorial in the `Help` menu in Wing and/or the **Wing IDE Quickstart Guide**.

Installing Pyramid

Please see the Pyramid website (part of the Pylons project), which provides complete instructions for installing the Pyramid framework. The procedure varies slightly by OS.

Like any Python package, Pyramid will install itself using whichever instance of Python runs the installer script. You should be using a Python version at least 2.6.

Pyramid projects are typically installed inside of a virtualenv, to maintain a "sandboxed" installation separate from your main Python installation. This allows Python packages that you install as part of your Pyramid project to be kept entirely separate from your system's main Python environment, and from any other virtualenvs that you may have. Creating or removing a virtualenv is just a couple of file system commands, so it's easy and quick to start a new one just to test an alternative configuration of your project. This makes it very easy to test "what-if" scenarios based on installing different versions of the packages relied upon by your project. For example, you could use a new virtualenv if you wanted to try serving your app using a newly released version of your ORM layer or your templating engine, or a newly released or beta version of Pyramid itself.

This How-To was developed with Pyramid version 1.3.

Configuring your Wing IDE Project

This section assumes your Pyramid project is called 'project' and is installed in a virtualenv at `.../project` where `...` is the full path to the location of your project. We also assume that you are running Wing IDE, that you have your current Wing Project open and saved as `.../project/project.wpr` (or whatever you chose to name your project).

Make sure that your Pyramid project directory (which should be the same as your virtualenv) is added to your Wing project with `Add Directory` in the `Project` menu, and that you have saved the project. There is no need to add the entire `.../project` directory to the Wing project, as that would include the entire `project/bin` area. Typical Pyramid project structure looks like `project/Project/project`. The Project (upper case) directory holds setup and README information for your project and the configuration files, ending in `.ini`, which allow you to start your project's server with different settings.

Ordinarily you'll have `project/Project/development.ini` which contains the settings (including enabling lots of logging, etc) that you run during development activities, and `project/Project/production.ini` which contains different settings (turning off most logging and any development-related security vulnerabilities such as open administrative access) that you'll use in production. But you can also create additional `.ini` files for

any purpose, such as when you want to simulate serving your project under different conditions, e.g. connecting to a different database server.

The one file you'll need to add to your Wing project from the `.../project` level of your directory structure is `.../project/bin/pserve`. Then open it in Wing and set it as Main Debug File from the `Debug` menu.

Next open up the `Python Shell` tool and type `import sys` followed by `sys.executable` to check whether Wing is using the Python that will be running the Pyramid server. If not, verify that the shell's status message does not indicate that it needs to be restarted to load configuration changes. If this message is present, restart the shell from its `Options` menu and try again. If the message is not present, open `Project Properties` and set the `Python Executable`, then restart the shell again and verify that `sys.executable` is correct. The interpreter used in this step will vary depending on whether your `.../project` directory is enabled as a virtualenv or not.

Once this is done, Wing's source analysis engine should be able to find and analyze your code and Pyramid. Analysis status messages may show up in the lower left of Wing's main window while analysis is in progress.

Debugging

To debug code running under Pyramid, place a copy of `wingdbstub.py`, which is located in your Wing IDE installation directory, in your `project/Project` directory, the same directory that holds your `.ini` files and which is set as the `Initial Directory` for your Wing project. Near the top of any Python file you wish to debug, place the following line:

```
import wingdbstub
```

Also click on the bug icon in the lower left of the main window and make sure that `Enable Passive Listen` is checked.

Then set a breakpoint on any location in your project's code that you know will be reached when an HTTP or AJAX request is made to your server, depending on what user actions in the browser you intend to follow with debugging. A common breakpoint location would be in one of what Pyramid calls your View Callables, which are the Python classes and/or methods called by the webserver depending on the URL and other parameters of the request. Or, if you need to debug lower levels of the stack, you can set breakpoints in the Pyramid source files themselves, or in the source of any other package (such as your ORM or template rendering system) that supports the handling of your web requests.

With a terminal window open, start your Pyramid server as you usually would, by issuing the command:

```
pserve --reload development.ini
```

from within your `project/Project` directory. `--reload` is a convenient option that restarts the server whenever you've saved any changes to your Pyramid project's source files. You don't have to use it, but Wing's debugger is still able to attach and operate correctly if you do. If you are using a different `.ini` file such as a `production.ini` or `testing.ini`, supply its name to pserve instead.

Load `http://localhost:5000/` or the page you want to debug in a browser. The port that your server uses (5000 in this example) is set in your `.ini` file, in a section that looks like the following:

```
[server:main]
use = egg:waitress#main
host = 0.0.0.0
port = 5000
```

Wing should stop on your breakpoint. Be sure to look aroung a bit with the `Stack Data` tool, and in Wing Pro the `Debug Probe` (a command line that works in the runtime state of your current debug stack frame). All the debugging tools are available from the `Tools` menu, if not already shown.

Notes on Auto-Completion

Wing provides auto-completion on Python code and within basic HTML elements, and can help a lot within the various templating languages that can be used in a Pyramid project.

The autocomplete information available to Wing is based on static analysis of your project files and any files Wing can find on your Python Path or via imports in other Python files.

Additionally, when the debugger is active and paused, Wing usess introspection of the live runtime state for any template or Python code that is active on the stack. As a result, it is often more informative to work on your source files while Wing's debugger is active and paused at a breakpoint, exception, or anywhere in the source code reached by stepping.

Debugging Mako Templates

A good choice of templating engine for the Pyramid projects of a Wing IDE user is Mako, because it allows the full syntax of Python in expression substitutions and control structures and this maximizes Wing's ability to help out. Mako templates are simply marked-up HTML files, and as such they cannot be directly stepped through using the debugger. However, they are compiled to `.py` files whenever the source file is altered, and you can set Wing debug breakpoints in the `.py` files corresponding to your templates.

Debugging Mako templates with Wing IDE requires one optional setting that can be made in your `.ini` file, usually `development.ini`. Under the `[app:main]` section, add the following line:

```
mako.module_directory=%(here)s/data/templates
```

This location will exist in most typical Pyramid projects. If yours does not have it you can create it, or point the setting to an existing location of choice. Without this setting (by default), mako templates are compiled in memory and not cached to disk. With this setting in place, your mako templates will be compiled to actual `.py` files in the desired location, with the same filename as the original template plus the `.py` extension appended to the end.

You should be able to set breakpoints within these `.mako.py` files just like anywhere else in your project. If necessary, add the following at the top of the template file:

```
<%! import wingdbstub %>
```

This uses mako's module-level import facility to drop the import directly into the compiled `.mako.py` file, and will prevent the import from disappearing when a template is automatically recompiled after its source file is changed.

Your `.mako.py` files will not be in one-to-one line correspondence with their `.mako` source files, but mako inserts tracking comments indicating original source line numbering.

Debugging without wingdbstub.py (experimental)

In some cases it may be more convenient to debug your Pyramid project files by launching your Pyramid server directly from Wing, rather than using `wingdbstub.py` as described above. In this approach, you use the `Debug Start` or `Restart` commands to start and restart your server, instead of launching it on the command line outside of Wing.

To try this, verify that you have set the `Main Debug File` to `.../project/bin/pserve` by opening the file, and selecting `Set Current as Main Debug File` from the `Debug` menu.

Then right click on the `pserve` file in the editor or Project tool and use `Properties...` to set its `Run Arguments` to `development.ini` or whatever `.ini` file you want to use with debugging, and then set the `Initial Directory` property to `.../project/Project` or wherever your `.ini` files are located.

Make sure that the `--reload` option is *not* supplied in the `Run Arguments` that you configure, as this will interfere with the debugger. You will need to press the restart debugging icon in the toolbar or select `Restart Debugging` from the `Debug` menu to restart the Pyramid server after making changes to Python files or templates.

Once this is done, press the green `Debug` icon in the toolbar or use `Start/Continue` in the `Debug` menu to start debugging. The `Debug I/O` tool in Wing, available in the `Tools` menu, will display any output from the server.

Related Documents

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

- **Wing IDE Reference Manual**, which describes Wing IDE in detail.

- Pyramid documentation

- **Wing IDE Quickstart Guide** which contains additional basic information about getting started with Wing IDE.

Thanks to Eric Ongerth for providing the initial version of this How-To.

3.4. 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 that can be used to write, test, and debug Python code that is written for Plone, a powerful web content management system. Wing provides auto-completion, call tips, debugger, and many other features that help you write, navigate, and understand Python code.

For more information on Wing IDE see the product overview. If you do not already have Wing IDE installed, download a free trial now.

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

Introduction

The instructions below are for the Plone 4 unified installer. If you are using an older version of Plone or use a source installation of Plone 4 that makes use of old style Products name space merging, please refer instead to the instructions for **Using Wing IDE with Zope**.

Note: We strongly discourage running Wing or development instances of Plone as root or administrator. This creates unnecessary security risks and will cause debugger configuration problems.

Configuring your Project

To set up your project, simply set the `Main Debug File` in `Project Properties` to the file `zinstance/bin/instance` within your Plone installation. This may instead be `zeocluster/bin/client1` with a ZEO install, or whatever name is given in the `.cfg` file. Wing will read the `sys.path` updates from that file so that it can find your Plone modules.

You may also need to set `Python Executable` in `Project Properties` (accessed from the `Project` menu) to the Python that is used in your Plone instance. For example, in a standalone install this may be `Python2.6/bin/python` or similar. The full path can be found by looking at the top of many of the scripts in `zinstance/bin` or `zeocluster/bin`.

For Plone 4, do **not** use the Zope2 support in `Project Properties` under the `Extensions` tab. This is not needed unless your Plone installation still uses old style Product name space merging.

Debugging with WingDBG

There are two ways to configure debugging. The method described in this sub-section uses a Zope control panel to turn debugging on and off and will debug only requests to a particular debug port. This is the most common way in which Plone is debugged with Wing IDE.

To get debugging working install WingDBG, the Wing debugger product, from zope/WingDBG-4.1.13.tar in your Wing installation by unpacking it into `zinstance/products` (or `zeocluster/products` in a zeo install).

Then edit your `etc/zope.conf` to change `enable-product-installation off` at the end to instead read `enable-product-installation on`. In a zeo install this file is located at `zeocluster/parts/client1/etc/zope.conf`.

Finally, click on the bug icon in the lower left of the IDE window and turn on `Enable Passive Listen` so the debugger listens for connections initiated from the outside.

Then start Plone and go into the Zope Management Interface from http://localhost:8080/ , click on `Control Panel`, and then on `Wing Debug Service` at the bottom. From here you can turn on debugging. The bug icon in lower left of Wing IDE's window should turn green after a while and then any page loads via port 50080 (http://localhost:50080/) will be debugged and will reach breakpoints. This port and other debugger options are configurable from the WingDBG control panel.

WingDBG in buildout-based Plone installations

In some new buildout-based Plone settings, WingDBG will not load until the `buildout.cfg` (generated by the template `plone4_buildout`) is edited to add the following just above `[zopepy]`:

```
products = ${buildout:directory}/products
```

Then rerun `bin/buildout -N` which will add a line like the following to your `parts/instance/etc/zope.conf` file:

```
products /path/to/your/products''
```

You will also need to add the specified products directory manually, and then place WingDBG in it.

WingDBG as an Egg

Encolpe Degoute has been maintaining a version of WingDBG that is packaged as an egg.

Creating an egg yourself is also possible as follows:

```
paster create -t plone Products.WingDBG
```

Then copy WingDBG/* to Products.WingDBG/Products/WingDBG.

Debugging Plone from the IDE

It is also possible to debug Plone without `WingDBG` by launching Plone directly from the IDE. This technique may be more convenient in some cases, and debugs all requests to the Plone instance (not just those on a special debug port). However, it takes more effort to set up and requires modifying the Plone sources.

To debug this way, set `zinstance/bin/instance` (or `zeocluster/bin/client1` in a zeo install) in your Plone installation as the `Main Debug File` in `Project Properties` (this should already be done from configuring your project earlier). Then right click on the file in the editor or Project view, select `Properties`, and set `Run Arguments` under the `Debug` tab to `fg`.

Then open the file `ctl.py` (search your Plone install to find it) and change the `AdjustedZopeCmd.doforeground` method by adding the following code before the last if/else block:

```
import os
if os.environ.has_key('WINGDB_ACTIVE'):
    p = os.path.dirname(command[1])
    sys.path.append(p)
    import run
    sys.argv = command[1:]
    run.run()
    return
```

This prevents Plone from running a sub-process, which breaks debugging because only the original parent process is debugged.

Note that this solution can take more time to launch than debugging with WingDBG since the entire startup process is debugged.

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.

3.5. Using Wing IDE with Zope

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

Wing IDE is an integrated development environment that can be used to develop, test, and debug Python code running under Zope2 or Zope3. Wing provides auto-completion, call tips, and other features that help you write, navigate, and understand Zope code. Wing's debugger can be used to debug code in the context of the running Zope server, in response to page loads from a browser, and can work with Zope's code reloading features to achieve a very short edit/debug cycle.

Wing's code intelligence and debugging support work with Products, External Methods, file system-based Scripts and Zope itself. Wing IDE is also useful for Zope-based frameworks like Plone (see **Plone Quickstart**).

For more information on Wing IDE see the product overview. If you do not already have Wing IDE installed, download a free trial now.

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

Before Getting Started

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.

Upgrading from earlier Wing versions

If you are upgrading from an older version of Wing and have previously used Wing with your Zope installation(s), you need to manually upgrade `WingDBG` in each Zope instance. Otherwise, debugging may fail.

The easiest way to do this is to go to the Zope Control Panel, click on `Wing Debug Service`, and then `Remove` the control panel. Then restart Zope. Next, go into your Wing project's `Extension Tab`, verify that you've got the `Zope Instance Home` set correctly, and press `Apply`. This will offer to re-install `WingDBG` with the latest version and will configure it to point to the new version of Wing.

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. If your zope instance is generated by buildout, set the main debug file to the `bin/instance` file (`bin\instance-script.py` on Windows) in your buildout tree by opening the file in Wing and select `Set Current as Main Debug File` in the `Debug` menu. This will set up the effective sys.path for the instance.

- **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. If the Wing Debugging Service does

not appear in the Control Panel, you may need to enable product loading in your zope.conf file by changing `enable-product-installation off` to `enable-product-installation on`.

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` (Wing Pro only) -- lets you interact with your paused debug process using a Python shell prompt
- `Watch` (Wing Pro only) -- 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` (Wing Pro only) -- 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 `Start / 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. Control-click on a source symbol to jump to its point of definition (or use Goto Selected Symbol in the Source menu). Wing Pro also includes a Source Assistant and Source Browser. The Source Assistant will display context appropriate call tips and documentation. 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.

Alternative Approach to Reloading

The `refresh.txt` techique for module reloading is discouraged in the Plone community. Another option for reloading both Zope and Plone filesystem-based code is `plone.reload` available from pypi at http://pypi.python.org/pypi/plone.reload. `plone.reload` will allow you to reload Python code that has been changed since the last reload, and also give you the option to reload any `zcml` configuration changes.

If you are using `buildout`, add `plone.reload` to the eggs and zcml sections of your `buildout.cfg` and re-run buildout.

To use `plone.reload`, assuming Zope is running on your local machine at port 8080, log into the ZMI as a Manager user, then go to http://localhost:8080/@@reload on your Zope instance with a web browser (append `@@reload` to the Zope instance root, not your Plone site if you are using Plone).

Notes:

- If you are using Plone, your Plone product's profile config files (*.xml files) get loaded through the ZMI at `/YourPlone/portal_setup` in the `import` tab.

- Code that uses a `@decorator` will still likely require a restart.

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 the 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 by building them from source code (contact Wingware for details).

- **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 `e:/Zope`. For more information on this, see **File Location Maps** and **Location Map Examples** in the Wing IDE manual.

- **Set the Zope Host** -- Go into Project Properties and set the Zope Host to match the host name used in configuring the File Location Map in the previous step. This is used to identify which host mapping should be applied to file names read from the `zope.conf` file.

- **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:

- If Zope or Plone on Windows is yielding a Site Error page with a notFoundError when run under Wing's debugger, you may need to go into the Zope Management Interface and delete the access rule (... accessRule.py ...). Now, Zope/Plone runs on port 8080, does not alter the configuration of port 80, and will work properly with Wing's debug port (50080 by default). If the URL for your front page is http://localhost:8080/default/front-page, the Wing IDE debug url will always be the same but with the different port: http://localhost:50080/default/front-page (Thanks for Joel Burton for this tip!)

- 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.

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.

3.6. Using Wing IDE with Turbogears

Wing IDE is an integrated development environment that can be used to develop, test, and debug Python code that is written for Turbogears, a powerful web development system. Wing provides auto-completion, call tips, a powerful debugger, and many other features that help you write, navigate, and understand Python code.

For more information on Wing IDE see the product overview. If you do not already have Wing IDE installed, download a free trial now. To get started using Wing, refer to the tutorial in the `Help` menu in Wing and/or the **Wing IDE Quickstart Guide**.

In order to debug Turbogears applications, you will need Wing 3.0 or later, since earlier versions did not support multi-threaded debugging.

> Note that some parts of this document are for Turbogears 1.x only, and others (as indicated) for Turbogears 2.x only.

Installing Turbogears

The Turbogears website provides complete instructions for installing Turbogears. The procedure varies slightly by OS. See also the Notes section below.

Configuring Turbogears 1.x to use Wing

This section assumes your Turbogears 1.x project is called `wingtest`. If not, substitute your project name in the following instructions.

- Go into the Turbogears instance directory `wingtest` and run Wing

- Add your instance directory to the project and save it as `wingtest.wpr` There is no need to add all of Turbogears to the project; just the instance should suffice.

- Open `start-wingtest.py` in Wing and set it as main debug file from the `Debug` menu

- Edit `start-wingtest.py` and add the following before the server is started:

      import os
      import cherrypy
      if os.environ.has_key('WINGDB_ACTIVE'):
          cherrypy.config.update({'autoreload.on': False})

This is needed to prevent creation of a sub-process controlled by the auto-restarter, which breaks debugging since Wing's debugger will not be running in the sub-process. If you omit this step, the symptom will be failure to stop on any breakpoints in your Turbogears application.

- Set a breakpoint on the `return` line of `Root.index()` in your `controllers.py` or somewhere else you know will be reached on a page load

- Start debugging in Wing from the toolbar or debug icon. If Wing issues a warning about `sys.settrace` being called in `DecoratorTools` select `Ignore this Exception Location` in the `Exceptions` tool in Wing and restart debugging. In general, `sys.settrace` will break *any* Python debugger but Wing and the code in DecoratorTools both take some steps to attempt to continue to debug in this case.

- Bring up the `Debug I/O` tool in Wing and wait until the server output shows that it has started

- Load `http://localhost:8080/` or the page you want to debug in a browser

- Wing should stop on your breakpoint. Be sure to look aroung a bit with the Stack Data tool and the in Wing Pro the Debug Probe (a command line that works in the runtime state of your current debug stack frame).

Configuring Turbogears 2.x to use Wing

Turbogears 2.0 changed some things about how Turbogears instances are packaged and launched, so the configuration is different than with Turbogears 1.x.

This section assumes your Turbogears 2.x project is called `wingtest`. If not, substitute your project name in the following instructions.

- Go into the Turbogears instance directory `wingtest` and run Wing

- Add your instance directory to the project and save it as `wingtest.wpr` There is no need to add all of Turbogears to the project; just the instance should suffice.

- Add also the `paster` to your project. Then open it and and set it as main debug file from the `Debug` menu

- Open up the Python Shell tool and type `import sys` followed by `sys.executable` to verify whether Wing is using the Python that will be running Turbogears. If not, open `Project Properties` and set the `Python Executable` to the correct one.

- Next right click on `paster` and select `File Properties`. Under the `Debug` tab, set `Run Arguments` to `serve development.ini` (do not include the often-used --reload argument, as this will interfere with debugging). Then also set `Initial Directory` to the full path of `wingtest`.

- Set a breakpoint on the `return` line of `RootController.index()` in your `root.py` or somewhere else you know will be reached on a page load

- Start debugging in Wing from the toolbar or debug icon. If Wing issues a warning about `sys.settrace` being called in `DecoratorTools` select `Ignore this Exception Location` in the `Exceptions` tool in Wing and restart debugging. In general, `sys.settrace` will break *any* Python debugger but Wing and the code in DecoratorTools both take some steps to attempt to continue to debug in this case.

- Bring up the `Debug I/O` tool in Wing and wait until the server output shows that it has started

- Load `http://localhost:8080/` or the page you want to debug in a browser

- Wing should stop on your breakpoint. Be sure to look aroung a bit with the Stack Data tool and in Wing Pro the Debug Probe (a command line that works in the runtime state of your current debug stack frame).

Notes for Turbogears 1.x

Turbogears 1.x will install itself into whichever instance of Python runs the installer script, and only certain versions of Python work with a given version of Turbogears.

If you want to avoid adding Turbogears to an install of Python that you are using for other purposes, you can install Python to a new location and dedicate that instance to Turbogears. On Linux, this can be done as follows (assuming you create `/your/path/to/turbogears` as the place to install):

- In a Python source dist do:

      ./configure --prefix=/your/path/to/turbogears
      make
      make install

- Download `tgsetup.py` (or from the Turbogears website

- Change to `/your/path/to/turbogears`

- Run `bin/python tgsetup.py --prefix=/your/path/to/turbogears` (this works in Turbogears 1.0.5 but in older versions you may need to edit `tgsetup.py` to replace `/usr/local/bin` with `/your/path/to/turbogears/bin`.

- Run `bin/tgadmin quickstart`

- Enter project name `wingtest` and defaults for the other options

Similar steps should work on Windows and OS X.

Notes for Turbogears 2.x

Turbogears 2.x uses `virtualenv` to separate what it installs from your main Python installation so in most cases you can install Turbogears 2.x using an installation of Python that you also use for other purposes. If, however, a clean or separate Python installation is desired, you can install Python to a new location and dedicate that instance to Turbogears. On Linux, this can be done as follows (assuming you create `/your/path/to/turbogears` as the place to install):

- In a Python source dist do:

  ```
  ./configure --prefix=/your/path/to/turbogears
  make
  make install
  ```

- Then install `easy_install` by running its setup script with the Python at `/your/path/to/turbogears/bin/python`.

- Whenever the Turbogears installation instructions call for invoking `easy_install` use the one in `/your/path/to/turbogears/bin`

Similar steps should work on Windows and OS X.

Related Documents

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

- **Wing IDE Reference Manual**, which describes Wing IDE in detail.

- Turbogears home page, which provides links to documentation.

- **Wing IDE Quickstart Guide** which contains additional basic information about getting started with Wing IDE.

3.7. Using Wing IDE with web2py

Wing IDE is an integrated development environment that can be used to write, test, and debug Python code that is written for web2py, an open source web development framework. Wing provides auto-completion, call tips, a powerful debugger, and many other features that help you write, navigate, and understand Python code.

For more information on Wing IDE see the product overview. If you do not already have Wing IDE installed, download a free trial now.

To get started using Wing, 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 web2py 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 web2py server is run from within Wing as opposed to attaching to a remote process.

Setting up a Project

In order to debug web2py code, you need the web2py sources. Download and unpack these, then launch Wing and then create a new project from the `Project` menu. Add the web2py directory and go to the `Project` view by right clicking and selecting `Add Directory`. Then right click on the file `web2py.py` in the project view and select `Set As Main Debug File`.

On Windows you will also need to:

1) Install pywin32

2) Use Python version 2.5 instead of Python 2.6 or greater (as of Dec 2009)

Debugging

Now you can debug web2py by launching it from Wing. Just click on the green `Debug` icon in the toolbar and wait for the web2py console to appear. Enter a password and start the server as usual.

Once web2py is running, open a file in Wing that you know will be reached when you load a page of your web2py application in your web browser. Place a breakpoint in the code and load the page in your web browser. Wing should stop at the breakpoint. Use the `Stack Data` tool or `Debug Probe` (in Wing Pro) to look around.

Notice that the `Debug Probe` is similar to running a shell from web2py (with `python web2py.py -S myApp -M`) but additionally includes your entire context. You can easily inspect or modify variables, manually make function calls, and continue debugging from your current context.

Setting Run Arguments

When you start debugging, Wing will show the `File Properties` for `web2py.py`. This includes a `Run Arguments` field under the `Debug` tab where you can add any web2py option. For example, adding `-a '<recycle>'` will give you somewhat faster web2py startup since it avoids showing the `Tk` dialogs and automatically opening a browser window. This is handy once you already have a target page in your browser. Run `python web2py.py --help` for a list of all the available options.

To avoid seeing the `File Properties` dialog each time you debug, un-check the "Show this dialog before each run" check box. You can access it subsequently with the `Current File Properties` item in the `Source` menu or by right clicking on the editor and selecting `Properties`.

Hung Cron Processes

Web2py may spawn cron sub-processes that fail to terminate on some OSes when web2py is debugged from Wing IDE. This can lead to unresponsiveness of the debug process until those sub-processes are killed. To avoid this, add the parameter `-N` to prevent the cron processes from being spawned.

Exception Reporting

Skip this section if you are using web2py version 1.62 or newer.

As shipped, web2py version 1.61 and earlier contain a catch-all exception handler to report unexpected errors in your web browser as tickets. This is useful when tracking problems on a live site.

To make debugging more convenient, change the `except Exception, exception` clause in the definition of `restricted` at the end of the file `src/gluon/restricted.py` in your web2py installation to read as follows:

```
except Exception, exception:
    # XXX Show exception in Wing IDE if running in debugger
    if __debug__ and 'WINGDB_ACTIVE' in os.environ:
        etype, evalue, tb = sys.exc_info()
        sys.excepthook(etype, evalue, tb)
    raise RestrictedError(layer, code, '', environment)
```

Now you will get exceptions reported in Wing's `Exceptions` tool and can conveniently move up or down the stack and inspect the program state at the time of the exception.

Better Static Auto-completion

Working in your code when the debugger is not runnng by default misses some auto-completion options because of how web2py works. For example, auto-completion after typing `db.` will fail because `db` is not explicitly defined. To fix this, you can add some hints for Wing as follows at the top of the file:

```
# XXX This makes auto-
completion work; also need to alter Python Path
# XXX in project properties.
if 0:
    import db
```

Then go into `Project properties` in the `Project` menu and add the following path under `Python Path`:

```
/path/to/web2py/applications/examples/models
```

Replace `/path/to` according to where you unpacked web2py. This path may vary depending on which app you are working with.

Now, typing `db.` should bring up an auto-completer with the contents of `db` even if the debugger is not running.

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.

3.8. Using Wing IDE with mod_wsgi

Wing IDE is an integrated development environment that can be used to write, test, and debug Python code that is running under mod_wsgi and other Python-based web development technologies. Wing provides auto-completion, call tips, a powerful debugger, and many other features that help you write, navigate, and understand Python code.

For more information on Wing IDE see the product overview. If you do not already have Wing IDE installed, download a free trial now.

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

Debugging Setup

When debugging Python code running under `mod_wsgi`, the debug process is initiated from outside of Wing IDE, and must connect to the IDE. This is done with `wingdbstub` according to the instructions in the **Debugging Externally Launched Code** section of the manual.

Because of how `mod_wsqi` sets up the interpreter, be sure to set `kEmbedded=1` in your copy of `wingdbstub.py` and use the debugger API to reset the debugger and connection as follows:

```
import wingdbstub
wingdbstub.Ensure()
```

Then click on the bug icon in lower left of Wing's window and make sure that `Passive Listen` is enabled. After that, you should be able to reach breakpoints by loading pages in your browser. : Disabling stdin/stdout Restrictions ::::::::::::::::::::::::::::::::::::

Literal block expected; none found.

In order to debug, may also need to disable the WSGI restrictions on stdin/stdout with the following mod_wsgi configuration directives:

```
WSGIRestrictStdin Off
WSGIRestrictStdout Off
```

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.

3.9. Using Wing IDE with mod_python

Wing IDE is an integrated development environment that can be used to write, test, and debug Python code that is run by the mod_python module for the Apache web server. Wing provides auto-completion, call tips, a powerful debugger, and many other features that help you write, navigate, and understand Python code.

For more information on Wing IDE see the product overview. If you do not already have Wing IDE installed, download a free trial now.

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

Introduction

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 takes control of all threads in a process, only one http request can be debugged at a time. In the technique described below, 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 block until the first request completes. 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. As a result, it is recommended that only one person debug mod_python based modules per Apache instance and production servers should not be debugged.

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.

Notes

In some cases, we've seen Wing fail to debug the second+ request to mod_python. If this happens, try the following variant of the above code:

```
import wingdbstub
import time

if wingdbstub.debugger != None:
  wingdbstub.debugger.StopDebug()
  time.sleep(2)
  wingdbstub.debugger.StartDebug()
```

This reinitialized debugging with each page load. The `time.sleep()` duration may be shortened, or may need to be lengthened if Wing does not manage to drop the debug connection and initiate listening for a new connection quickly enough.

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.

3.10. Using Wing IDE with Paste and Pylons

Wing IDE is an integrated development environment that can be used to write, test, and debug Python code that is written for Paste and Pylons (which is based on Paste). Wing provides auto-completion, call tips, a powerful debugger, and many other features that help you write, navigate, and understand Python code.

For more information on Wing IDE see the product overview. If you do not already have Wing IDE installed, download a free trial now. To get started using Wing, refer to the tutorial in the `Help` menu in Wing and/or the **Wing IDE Quickstart Guide**.

In order to debug Pylons and Paste applications, you will need Wing 3.0 or later, since earlier versions did not support multi-threaded debugging.

Installing Paste and/or Pylons

The Pylons website and Paste website provide complete instructions for installing Pylons or Paste

Debugging in Wing IDE

Paste and Pylons can be set to run in an environment that spawns and automatically relaunches a sub-process for servicing web requests. This is used to automatically restart the server if for some reason it crashes. However, this does not work with Wing's debugger since the debugger has no way to cause the sub-process to be debugged when it is started by the main process.

To avoid this, do not specify the `--reload` flag for Paste. Place the following in a file that you add to your project and set as the main debug file:

```
from paste.script.serve import ServeCommand
ServeCommand("serve").run(["development.ini"])
```

This may vary somewhat, as necessary for your application.

Debugging Mako Templates

Wing cannot debug Mako templates directly, but it is possible to debug them through the `.py` translation (stored in `data/templates` in the Pylon tree).

Related Documents

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

- **Wing IDE Reference Manual**, which describes Wing IDE in detail.

- Pylons home page, which provides links to documentation.

- Paste home page, which provides links to documentation.

- **Wing IDE Quickstart Guide** which contains additional basic information about getting started with Wing IDE.

3.11. Using Wing IDE with Webware

Wing IDE is an integrated development environment that can be used to write, test, and debug Python code that is written for Webware, an open source web development framework. Wing provides auto-completion, call tips, a powerful debugger, and many other features that help you write, navigate, and understand Python code.

For more information on Wing IDE see the product overview. If you do not already have Wing IDE installed, download a free trial now.

To get started using Wing, 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 de-
bugging scenario you can use the `Default Project` and simply add your source code
directory to it by using `Add Directory` 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
```

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

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 `File
   Properties`. 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, use the `Set Main Debug
   File` item in the `Debug` menu to cause Wing to always launch this file when
   debug is started, 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.

3.12. Debugging Web CGIs with Wing IDE

Wing IDE is an integrated development environment that can be used to write, test, and debug CGI scripts written in Python. Debugging takes place in the context of the web server, as scripts are invoked during a browser page load. Wing also provides

auto-completion, call tips, and many other features that help you write, navigate, and understand Python code.

For more information on Wing IDE see the product overview. If you do not already have Wing IDE installed, download a free trial now.

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

Introduction

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.

Tips and Tricks

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. The work-around is to load specific parts of the page in the browser by entering the 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 `wingdb-stub` or calling the debugger API:

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

(In Python 3.x, use `print()` instead of `print`)

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 (in Python 3.x replace `print` with `print()`):

```
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 `kLogFile` to receive extra information from the debug server, in order to debug problems connecting back to Wing IDE.

(5) 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.

(6) 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.

(7) 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.

How-Tos for GUI Development

The following How-Tos provide tips and short cuts for using a number of popular GUI development frameworks with Wing IDE.

4.1. Using Wing IDE with wxPython

Wing IDE is an integrated development environment that can be used to write, test, and debug Python code that is written for the powerful wxPython cross-platform GUI development toolkit. Wing provides auto-completion, call tips, a powerful debugger, and many other features that help you write, navigate, and understand Python code.

For more information on Wing IDE see the product overview. If you do not already have Wing IDE installed, download a free trial now.

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

Introduction

wxPython is a good choice for GUI developers. It 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 provide a GUI builder for wxPython, it does provide the most advanced 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, the Finder or OS X, or by typing `wing4.1` on the command line on Linux other Posix systems. Once Wing has started, you may want to switch to reading this How-To from the `Help` menu. This will add links to the functionality of the application.

- Select **Show Python Environment** 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 may be located within your Python installation at `site-packages/wx/demo/demo.py`, or `Lib/site-packages/wx/demo/demo.py`, or `c:\Program Files\wxPython2.6 Docs and Demos\demo`, or similar location. On Linux it may be part of a separate wx examples package, for example on Ubuntu 6.06 LTS the demo is in the package `wx2.6-examples`, is installed in `/usr/share/doc/wx2.6-examples/examples/wxPython`, and some files in this directory need to be `gunzip''ed before the demo will work. Once you've opened ''demo.py`, select `Add Current File` from the `Project` menu. If you can't find `demo.py` but have other wxPython code that works, you can also just use that. However, the rest of this document assumes you're using `demo.py` so you will have to adapt the instructions.

- Set `demo.py` as main entry point for debugging using the `Set Main Debug File` item in 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 `Start / 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. Try out the various demos from the tree on the left of the wxPython demo app.

Important: In earlier wxPython 2.6 versions, 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 `ImageBrowser.py` (located in the same directory as `demo.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` (Wing Pro only) 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` (Wing Pro only) -- browses data for all modules in `sys.modules`
- `Watch` (Wing Pro only) -- 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` or `site-packages/wx` inside your Python installation to your project file with the `Add Directory` item in the `Project` menu.

- After doing so, Wing may consume significant CPU for some time, 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 `Browse Project Classes` mode at the top of the source browser. This is generally the best view to use for wxPython. If you duse the `Browse Project Modules` view, it helps to select `Hide Inherited Classes` from the `Options` menu in the browser.

- Use the right-click menu to zoom to base classes. In general in Wing, 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 add `Enter` to the **Completion Keys** 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 may be sluggish at first, once again due to the large number of symbols imported into most wxPython files with `from wx import *`. However, this should 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. Note that this tool is also integrated with the source browser, and with the auto-completer in the editor, Python Shell, and Debug Probe (in Wing Pro).

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. This is done in Preferences in the `Files Reloading` area.

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.

4.2. Using Wing IDE with PyQt

Wing IDE is an integrated development environment that can be used to write, test, and debug Python code that is written for the PyQt cross-platform GUI development toolkit. Wing provides auto-completion, call tips, a powerful debugger, and many other features that help you write, navigate, and understand Python code.

For more information on Wing IDE see the product overview. If you do not already have Wing IDE installed, download a free trial now.

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

Introduction

PyQt is a commercial GUI development environment that runs with native look and feel on Windows, Linux/Unix, Mac OS, and mobile devices. While Wing IDE does not include a GUI builder for PyQt, it does provide the most advanced 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, PyQt, and Wing. The **Wing IDE Quickstart Guide** provides installation instructions for Wing.

- Start Wing from the Start menu on Windows, the Finder on OS X, or by typing `wing4.1` on the command line on Linux other Posix systems. Once Wing has started, you may want to switch to reading this How-To from the `Help` menu. This will add links to the functionality of the application.

- Select **Show Python Environment** 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 `examples/demos/qtdemo/qtdemo.py` into Wing IDE (located within your Python installation) and select `Add Current File` from the `Project` menu.

- Set `qtdemo.py` as main entry point for debugging with `Set Main Debug File` in 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 `Start / 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`. You can visit this dialog again later by right clicking on `qtdemo.py` in the Project view and selecting `File Properties` or by right clicking on the editor.

- 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 `menumanager.py` from the `examples/demos/qtdemo` directory and set a breakpoint on the first line of the method `itemSelection`. Once set, this breakpoint should be reached whenever you click on a button in the `qtdemo` application.

- Use the `Stack Data` tool in the `Tools` menu to look around the stack and the locals and globals for the selected stack frame.

- Select `Debug Probe` (Wing Pro only) 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 and also useful for writing new code in the context of the live runtime environment.

- Notice also that when the debugger is active, typing in code that is on the stack (such as in `itemSelected`) shows auto-completion in the editor and calltips and documentation in the Source Assistant tool that is sourced from the live runtime state of your application.

See the **Wing IDE Tutorial** and **Quick start** for more information.

Test Driving the Source Browser

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

- Add package `Lib/site-packages` or `site-packages` (inside your Python installation) to your project with the `Add Directory` item in the `Project` menu. On OS X this is located inside your `Python.framework/Versions/#.#/lib/python#.#` directory.

- Next bring up the `Source Browser` from the `Tools` menu. You can select the view style at the top of the window, to browse by modules, by classes, or only the current file. The `Options` menu on the right will filter what types of symbols are being displayed in the browser.

- Double clicking on the browser will show the corresponding source code in the source editor area.

- Use the right-click menu on the Source Browser 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 **Completion Keys** 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. Note that this tool is also integrated with the source browser, and with the auto-completer in the editor, Python Shell, and Debug Probe (in Wing Pro).

Using a GUI Builder

Wing IDE doesn't currently include a GUI builder for PyQt but it can be used with an external GUI builder. Wing will automatically reload files that are written by the GUI builder, making for a fairly seamless integration.

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 model-driven GUIs also tend to be more maintainable, and the Qt widget set was designed specifically to make hand-coding easy.

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 and downloads.

- **Wing IDE Quickstart Guide** which contains additional basic information about getting started with Wing IDE.

4.3. Using Wing IDE with GTK and PyGObject

Wing IDE is an integrated development environment that can be used to edit, test, and debug Python code that is written for GTK using PyGObject. Wing provides auto-completion, call tips, a powerful debugger, and many other features that help you write, navigate, and understand Python code.

For more information on Wing IDE see the product overview. If you do not already have Wing IDE installed, download a free trial now.

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

Auto-Completion

PyGObject uses lazy (on-demand) loading of functionality to speed up startup of applications that are based on it. This prevents Wing's analysis engine from inspecting PyGObject-wrapped APIs and thus the IDE fails to offer auto-completion.

To work around this, use Fakegir, which is a tool to build a fake Python package of PyGObject modules that can be placed onto the `Python Path` defined in Wing's `Project Properties`.

Fakegir's `README.md` provides usage details.

Since Wing uses the configured `Python Path` not just for the editor but also when debugging or executing code, it is necessary to remove the fake module directory from sys.path, if present, before importing any PyGObject-provided modules:

```
import sys, os
kFakegirCache = os.path.join(os.path.expanduser('~'), '.cache/fakegir/')
if kFakegirCache in sys.path:
  sys.path.remove(kFakegirCache)
```

Once this is done Wing should offer auto-completion for all PyGObject-provided modules and should be able to execute and debug your code without disruption.

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.

4.4. Using Wing IDE with PyGTK

Wing IDE is an integrated development environment that can be used to edit, test, and debug Python code that is written for PyGTK and GTK+, a mature open source GUI development toolkit. Wing provides auto-completion, call tips, a powerful debugger, and many other features that help you write, navigate, and understand Python code.

For more information on Wing IDE see the product overview. If you do not already have Wing IDE installed, download a free trial now.

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

Introduction

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, its 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 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, the Finder or OS X, or by typing `wing4.1` on the command line on Linux other Posix systems. Once Wing has

started, you may want to switch to reading this How-To from the `Help` menu. This will add links to the functionality of the application.

- Select **Show Python Environment** 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 with `Set Main Debug File` in the `Debug` menu.

- Save your project to disk. Use a name ending in `.wpr`.

- You should now be able to debug your PyGTK application from within Wing. If you see ImportErrors on the PyGTK modules, you will need to add **Python Path** 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 IDE Pro's Source Assistant, you may need to run a script that converts from PyGTK's defs files into Python interface files that Wing's source analyser can read. This is only necessary if you are working with PyGTK significantly different than version 2.7.4, because Wing ships with pre-built interface information for PyGTK 2.7.4. If you do need to build interface files, do so as follows:

- 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 (in Wing IDE Pro) Source Assistant information when you `import gtk` and type `gtk.` in the editor.

With newer PyGTK versions, it may be necessary to make modifications to the `pygtk_to_pi.py` script to track changes in the nature of the source base.

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. This is done in Preferences in the `Files / Reloading` area.

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. 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.

Unfortunately not all of our patches have been merged into the current GTK sources, although we have contributed patches in all cases so they can be retrieved from the source forge bug tracker as well.

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.

4.5. Using Wing IDE with matplotlib

Wing IDE is an integrated development environment that can be used to speed up the process of writing and debugging Python code that is written for matplotlib, a powerful 2D plotting library. Wing provides auto-completion, call tips, a powerful debugger, and many other features that help you write, navigate, and understand Python code.

For more information on Wing IDE see the product overview. If you do not already have Wing IDE installed, download a free trial now.

Note: This document contains only matplotlib specific tips; please refer to the tutorial in the `Help` menu in Wing and/or the **Wing IDE Quickstart Guide**.

Working in the Python Shell

Users of matplotlib often work interactively in the Python command line shell. For example, two plots could be shown in succession as follows:

```
from pylab import plot,show,close
x = range(10)
plot(x)
show()
y = [2, 8, 3, 9, 4]
plot(y)
close()
```

In some environments, the `show()` call above will block until the plot window is closed. By default Wing IDE modifies the matplotlib event loop in such a way that the `show()` call will not block when entered in the integrated Python Shell, and the plot window will be updated continuously as additional commands are typed. This allows for easier interactive testing of new code and plots.

Code from the editor can be executed in the Python Shell using the `Evaluate File in Python Shell` item in the `Source` menu or with the `Evaluate Selection in Python Shell` item in the editor context menu (right click). By default the Python Shell restarts before evaluating a whole file; this can be disabled in the Python Shell's `Options` menu.

This special event loop support has been implemented for the `TkAgg`, `GTKAgg`, `WXAgg` (for wxPython 2.5+) and `Qt4Agg` backends. It will not work with other backends.

Working in the Debugger

When executing code that includes `show()` in the debugger, Wing will block within the `show()` call just as Python would outside of the debugger if launched on the same file. This is by design, since the debugger seeks to replicate Python run non-interactively.

To work interactively with matplotlib code launched in the debugger, you can set a breakpoint on `show()` in the code and then work in the Debug Probe. Wing adds an item `Evaluate Selection in Debug Probe` to the editor context menu (right click) when the debugger is active.

Trouble-shooting

If `show()` blocks when typed in the Python Shell or Debug Probe, if plots fail to update, or if you run into other event loop problems working with matplotlib you can:

(1) Try the following as a way to switch to another backend before issuing any other commands:

```
import matplotlib
matplotlib.use('TkAgg')
```

(2) Try disabling the matplotlib support entirely in `Project Properties` under the `Extensions` tab and then restart the Python Shell from its `Options` menu and restart your debug process, if any. However, this prevents interactive use of matplotlib in the Python Shell and Debug Probe.

Please email support@wingware.com if you cannot resolve problems without disabling Wing's matplotlib support.

Related Documents

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

- **Wing IDE Reference Manual**, which describes Wing IDE in detail.

- The matplotlib home page, which provides links to documentation.

- **Wing IDE Quickstart Guide** which contains additional basic information about getting started with Wing IDE.

How-Tos for Rendering and Compositing Systems

The following How-Tos provide tips and short cuts for using a number of rendering and compositing systems with Wing IDE.

5.1. Using Wing IDE with Blender

Wing IDE is an integrated development environment that can be used to develop, test, and debug Python code written for Blender, an open source 3D content creation system. Wing provides auto-completion, call tips, a powerful debugger, and many other features that help you write, navigate, and understand Python code.

For more information on Wing IDE see the product overview. If you do not already have Wing IDE installed, download a free trial now.

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

Introduction

Blender's loads Python scripts in a way that makes them difficult to debug in a Python debugger. The following stub file can be used to work around these problems:

```
import os
import sys

# MODIFY THESE:
winghome = r'c:\Program Files\Wing IDE 2.1'
scriptfile = r'c:\src\test\blender.py'
```

```
os.environ['WINGHOME'] = winghome
if winghome not in sys.path:
    sys.path.append(winghome)
#os.environ['WINGDB_LOGFILE'] = r'c:\src\blender-debug.log'
import wingdbstub
wingdbstub.debugger.StartDebug()

def runfile(filename):
    execfile(filename)
runfile(scriptfile)
```

To use this script:

1) Modify `winghome` & `scriptfile` definitions where indicated to the wing installation directory and the script you want to debug, respectively. When in doubt, the location to use for `winghome` is given as the `Install Directory` in your Wing IDE About box (accessed from `Help` menu).

2) Run blender

3) Click on upper left icon and select text editor

4) Click on icon to right of "File" to display text editor pane

5) Select File -> Open from the *bottom* menu bar and select this file to open

Once the above is done you can debug your script by executing this blenderstub file in blender. This is done using File -> Run Python Script from the bottom menu or by the Alt-P key, though Alt-P seems to be sensitive to how the focus is set.

Note that you will need to turn on passive listening for debug connections in Wing, which is most easily done by clicking on the bug icon in the lower left of the main window and selecting `Enable Passive Listen` in the popup menu that appears.

Related Documents

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

- **Wing IDE Reference Manual**, which describes Wing IDE in detail.

- Blender home page, which provides links to documentation.

- **Wing IDE Quickstart Guide** which contains additional basic information about getting started with Wing IDE.

5.2. Using Wing IDE with Autodesk Maya

Wing IDE is an integrated development environment that can be used to develop, test, and debug Python code written for Autodesk Maya, a commercial 3D modeling application. Wing provides auto-completion, call tips, a powerful debugger, and many other features that help you write, navigate, and understand Python code.

For more information on Wing IDE see the product overview. If you do not already have Wing IDE installed, download a free trial now.

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

Debugging Setup

When debugging Python code running under `Maya`, the debug process is initiated from outside of Wing IDE, and must connect to the IDE. This is done with `wingdbstub` according to the instructions in the **Debugging Externally Launched Code** section of the manual.

Because of how `Maya` sets up the interpreter, be sure to set `kEmbedded=1` in your copy of `wingdbstub.py` and use the debugger API to ensure the debugger is connected to the IDE before any other code executes as follows:

```
import wingdbstub
wingdbstub.Ensure()
```

Then click on the bug icon in lower left of Wing's window and make sure that `Passive Listen` is enabled. After that, you should be able to reach breakpoints by causing the scripts to be invoked from Maya.

To use the `mayapy` executable found in the `Maya` application directory to run Wing's Python Shell tool and to debug standalone Python scripts, enter the full path of the mayapy file (mayapy.exe on Windows) in the **Python Executable field of the Project Properties dialog**.

Better Static Auto-completion

Maya's Python support scripts do not come with source code, but rather only with `pyc` files. Because Wing cannot statically analyze those files, it will fail to offer auto-completion for them unless `.pi` files are used. A set of .pi files generated by the PyMEL project can be found in Maya 2011 or in the PyMEL distribution.

- Maya 2011 ships with .pi files in the `devkit/pymel/extras/completion/pi` sub-directory of the Maya 2011 install directory.

- For other Maya versions, .pi files from the PyMEL distribution at http://code.google.com/p/pymel/ may be used. PyMEL does not need to be installed or used to make use of the .pi files; it's enough to simply unpack the source distribution. The pi directory within the PyMEL 1.0.2 distribution is `extras/completion/pi`

Add the pi directory to the list of interface file directories that Wing uses by adding it to the **Interface File Path** preference in the Source Analysis -> Advanced preference page. After adding the directory to the path, Wing will offer auto-completion if you `import xxx` and then type `xxx.`

Additional Information

Some additional information about using Wing IDE with Maya can be found in For Python: Maya 'Script Editor' Style IDE. This includes extension scripts for more closely integrating Wing Pro and Maya and some additional details. For example, sending Python and MEL code to Maya from Wing is explained here

See also the section `Using Wing IDE with Maya` in Autodesk Maya Online Help: Tips and tricks for scripters new to Python.

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.

5.3. Using Wing IDE with NUKE and NUKEX

Wing IDE is an integrated development environment that can be used to write, test, and debug Python code that is written for The Foundry's NUKE and NUKEX digital compositing tool. Wing provides auto-completion, call tips, a powerful debugger, and many other features that help you write, navigate, and understand Python code.

For more information on Wing IDE see the product overview. If you do not already have Wing IDE installed, download a free trial now.

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

Project Configuration

First, launch Wing IDE and create a new project from the `Project` menu and save it to disk. Files can be added to the project with the `Project` menu. This is not a requirement for working with NUKE but recommended so that Wing IDE's source analysis, search, and revision control features know which files are part of the project.

Next, make sure Wing IDE is using NUKE's Python installation, or a Python that matches NUKE's Python version.

Configuring for Licensed NUKE/NUKEX

If you have NUKE or NUKEX licensed and are not using the Personal Learning Edition, then you can create a script to run NUKE's Python in terminal mode and use that as the `Python Executable` in Wing's Project Properties. For example on OS X create a script like this:

```
#!/bin/sh
/Applications/Nuke6.3v8/Nuke6.3v8.app/Nuke6.3v8 -t -i "$@"
```

Then perform `chmod +x` on this script to make it executable. Next, you will make the following changes in `Project Properties`, from the `Project` menu in Wing:

- Set `Python Executable` to point to this script

- Change `Python Options` under the `Debug` tab to `Custom` with a blank entry area (no options instead of `-u`)

Apply these changes and Wing will use NUKE's Python in its Python Shell (after restarting from its `Options` menu), for debugging, and for source analysis.

Configuring for Personal Learning Edition of NUKE

The above will not work in the Personal Learning Edition of NUKE because it does not support terminal mode. In that case, install a Python version that matches NUKE's

Python and use that instead. You can determine the correct version to use by by looking at `sys.version` in NUKE's Script Editor. Then point Wing to that Python with `Python Executable` in `Project Properties`. Using a matching Python version is a good idea to avoid confusion caused by differences in Python versions, but is not critical for Wing to function. However, Wing must be able to find *some* Python version or many of its features will be disabled.

Additional Project Configuration

When using Personal Learning Edition, and possibly in other cases, some additional configuration is needed to obtain auto-completion on the NUKE API also when the debugger is not connected or not paused. The API is located inside the NUKE installation, in the `plugins` directory. The `plugins` directory (parent directory of the `nuke` package directory) should be added to the `Python Path` configured in Wing's `Project Properties` (as accessed from the `Project` menu). On OS X this directory is within the NUKE application bundle, for example `/Applications/Nuke6.3v8/Nuke6.3v8.app/Contents/MacOS/plugins`.

Replacing the NUKE Script Editor with Wing IDE Pro

Wing IDE Pro can be used as a full-featured Python IDE to replace NUKE's Script Editor component. This is done by downloading and configuring NukeExternalControl.

First set up and test the client/server connection as described in the documentation for NukeExternalControl. Once this works, create a Python source file that contains the necessary client-side setup code and save this to disk.

Next, set a breakpoint in the code after the NUKE connection has been made, by clicking on the breakpoint margin on the left in Wing's editor or by clicking on the line and using `Add Breakpoint` in the `Debug` menu or the breakpoint icon in the toolbar.

Then debug the file in Wing IDE Pro by pressing the green run icon in the toolbar or with `Start/Continue` in the `Debug` menu. After reaching the breakpoint, use the `Debug Probe` in Wing to work interactively in that context.

You can also work on a source file in Wing's editor and evaluate selections within the file in the `Debug Probe`, by right-clicking on the editor.

Both the `Debug Probe` and Wing's editor should offer auto-completion on the NUKE API, at least while the debugger is active and paused in code that is being edited. The `Source Assistant` in Wing IDE Pro provides additional information for symbols in the auto-completer, editor, and other tools in Wing.

This technique will not work in Wing IDE Personal because it lacks the `Debug Probe` feature. However, debugging is still possible using the alternate method described in the next section.

Debugging Python Running Under NUKE

Another way to work with Wing IDE and NUKE is to connect Wing IDE directly to the Python instance running under NUKE. In order to do this, you need to import a special module in your code, as follows:

```
import wingdbstub
```

You will need to copy `wingdbstub.py` out of your Wing IDE installation and may need to set `WINGHOME` inside `wingdbstub.py` to the location where Wing IDE is installed if this value is not already set by the Wing IDE installer. On OS X, `WINGHOME` should be set to the `Contents/MacOS` directory within Wing's `.app` folder.

Before debugging will work within NUKE, you must also set the `kEmbedded` flag inside `wingdbstub.py` to `1`.

Next click on the bug icon in the lower left of Wing IDE's main window and make sure that `Enable Passive Listen` is checked.

Then execute the code that imports the debugger. For example, right click on one of NUKE's tool tabs and select `Script Editor`. Then in the bottom panel of the Script Editor enter `import wingstub` and press the `Run` button in NUKE's Script Editor tool area. You should see the bug icon in the lower left of Wing IDE's window turn green, indicating that the debugger is connected.

If the import fails to find the module, you may need to add to the Python Path as follows:

```
import sys
sys.path.append("/path/to/wingdbstub")
import wingdbstub
```

After that, any breakpoints set in Python code should be reached and Wing IDE's debugger can be used to inspect, step through code, and try out new code in the live runtime.

For example, place the following code in a module named `testnuke.py` that is located in the same directory as `wingdbstub.py` or anywhere on the `sys.path` used by NUKE:

```
def wingtest():
  import nuke
  nuke.createNode('Blur')
```

Then set a breakpoint on the line `import nuke` by clicking in the breakpoint margin to the left, in Wing's editor.

Next enter the following and press the `Run` button in NUKE's Script Editor (just as you did when importing wingdbstub above):

```
import testnuke
testnuke.wingtest()
```

As soon as the second line is executed, Wing should reach the breakpoint. Then try looking around with the `Stack Data` and `Debug Probe` (in Wing Pro only).

Debugger Configuration Detail

If the debugger import is placed into a script file, you may also want to call `Ensure` on the debugger, which will make sure that the debugger is active and connected:

```
import wingdbstub
wingdbstub.Ensure()
```

This way it will work even after the Stop icon has been pressed in Wing, or if Wing is restarted or the debugger connection is lost for any other reason.

For additional details on configuring the debugger see **Debugging Externally Launched Code**.

Limitations and Notes

When Wing's debugger is connected directly to NUKE and at a breakpoint or exception, NUKE's GUI will become unresponsive because NUKE scripts are run in a way that prevents the main GUI loop from continuing while the script is paused by the debugger. To regain access to the GUI, continue the paused script or disconnect from the debug process with the `Stop` icon in Wing's toolbar.

NUKE will also not update its UI to reflect changes made when stepping through a script or otherwise executing code line by line. For example, typing `import nuke;`

`nuke.createNode('Blur')` in the `Debug Probe` will cause creation of a node but NUKE's GUI will not update until the script is continued.

When the NUKE debug process is connected to the IDE but not paused, setting a breakpoint in Wing will display the breakpoint as a red line rather than a red dot during the time where it has not yet been confirmed by the debugger. This can be any length of time, if NUKE is not executing any Python code. Once Python code is executed, the breakpoint should be confirmed and will be reached. This delay in confirming the breakpoint does not occur if the breakpoint is set while the debug process is already paused, or before the debug connection is made.

These problems should only occur when Wing IDE's debugger is attached directly to NUKE, and can be avoided by working through `NukeExternalControl` instead, as described in the first part of this document.

Related Documents

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

- **Wing IDE Reference Manual**, which describes Wing IDE in detail.

- NUKE/NUKEX home page, which provides links to documentation.

- **Wing IDE Quickstart Guide** which contains additional basic information about getting started with Wing IDE.

How-Tos for Other Libraries

The following How-Tos provide tips and short cuts for using a number of other popular development frameworks with Wing IDE.

6.1. Using Wing IDE with Twisted

Wing IDE is an integrated development environment that can be used to write, test, and debug Python code that is written for Twisted. Wing provides auto-completion, call tips, a powerful debugger, and many other features that help you write, navigate, and understand Python code.

For more information on Wing IDE see the product overview. If you do not already have Wing IDE installed, download a free trial now.

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

Installing Twisted

The Twisted website provides complete instructions for installing and using Twisted.

Debugging in Wing IDE

To debug Twisted code launched from within Wing IDE, create a file with the following contents and set it as your main debug file by adding it to your project and then using the `Set Main Debug File` item in the `Debug` menu:

```
from twisted.scripts.twistd import run
import os
try:
```

```
        os.unlink('twistd.pid')
    except OSError:
        pass
run()
```

Then go into the `File Properties` for this file (by right clicking on it) and set `Run Arguments` to something like:

```
-n -y name.tac
```

The `-n` option tells Twisted not to daemonize, which would cause the debugger to fail because sub-processes are not automatically debugged. The `-y` option serves to point Twisted at your `.tac` file (replace `name.tac` with the correct name of your file instead).

You can also launch Twisted code from outside of Wing using the module `wingdbstub.py` that comes with Wing. This is described in **Debugging Externally Launched Code** in the manual.

Related Documents

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

- **Wing IDE Reference Manual**, which describes Wing IDE in detail.

- Twisted home page, which provides links to documentation.

- **Wing IDE Quickstart Guide** which contains additional basic information about getting started with Wing IDE.

6.2. Using Wing IDE with Cygwin

Wing IDE is an integrated development environment that can be used to write, test, and debug Python code that is written for cygwin, a Linux/Unix like environment for Microsoft Windows. Wing provides auto-completion, call tips, a powerful debugger, and many other features that help you write, navigate, and understand Python code.

For more information on Wing IDE see the product overview. If you do not already have Wing IDE installed, download a free trial now.

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

Configuration

To write and debug code running under cygwin, download and install Wing IDE for Windows on your machine. There is no Wing IDE for cygwin specifically but you can set up Wing IDE for Windows to work with Python code that is running under cygwin.

Cygwin has a different view of the file system than the paths used by Windows applications. This causes problems when code is debugged since Wing cannot find the files referenced by their cygwin name.

The solution to this problem is to treat Python running under cygwin as if it were running on a separate system. This is done using Wing's **external launch / remote debugging support**. In this model, you will always launch your Python code from cygwin rather than from Wing's menus or toolbar.

When setting this up according to the instructions provided by the above link, use cygwin paths when setting up `WINGHOME` in `wingdbstub.py`.

You will also need to set up a file location translation map from your cygwin names (usually by default something like `/c/path/to/files` maps to `C:\path\to\files`), or set things up in cygwin's configuration so that the cygwin pathname is equivalent to the win32 pathname. For the latter, an example would be to set up `/src` in cygwin to point to the same dir as `\src` in win32 (which is `src` at top level of the main drive, usually `c:\src`). Wing will ignore the difference between forward and backward slashes in path names. An easy way to determine the correct cygwin file path to use is to place `assert 0` into a file and refer to the traceback shown in the Exceptions tool in Wing when the file is debugged via `wingdbstub`.

Related Documents

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

- **Wing IDE Reference Manual**, which describes Wing IDE in detail.

- Cygwin home page, which provides links to documentation.

- **Wing IDE Quickstart Guide** which contains additional basic information about getting started with Wing IDE.

6.3. Using Wing IDE with pygame

Wing IDE is an integrated development environment that can be used to write, test, and debug Python code that is written for pygame, an open source framework for game development with Python.. Wing provides auto-completion, call tips, a powerful debugger, and many other features that help you write, navigate, and understand Python code.

For more information on Wing IDE see the product overview. If you do not already have Wing IDE installed, download a free trial now.

This document contains only pygame-specific tips. To get started using Wing, 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, some versions of pygame running in full screen mode may not work properly and may crash Wing. If that is the case, use window mode instead while debugging.

This problem exists with other Python IDEs as well; we have not yet determined what the cause is and it appears to have been fixed in newer pygame versions.

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.

6.4. Using Wing IDE with scons

Wing IDE is an integrated development environment that can be used to edit, test, and debug Python code that is written for scons, an open source software construction or build control framework that uses Python. Wing provides auto-completion, call

tips, a powerful debugger, and many other features that help you write, navigate, and understand Python code.

For more information on Wing IDE see the product overview. If you do not already have Wing IDE installed, download a free trial now.

To get started using Wing, 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 any Python 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 scons from the command line (likely the preferred method) rather than from within Wing IDE, you will need to use `wingdbstub` to initiate debugging, 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.

6.5. Using Wing IDE with IDA Python

Wing IDE is an integrated development environment that can be used to write, test, and debug Python code that is written for Hex-Rays IDA multi-processor disassembler and debugger. Wing provides auto-completion, call tips, a powerful debugger, and many other features that help you write, navigate, and understand Python code.

For more information on Wing IDE see the product overview. If you do not already have Wing IDE installed, download a free trial now.

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

Debugging IDA Python in Wing IDE

IDA embeds a Python interpreter that can be used to script the system. In order to debug Python code that is run within IDA, you need to import a special module in your code, as follows:

```
import wingdbstub
wingdbstub.Ensure()
```

You will need to copy `wingdbstub.py` out of your Wing IDE installation and may need to set `WINGHOME` inside `wingdbstub.py` to the location where Wing IDE is installed (or on OS X to `Contents/MacOS` within Wing's `.app` folder) if this value is not already set. Even though this is an embedded instance of Python, leave the `kEmbedded` flag set to `0`.

Next click on the bug icon in the lower left of Wing IDE's main window and make sure that `Enable Passive Listen` is checked. Then restart IDA and the debug connection should be made as soon as the above code is executed, as indicated by the color of the bug icon in Wing IDE.

At that point, any breakpoints set in Python code should be reached and Wing IDE can be used to inspect the runtime state, step through code, and try out new code in the live runtime.

For details see **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.

- Hex-Rays IDA 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 IronPython

Wing IDE is an integrated development environment for the Python programming language. Wing can be used to speed up the process of writing code for IronPython, which is an implementation of Python for the .NET runtime.

For more information on setting up Wing IDE with IronPython, see IronPython and the Wing IDE: Using the Wing Python IDE with IronPython. This article provides a script to help with setting up auto-completion for the .NET framework, and some information on how to get Wing to execute your code in IronPython. It was written by Michael Foord, co-author of the book IronPython in Action.

The script the article refers to is now shipped with Wing; it's the src/wingutils/generate_pi.py file in the Wing IDE install directory.

Related Documents

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

- **Wing IDE Reference Manual**, which describes Wing IDE in detail.

- IronPython home page, which provides links to documentation.

- **Wing IDE Tutorial**

- **Wing IDE Quickstart Guide** which contains additional basic information about getting started with Wing IDE.

7.1. 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.

Long strings are also truncated by default when they are sent to the IDE from the debug process. The maximum displayable length of strings is controlled with the **Huge String Threshold** preference.

To view all of a truncated string, right click on them in the Stack Data tool and select `Show Detail` from the popup that appears. Alternatively in Wing Professional you can use the Debug Probe or Watch 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.

7.2. Debugging C/C++ and Python together

Wing's debugger is for Python code only 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 can be done either by launching the debug process from Wing and attaching the C/C++ debugger to it, or by launching the debug process with the C/C++ debugger and then initiating debug in Wing by importing `wingdbstub` in your Python code. To configure `wingdbstub`, see the manual section on **Debugging Externally Launched Code**.

To debug the C/C++ code you need to be running with a copy of Python compiled from sources with debug symbols.

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

7.3. Debugging Extension Modules on Linux/Unix

Gdb can be used as a tool to aid in debugging C/C++ extension modules written for Python, while also running code in Wing's Python debugger.

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

Preparing Python

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 to add `OPT=-g`, and (3) type `make`. Once the build is complete you can optionally install it with `make install` (see the `README` first if you don't want to install into `/usr/local/lib/python`) but you can also run Python in place without installing it.

When this is complete, compile your extension module against that version of Python.

Starting Debug

In order to run code both within Wing's Python debugger and gdb, launch your debug process from Wing first, then note the process ID shown in the tooltip that appears when you hover the mouse over the debug icon in the lower left of Wing's main window.

Next, start gdb. First running `emacs` and then typing `Esc-X gdb` within emacs is one way of doing this, and this makes it easier to set breakpoints and view code as you go up and down the stack or step through it.

Within gdb, type `attach <pid>` where `<pid>` is replaced with the process ID reported by Wing. This will pause the process as it attaches, which gives you a chance to set breakpoints (in `emacs` you can do this with `Ctrl-X Space` while working in the editor). When you're ready to continue the process, type `c` in gdb.

You are now debugging both at the Python and C/C++ level. You should be able to pause, step, and view data in Wing whenever gdb is not paused. When gdb is paused, Wing's debugger cannot be used until the process is continued at the gdb level.

Tips and Tricks

(1) 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 (for example pystack will print the Python stack, pylocals will print the Python locals, and pyframe prints the current Python stack frame)

(2) 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).

(3) 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)
```

For more information see Debugging with Gdb in the Python wiki.

(4) If you are launching code in a way that requires you to set `LD_LIBRARY_PATH` and this is not working, check whether this value is set in `.cshrc`. This file is read each time gdb runs so may overwrite your value. To work around this, set `LD_LIBRARY_PATH` in `.profile` instead. This file is read only once at login time.

(5) Some older versions of gdb will 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.

7.4. 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.

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 `-nograb`

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 asyncronous 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.

7.5. 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. 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.

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 program 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/tserver` within `WINGHOME` (the Wing IDE installation directory, or on OS X `Contents/MacOS` in Wing's `.app` folder). 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 with in 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 `kSupportedMain-loops` in `_extensions.py`

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

7.6. 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 4.1
```

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.