



## How-Tos

### Version 6.1.5-1

This collection of How-Tos explains how to get started using Wing with specific Python frameworks, tools, and libraries for web and GUI development, 2D and 3D modeling, compositing, and rendering, game development, scientific analysis, and much more.

These How-Tos assume that you are familiar with the Python framework or tool being discussed and that you already know how to use Wing. To learn more about Wing see the [Quick Start Guide](#) or [Tutorial](#).

---

*Wingware, the feather logo, Wing Python IDE, Wing Pro, Wing Personal, Wing 101, Wing IDE, Wing IDE 101, Wing IDE Personal, Wing IDE Professional, Wing IDE Pro, Wing Debugger, and "The Intelligent Development Environment for Python Programmers" are trademarks or registered trademarks of Wingware in the United States and other countries.*

*Disclaimers: The information contained in this document is subject to change without notice. Wingware shall not be liable for technical or editorial errors or omissions contained in this document; nor for incidental or consequential damages resulting from furnishing, performance, or use of this material.*

*Hardware and software products mentioned herein are named for identification purposes only and may be trademarks of their respective owners.*

---

Copyright (c) 1999-2019 by Wingware. All rights reserved.

Wingware  
P.O. Box 400527  
Cambridge, MA 02140-0006  
United States of America

# Contents

<b>How-Tos</b>	<b>1</b>
Wing Python IDE Quick Start Guide	1
Install Python	1
Set up a Project	1
Configuring the UI	1
Navigating Code	1
Editing Code	2
Debugging Code	3
Other Features	3
Related Documents	4
How-Tos for Web Development	4
2.1. Using Wing with Django	4
Installing Django	5
Quick Start with Wing Pro	5
Existing Django Project	5
New Django Project	5
Django-specific Actions	6
Usage Tips	6
Debugging Exceptions	6
Debugging Django Templates	6
Notes on Auto-Completion	7
Running Unit Tests	7
Django with Buildout	7
Manual Configuration	7
Configuring the Project	7
Configuring the Debugger	7
Launching from Wing	8
Launching Outside of Wing	8
Debugging Django Templates	8
Related Documents	8
2.2. Using Wing with web2py	9
Introduction	9
Setting up a Project	9
Debugging	9
Setting Run Arguments	10
Hung Cron Processes	10

Better Static Auto-completion	10
Exception Reporting in Old Web2Py Versions	10
Related Documents	11
2.3. Using Wing with Flask	11
Debugging in Wing	11
Setting up Auto-Reload with Wing Pro	12
Related Documents	12
2.4. Using Wing with Pyramid	12
Installing Pyramid	12
Configuring your Wing Project	13
Debugging	13
Notes on Auto-Completion	14
Debugging Mako Templates	14
Debugging without wingdbstub.py (experimental)	15
Related Documents	15
2.5. Using Wing with Plone	15
Introduction	16
Configuring your Project	16
Debugging with WingDBG	16
WingDBG in buildout-based Plone 4 installations	16
WingDBG as an Egg	17
Related Documents	17
2.6. Using Wing with Zope	17
Before Getting Started	17
Upgrading from earlier Wing versions	18
Quick Start on a Single Host	18
Starting the Debugger	18
Test Drive Wing	18
Setting Up Auto-Refresh	19
Alternative Approach to Reloading	20
Setting up Remote Debugging	20
Trouble Shooting Guide	21
Related Documents	21
2.7. Using Wing with Turbogears	21
Installing Turbogears	21
Configuring Turbogears 1.x to use Wing	22
Configuring Turbogears 2.x to use Wing	22
Notes for Turbogears 1.x	23

Notes for Turbogears 2.x	23
Related Documents	24
2.8. Using Wing with Google App Engine	24
Creating a Project	24
Configuring the Debugger	24
Using the Debugger	25
Improving Auto-Completion and Goto-Definition	25
Trouble-shooting	26
Related Documents	26
2.9. Using Wing with mod_wsgi	26
Debugging Setup	26
Disabling stdin/stdout Restrictions	27
Related Documents	27
2.10. Remote Web Development	27
Setting up SSH Access	27
Installing the Remote Agent	27
Setting up a Project	28
Initiating Debug	28
Debugging Code	29
Managing Permissions	29
Resources	30
2.11. Using Wing with mod_python	30
Introduction	30
Quick Start	30
Example	31
Related Documents	31
2.12. Using Wing with Pylons	31
Installing Pylons	31
Debugging in Wing	31
Debugging Mako Templates	32
Related Documents	32
2.13. Using Wing with Webware	32
Introduction	32
Setting up a Project	32
Starting Debug	33
Related Documents	33
2.14. Debugging Python CGI Scripts with Wing	34
Introduction	34

Tips and Tricks	34
How-Tos for GUI Development	35
3.1. Using Wing with wxPython	35
Introduction	35
Installation and Configuration	36
Test Driving the Debugger	36
Using a GUI Builder	37
Related Documents	37
3.2. Using Wing with PyQt	37
Introduction	38
Installation and Configuration	38
Test Driving the Debugger	38
Test Driving the Source Browser	38
Using a GUI Builder	39
Related Documents	39
3.3. Using Wing with GTK and PyGObject	39
Auto-Completion	39
Related Documents	40
3.4. Using Wing with PyGTK	40
Introduction	40
Installation and Configuration	40
Auto-completion and Source Assistant	41
Using a GUI Builder	41
Details and Notes	41
Related Documents	42
How-Tos for Scientific and Engineering Tools	42
4.1. Using Wing with matplotlib	42
Working in the Python Shell	42
Working in the Debugger	42
Trouble-shooting	43
Related Documents	43
4.2. Using Wing with Jupyter Notebooks	43
Setting up Debug	43
Working with the Debugger	44
Editing Code	45
Stopping on Exceptions	45
Fixing Failure to Debug	46
Reloading Changed Modules	46

Related Documents	46
How-Tos for Modeling, Rendering, and Compositing Systems	46
5.1. Using Wing with Blender	46
Working with Blender	47
Related Documents	47
5.2. Using Wing with Autodesk Maya	48
Debugging Setup	48
Using Maya's Python in Wing	48
Better Auto-completion	48
Maya 2018	49
Maya 2016	49
Maya 2011+	49
Older Versions	49
Additional Information	49
Related Documents	49
5.3. Using Wing with NUKE and NUKEX	49
Project Configuration	50
Configuring for Licensed NUKE/NUKEX	50
Configuring for Personal Learning Edition of NUKE	50
Additional Project Configuration	50
Replacing the NUKE Script Editor with Wing Pro	51
Debugging Python Running Under NUKE	51
Debugger Configuration Detail	52
Limitations and Notes	52
Related Documents	52
5.4. Using Wing with Source Filmmaker	53
Debugging Setup	53
Related Documents	53
How-Tos for Educational Tools	53
6.1. Using Wing with Raspberry Pi	54
Introduction	54
Remote Development with Wing Pro	54
Manual Configuration for Wing Personal	55
Installing and Configuring the Debugger	55
Invoking the Debugger	56
Configuration Details	57
Trouble-Shooting	57
Setting up Wifi on a Raspberry Pi	57

Related Documents	58
6.2. Using Wing with pygame	58
Debugging pygame	58
Related Documents	58
How-Tos for Other Frameworks and Tools	58
7.1. Using Wing with virtualenv	58
Creating a New Virtualenv	59
Working on a Remote Host	59
Using an Existing Virtualenv	59
Activating the Virtualenv	59
Related Documents	60
7.2. Using Wing with Vagrant	60
Prerequisites	60
Creating a Project	60
How It Works	60
Usage Hints	61
Synced Folders	61
Password-less Private Keys	61
Related Documents	61
7.3. Using Wing with Windows Subsystem for Linux	61
Prerequisites	61
Creating a Project	61
Setting up WSL	62
Related Documents	62
7.4. Using Wing with Twisted	63
Installing Twisted	63
Debugging in Wing	63
Related Documents	63
7.5. Using Wing with Cygwin	63
Configuration	64
Related Documents	64
7.6. Using Wing with scons	64
Debugging scons	64
Related Documents	65
7.7. Using Wing with IDA Python	65
Debugging IDA Python in Wing	65
Related Documents	66
7.8. Using Wing with IronPython	66

Project Configuration	66
Related Documents	66
7.9. Handling Large Values and Strings in the Debugger	66
7.10. Debugging C/C++ and Python together	67
7.11. Debugging Extension Modules on Linux/Unix	67
Preparing Python	67
Starting Debug	67
Tips and Tricks	67
7.12. Debugging Code with XGrab* Calls	68
7.13. Debugging Non-Python Mainloops	68
7.14. Debugging Code Running Under Py2exe	70



# Wing Python IDE Quick Start Guide

This is a minimalist guide for getting started quickly with Wing's Python IDE features. For a more in-depth introduction, try the [Tutorial](#).

## Install Python

If you don't already have it on your system, install [Python](#). You may need to restart Wing after doing so.

## Set up a Project

After Wing is running, create a new project from the `Project` menu. Then configure your project with the following steps:

1. Use `Add Existing Directory` in the `Project` menu to your sources to the project. It's best to constrain this to the directories you are actively working with and let Wing find the libraries you use through the `PYTHONPATH`.
2. Use `Project Properties` in the `Project` menu to set `Python Executable` to the `python.exe` or other interpreter executable you want to use with your project. This is typically the full path that is in `sys.executable` in the desired Python installation.
3. If your code alters `sys.path` or loads modules in a non-standard way then you may need to set `Python Path` so that Wing can find your modules for auto-completion, refactoring, debugging, testing, other features.
4. You may want to right-click on your main entry point in the `Project` tool and select `Set As Main Debug File` so that debugging always starts there.
5. Use `Save Project As` in the `Project` menu to save your project to disk.

**Note:** Wing may consume significant CPU time when it first analyzes your code base. Progress is indicated in the lower left of the IDE window. Once this is done, the results are cached across sessions and Wing should run with a snappy and responsive interface.

See [Project-Wide Properties](#) and [Per-File Properties](#) for a description of all available properties. See [Source Code Analysis](#) for background on how Wing's source analysis system works.

## Configuring the UI

You are now ready to start working with code, but may want to make a few configuration changes first:

**Key Bindings** - Wing can emulate VI/Vim, Visual Studio, Emacs, Eclipse, and Brief editors, selected with the `User Interface > Keyboard > Personality` preference.

**Tab Key** - The default tab key action depends on file type, context, and whether or not there is a selection. This can be changed from the `User Interface > Keyboard > Tab Key Action` preference.

There are many other options in `Preferences`.

## Navigating Code

Wing provides many ways to get around your code quickly:

**Goto-definition** is available from the toolbar, `Source` menu, and by right-clicking on symbols in the editor or shells. Use the browser-like forward/back history buttons at the top left of the editor to return from visiting a point of definition.

**Source Index** menus at the top of the editor provide quick access to other parts of a source file.

**Find Symbol** in the `Source` menu in Wing Pro and Wing Personal jumps to a symbol defined in the current file by typing a fragment of its name. `Find Symbol in Project` in Wing Pro works the same way but searches all files in the project.

**Open From Project** in the `File` menu in Wing Pro and Wing Personal is a similar interface for quickly opening project files.

**Find Points of Use** in Wing Pro, when you right-click on a symbol shows where that symbol is being used. Wing distinguishes between separate but like-named symbols.

**Source Browser** in the tools menu in Wing Pro and Wing Personal provides module or class oriented display of the structure of your code. Show both the `Source Browser` and `Source Assistant` for detailed information about selected symbols.

**Mini-search** (Wing Pro and Wing Personal) is a powerful keyboard-driven search and replace facility. The key bindings listed in the `Mini-search` area of the `Edit` menu will display the search entry area at the bottom of the screen.

**Search** in the `Tools` menu provides incremental text, wildcard, and regular expression search and replace in selections and the current file.

**Search in Files** in the `Tools` menu in Wing Pro and Wing Personal provides wildcard and regular expression search and replace in filtered sets of files, directories, named file sets, and within the project.

**Toolbar search** is another quick way to search the current file.

## ***Editing Code***

Wing's editor focuses on fast error-free Python coding:

**Auto-completion** in Wing's editor speeds up typing and reduces coding errors. The auto-completer uses `Tab` by default for completion, but this can be changed in the `Editor > Auto-Completion > Completion Keys` preference. This feature is disabled by default in Wing 101.

**Call Tips and Documentation** shown in the `Source Assistant` (Wing Pro and Wing Personal only) update as you move through your code or work in the shells.

**Auto-indent** while typing in Wing Pro and Wing Personal matches the file's existing indentation. When multiple lines are pasted, they are re-indented according to context (a single `Undo` reverts any unwanted indentation change). Wing also provides an `Indentation` tool for converting a file's indentation style.

**Auto-Editing** in Wing Pro implements a range of operations such as auto-entering closing parentheses, brackets, braces, and quotes. Among other things, Wing also auto-enters invocation arguments, manages new blocks with the `:` key, and corrects out-of-order typing.

Auto-editing operations can be enabled and disabled in the `Editor > Auto-Editing` preferences group. The default set includes those operations that don't affect finger memory. The others are well worth learning.

For details, see [Auto-Editing](#).

**Refactoring** in Wing Pro supports automated renaming and moving of symbols, extracting functions or methods, and introducing variables more quickly than by manually editing code.

**Multiple Selections** can be made with the `Editor > Multiple Selections` menu or the multiple selections toolbar item, and by pressing `Ctrl` (or `Command` on the Mac) while making a selection with the mouse. Once multiple selections have been made, edits made will be applied to all the selections at once.

**Snippets** in Wing Pro are included in Wing's auto-completer as a quick way to enter commonly repeated coding motifs for coding standards, documentation, testing, and so forth. Data entry for snippet arguments is inline in the editor. Use the `Tab` key to move between the fields. Edit or add snippets in the `Snippets` tool.

**Turbo Completion** in Wing Pro is an optional auto-completion mode made possible by Wing's powerful source analysis engine. When the `Editor > Auto-Editing > Python Turbo Mode` preference is enabled, Wing turns every non-symbol key into a completion key in contexts where a new symbol name is

not being typed. The modifier keys can be used alone to escape from the completer in the rare cases when Wing fails to provide the desired completion.

**Code Selection** from the `Edit > Select` menu makes selecting whole statements, blocks, or scopes a snap, before copying, editing, or searching through them.

## ***Debugging Code***

Wing's debugger is a powerful tool for finding and fixing bugs, understanding unfamiliar code, and writing new code interactively. You can launch code from the `Debug` menu or toolbar, or from the `Python Shell` (click on the bug icon in the top right of the shell to enable debugging there).

Breakpoints can be set by clicking on the breakpoint margin of the editor and debugging is started from the toolbar or `Debug` menu. The `Stack Data` tool is used to inspect or change program data. Debug process I/O is shown in the `Debug I/O` tool, or optionally in an external console.

**Interactive Debugging** is supported by Wing Pro's `Debug Probe`, which provides an interactive Python prompt that executes code in the current debug stack frame. When the debugger is paused Wing also uses the live runtime state to fuel the auto-completer in the editor, `Source Assistant`, goto-definition, and other tools.

**Conditional Breakpoints** can be used in Wing Pro to isolate and understand complex bugs by stopping before they occur. Using a conditional breakpoint to isolate a broken case and the `Debug Probe` to design a fix is far more accurate and productive than relaunching code repeatedly.

**Move Program Counter** is also supported in Wing Pro, in the innermost stack frame by right clicking in the editor and selecting `Move Program Counter Here`.

**Watching Values** in Wing Pro by right-clicking on the editor or any of the data views tracks values over time by symbolic name or object reference in the `Watch` tool. Expressions can be also be watched.

**Launch Configurations** in the `Project` menu in Wing Pro and Wing Personal can be used with `Named Entry Points` in the `Debug` menu define different runtime environments for debugging, executing, and unit testing your code.

## ***Other Features***

Wing contains many other features, including:

**Python Shell** -- Wing's `Python Shell` lets you try out code in a sandbox process kept isolated from Wing and your debug process. Code run here can optionally be debugged. To enable this, click the bug icon in the top right of the `Python Shell`. In Wing Pro and Wing Personal, shell provides auto-completion, goto-definition, and is integrated with the `Source Assistant`.

**Unit Testing** in Wing Pro's `Testing` tool works with unittest, doctest, pytest, nose, and Django unit tests. You can run tests suites, view the results, and debug tests.

**Version Control** in Wing Pro supports Mercurial, Git, Subversion, Perforce, Bazaar, and CVS version control systems. Wing should auto-detect which systems are used in your project and show the appropriate additional menubar menus and tools in the `Tools` menu. Right-click on the editor, `Project` tool, or items in the version control tool to compare that file or directory to the repository with `Compare to Repository`. Wing will display both versions with differences highlighted and the added toolbar tools can be used to move through and merge differences. This capability is also available for comparing two files or directories, and a modified buffer to its disk file, by clicking on the `Difference/Merge` icon in the toolbar.

**Remote Development** is very easy to do in Wing Pro by setting up SSH access to a remote host, and then configuring Wing to access that host from the `Remote Hosts` item in the `Project` menu, setting the `Python Executable` in `Project Properties` to that remote host, and adding remote files and directories to your project. Wing can edit, debug, test, search, inspect, and manage files, run the `Python Shell`, and execute `OS Commands` on the remote host in the same way as it does when working locally.

**Running Command Lines** in Wing Pro and Wing Personal's `OS Commands` tool makes it possible to set up and easily execute external tools. This can also be used to set up a build command that will be executed automatically before each debug sessions.

**User Interface Customization** in `Preferences` gives you control of the overall layout and color of the IDE, among many other options. Right click on the tabs for layout options, or drag tool and editor tabs to move them or create new splits. Right click on the toolbar to configure which tools are visible or to add your own. Wing also supports defining [sharable color palettes](#) and [syntax colors](#).

**Perspectives** in Wing Pro and Wing Personal let you save named tool panel configurations.

**Many Other Features** such as bookmarks, line editing, code folding, macros are also available. You can also [extend Wing by writing Python scripts](#).

### Note

We welcome feedback, which can be submitted with `Submit Feedback` in Wing's `Help` menu, or by emailing at [support@wingware.com](mailto:support@wingware.com)

## Related Documents

For more information see:

- [Wing Tutorial](#), a detailed guided tour for Wing.
- How-Tos for [Django](#), [Flask](#), [matplotlib](#), [PyQt](#), [wxPython](#), [Plone](#), [Autodesk Maya](#), [NUKE/NUKEX](#), [PyGame](#), and [many others](#)
- [Wing Reference Manual](#), which describes Wing 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.

### 2.1. Using Wing with Django

#### Note

**"Wing is really the standard by which I judge other IDEs. It opens, it works, and does everything it can do to stay out of my way so I can be productive. And its remote debugging, which I use when I'm debugging Django uWSGI processes, makes it a rock star!" -- Andrew M**

[Wing](#) is a Python IDE that can be used to develop, test, and debug Python code written for the [Django](#) web development framework.

Wing can also be used to step through and debug Django templates, and Wing Pro 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.

Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing for Django. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the [Quickstart Guide](#).

### **Installing Django**

The [Django website](#) provides complete instructions for installing Django.

### **Quick Start with Wing Pro**

If you have Wing Pro and Django 1.4 or later, the fastest way to get started using Wing with Django is to use the provided Django extensions. If you have Wing Personal, skip ahead to the Manual Configuration section below.

### **Existing Django Project**

To set up a Wing Pro 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` (or `settings` package) 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 8000` as the main debug file, turns on child process debugging in Project Properties (for debugging auto-reloaded code), 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`.

If `settings` is a package in your project (instead of a `settings.py` file), you will need to set `TEMPLATE_DEBUG=True` manually in the appropriate place(s) in your settings.

Now you should be able to start Django in Wing'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 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 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 create a new Wing project, add the files, and configure the project for use with Django as described in the Existing Django Project sub-section above.

## Django-specific Actions

The Django menu that auto-activates for Django projects also contains special actions for running sync db, generating SQL for a selected app, running validation checks, running unit tests, and restarting the integrated Python Shell with the Django environment.

This menu also allows starting a new Django app. This action creates the app and adds it to `INSTALLED_APPS` in `settings.py`. If `settings` is a package, you will need to manually add the new Django app to `INSTALLED_APPS` in the appropriate place(s) in your settings.

This functionality is implemented as an open source plugin that can be found in `scripts/django.py` in the install directory listed in Wing's About box. This code can be user-modified by working from the existing functionality as examples. For detailed information on writing extensions for Wing, see the [Scripting and Extending Wing](#) chapter.

## Usage Tips

### 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.

### Debugging Django Templates

#### Note

Django 1.9 reimplemented the template engine in a way that partially broken template debugging due to missing information in the new template implementation. Currently stopping in templates works but not in tags invoked via `extends` since there is no way to find the correct template file name in that context. See <https://code.djangoproject.com/ticket/25848>.

The above-described project setup scripts enable template debugging automatically. 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 paused or at a breakpoint, particularly in Django templates where static analysis is not as effective as it is in Python code.

### ***Running Unit Tests***

In Wing Pro, 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 different settings, 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` or the `settings` package.

### ***Manual Configuration***

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

### ***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***

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

## Launching from Wing

When Django is launched from Wing, you must enable `Debug Child Processes` under the `Debug/Execute` tab of `Project Properties` so that Wing can debug auto-reloaded processes. This way Django can immediately load changes you make to code without requiring a restart of Django.

Next find `manage.py` in your project, right click to select `File Properties...`, and set the `Run Arguments` to your desired launch arguments. For example:

```
runserver 8000
```

Child process debugging is not available in Wing Personal, where instead you will need to add `--noreload` to the run arguments for `manage.py`, like this:

```
runserver --noreload 8000
```

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.

## Launching Outside of Wing

Another method of debugging Django is to use `wingdbstub.py` to initiate debugging when Django is started from outside of Wing. This method can be used to debug a Django instance remotely or to enable debugging reloaded Django processes with Wing Personal.

This is done by placing a copy of `wingdbstub.py`, which is located in the install directory listed in Wing's `About` box, into 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 installation (on OS X, to the name of 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 `Accept Debug Connections` preference is enabled in Wing and start Django. The Django process should connect to Wing 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 once you request a page load in your browser that leads to one of your `import wingdbstub` statements.

To debug remotely, refer to [Debugging Externally Launched Remote Code](#) in the Wing Pro reference manual.

## Debugging Django Templates

To enable debugging of Django templates, you will need to take the following two steps:

1. Set `TEMPLATE_DEBUG` to `True` in your Django application's `settings.py` file or settings package,
2. Be sure that Wing'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.

## Related Documents

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



- [Remote Web Development](#) describes how to set up development to a remote host.
- [Django home page](#), which provides links to documentation.
- [Quickstart Guide](#) which contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#), which describes Wing in detail.

## 2.2. Using Wing with web2py

[Wing](#) is a Python IDE that can be used to develop, test, and debug Python code written for [web2py](#), an open source web development framework. Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing for web2py. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's `Help` menu or read the [Quickstart Guide](#).

### Introduction

Wing allows you to debug Python code and templates running under web2py 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 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

Download and install web2py. On some OSes you can use the regular install, but at least on Windows you need to use the web2py sources instead because the regular install is missing modules necessary for debugging. When the sources are being used, you will also need to install Python if you don't already have it.

Then launch Wing and create a new project from the `Project` menu. Select `web2py` as your project type, and point the `Python Executable` at the Python executable (`python` or `python.exe`) used for web2py. Click `OK` and then save the project (for example, as `web2py.wpr` within the web2py directory).

Next add the web2py directory to your project by going to the `Project` view, right clicking, and selecting `Add Directory`. After the project view populates, find and right click on the file `web2py.py` and select `Set As Main Debug File`.

On Windows, if you are working from sources, you may also need to install [pywin32](#)

### Debugging

You can now debug web2py by clicking on the green `Debug` icon in Wing's toolbar and waiting 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.

An example is to set a breakpoint in `applications/welcome/views/default/index.html`, which is loaded when you go to the URL `http://127.0.0.1:8000/welcome/default/index` (assuming local web2py install running on port 8000).

Notice that breakpoints work both in Python code and HTML template files.

Wing's Debug Probe (in the Tools menu) is similar to running a shell from web2py (with `python web2py.py -S myApp -M`) but additionally includes your entire context and provides auto-completion. 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. 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.

### **Better Static Auto-completion**

Working in your code when the debugger is not running 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.

### **Exception Reporting in Old Web2Py Versions**

This section is only relevant if you are using a very old web2py, before version 1.62 .

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 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.

### ***Related Documents***

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

- [Remote Web Development](#) describes how to set up development on a remote host.
- [Quickstart Guide](#) which contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#), which describes Wing in detail.

## **2.3. Using Wing with Flask**

[Wing](#) is a Python IDE that can be used to develop, test, and debug Python code written for the [Flask](#) web development framework. Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing for Flask. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's `Help` menu or read the [Quickstart Guide](#).

### ***Debugging in Wing***

To debug Flask in Wing you need to turn off Flask's built-in debugger, so that Wing's debugger can take over reporting exceptions.

To do this, you can set up your main entry point as in the following example:

```
from flask import Flask
app = Flask(__name__)

...

if __name__ == "__main__":
    import os
    if 'WINGDB_ACTIVE' in os.environ:
        app.debug = False
    app.run()
```

Notice that this turns off Flask's debugging support only if Wing's debugger is present.

Once this is done, use `Set Main Debug File` in the `Debug` menu to set this file as your main debug file in Wing. Then you can start debugging from the IDE, see Flask's output in the `Debug I/O` tool, and load pages from a browser to reach breakpoints or exceptions in your code.

Use `Restart Debugging` in the `Debug` menu or the restart icon in the toolbar to quickly restart Flask after making changes to your code. Or if you have Wing Pro you can automate this as described in the next section.

Passing the `--no-debug` flag or setting environment variable `FLASK_DEBUG=0` are other documented ways to turn off Flask's debug support, although we've had some reports of `--no-debug` failing to function as expected.

### Setting up Auto-Reload with Wing Pro

With the above configuration, you will need to restart Flask whenever you make a change to your code. If you have Wing Pro, you can avoid this by replacing the `app.run()` line in the above example with the following:

```
app.run(use_reloader=True)
```

Then enable `Debug Child Processes` under the `Debug/Execute` tab in `Project Properties` from the `Project` menu. This tells Wing Pro to debug also child processes created by Flask, including the reloader process.

Now Flask will automatically restart on its own whenever you save an already-loaded source file to disk.

You can add additional files for Flask to watch as follows:

```
watch_files = ['/path/to/file1', '/path/to/file2']
app.run(use_reloader=True, extra_files=watch_files)
```

Whenever any of these additional files changes, Flask will also automatically restart.

### Related Documents

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

- [Remote Web Development](#) describes how to set up development to a remote host.
- [Flask home page](#), which provides links to documentation.
- [Quickstart Guide](#) which contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#), which describes Wing in detail.

## 2.4. Using Wing with Pyramid

[Wing](#) is a Python IDE that can be used to develop, test, and debug Python code written for the [Pyramid](#), web development framework. Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing for Pyramid. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's `Help` menu or read the [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 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, 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 `wingdbstub.py` (from the install directory listed in Wing's `About` box) into 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 `Accept Debug Connections` 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 around 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 uses 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 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 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 provides many other options and tools. For more information:

- [Remote Web Development](#) describes how to set up development to a remote host.
- [Pyramid documentation](#)
- [Quickstart Guide](#) which contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#), which describes Wing in detail.

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

## **2.5. Using Wing with Plone**

### ***Note***

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

[Wing](#) is a Python IDE that can be used to develop, test, and debug Python code written for the [Plone](#) content management system. Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing for Plone. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the [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 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 `Python2.7/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**

To get debugging working you must install WingDBG, the Wing debugger product, from `zope/WingDBG-6.1.5.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 Accept Debug Connections so the debugger listens for connections initiated from the outside.

Then start Plone and go into the 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'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 4 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.

## Related Documents

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

- [Plone home page](#), which provides links to documentation.
- [Quickstart Guide](#) which contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#), which describes Wing in detail.

## 2.6. Using Wing with Zope

### Note

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

[Wing](#) is a Python IDE that can be used to develop, test, and debug Python code written for 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 is also useful for Zope-based frameworks like [Plone](#) (see [Plone How-To](#)).

Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing for Zope. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's `Help` menu or read the [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 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 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 with Zope running on the same host as the IDE:

- **Install Zope** -- You can obtain Zope from [zope.org](http://zope.org). Version 2.5.1 or newer will work with Wing.
- **Install Wing** -- You will need [Wing 2.1](#) or later. See [Installing](#) for details.
- **Configure Wing** -- 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**

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

**Run to a Breakpoint** -- Open up your Zope code in Wing 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'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` technique 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 X11 (Linux/Unix) or Remote Desktop (Windows) to run Wing 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 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 your Wing 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 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 [Manually Configured File Location Maps](#) and [Location Map Examples](#) in the Wing 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 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 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 on two separate hosts, you should also inspect the `ide.log` file on the Wing host (located in the [User Settings Directory](#)). It contains additional logging information from the Wing process.

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

## Related Documents

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

- [Remote Web Development](#) describes how to set up development to a remote host.
- [Zope home page](#), which contains much additional information for Zope programmers.
- [Quickstart Guide](#) which contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#), which describes Wing in detail.

## 2.7. Using Wing with Turbogears

[Wing](#) is a Python IDE that can be used to develop, test, and debug Python code written for the [Turbogears](#), web development framework. Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing for Turbogears. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's `Help` menu or read the [Quickstart Guide](#).

### Note

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 around 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 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 around 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 provides many other options and tools. For more information:

- [Remote Web Development](#) describes how to set up development to a remote host.
- [Turbogears home page](#), which provides links to documentation.
- [Quickstart Guide](#) which contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#), which describes Wing in detail.

## **2.8. Using Wing with Google App Engine**

[Wing](#) is a Python IDE that can be used to develop, test, and debug Python code written for [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.

Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing for Google App Engine. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's `Help` menu or read the [Quickstart Guide](#).

### **Creating a Project**

Before trying to configure a Wing project, first install and set up Google App Engine and verify that it is working by starting it outside of Wing 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.

Then create a project in Wing using `New Project` in the `Project` menu and selecting `Google App Engine` as the project type. Then use `Add Directory` in the `Project` menu to add your source directories to the project. You should also add at least `dev_appserver.py`, which is located in the top level of the Google SDK directory.

Next 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, which is then 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.

Next you need to go into `Project Properties` and set `Debug/Execute > Debug Child Processes` to `Always Debug Child Processes`. This is needed because App Engine creates more than one process.

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

### **Configuring the Debugger**

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

You can debug code running under 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 error 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.

For most projects, you'll need to add at least `--max_module_instances=1` to the run arguments, and you may also want to add `--threadsafe_override=false`. These command line arguments disable some of GAE's threading and concurrency features that can prevent debugging from working properly.

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. Or send the `--skip_sdk_update_check` argument on the command line to `dev_appserver.py` to disable this.

### **Using the Debugger**

After you have configured 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'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 interactive 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. You can also see data values by hovering the mouse over symbols in the editor or **Debug Probe** and you can press **F4** to go to the point of definition.

Continuing with the green run button in the toolbar will complete the page load in the browser, unless a breakpoint or exception is reached first.

To set up multiple entry points, use **Named Entry Points** in the **Debug** menu. These can contain different commands lines and environment for `dev_appserver.py`.

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 restart the debug process after edits are made. However, if you try the browser reload too quickly, while App Engine is still restarting, then it may not respond or breakpoints may be missed.

To learn more about the debugger, try the **Tutorial** in Wing's **Help** menu.

### **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 `_run_file` in `dev_appserver.py` and start debugging. Then, after `script_name` has been set, in the **Debug Probe** tool (in Wing Pro only) type the following:

```
os.pathsep.join(_PATHS.script_paths(script_name))
```

Copy this to the clipboard and open up the file properties for `dev_appserver.py` by right-clicking on the file. Then, in **Project Properties** 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.

If you use more than one app within your project with multiple Named Entry Points, you'll want to set this **Python Path** into the **Named Entry Points's Launch Configuration** environment instead of placing it in **Project Properties**.

### ***Trouble-shooting***

App Engine runs code in a secure environment that prevents access to some system information, including process ID. This causes some of the sub-processes created by App Engine to be shown with process id -1. In this case they are not listed as children of the parent process and you will need to kill both processes, one at a time, from the toolbar or **Debug** menu.

Windows users may need 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 (this is a Windows runtime bug). One possible symptom of this is repeated 302 redirects that prevent logging in or other use of the site.

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.

If you have unchecked the "Show this dialog before each run" checkbox in the debug dialog shown when launching `dev_appserver.py` and need to alter the command line arguments or other values there, you can access the dialog by right clicking on `dev_appserver.py` in the editor or **Project** and selecting **Properties**.

### ***Related Documents***

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

- [Remote Web Development](#) describes how to set up development to a remote host.
- [Google App Engine home page](#), which provides links to downloads and documentation.
- [Wing Quickstart Guide](#) which contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#), which describes Wing in detail.

## ***2.9. Using Wing with mod\_wsgi***

[Wing](#) is a Python IDE that can be used to develop, test, and debug Python code that is running under [mod\\_wsgi](#). Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing for `mod_wsgi`. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the [Quickstart Guide](#).

### ***Debugging Setup***

When debugging Python code running under `mod_wsgi`, the debug process is initiated from outside of Wing, 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_wsgi` 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 `Accept Debug Connections` is checked. After that, you should be able to reach breakpoints by loading pages in your browser.

### ***Disabling stdin/stdout Restrictions***

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 provides many other options and tools. For more information:

- [Remote Web Development](#) describes how to set up development to a remote host.
- [Quickstart Guide](#) which contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#), which describes Wing in detail.

## ***2.10. Remote Web Development***

[Wing Pro](#) is a Python IDE that can be used to develop, test, and debug websites running on a remote server, VM, or other system where an IDE cannot be installed. Debugging takes place in the context of the web server or web framework, as code is invoked by browser page load. Wing provides auto-completion, call tips, find uses, and many other features that help you work with Python code.

If you do not already have Wing Pro installed, [download it now](#).

This document focuses on how to configure Wing for remote web development. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's `Help` menu or read the [Quickstart Guide](#).

### ***Setting up SSH Access***

Wing Pro's [remote development support](#) assumes that you have already set up password-less remote access to the remote host. This means that you should have your SSH key loaded into the SSH agent so that you can type `ssh hostname` or equivalent on the command line to connect to the remote host without entering a password.

If you don't already have this working, see [SSH Setup Details](#) for instructions.

### ***Installing the Remote Agent***

The next step is to set up a remote host configuration from `Remote Hosts` in the `Project` menu. Press the `+` icon to add a new remote host configuration.

You will need to choose an identifier that Wing will use to refer to the host, enter the hostname (optionally in `username@hostname` form) and specify where on the remote host you want Wing to install its remote agent (this directory is referred to as `WINGHOME`). You will only rarely need to specify any of the other values in a remote host configuration. For now, leave them blank. For example:



Once you submit the dialog for creating the configuration, Wing will try to probe that remote host. It will fail and prompt to install the remote agent like this:



Press `Install Remote Agent`. After the installation is complete, Wing probes again and should show a success dialog with some information about the remote host:



Next return to the `Remote Hosts` dialog and specify that the remote host configuration you've just created should be shared, so that it isn't just stored in the currently open project:



### **Setting up a Project**

Now it's time to set up a project that accesses your files on the remote host. To do this, select `New Project` in the `Project` menu and use the `Generic Python` project type:



There is also a project type for remote projects, but let's use the default project type now so that you will see where the relevant configuration takes place.

Click `OK` to create the project, and then go into `Project Properties` from the `Project` menu. Set the `Python Executable` to `Remote` and select the remote host you've just configured:



Next add your files to the project with the `Add Existing Directory` item in the `Project` menu. Press the `Browse...` button next to the `Directory` field to display a dialog that browses the file system on the remote host. Go into the directory you want to add and press `Select Directory`.

Wing will spend some time scanning the disk and analyzing files but you should already be able to open and edit source files stored on the remote host from the `Project` tool.

### **Initiating Debug**

This How-To assumes you're going to be launching your web server or web framework from outside of Wing and want to debug-enable code that is invoked as you browse your development website. The way Wing does this is by providing a module `wingdbstub.py` that you can import to initiate debug.

When you installed the remote agent above, Wing wrote a preconfigured copy of `wingdbstub.py` to the `WINGHOME` directory you specified in your remote host configuration. Copy that file to the same directory as your code and add the following to your code before it reaches anything you'll want to debug:

```
import wingdbstub
```

Next tell Wing to listen for connections from an externally launched debug process. This is done by clicking on the bug icon in the lower left of Wing's window and checking on `Accept Debug Connections`:



If you hover over the bug icon, Wing shows that it is listening for connections, both on the local host and on the configured remote host:



**Note:** If you are using `apache` or otherwise run your code as a user that is different from the one used in your remote host configuration, you'll need to adjust permissions on some files as described in the section [Managing Permissions](#) below.

### ***Debugging Code***

Now you can set some breakpoints by clicking on the breakpoint margin to the left of your code. For example:



Once this is done you should be able to point your browser at a web page that executes code with a breakpoint, and Wing should stop on it:



Use `Stack Data` to view debug data:



Or just interact on the command line within the current stack frame in the `Debug Probe` tool:



Both of these tools are accessible from Wing's `Tools` menu.

This technique should work with any web development framework. See [Web Development How-Tos](#) for details on using Wing with specific frameworks.

### ***Managing Permissions***

If you are debugging code that is running under Apache or another web server, the code is being run as a user that may not be the same as the one you used when installing the remote agent. For example, your remote host configuration may set `Host Name` to `devel@192.168.0.50` while the code is actually run by the user `apache`.

In this case, Wing will not accept the debug connection because the security token from the user running the code does not match what Wing is expecting from the way it installed and configured the remote agent.

To solve this, go into the `WINGHOME` where you installed the remote agent and change the permissions of the file `wingdebugpw` so that the user running the code can also read it. For example, if both your users are members of the group `webdev` then you can do this:

```
chgrp webdev wingdebugpw
```

A less secure solution is just to change the permissions of this file so everyone can read it:

```
chmod 644 wingdebugpw
```

The disadvantage of this approach is that other users could potentially use the contents of this file to connect to your instance of Wing against your will.

## Resources

- [Web Development How-Tos](#) contains instructions for using Wing with specific web development frameworks, such as [Django](#), [Flask](#), [Pyramid](#), [web2py](#), and others.
- [Remote Hosts](#) documentation provides details for configuring Wing Pro to work with remote hosts.
- [Quick Start](#) provides an introduction to Wing's features.
- [Tutorial](#) takes you through Wing's features step by step.

## 2.11. Using Wing with mod\_python

[Wing](#) is a Python IDE that can be used to develop, test, and debug Python code that is run by the [mod\\_python](#) module for the Apache web server. Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing for mod\_python. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's `Help` menu or read the [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 install directory listed in Wing's `About` box) 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 [Manually Configured Remote Debugging](#) section of the Wing 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. 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.
- Allow debug connections to Wing by setting the `Accept Debug Connections` 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"):
    wingdbstub.Ensure()
    return "I am saying %s" % what
```

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

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

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

### **Related Documents**

- [Remote Web Development](#) describes how to set up development to a remote host.
- [Mod\\_python Manual](#), describes how to install, configure, and use `mod_python`.
- [Quickstart Guide](#) contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#), which describes Wing in detail.

## **2.12. Using Wing with Pylons**

Wing is a Python IDE that can be used to develop, test, and debug Python code written for [Pylons](#). Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing for Pylons. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's `Help` menu or read the [Quickstart Guide](#).

### **Installing Pylons**

The [Pylons website](#) provide complete instructions for installing Pylons.

### **Debugging in Wing**

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

- [Remote Web Development](#) describes how to set up development to a remote host.
- [Pylons home page](#), which provides links to documentation.
- [Quickstart Guide](#) which contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#), which describes Wing in detail.

## ***2.13. Using Wing with Webware***

[Wing](#) is a Python IDE that can be used to develop, test, and debug Python code written for [Webware](#), an open source web development framework. Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing for Webware. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's `Help` menu or read the [Quickstart Guide](#).

### ***Introduction***

Wing 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 Pro, the `Debug Probe` tab allows you to interactively execute methods on objects and get values of variables that are available in the context of the running web app.

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

### ***Setting up a Project***

Though Wing supports the notion of "Projects" for organizing one's work for this debugging scenario you can use the `Default Project` and simply add your source code directory to it by using `Add Directory` 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 `Webware MakeAppDir.py` script creates a default directory structure and this example assumes 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 provides many other options and tools. For more information:

- [Remote Web Development](#) describes how to set up development to a remote host.
- [Quickstart Guide](#) which contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#), which describes Wing in detail.

## 2.14. Debugging Python CGI Scripts with Wing

**Wing** is a Python IDE that can be used to develop, 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, find uses, and many other features that help you write, navigate, and understand Python code.

Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing in order to work with CGIs. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the [Quickstart Guide](#).

### Introduction

To set up your CGIs for debugging with Wing, 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. 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 `wingdbstub` 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.

(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.

### 3.1. Using Wing with wxPython

[Wing](#) is a Python IDE that can be used to develop, test, and debug Python code written for the [wxPython](#) cross-platform GUI development toolkit. Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing for wxPython. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's `Help` menu or read the [Quickstart Guide](#).

#### Introduction

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

While Wing 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 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 [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 `wing6.1` 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. 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. 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

### ***Using a GUI Builder***

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

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

## ***3.2. Using Wing with PyQt***

[Wing](#) is a Python IDE that can be used to develop, test, and debug Python code written for the [PyQt](#) cross-platform GUI development toolkit. Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing for PyQt. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's `Help` menu or read the [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 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 for use with PyQt:

- Install Python, PyQt, and Wing. The [Quickstart Guide](#) provides installation instructions for Wing.
- Start Wing from the Start menu on Windows, the Finder on OS X, or by typing `wing6.1` 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 (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 Tutorial](#) and [Quick start](#) for more information.

## **Test Driving the Source Browser**

Don't forget to check out Wing Pro and Wing Personal'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 [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 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 provides many other options and tools. For more information:

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

## **3.3. Using Wing with GTK and PyGObject**

[Wing](#) is a Python IDE that can be used to develop, test, and debug Python code written for [GTK](#) using [PyGObject](#). Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing for GTK and PyGObject. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's `Help` menu or read the [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 added to the `Source Analysis > Advanced > Interface File Path` in preferences. The parent directory of the generated gi package should be added; if the defaults are used, the directory to add is `~/.cache/fakegir`.

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

Don't add the Fakedir produced package to the `Python Path` defined in Wing's `Project Properties` because code will not work if the fake module is actually on `sys.path` when importing any PyGObject-provided modules.

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 provides many other options and tools. For more information:

- [Wing Reference Manual](#), which describes Wing in detail.
- [Quickstart Guide](#) which contains additional basic information about getting started with Wing.

## **3.4. Using Wing with PyGTK**

Wing is a Python IDE that can be used to develop, test, and debug Python code written for [PyGTK](#) and [GTK+](#), an open source GUI development toolkit. Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing for PyGTK. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's `Help` menu or read the [Quickstart Guide](#).

### **Introduction**

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

- Install Python and Wing. See the generic [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 `wing6.1` 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 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 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 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 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 provides many other options and tools. For more information:

- [Wing Reference Manual](#), which describes Wing in detail.
- [Quickstart Guide](#) which contains additional basic information about getting started with Wing.

## How-Tos for Scientific and Engineering Tools

The following How-Tos provide tips and short cuts for using Wing with tools for scientific and engineering data analysis and visualization.

### 4.1. Using Wing with matplotlib

[Wing](#) is a Python IDE that can be used to develop, test, and debug Python code written for [matplotlib](#), a powerful plotting library. Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing for matplotlib. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's `Help` menu or read the [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 by typing the following:

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

In some environments, the `show()` call above will block until the plot window is closed. To avoid this, Wing 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. In fact `show()` is not needed at all here since Wing automatically shows and updates plots once `plot()` is called, but calling it is not a problem. 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`, `GtkCairo`, `WXAgg` (for wxPython 2.5+), `Qt5Agg`, `Qt4Agg`, and `MacOSX` 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 set a breakpoint where you want to stop, and then work in the Debug Probe. Type `show()` in the Debug Probe if your plot is not yet visible, or call `matplotlib.interactive(True)` to put matplotlib into interactive mode so that calling `show()` is no longer necessary.

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) When working in the Debug Probe, evaluate the imports that set up matplotlib first, so that Wing can initialize its event loop support before `show()` is called. Evaluating a whole file at once in the Debug Probe (but not the Python Shell) will cause `show()` to block if matplotlib was not previously imported.

(2) In case there is a problem with the specific matplotlib backend that you are using, try the following as a way to switch to another backend before issuing any other commands:

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

(3) 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](mailto:support@wingware.com) if you run into problems that you cannot resolve.

### ***Related Documents***

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

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

## ***4.2. Using Wing with Jupyter Notebooks***

[Wing](#) is a Python IDE that can be used to develop, test, and debug Python code written for [Jupyter](#), an open source scientific notebook system. Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing for Jupyter. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's `Help` menu or read the [Quickstart Guide](#).

### ***Setting up Debug***

Since Jupyter is started outside of Wing, you will need to initiate debug from your code or from the Jupyter notebook. There are a few configuration options that need to be set correctly for this to work properly.

**Limitation:** Jupyter does not provide a usable filename for code that resides directly in a notebook `.ipynb` file (it is simply set to names like `<ipython-input-1>`). As a result you cannot stop in or step through code in the notebook itself. Instead, you need to place your code in a Python file that is imported into the notebook, and then set breakpoints and step through code in the Python file.

### ***Configure wingdbstub.py***

To initiate debug, you will need to copy `wingdbstub.py` out of your Wing installation (on OS X it is located in `Contents/Resources` within the `.app` bundle) and place it in the same directory as your `.ipynb` file.

You may need to set `WINGHOME` inside of `wingdbstub.py` to the installation location of Wing. This is set automatically during installation of Wing except on OS X, on Windows if you use the zip installer, and

on Linux if you use the tar installer to install Wing. An alternative to editing `wingdbstub.py` is just to set the environment variable `WINGHOME` before you run `jupyter notebook`.

### Listen for Debug Connections

Next, tell Wing to listen for externally initiated debug connections by clicking on the bug icon in the lower left of Wing's window and checking on `Accept Debug Connections`.

### Starting Debug

Now add code like the following to the top of your Jupyter notebook:

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

When you run that cell, Wing will start debugging Jupyter. You should see Wing's toolbar change and the `Stack Data` tool should show one running process:



### Working with the Debugger

To try out debugging, save a file named `testdebug.py` in the same directory as your `.ipynb` file with the following contents:

```
def run():
    print("Hello world")
    x = 1
    print("Done")
```

Open this in Wing and place a breakpoint on the first line by of the body of `run()` by clicking on the breakpoint margin to the left, as follows:




Now add the following cell to your Jupyter notebook:

```
import testdebug
testdebug.run()
```

When you execute that cell, Wing should stop on the breakpoint in `testdebug.py`:



Now you can use the toolbar icons to step through code, view data in the `Stack Data` tool in Wing, interact in the context of the current debug stack frame with the `Debug Probe` (Wing Pro only), and use all of Wing's other debugging features on your code. See the `Tutorial` in Wing's `Help` menu for more detailed information on Wing's debugging capabilities.

To complete execution of your cell, press the green continue arrow  in the toolbar. Now if you execute the cell again, you should reach your breakpoint a second time. Then continue again to complete execution of the cell.

## Editing Code



Now try edit code in `testdebug.py` to change `Hello world` to `Hello everyone` and save the file. If you execute your cell again in Jupyter you'll notice the text being output has not changed. This is because the module has already been imported by Python and Jupyter is not automatically reloading it. To load your changes you'll need to restart the kernel from Jupyter's toolbar or its `Kernel` menu. In many cases `Restart` and `Run All` in the `Kernel` menu will be the most efficient way to reload your code and get back to your breakpoint.

Try selecting the `Source Assistant` from Wing's `Tools` menu and then adding some other code in `testdebug.py`, for example add `z = yy` for your code reads as follows:

```
def run():
    print("Hello everyone")
    z = yy
    print("Done")
```

Notice that Wing offers auto-completion and updates the `Source Assistant` with call tips, documentation, and other information about what you are typing, or what you have selected in the auto-completer. If a debug process is live and the code you are typing is in the stack, Wing includes also symbols found through inspection of the live runtime state in the auto-completer. In some code (but not the above example) this can include information Wing was not able to find through static analysis of the Python code.

Working in live code like this is a great way to write new code in the `Debug Probe`, where you can try it out immediately.

Or, you can work in the editor and try out selected lines of code by pressing the  icon in top right of the `Debug Probe` to make an active range. Once that is done, you can execute those lines repeatedly by pressing the  icon in the `Debug Probe`:



## Stopping on Exceptions

Since Jupyter handles all exceptions that occur while executing a cell, Wing will not stop on most exceptions in your code. Instead, you will get the usual report in the notebook output area.

Try this by now by restarting the Jupyter kernel and executing your edited copy of `testdebug.py`, which should read as follows:

```
def run():
    print("Hello everyone")
    z = yy
    print("Done")
```

Jupyter will report the exception in the notebook (undefined symbol `yy`), but Wing will not stop on it.

It is possible to get Wing to stop on exceptions, although currently the only way to do that is to edit code in IPython's `interactiveshell.py`. You can easily find that by setting a breakpoint in `run()` as before and going up the stack in Wing using the `Stack Data` or `Call Stack` tool. Then add the following code to the final `except:` clause in `InteractiveShell.runcode`. This will log the exception, which Wing takes as a clue that it should report the exception to the user:

```
if 'WINGDB_ACTIVE' in os.environ:
    import logging
    logging.exception(sys.exc_info()[1])
```

You will need to restart the Jupyter kernel after making this change. Then try executing your cell again and you will see Wing now reports the exception:



You can continue as usual from the exception and it will also be reported in the Jupyter notebook.

### ***Fixing Failure to Debug***



If you accidentally disconnect Wing's debugger from Jupyter, for example by pressing the red stop icon in Wing's toolbar, you can reestablish the debug connection at any time by re-executing the first cell we set up above, or by placing the following code into any other code that gets executed:

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

Note that if you plan to restart the Jupyter kernel every time you start debug then you don't need the `wingdbstub.Ensure` line. This makes sure that debug is active and connected to the IDE, so it is only needed if the debug connection has been dropped since the first time `wingdbstub` was imported.

If debugging stops working entirely and this does not solve it, you will need to restart the Jupyter kernel from its toolbar or `Kernel` menu and then re-execute the above code to start debugging again.

### ***Reloading Changed Modules***

The instructions above rely on restarting of the kernel as the way to reload changed code into Jupyter. Module reloading is also an option, making it possible to reload code without restarting the kernel.

Simple module reloading can be done using Python's builtin function `reload()` (or in Python 3.x instead `imp.reload()` after `import imp`). For details see [instructions for reloading in IPython](#).

Or, for more complex cases, the [autoreload extension](#) for IPython may help.

In general module reload can be problematic if old program state is not cleared correctly, and the complexity of this depends on the modules being used and their implementations. Simply restarting the kernel is always the safest option.

### ***Related Documents***

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

- [Wing Reference Manual](#), which describes Wing in detail.
- [Jupyter home page](#), which provides links to documentation.
- [Quickstart Guide](#), which contains additional basic information about getting started with Wing.
- [Tutorial](#), which provides a gentler introduction to Wing's features.

## **How-Tos for Modeling, Rendering, and Compositing Systems**

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

### ***5.1. Using Wing with Blender***

[Wing](#) is a Python IDE that can be used to develop, test, and debug Python code written for [Blender](#), an open source 3D content creation system. Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing for Blender. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the [Quickstart Guide](#).

### **Working with Blender**

Blender 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 (x86)\Wing IDE 6.0'
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.Ensure()

def runfile(filename):
    if sys.version_info < (3, 0):
        execfile(filename)
    else:
        import runpy
        runpy.run_path(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 Wing's About box (accessed from **Help** menu).
2. Run blender
3. Press **Shift-F11** to display the text editor
4. Press **Alt-O** to browse for a file 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 the **Run Script** button on the bottom toolbar 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 listening for externally initiated 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 **Accept Debug Connections** in the popup menu that appears.

### **Related Documents**

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

- [Wing Reference Manual](#), which describes Wing in detail.
- [Blender home page](#), which provides links to documentation.
- [Quickstart Guide](#), which contains additional basic information about getting started with Wing.
- [Tutorial](#), which provides a gentler introduction to Wing's features.



## 5.2. Using Wing with Autodesk Maya

Wing is a Python IDE that can be used to develop, test, and debug Python code written for [Autodesk Maya](#), a commercial 3D modeling application. Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing for Maya. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's `Help` menu or read the [Quickstart Guide](#).

### Debugging Setup

When debugging Python code running under Maya, the debug process is initiated from outside of Wing, and must connect to the IDE. This is done with `wingdbstub` according to the detailed instructions in the [Debugging Externally Launched Code](#) section of the manual. In summary, you will need to:

1. Copy `wingdbstub.py` from your Wing installation into a directory that will be on the `sys.path` when Python code is run by Maya. You may need to inspect that (after `import sys`) first from Maya, or you can add to the path with `sys.path.append()` before importing `wingdbstub`.
2. Because of how Maya sets up the Python interpreter, be sure to set `kEmbedded=1` in your copy of `wingdbstub.py`
3. In your code, where you initiate debugging, use the debugger API to ensure the debugger is connected to the IDE before any other code executes, as follows:

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

4. In some cases you may need to edit `wingdbstub.py` to set `WINGHOME` to point to the directory where Wing is installed. This is usually set up automatically by Wing's installer, but won't be if you are using the `.zip` installation of Wing. Note that if you edit `wingdbstub.py` after Maya has already imported it then you will need to restart Maya to get it to import the modified `wingdbstub`.
5. Then click on the bug icon in lower left of Wing's window and make sure that `Accept Debug Connections` is checked.

At this point, you should be able to reach breakpoints by causing the scripts to be invoked from Maya. In Maya 2018 at least, running a script does not set up the file name in the compiled Python code correctly, so breakpoints only work in modules that are imported into your top-level script. Breakpoints in the main script may work in older Maya versions.

Once debugging starts, when a breakpoint or exception is reached, Wing should come to the front and show the place where the debugger stopped. Although the code is running inside Maya, editing and debugging happens inside Wing.

### Using Maya's Python in Wing

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 Auto-completion

Setting `Python Executable` in Wing's `Project Properties`, as described above, is also needed to obtain auto-completion for Maya's Python API.

In some versions of Maya, additional configuration is needed before Wing will be able to statically analyze the files in the Python API and offer auto-completion for them. The solution to this depends on the version of Maya.



## Maya 2018

Maya 2018 ships with .pi files in the `devkit/pymel/extras/completion/pi` subdirectory of the Maya 2018 install directory. This can be added to the Source Analysis > Advanced > Interface File Path preference in Wing.

## Maya 2016

Maya 2016 is missing necessary developer files so you will need to download and install the [Maya 2016 devkit](#) which should create `devkit\other\pymel\extras\completion\py\maya\api` in your Maya installation. This can then be used by making the following edits:

```
In "OpenMaya.py" add "from _OpenMaya_py2 import *"
In "OpenMayaAnim.py" add "from _OpenMayaAnim_py2 import *"
In "OpenMayaRender.py" add "from _OpenMayaRender_py2 import *"
In "OpenMayaUI.py" and "from _OpenMayaUI_py2 import *"
```

This method is based on [this forum post](#).

Instead of editing files in the Maya installation, it is also possible to add .pi files with the added source. For example, placing `OpenMaya.pi` with contents `from _OpenMaya_py2 import *` in the same directory as `OpenMaya.py` causes Wing to merge the analysis of the \*.pi file with what is found in the \*.py file.

Alternatively, place these files in another directory that is added to the Source Analysis > Advanced > Interface File Path preference in Wing.

You will also want to set the Python Executable in Wing's Project Properties to `mayapy.exe` so that the API is on the Python Path and you are using the correct version of Python.

## Maya 2011+

Maya 2011+ before 2016 also shipped with .pi files that can be used as described for Maya 2018 above.

## Older Versions

For older Maya versions, .pi files from the PyMEL distribution at <http://code.google.com/p/pymel/> may be used. Just unpack the distribution and add `extras/completion/pi` to the Source Analysis > Advanced > Interface File Path preference in Wing.

## Additional Information

Some additional information about using Wing with Maya can be found in the [mel wiki](#) under the wing tag.

See also the section [Using Wing with Maya](#) in [Autodesk Maya Online Help: Tips and tricks for scripters new to Python](#).

## Related Documents

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

- [Wing Reference Manual](#), which describes Wing in detail.
- [Quickstart Guide](#) which contains additional basic information about getting started with Wing.

## 5.3. Using Wing with NUKE and NUKEX

[Wing](#) is a Python IDE that can be used to develop, test, and debug Python code written for The Foundry's [NUKE](#) and [NUKEX](#) digital compositing tool. Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing for NUKE and NUKEX. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's `Help` menu or read the [Quickstart Guide](#).

### **Project Configuration**

First, launch Wing 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's source analysis, search, and revision control features know which files are part of the project.

Next, make sure Wing 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. On Windows, you can create a batch file like this:

```
@echo off
"c:\Program Files\Nuke7.0v9\Nuke7.0.exe" -t -i %*
```

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 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 Pro**

Wing 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 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 Pro provides additional information for symbols in the auto-completer, editor, and other tools in Wing.

This technique will not work in Wing 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 and NUKE is to connect Wing 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 the install directory listed in Wing's `About` box and may need to set `WINGHOME` inside `wingdbstub.py` to the location where Wing is installed if this value is not already set by the Wing installer. On OS X, `WINGHOME` should be set to the full path of 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's main window and make sure that `Accept Debug Connections` 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'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, breakpoints set in Python modules should be reached and Wing's debugger can be used to inspect, step through code, and try out new code in the live runtime. Breakpoints set in the script itself won't be hit, though, due to how Nuke loads the script so code to be debugged should be put in modules that are imported.

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'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 provides many other options and tools. For more information:

- [Wing Reference Manual](#), which describes Wing in detail.
- [NUKE/NUKEX home page](#), which provides links to documentation.

- [Quickstart Guide](#) which contains additional basic information about getting started with Wing.

## 5.4. Using Wing with Source Filmmaker

Wing is a Python IDE that can be used to develop, test, and debug Python code written for [Source Filmmaker \(SFM\)](#), a movie-making tool built by Valve using the Source game engine. Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing for Source Filmmaker. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's `Help` menu or read the [Quickstart Guide](#).

### Debugging Setup

Wing can debug Python code that's saved in a file, but not code entered in the Script Editor window. As of version 0.9.8.5 (released May 2014), this includes scripts run from the main menu. In all versions, code in imported modules may be debugged.

When debugging Python code running under *SFM*, the debug process is initiated from outside of Wing, and must connect to the IDE. This is done with `wingdbstub`, as described in in the [Debugging Externally Launched Code](#) section of the manual. Because of how *SFM* sets up the interpreter, you must set `kEmbedded=1` in your copy of `wingdbstub.py`.

As of May 2014, *SFM* comes with `wingdbstub.py` in the site-packages directory in its Python installation. If an older version of *SFM* is being used or if Wing is installed into a nonstandard directory, copy `wingdbstub.py` from your Wing install directory to the site-packages directory. The default location of the site-packages directory is:

```
<STEAM>\steamapps\common\SourceFilmmaker\game\sdktools\python\2.7\win32\Lib\site-packages
```

Before debugging, click on the bug icon in lower left of Wing's window and make sure that `Accept Debug Connections` is checked. After that, you should be able to reach breakpoints by causing the scripts to be invoked from *SFM*.

To start debugging and ensure there's a connection from the *SFM* script being debugged to Wing, execute the following before any other code executes:

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

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

### Related Documents

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

- [Wing Reference Manual](#), which describes Wing in detail.
- [Quickstart Guide](#) which contains additional basic information about getting started with Wing.
- [Wing Tutorial](#) which provides a tour of Wing's feature set.

## How-Tos for Educational Tools

The following How-Tos provide tips and short cuts for using Wing with Python-related hardware and libraries used in education.

## 6.1. Using Wing with Raspberry Pi

### Note

"Within a couple of minutes I could fence in and eliminate an error with the handling of a GPRS modem attached to the Raspberry Pi that before I was trying to hunt down for hours."

-- Robert Rottermann, [redCOR AG](#)

Wing is a Python IDE that can be used to develop, test, and debug Python code running on the [Raspberry Pi](#). Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing for Raspberry Pi. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's `Help` menu or read the [Quickstart Guide](#).

### Introduction

Wing does not run on the Raspberry Pi, but you can set up Wing on a computer connected to the Raspberry Pi to work on and debug Python code remotely.

If you have Wing Pro, then you can set up development on the Raspberry Pi very quickly using Wing Pro's remote development features, as described in the following section.

If you have Wing Personal, you will need to set up remote file sharing and debugging manually, as described in `Manual Configuration for Wing Personal` below.

In either case you will first need a TCP/IP network connection between the machine where Wing is running and the Raspberry Pi. The easiest way to connect the Raspberry Pi to your network is with ethernet, or see the instructions at the end of this document for configuring a wifi connection.

### Remote Development with Wing Pro

To use Wing Pro's remote development capabilities with the Raspberry Pi, take the following steps:

- If you do not already have Wing installed, [download it now](#) on Windows, Linux, or OS X.
- Make sure you can connect to the Raspberry Pi from the machine where Wing IDE will be running, using `ssh` (or `PuTTY` on Windows) without entering a password. You need to set up the SSH keys on each machine, and load them into your SSH user agent via `ssh-add` (or in `Pageant` on Windows).
- Start up Wing and use `New Project` from the `Project` menu to create a project. Select project type `Raspberry Pi`. Fill in the fields as follows:
  - **Identifier** -- Set this to `rasp` or some other string to identify the Raspberry Pi
  - **Host Name** -- Set this to the string you use to SSH into the Raspberry Pi. In most cases you'll need both a username and IP address, such as `pi@192.168.0.2`.
  - **WINGHOME** -- Set this to the full path of the location where you would like Wing to install its remote agent on the Raspberry Pi, for example `/home/pi/winghome`.

Note that you can edit your configuration later, or add remote hosts to any project, from the `Remote Hosts` item in the `Project` menu.

- Next click `OK` to create that remote host. Wing will attempt to contact the Raspberry Pi and fail. Press the `Install Remote Agent` button to install the remote agent in the directory you specified as `WINGHOME` above. Wing will run the installation and try again to establish a connection



to the remote agent running on the Raspberry Pi. If this fails, details of the SSH command's output will be given in the resulting dialog.

- Once you have the remote agent working, go into `Project Properties` and set `Python Executable` to `Remote` and choose the remote host definition you just created above. Then click `OK` to save the project configuration. If you have not already done so, save the project to disk using a name ending in `.wpr`, for example `raspremote.wpr`.
- Next right click on the `Project` tool in Wing and select `Add Existing Directory`. In the resulting dialog, press `Browse` to choose directories on the Raspberry Pi.

Once this is done, you can open files from the `Project` tool, with `Open From Project` and in other ways, and work with them as if they were on your local machine. That includes debugging, running unit tests, issuing revision control commands, searching, running a Python Shell or OS Commands remotely, and using other features like goto-definition, find uses, and refactoring.

### **Manual Configuration for Wing Personal**

This section describes how to set up remote debugging on a Raspberry Pi manually, for Wing Personal. These instructions also work with Wing Pro but it is much easier to use Wing Pro's remote development features (see instructions above).

To do this, you will first need (1) a network connection between the Raspberry Pi and the computer where Wing will be running, and (2) a way to share files from the machine running Wing and the Raspberry Pi. For file sharing, use `Samba`, or simply transfer a copy of your files to the Raspberry Pi using `scp` or `rsync`.

### **Installing and Configuring the Debugger**

Once you have a network connection and some sort of file sharing set up, the next step is to install and configure Wing's debugger. This is done as follows:

- If you do not already have Wing installed, [download it now](#) on Windows, Linux, or OS X.
- Download the [Raspberry Pi debugger package](#) to your Raspberry Pi and unpack it with `tar xzf wingide-debugger-raspbian-6.1.5-1.tar.bz2`. This creates a directory named `wing-debugger-raspbian-6.1.5-1`.
- Launch Wing and make sure that `Accept Debug Connections` is checked when you click on the bug icon in the lower left of Wing's main window. Hovering the mouse over the bug icon will show additional status information, including the port Wing is listening on, which should be `50005` by default.
- Copy `wingdebugpw` from the machine where you have Wing installed to the Raspberry Pi and place it into the directory `wing-debugger-raspbian-6.1.5-1`. This file is located in the `Settings Directory`, which is listed 5th in Wing's `About` box.
- On the Raspberry Pi, use `/sbin/ifconfig` to determine the IP address of the Raspberry Pi (not `127.0.0.1`, but instead the number listed under `eth0` or `wlan0` if you're using wifi).
- On the host where Wing is running (not the Raspberry Pi), establish an `ssh` reverse tunnel to the Raspberry Pi so the debugger can connect back to the IDE. On Linux and OS X this is done as follows:

```
ssh -N -R 50005:localhost:50005 <user>@<rasp_ip>
```

You'll need to replace `<user>@<rasp_ip>` with the login name on the Raspberry Pi and the ip address from the previous step.

The `-f` option can be added just after `ssh` to cause `ssh` to run in the background. Without this option, you can use `Ctrl-C` to terminate the tunnel. With it, you'll need to use `ps` and `kill` to manage the process.

On Windows, use [PuTTY](#) to configure an ssh tunnel using the same settings on the Connections > SSH > Tunnels page: Set Source port to 50005, Destination to localhost:50005, and select the Remote radio button, then press the Add button. Once this is done the tunnel will be established whenever PuTTY is connected to the Raspberry Pi.

- In Wing's Preferences, use the Debugger > Advanced > Location Map preference to set up a mapping from the location of your files on the remote host (the Raspberry Pi) and the machine where the IDE is running.

For example, if you have files in `/home/pi/` on your Raspberry Pi that match those in `/Users/pitest/src/` on the machine where Wing is running, then you would add those two to the location mapping for 127.0.0.1, with `home/pi/` as the remote directory and `/Users/pitest/src/` as the local directory. On Windows the IP address to use in the location map may instead be the IP address of the host where Wing is running. This depends on the peer ip that is reported on the IDE side for sockets opened through the pipe.

Don't add a location map for the Raspberry Pi's ip address because your ssh tunnel makes it look like the connection is coming from the local host where the IDE is running.

### ***Invoking the Debugger***

There are two ways to invoke the debugger: (1) from the command line, or (2) from within your Python code. The latter is useful if debugging code running under a web server or other environment not launched from the command line.

### **Debugging from the Command Line**

To invoke the debugger without modifying any code, use the following command:

```
wing-debugger-raspbian-6.1.5-1/wingdb yourfile.py arg1 arg2
```

This is the same thing as `python yourfile.py arg1 arg2` but runs your code in Wing's debugger so you can stop at breakpoints and exceptions in the IDE, step through your code, and interact using the Debug Probe in the Tools menu.

By default this runs with `python` and connects the debugger to `localhost:50005`, which matches the above configuration. To change which Python is run, set the environment variable `WINGDB_PYTHON`:

```
export WINGDB_PYTHON=/some/other/python
```

Use the [Tutorial](#) in Wing's Help menu to learn more about the features available in Wing.

### **Starting Debug from Python Code**

To start debug from within Python code that is already running, edit `wing-debugger-raspbian-6.1.5-1/wingdbstub.py` and change the line `WINGHOME = None` to `WINGHOME = /home/pi/wing-debugger-raspbian-6.1.5-1` where `/home/pi` should be replaced with the full path where you unpacked the debugger package earlier. Use `pwd` to obtain the full path if you don't know what it is.

Copy your edited `wingdbstub.py` into the same directory as your code and add `import wingdbstub` to your code. This new line is what initiates debugging and connects back to the IDE through the ssh tunnel.

An alternative to editing `wingdbstub.py` is to set `WINGHOME` in the environment instead with a command like `export WINGHOME=/home/pi/wing-debugger-raspbian-6.1.5-1`.



## Configuration Details

If for some reason you can't use port 50005 as the debug port on either machine, this can be changed on the Raspberry Pi with `kHostPort` in `wingdbstub.py` or with the `WINGDB_HOSTPORT` environment variable. To change the port the IDE is listening on, use the Debugger > Listening > Server Port preference and or Debug Server Port in Project Properties in Wing.

If this is done, you will need to replace the port numbers in the ssh tunnel invocation in the following form:

```
ssh -N -R <remote_port>:localhost:<ide_port> <user>@<rasp_ip>
```

The first port number is the port specified in `kHostPort` or with `WINGDB_HOSTPORT` environment variable, and the second one is the port set in Wing's preferences or Project Properties.

On Windows using PuTTY, the `Source` port is the port set with `kHostPort` or `WINGDB_HOSTPORT` on the Raspberry Pi, and the port in the `Destination` is the port Wing is configured to listen on.

Refer to the documentation for `ssh` or `PuTTY` for details.

## Trouble-Shooting

There are several ways in which a debug configuration can fail and when a connection cannot be established to the IDE code will run without debug. Additional diagnostic output is needed to find the cause of most problems. This is done by setting an extra environment variable before initiating debug on the Raspberry Pi:

```
export WINGDB_LOGFILE=/home/pi/debug.log
```

Hovering the mouse over the bug icon in the lower left of Wing's window will show if a debug connection is active. Wing also adds icons to the toolbar while debugging.

If Wing is not receiving a connection, check the reverse ssh tunnel, make sure that `wingdebugpw` was copied, and check that Wing is listening for debug connections.

If Wing is receiving a connection but breakpoints are not reached or source code is not shown when reaching an exception, check your location map preference. A good way to test this is to add a deliberate unhandled exception to your code (such as `assert 0`) to see if Wing's debugger stops but fails to show the source code. The location map must be correct for Wing to show the source code.

## Setting up Wifi on a Raspberry Pi

It is possible to easily and cheaply connect a Raspberry Pi 2 to a wifi network. Here are instructions for doing this using an Edimax EW-7811Un wifi USB card (although other cards may also work) for a passphrase-protected wifi network:

- Plug in the USB wifi card and reboot your Raspberry Pi
- Edit `/etc/network/interfaces` and comment out the interface for `wlan1`. Nothing works if this is not done.
- Edit `/etc/wpa_supplicant/wpa_supplicant.conf` and add the following to the end:

```
network={
    ssid="<yourssid>"
    scan_ssid=1
    key_mgmt=WPA-PSK
    psk="<yourpassphrase>"
}
```

Replace `<yourssid>` your wifi network name and `<yourpassphrase>` with your wifi passphrase. Be sure to use exactly the above with no changes in spacing and with the quotes for the ssid and passphrase but not for other things. Otherwise nothing works and you won't get any usable error messages.

- Restart your Raspberry Pi again and wifi should work.

### ***Related Documents***

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

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

## **6.2. Using Wing with pygame**

[Wing](#) is a Python IDE that can be used to develop, test, and debug Python code written for [pygame](#), an open source framework for game development with Python. Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing for pygame. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's `Help` menu or read the [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 provides many other options and tools. For more information:

- [Wing Reference Manual](#), which describes Wing in detail.
- [Quickstart Guide](#) which contains additional basic information about getting started with Wing.

## **How-Tos for Other Frameworks and Tools**

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

### ***7.1. Using Wing with virtualenv***

[Wing](#) is a Python IDE that can be used to develop, test, and debug Python code. Wing supports [virtualenv](#), providing auto-completion, call tips, goto-definition, find uses, refactoring, a powerful debugger, unit testing, and many other features that help you navigate, understand, and write Python code.

Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing for virtualenv. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's `Help` menu or read the [Quickstart Guide](#).

## ***Creating a New Virtualenv***

If you are starting a new project from scratch and want to create a new virtualenv for the project, select `New Project` from the `Project` menu and use the `Create New Virtualenv` project type. You will need to enter the following values:

`Name` is the name for your virtualenv directory.

`Parent Directory` is the directory where the virtualenv directory will be created.

`Python Executable` selects the base Python installation to use. In Python 2, you must install virtualenv into the selected Python first, if it's not already present.

`Inherit global site-packages` controls whether to use the `--system-site-packages` option when running virtualenv. When checked, the virtualenv will be able to use packages installed into the base Python installation. Otherwise, it will be completely isolated from the base install, other than its use of Python's standard libraries.

`Auto-save project` controls whether Wing automatically saves its project file to the virtualenv directory. When checked, the project is named using the `Name` entered above plus `.wpr` and is stored in the top level of the virtualenv directory. In Wing Pro, which separates sharable project data from user-specific data, a second file ending in `.wpu` will also be written.

After submitting the `New Project` dialog, Wing will create the virtualenv, set the `Python Executable` in `Project Properties` to the `python` inside the virtualenv directory, and add the virtualenv directory to the project.

Now source analysis, executing, debugging, and testing in Wing will use the new virtualenv, as long as the project you just created is open. You will need to restart the `Python Shell` tool in Wing before it uses the newly created virtualenv.

## ***Working on a Remote Host***

Wing Pro can also create a new virtualenv on a remote host. This is done the same way as described above, except you will choose `Remote` for `Python Executable` and then select or create a remote host configuration. The Python used for creating and running the virtualenv on the remote host is the one chosen in the remote host configuration's `Python Executable`.

If you check `Auto-save Project` you will need to mark your remote host configuration as shared in the remote host configuration management dialog, which can be accessed from `Remote Hosts` in the `Project` menu or with the `Manage` button under `Python Executable` in the `New Project` dialog. This is needed so remote host configuration is stored locally, while the project will be stored on the remote host.

## ***Using an Existing Virtualenv***

To use an existing virtualenv with Wing, simply set the `Python Executable` in Wing's `Project Properties` to the `python` inside your virtualenv directory. Wing uses this to determine the environment to use for source analysis and to execute, test, and debug your code.

The easiest way to find the correct value to set is to launch your virtualenv Python outside of Wing and run `import sys; print(sys.executable)`. Then use the full path that prints as the `Python Executable` in Wing.

## ***Activating the Virtualenv***

If you follow the above instructions, Wing will automatically activate the virtualenv while you're using your project.

An alternative approach is to leave `Python Executable` unset and instead activate the virtualenv on the command line and then start Wing from the command line so that it inherits the virtual environment. However, setting `Python Executable` is preferable so that Wing switches virtual environments when you switch projects, without restarting the IDE.

## Related Documents

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

- [Wing Reference Manual](#), which describes Wing in detail.
- [Wing Tutorial](#)
- [Quickstart Guide](#)

## 7.2. Using Wing with Vagrant

[Wing](#) is a Python IDE that can be used to develop, test, and debug Python code running on [Vagrant](#) containers.

This document describes how to configure Wing for Vagrant. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's `Help` menu or read the [Quickstart Guide](#).

### Prerequisites

This guide assumes you have already installed and started using [Vagrant](#). Wing supports working with Vagrant boxes running Linux (32-bit or 64-bit) or OS X. Other OS types, such as FreeBSD, won't work.

If you do not already have Wing installed, [download it now](#).

### Creating a Project

Creating a Wing project for an existing Vagrant container is easy:

1. Start your container with `vagrant up`
2. Use `New Project` from the `Project` menu and then select `Connect to Vagrant` as your project type
3. Fill in the full path to your Vagrant project directory (which contains your `Vagrantfile`) and press `OK`
4. Only on Windows with PuTTY, you will be asked to convert Vagrant's private key into a PuTTY key. To do this, Wing starts `puttygen` with the private key already loaded into it. Press `Save private key` to save the key as `private_key.ppk` in the current directory. Confirm saving without password (the original also doesn't have a password) and then quit `puttygen` to continue the project setup process in Wing.
5. Install the remote agent when prompted.

Once this is done, Wing should confirm that the remote agent installation is working. Then you can add files and directories and configure your project from the `Project` menu. You will be able to edit, debug, test, search, and manage files on the container, or launch commands running in the container from Wing's `OS Commands` tool.

To learn more about Wing's remote development capabilities, see [Remote Hosts](#).

To learn more about Wing's features, take a look at the [Tutorial](#) or [Quickstart Guide](#).

### How It Works

Wing uses `vagrant ssh-config` to inspect your Vagrant container and fill in the necessary settings in Wing's project file.

To see the settings that Wing created during `New Project`, take a look at `Project Properties` from the `Project` menu. The `Python Executable` was set to point to a remote host named `vagrant`. Click on `Edit here` or use `Remote Hosts` in the `Project` menu to access the remote host configuration. The values that Wing sets up are: `Identifier`, `Hostname`, and `WINGHOME` under the `General` tab, and `SSH Port` and `Private Key` under the `Options` tab. Settings these values manually achieves exactly the same results as using `New Project`.

The installation process will also add the Vagrant container's SSH host key as a known host in `~/.ssh/known_hosts` (or in the registry if using PuTTY on Windows).

### **Usage Hints**

#### **Synced Folders**

As far as Wing is concerned, all files and directories are located in the Vagrant container and Wing never accesses local copies of the files maintained by Vagrant's synchronization commands.

If you need to update your local copies of files for some other reason while working with Wing, run `vagrant rsync`, or set up continuous synchronization with `vagrant rsync-auto`.

#### **Password-less Private Keys**

Vagrant uses password-less private keys by default. If you want to use private keys with passwords, you'll need to load the key into the SSH user agent (`ssh-agent` or `pageant` for PuTTY) and change `Options > Private Key` in Wing's `vagrant` remote host configuration to `Use SSH User Agent`.

#### **Related Documents**

For more information see:

- [Remote Hosts](#) for details on configuring remote development.
- [Quickstart Guide](#) which contains basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#), which describes Wing in detail.
- [Vagrant home page](#), which provides downloads and documentation.

### **7.3. Using Wing with Windows Subsystem for Linux**

Wing is a Python IDE that can be used to develop, test, and debug Python code running on [Windows Subsystem for Linux](#) (WSL).

This document describes how to configure Wing for WSL. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's `Help` menu or read the [Quickstart Guide](#).

#### **Prerequisites**

To use Wing with [WSL](#) you will need to install Python on Linux and be able to SSH from Windows into Linux under WSL without entering a password. If you haven't already done this, see [Setting up WSL](#) below before trying to create a project in Wing.

If you do not already have Wing installed, [download it now](#).

#### **Creating a Project**

To create a Wing project that accesses Linux under WSL:

1. Use `New Project` from the `Project` menu and then select `Connect to WSL` as your project type.
2. Set `Identifier` to `wsl` or some other short name for the WSL-hosted Linux.
3. Set `Host Name` to `username@127.0.0.1` where `username` is replaced with the user name running on Linux. The user name is needed even if it is the same as the user running on Windows.
4. Set `Python Executable` to `/usr/bin/python3` or the full path to the Python executable you wish to use on Linux.
5. If you are running the SSH server on Linux under a non-standard port, set `SSH Port` under the `Options` tab.

6. Leave the rest of the configuration values set to their defaults, press `OK`, and install the remote agent when prompted

Once this is done, Wing should confirm that the remote agent installation is working. Then you can add files and directories and configure your project from the `Project` menu. You will be able to edit, debug, test, search, and manage files on the WSL-hosted Linux installation, or launch commands running on Linux from Wing's `OS Commands` tool.

To learn more about Wing's remote development capabilities, see [Remote Hosts](#).

To learn more about Wing's features, take a look at the [Tutorial](#) or [Quickstart Guide](#).

## Notes

Selecting the `Connect to WSL` project type when creating your project automatically unchecks `Use SSH Tunnel for username@localhost` under the `Advanced` tab of the remote host configuration. Other than this, the remote host configuration created in the above instructions is the same as for any other remote host.

## Setting up WSL

Here is one way to set up WSL with Ubuntu as the Linux distribution and PuTTY as the SSH client:

### Enable WSL and Install Ubuntu Linux:

- Enable WSL in Windows 10+. This is done in the Settings app under `Apps > Apps & features > Related settings / Programs and features > Turn Windows features on and off`. Restart when prompted.
- Install Ubuntu from the Microsoft Store.
- Install Python in Ubuntu with `sudo apt-get install python3`

### Set up password-less SSH access to Ubuntu:

- Install [PuTTY](#) if you don't already have it. You should install all the tools from the MSI installer.
- Create an SSH key pair and add it to `~/.ssh/authorized_keys` on Ubuntu, as described in [Working with PuTTY](#).

### Each time you restart Windows or Ubuntu:

- Start `pageant` from Windows and right-click on the icon that appears in the lower right of Windows to load your private key into Pageant.
- Run `sudo service ssh --full-restart` on Ubuntu to make sure the SSH server is started.

Now running `plink username@127.0.0.1` on Windows should connect to Ubuntu without prompting for a password.

## Notes

In the unlikely event that you already have an SSH server running somewhere on your system on port 22, you will need to change the port number used on Ubuntu in `/etc/ssh/ssh_config` and then execute `sudo service ssh --full-restart` again.

Other Linux distributions are available in the Microsoft Store.

It is also possible to use Open SSH instead of PuTTY. See [Working with OpenSSH](#) for details.

## Related Documents

For more information see:

- [Remote Hosts](#) for details on configuring remote development.
- [Quickstart Guide](#) which contains basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.

- [Wing Reference Manual](#), which describes Wing in detail.

## 7.4. Using Wing with Twisted

[Wing](#) is a Python IDE that can be used to develop, test, and debug Python code written for [Twisted](#). Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing for Twisted. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's `Help` menu or read the [Quickstart Guide](#).

### Installing Twisted

The [Twisted website](#) provides complete instructions for installing and using Twisted.

### Debugging in Wing

To debug Twisted code launched from within Wing, 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 provides many other options and tools. For more information:

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

## 7.5. Using Wing with Cygwin

[Wing](#) is a Python IDE that can be used to develop, test, and debug Python code written for [cygwin](#), a Linux/Unix like environment for Microsoft Windows. Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).



This document describes how to configure Wing for Cygwin. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the [Quickstart Guide](#).

## Configuration

To write and debug code running under cygwin, download and install Wing for Windows on your machine. There is no Wing for cygwin specifically but you can set up Wing 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`. Wing will look for the `wingdebugpw` file in a cygwin `~/.wingide6` directory if it exists and is not empty; otherwise the standard win32 User Settings directory will be used.

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 provides many other options and tools. For more information:

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

## 7.6. Using Wing with scons

[Wing](#) is a Python IDE that can be used to develop, test, and debug Python code written for [scons](#), an open source software construction or build control framework that uses Python. Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing for scons. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's **Help** menu or read the [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, you will need to use `wingdbstub` to initiate debugging, as described in [Debugging Externally Launched Code](#).

### **Related Documents**

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

- [Wing Reference Manual](#), which describes Wing in detail.
- [Quickstart Guide](#) which contains additional basic information about getting started with Wing.

## **7.7. Using Wing with IDA Python**

Wing is a Python IDE that can be used to develop, test, and debug Python code written for [Hex-Rays IDA](#) multi-processor disassembler and debugger. Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing for IDA. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's `Help` menu or read the [Quickstart Guide](#).

### **Debugging IDA Python in Wing**

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 installation and may need to set `WINGHOME` inside `wingdbstub.py` to the location where Wing is installed (on OS X, the name of 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's main window and make sure that `Accept Debug Connections` 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.

At that point, any breakpoints set in Python code should be reached and Wing 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 provides many other options and tools. For more information:

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

## **7.8. Using Wing with IronPython**

[Wing](#) is a Python IDE that can be used to develop, test, and debug Python code written with [IronPython](#). Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional programmers, and Wing Personal is a free alternative with reduced feature set.

If you do not already have Wing installed, [download it now](#).

This document describes how to configure Wing for IronPython. To get started using Wing as your Python IDE, please refer to the tutorial in Wing's `Help` menu or read the [Quickstart Guide](#).

### **Project Configuration**

For more information on setting up Wing with IronPython, see [IronPython and the Wing: 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 install directory.

### **Related Documents**

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

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

## **7.9. 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 Pro, 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.

## 7.10. 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 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.

## 7.11. 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.

## 7.12. Debugging Code with XGrab\* Calls

Under X11, 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) Most Linux systems offer some way to break through X11 pointer and keyboard grabs.

For example, `x.org` installations define a key symbol that releases all pointer and keyboard grabs. You can map a key sequence to it with `xdotool` as in the following example:

```
xdotool ctrl+alt+n XF86Ungrab
```

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

When this option is not available, another option is to move processing into a timer or idle task so it occurs after the grab has been released.

(3) If all else fails, you can log in remotely, use `ps` to find the debug process, and kill it with `kill` or `kill -9`. This will unlock your X windows display.

(4) Setting `DISPLAY` to send your debug process window to another X display avoids tying up Wing in this way. The remote display will release its grabs once you kill the debug process from the IDE.

## 7.13. 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 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) to control the debug process from the IDE and to inform the IDE when events (such as reaching a breakpoint or exception) occur in the debug process.

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

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

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

### **Supporting Non-Python Mainloops**

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

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

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

Mainloop hooks are written as separate modules that are placed into `src/debug/tserver` within `WINGHOME` (the Wing installation directory, or on OS X `Contents/Resources` 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 `kSupportedMainloops` 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 <https://wingware.com/support>. We will be happy to assist you, and welcome the contribution of any hooks you may write.

## 7.14. 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\lib\site-packages
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 `Accept Debug Connections` in Wing 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.