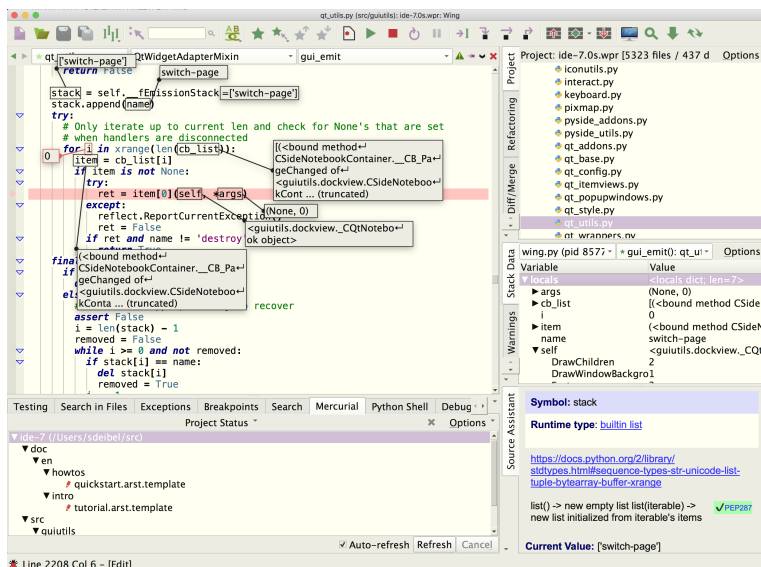




## How-Tos



### Version 7.2.9

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, scientific analysis, compositing, rendering, game development, and much more.

These How-Tos assume that you know how to use the Python framework or tool being discussed and that you are already somewhat familiar with 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" 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) 1999-2020 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>
How-Tos for Containers	1
1.1. Using Wing with virtualenv	2
Creating a New Virtualenv	2
Using an Existing Virtualenv	3
Activating the Virtualenv	3
Using Virtualenv with Anaconda	4
Related Documents	4
1.2. Using Wing with Anaconda	5
Configuring Your Project	5
Creating a New Anaconda Environment	6
About Anaconda Environments	6
Related Documents	7
1.3. Using Wing with Cygwin	8
Project Configuration	8
Debugger Configuration	8
File Paths	9
Related Documents	9
How-Tos for Scientific and Engineering Tools	10
2.1. Using Wing with Matplotlib	11
Working Interactively	11
Debugging	12
Trouble-shooting	13
Related Documents	13
2.2. Using Wing with Jupyter Notebooks	15
Setting up Debug	15
Working with the Debugger	16
Editing Code	18
Stopping on Exceptions	19

Fixing Failure to Debug	20
Reloading Changed Modules	21
Related Documents	21
2.3. Using Wing with PyXLL	22
Introduction	22
Installation and Configuration	23
Debugging Python Code in Excel	23
Trouble-shooting	24
Related Documents	24
How-Tos for Web Development	25
3.1. Using Wing with Django	26
Automated Configuration	26
Existing Django Project	27
New Django Project	27
Automated Django Tasks	28
Remote Development	28
Manual Configuration	28
Configuring the Project	28
Configuring the Debugger	29
Launching from Wing	29
Launching Outside of Wing	29
Debugging Django Templates	30
Usage Tips	30
Debugging Exceptions	30
Template Debugging	31
Better Auto-Completion	31
Running Unit Tests	31
Django with Buildout	31
Related Documents	31
3.2. Using Wing with Flask	32
Project Configuration	32
Remote Development	33

Debugging Flask in Wing	33
Setting up Auto-Reload with Wing Pro	33
Related Documents	34
3.3. Using Wing with Pyramid	35
Creating a Wing Project	35
Debugging	36
Launching from Wing	36
Auto-reloading Changes	36
Launching Outside of Wing	36
Notes on Auto-Completion	37
Debugging Jinja2 Templates	37
Debugging Mako Templates	37
Remote Development	38
Related Documents	38
3.4. Using Wing with web2py	39
Introduction	39
Setting up a Project	39
Remote Development	40
Debugging	40
Usage Tips	41
Setting Run Arguments	41
Hung Cron Processes	41
Better Auto-completion	41
Related Documents	41
3.5. Using Wing with mod_wsgi	42
Debugging Setup	42
Disabling stdin/stdout Restrictions	43
Remote Development	43
Related Documents	43
How-Tos for GUI Development	44
4.1. Using Wing with wxPython	45
Introduction	45

Installation and Configuration	45
Test Driving the Debugger	46
Using a GUI Builder	47
Related Documents	47
4.2. Using Wing with PyQt	48
Introduction	48
Installation and Configuration	48
Test Driving the Debugger	49
Using a GUI Builder	50
Related Documents	50
4.3. Using Wing with GTK and PyGObject	51
Introduction	51
Installation and Configuration	51
Test Driving the Debugger	52
Improving Auto-Completion	52
Using a GUI Builder	53
Related Documents	53
How-Tos for Modeling, Rendering, and Compositing Systems	54
5.1. Using Wing with Blender	55
Working with Blender	55
Related Documents	56
5.2. Using Wing with Autodesk Maya	57
Debugging Setup	57
Using Maya's Python in Wing	58
Better Static Auto-completion	58
Maya 2018	58
Maya 2016	58
Maya 2011+	59
Older Versions	59
Additional Information	59
Related Documents	59
5.3. Using Wing with NUKE and NUKEX	60

Project Configuration	60
Configuring for Licensed NUKE/NUKEX	60
Configuring for Personal Learning Edition of NUKE	61
Additional Project Configuration	61
Replacing the NUKE Script Editor with Wing Pro	61
Debugging Python Running Under NUKE	62
Debugger Configuration Detail	63
Limitations and Notes	63
Related Documents	64
5.4. Using Wing with Source Filmmaker	65
Debugging Setup	65
Related Documents	66
How-Tos for Educational Tools	67
6.1. Using Wing with Raspberry Pi	68
Introduction	68
Remote Development with Wing Pro	69
Manual Configuration for Wing Personal	69
Installing and Configuring the Debugger	70
Invoking the Debugger	70
Access Control	71
Configuration Details	71
Trouble-Shooting	72
Setting up Wifi on a Raspberry Pi	72
Related Documents	73
6.2. Using Wing with pygame	74
Project Configuration	74
Debugging	75
Related Documents	75
Unmaintained How-Tos	76
7.1. Using Wing with Twisted	77
Project Configuration	77
Remote Development	78

Debug Configuration	78
Related Documents	78
7.2. Using Wing with Plone	79
Introduction	79
Configuring your Project	80
Debugging	80
Debugging with WingDBG	80
WingDBG in Buildout-based Plone 4 Installations	81
Related Documents	81
7.3. Using Wing with Turbogears	82
Project Configuration	82
Debugging	83
Remote Development	83
Related Documents	83
7.4. Using Wing with Zope	84
Quick Start on a Single Host	84
Starting the Debugger	85
Test Drive Wing	85
Setting Up Auto-Refresh	86
Alternative Approach to Reloading	87
Setting up Remote Debugging	87
Upgrading from Earlier Wing Versions	88
Trouble Shooting Guide	88
Related Documents	89
7.5. Using Wing with mod_python	90
Introduction	90
Quick Start	90
Example	91
Remote Development	91
Related Documents	91
7.6. Debugging Code Running Under Py2exe	93
Configuring the Debugger	93

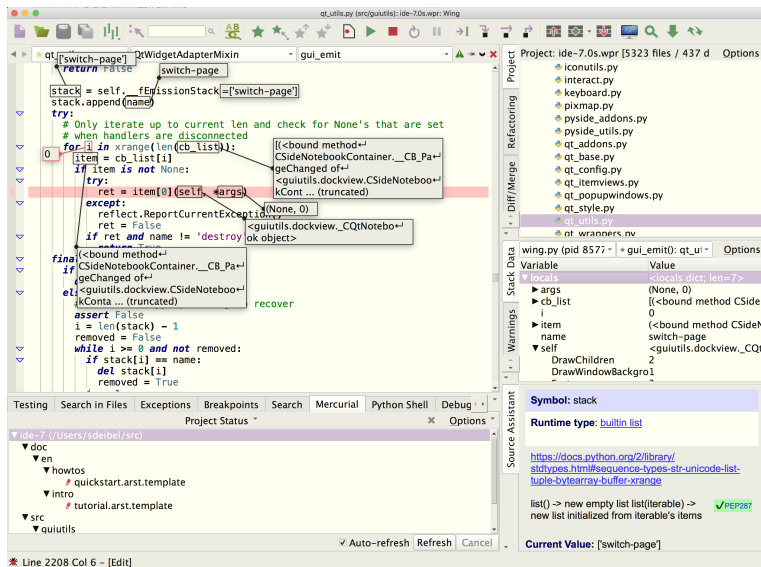


Related Documents	94
7.7. Using Wing with IDA Python	95
Debugging IDA Python in Wing	95
Related Documents	96
7.8. Using Wing with IronPython	97
Project Configuration	97
Related Documents	97

## **How-Tos for Containers**

The following How-Tos explain how to get started using Wing with containers, virtual machines, and remote hosts.

### 1.1. Using Wing with virtualenv



**Wing** is a Python IDE that can be used to develop, test, and debug Python code running in **virtualenv**.

Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional developers, 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.

**Packages** lets you specify packages to install into the new **virtualenv**. This is either a space-separated list of pip package specifications, or a file that contains one package specification per line. In either case, the package specifications may be anything accepted by pip, such as a package name, **package==version**, and **package>=version**.

**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.

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

**Upgrade pip** selects whether Wing should upgrade pip in the virtualenv before installing any packages, to compensate for the fact that virtualenv installs an old version of pip even if the base Python installation has a newer one.

**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 command that activates the environment, 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.

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

To use an existing virtualenv with Wing, simply set the **Python Executable** in Wing's **Project Properties** to **Activated Env** and enter the command that activates the environment. Wing uses this to determine the environment to use for source analysis and to execute, test, and debug your code. In this case, Wing starts Python by running **python** in that environment.

**Python Executable** can also be set to **Command Line** to enter the full path to the virtualenv's **python.exe** or **python**. The easiest way to find the correct value to set is to launch your virtualenv Python outside of Wing and execute the following:

```
import sys
print(sys.executable)
```

### ***Activating the Virtualenv***

If you followed 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, this is not recommended because the inherited environment may conflict with virtual environments used by other projects.

### ***Using Virtualenv with Anaconda***

Anaconda implements its own named environments, created by **conda create** but it is also possible to use virtualenv with Anaconda. This works in the same way, except that on Windows Wing will automatically call **conda activate base** before it sets up your virtualenv. This is needed to avoid failure to import some modules as a result of missing environment. See About Anaconda Environments in the [Anaconda How-To](#) for details.

### ***Related Documents***

For more information see:

- [Quickstart Guide](#) contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#) documents Wing in detail.

## 1.2. Using Wing with Anaconda



**Wing** is a Python IDE that can be used to develop, test, and debug Python code run with the [Anaconda Distribution](#) of Python.

Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional developers, 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 use with Anaconda 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](#).

### Configuring Your Project

To use Anaconda with an existing Wing project, simply set the **Python Executable** in your **Project Properties** in the **Project** menu to the interpreter that you want to use. There are several options for this:

**Command Line** can be selected to enter the full path to Anaconda's **python.exe** or **python**. In many cases, Wing will automatically find Anaconda and include it in the drop down menu to the right of the file selector shown for this option. The Python executable for Anaconda is typically located at the top level of the installation on Windows and in the **bin** sub-directory on other OSes. Another way to find the correct full path to use is to start Anaconda outside of Wing and then type the following:

```
import sys
print(sys.executable)
```

**Activated Env** can be selected to use an existing environment created with **conda create** or **virtualenv**. This should be the command that activates the environment, for example

**activate venv1**. In this case, Wing starts Python by running **python** in that environment. If Anaconda is installed in a default location, Wing will find your existing environments, which can be selected with the drop down menu to the right of this field.

If you are creating a new Wing project and want to use Anaconda, select **New Project** from the **Project** menu and configure **Python Executable** in the **New Project** dialog in the same way as described above.

In most cases, setting **Python Executable** is all that you need to do. Wing will start using your Anaconda installation immediately for source intelligence, for the next debug session, and in the integrated **Python Shell** after it is restarted from its **Options** menu.

### ***Creating a New Anaconda Environment***

Wing can create a new Anaconda environment with **conda create** at the same time that it creates a new project. To do this, select **New Project** from the **Project** menu and then choose **Create New Anaconda Environment** as the project type.

You will need to enter the name for the new environment, choose the location to write the new environment, select the installation directory of the Anaconda that you want to use, and specify at least one package to install into the new environment.

Package specifications may either be entered directly into the **New Project** dialog, in a space-separated list, or placed into a **requirements.txt** file, with one package specification per line. In both cases, the package specifications may be anything accepted by **conda install** including just the package name, **package==version**, or **package>=version**:

```
flask unicorn numpy==1.17.4 django>=3.1
```

When the **New Project** dialog is submitted, it will run **conda create** and then configure the project to use the new environment.

### ***About Anaconda Environments***

On Windows, Anaconda may fail to load DLLs when its **python.exe** is run directly without using a named environment. This is due to the fact that by default the Anaconda installer no longer sets the **PATH** that it needs to run, in order to avoid conflicting with different Python installations on the same system. A typical error message looks like this:

```
builtins.ImportError:
IMPORTANT: PLEASE READ THIS FOR ADVICE ON HOW TO SOLVE THIS ISSUE!
Importing the numpy c-extensions failed.
...
Original error was: DLL load failed: The specified module could not be found.
```

The exact message you see will vary depending on which packages you are using, or you may not run into this at all if you are not using packages that are affected by it.

## How-Tos for Containers

This may occur when running Anaconda Python outside of Wing without using a named Anaconda environment or when using virtualenv with Anaconda. The solution on the command line is to call **conda activate base** before starting Anaconda or activating the virtualenv.

The problem should not appear in Wing because it detects when Anaconda is being used and automatically activates the base environment before launching Anaconda.

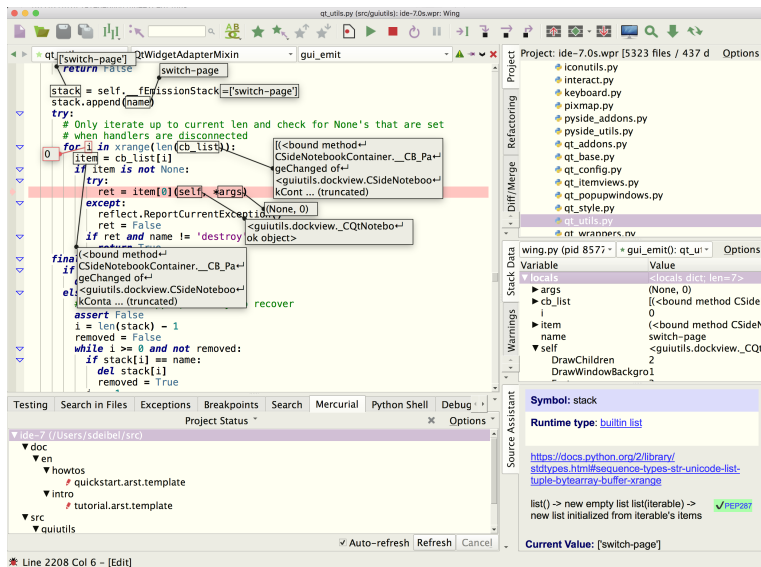
### ***Related Documents***

For more information see:

- [Quickstart Guide](#) contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#) documents Wing in detail.



### 1.3. 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 developers, 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](#).

**Note:** This document was last tested with cygwin 3.6.

#### Project Configuration

To write and debug code running under cygwin, download and install Wing for Windows on your machine. Wing does not run on cygwin but you can set up Wing for Windows to work with Python code that is running under cygwin.

This is done by creating a project with **New Project** in the **Project** menu, and then adding the Windows-side copies of your source files to the project with **Add Existing Directory**, which is also in the **Project** menu.

#### Debugger Configuration

To debug code running on cygwin, follow the instructions for [Debugging Externally Launched Code](#). In this model, you will always launch your Python code from cygwin and not from Wing's menus or toolbar.

## How-Tos for Containers

When setting this up, use cygwin paths for **WINGHOME** in **wingdbstub.py** because this file will be used on the cygwin side.

### ***File Paths***

In many cases, it's easiest to configure cygwin pathnames to be equivalent to the Windows pathnames. An example would be to set up **/src** in cygwin to point to the same directory as **\src** on Windows, which is **src** at top level of the main drive, usually **c:\src**.

If this is not possible, you should be sure to add all the sources you need to work with to your project in Wing. This way, Wing can automatically find all your files and use a hash on the contents of the file to identify which Windows-side files are the same as the cygwin files. See [File Location Maps](#) for details.

### ***Related Documents***

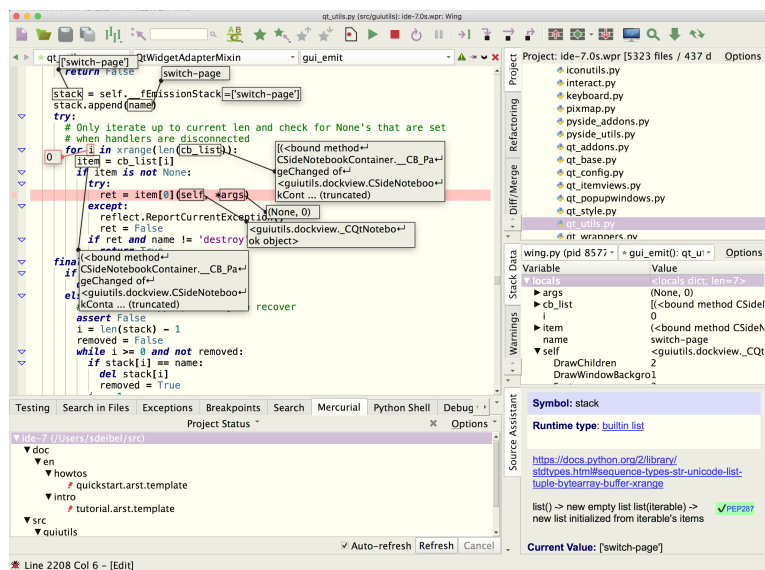
For more information see:

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

## **How-Tos for Scientific and Engineering Tools**

The following How-Tos explain how to get started using Wing with tools for scientific and engineering data analysis and visualization.

## 2.1. 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 numerical and scientific plotting library.

Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional developers, 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 Interactively

Wing supports interactive development and debugging of Python code designed for the Matplotlib numerical and scientific plotting library, so plots can be shown and updated from the command line. For example, two plots could be shown in succession by typing the following into Wing's *integrated* *Python Shell*, one line at a time:




```
from matplotlib.pyplot import plot, show
x = range(10)
plot(x)
show()
y = [2, 8, 3, 9, 4] * 2
plot(y)
```

Wing sets up the environment so that **show()** runs to completion and immediately returns you to the prompt, rather than waiting until the plot is closed. In addition, Wing calls Matplotlib's main loop to keep plots windows interactive and updating while you are at the prompt. This allows plots to be added or changed without restarting a process or interrupting your work flow.

## Evaluating Files and Selections

Code from the editor can be executed in the **Python Shell** with the **Evaluate ... in Python Shell** items in the **Source** menu and in the editor's right-click context menu. By default the **Python Shell** restarts automatically before evaluating a whole file, but this can be disabled in its **Options** menu.


## Active Ranges

Wing also allows you to set a selected range of lines in the editor as the "active range" for the **Python Shell** by clicking the  icon in the top right of the **Python Shell** tool. Wing highlights and maintains the active range as you edit it in the editor, and it can be re-evaluated easily with the  icon that appears in the top right of the **Python Shell** once an active range has been set into it. Use the  icon to clear the active range from the editor and shell.

## Supported Backends

Interactive development is supported for the **TkAgg**, **GTKAgg**, **GtkCairo**, **WXAgg** (for wxPython 2.5+), **Qt5Agg**, **Qt4Agg**, **MacOSX**, and **WebAgg** backends. It will not work with other backends.

## Debugging

Code can be debugged either by launching a file with  in the toolbar (or **Start/Continue** the **Debug** menu) or by enabling debug in the integrated **Python Shell** and working from there. In either case, Wing can be used to reach breakpoints or exceptions, step through code, and view the program's data. For general information on using Wing's debugger see the [Debugger Quick Start](#).

When executing code that includes **show()** in the debugger, Wing will block within the **show()** call just as Python would if launched on the same file. This is by design, since the debugger seeks to replicate as closely as possible how Python normally runs.

However, interactive development from a breakpoint or exception is still possible, as described below. This capability can be used to load setup code before interacting with Matplotlib, or to try out a fix when an exception has been reached.

## Interactive Debugging from the Debug Console (Wing Pro only)

Whenever the debugger is stopped at a breakpoint or exception, Wing Pro's **Debug Console** provides a command prompt that may be used to inspect and interact with the paused debug process. Commands entered here run in the context of the currently selected debug stack frame.

The tool implements the same support for interactive development provided by the **Python Shell**, so plots may be shown and modified interactively whenever Wing's debugger is paused. Once the debug process is continued, Wing switches off interactive mode and returns to behaving in the same way that Python would when running the code outside of the debugger.

### Note

Interactive development from the **Debug Console** requires that you have already imported **matplotlib** in the code that you are debugging or in a previous command entered in the console. Otherwise **show()** may block and plots won't be updated.

### Interactive Debugging from the Python Shell

Another way to combine the debugger with interactive development is to turn on both **Enable Debugging** and **Enable Recursive Prompt** in the **Python Shell's Options** menu. This causes Wing to add a breakpoint margin to the **Python Shell** and to stop in the debugger if an exception or breakpoint is reached, either in code in the editor or code that was entered into the **Python Shell**.

The option **Enable Recursive Prompt** causes Wing to show a new recursive prompt in the **Python Shell** whenever the debugger is paused, rather than waiting for completion of the original command before showing another prompt. Showing or updating plots from recursive prompts works interactively in the same way as described earlier.

If another exception or breakpoint is reached, Wing stops at those as well, recursively to any depth. Continuing the debug process from a recursive prompt completes the innermost invocation and returns to the previous recursive prompt, unless another exception or breakpoint is reached first.

### Trouble-shooting

If **show()** blocks when typed into the **Python Shell**, if plots fail to update, or if you run into other event loop problems while working with Matplotlib, then the following may help solve the problem:

(1) When working in the **Debug Console**, 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 Console** (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')
```

Instead of **TkAgg** you may also try other supported backends, including **Qt5Agg** (which requires that Qt5 is installed) or **WebAgg** (which uses a web browser for plot display).

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

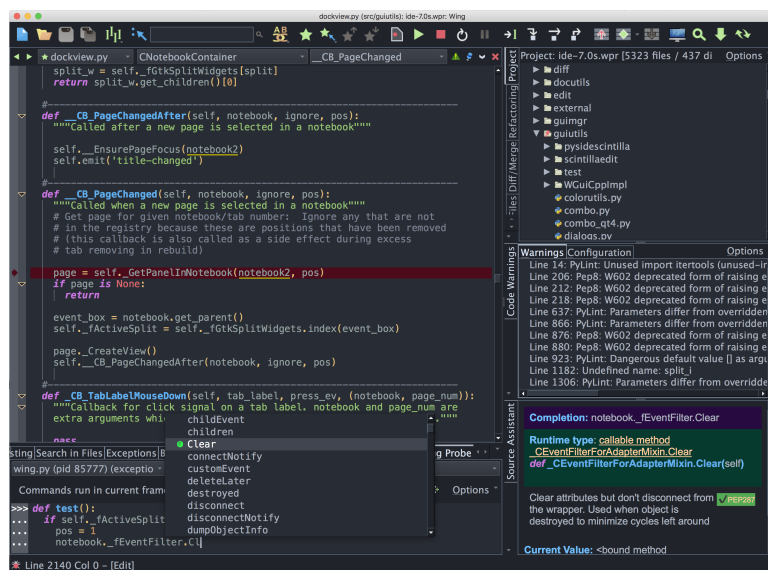
### Related Documents

For more information see:

## How-Tos for Scientific and Engineering Tools

- [The Matplotlib website](#)
- [Quickstart Guide](#) contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#) documents Wing in detail.

## 2.2. 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 developers, 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. An alternative to editing `wingdbstub.py` is just to set the environment variable `WINGHOME` before you run `jupyter notebook`.

### Listen for Debug Connections

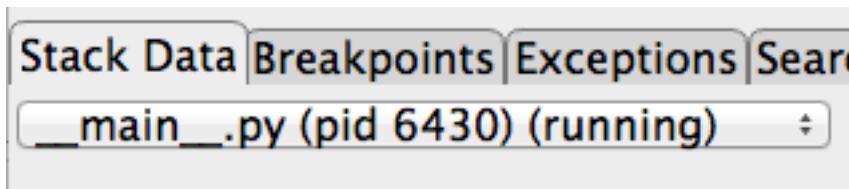
Next, tell Wing to listen for externally initiated debug connections by clicking on the bug icon in the lower left of Wing's window and checking on **Accept Debug Connections**.

### 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:

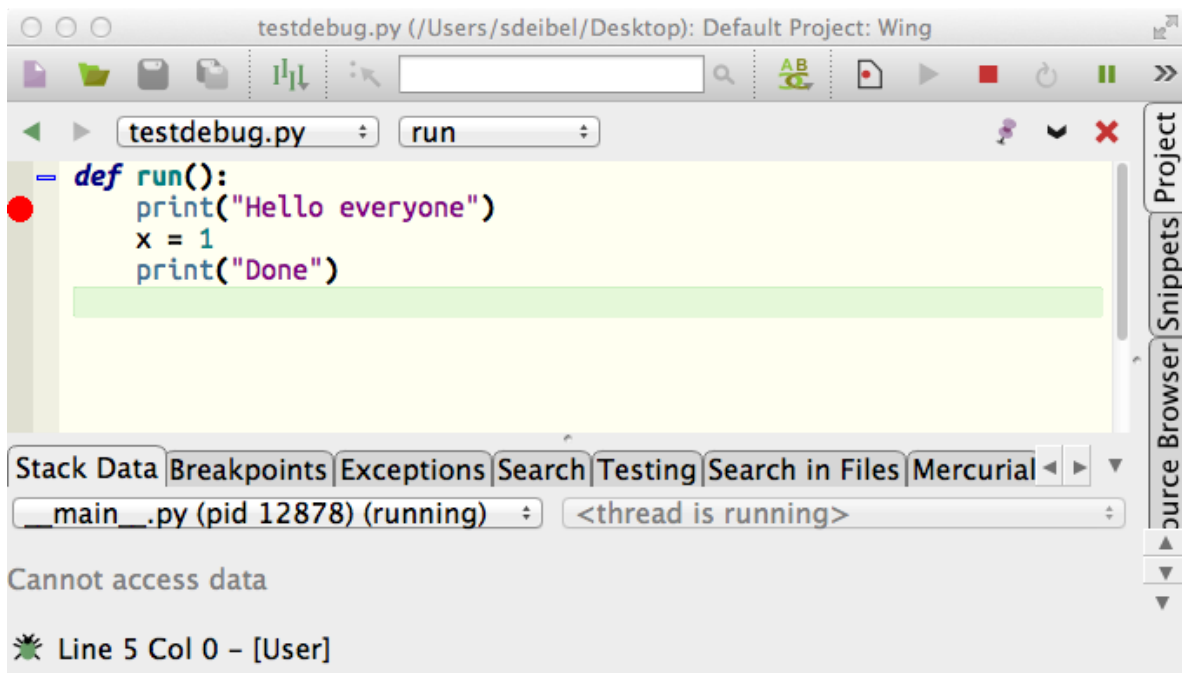


### Working with the Debugger

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")
```

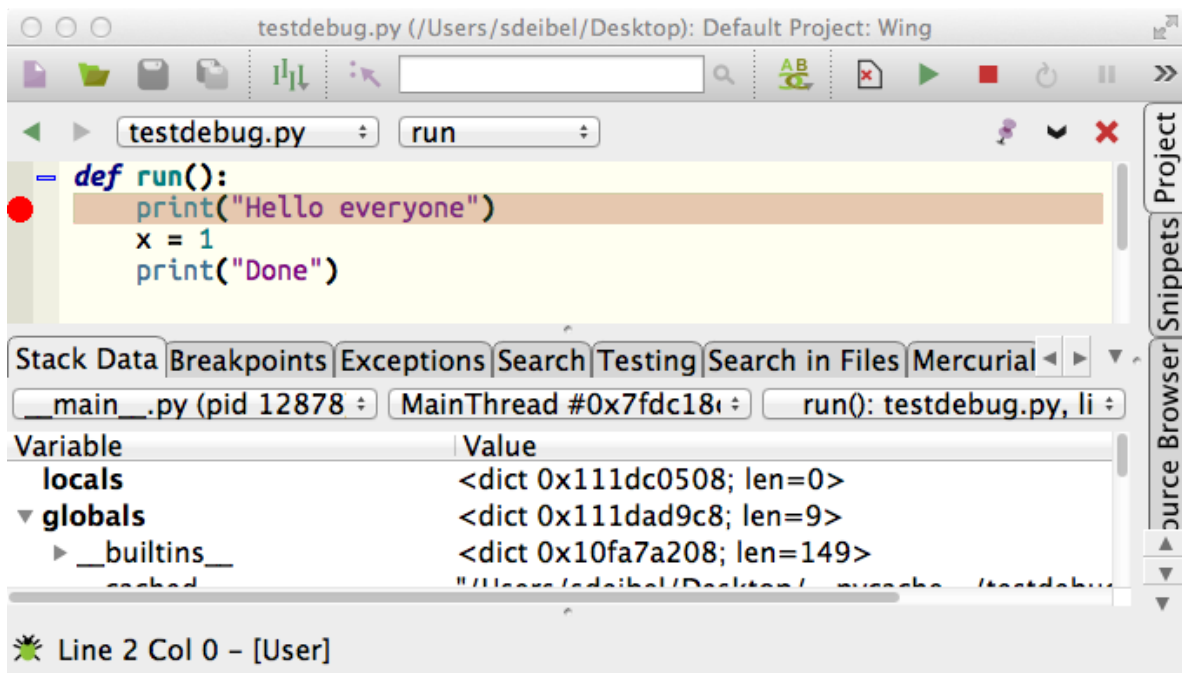
Open this in Wing and place a breakpoint on the first line by of the body of `run()` by clicking on the breakpoint margin to the left, as follows:




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 Console** (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 editing 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.

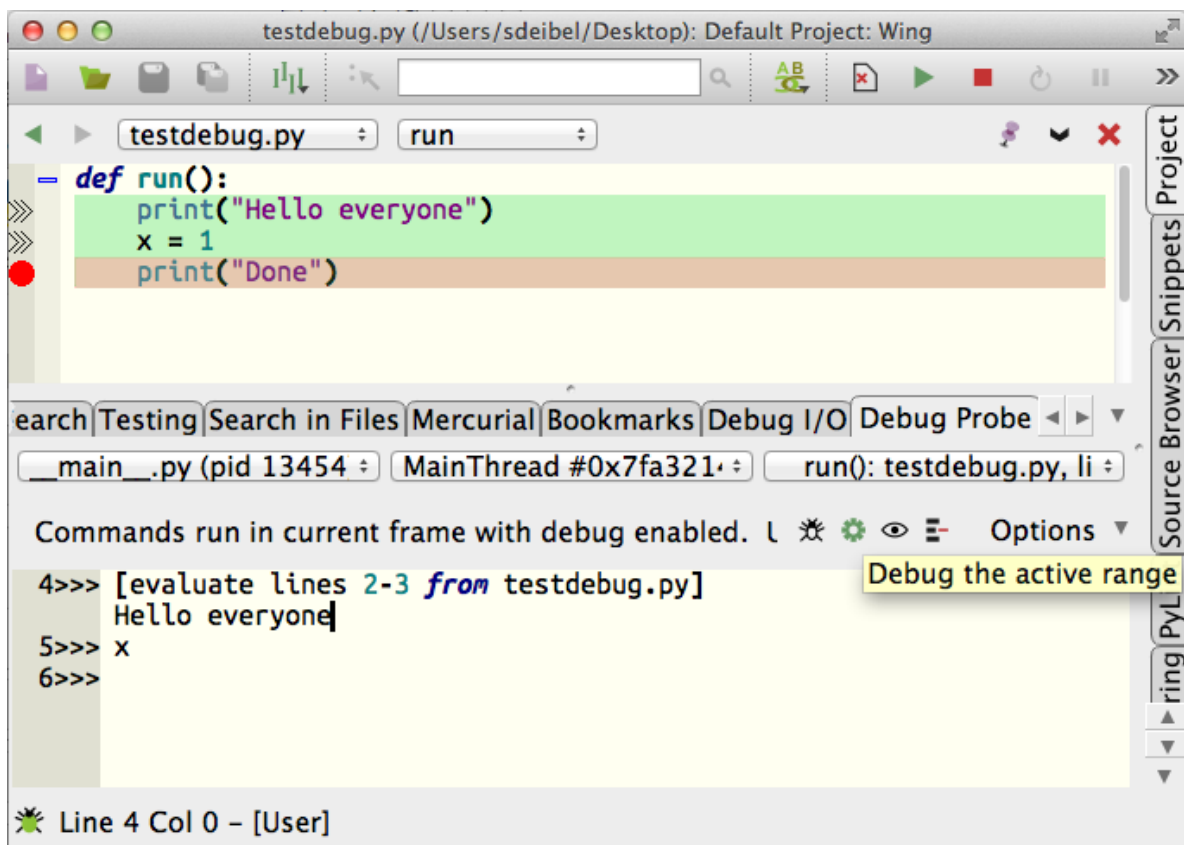
Try selecting the **Source Assistant** from Wing's **Tools** menu and then adding some other code in **testdebug.py**, for example add **z = yy** for your code reads as follows:

```
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 active and the code you are typing is on 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 Console**, 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 Console** to make an active range. Once that is done, you can execute those lines repeatedly by pressing the  icon in the **Debug Console**:



### 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:

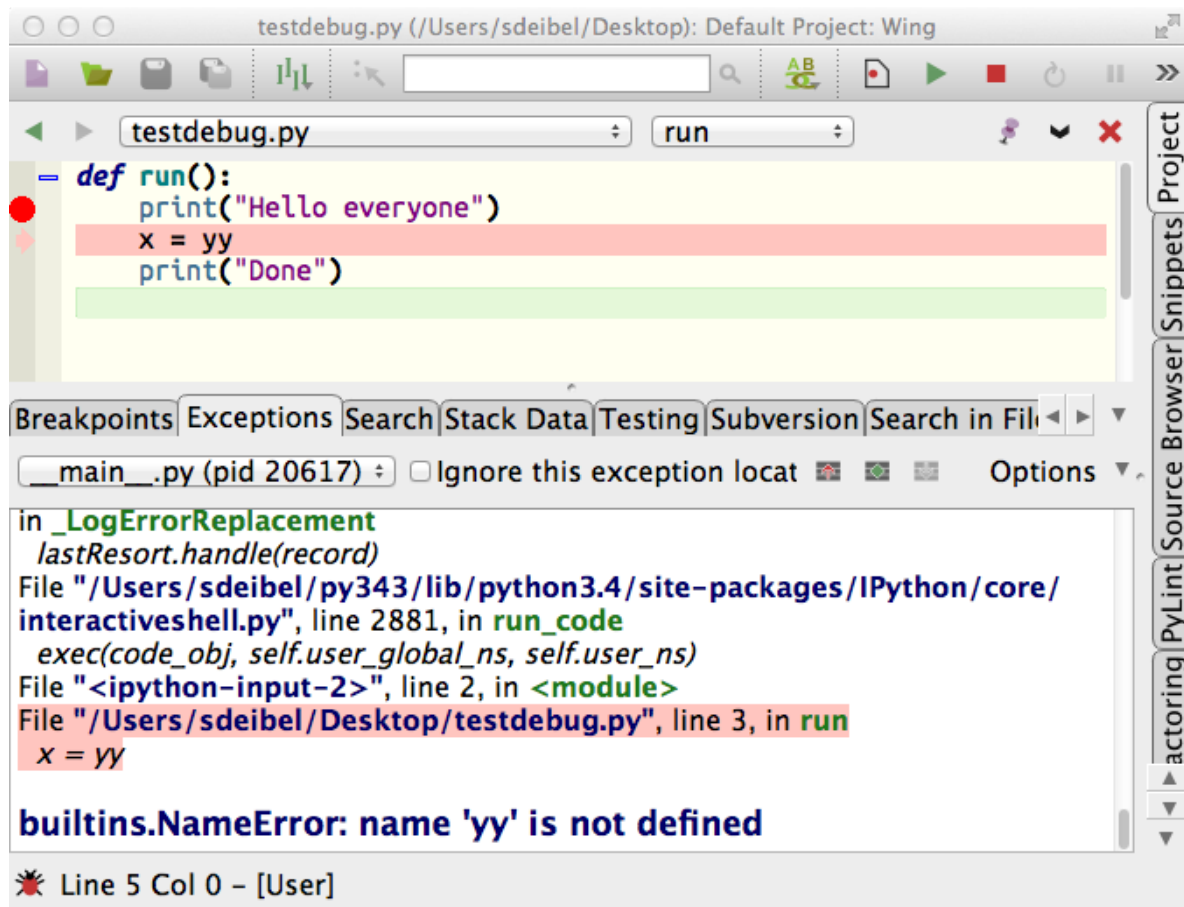
```
def run():
    print("Hello everyone")
    z = yy
    print("Done")
```

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 even though it is being handled:

```
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***

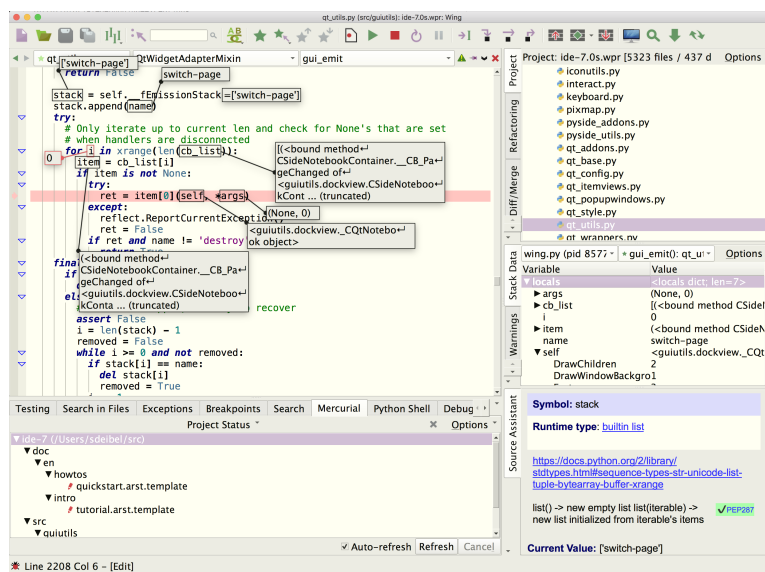
Wing provides many other options and tools. For more information:

- [Jupyter website](#)
- [Quickstart Guide](#) contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#) documents Wing in detail.

## 2.3. Using Wing with PyXLL

### Note

"Out of all the Python IDEs available I found Wing to have the fastest and easiest to use debugger by far. Using it to debug Python code running in Excel with PyXLL is a joy!" -- Tony Roberts



Wing is a Python IDE that can be used to develop, test and debug Microsoft Excel add-ins written in Python with PyXLL.

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

This document focuses on configuring Wing to debug Python code running in Excel. To learn more about Wing in general, please refer to the **Tutorial** in Wing's **Help** menu or read the **Quickstart Guide**.

### Introduction

PyXLL is a commercial product that embeds Python into Microsoft Excel on Windows. It allows you to expose Python code to Excel as worksheet functions, macros, menus, and ribbon toolbars.

PyXLL add-ins can be developed, tested, and debugged using Wing. Wing's **remote debugger** is used to connect to Excel in order to debug the Python code.

### ***Installation and Configuration***

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

- Install PyXLL as described in the [PyXLL user guide](#). Be sure to follow this guide to the end and install the optional PyXLL wheel using pip.
- [Install Wing](#) if you don't already have it.
- Launch Wing from the **Start** menu on Windows.
- Create a new project in Wing with **New Project** in the **Project** menu. Select **Empty Python Project** as the project type and set **Python Executable** to **Command Line** and then enter the full path to the Python you are using with PyXLL. This is the same value used for **executable** in the PyXLL config file.
- Locate the folder where you have installed PyXLL and in Wing select **Add Existing Directory** from the **Project** menu to add it to your project. Also add any other directories that store the source code you are working on.
- Save your project to disk with **Save Project As** in the **Project** menu.

### ***Debugging Python Code in Excel***

This section describes how to debug Python code running in the Excel process through the PyXLL add-in.

- Copy **wingdbstub.py** from the **Install Directory**, listed in Wing's **About** box, accessed from the **Help** menu, into a directory listed on the **pythonpath** in your PyXLL config file. If you are just starting with PyXLL, this could be the **examples** folder in your PyXLL folder.
- Open your copy of **wingdbstub.py** and make the following changes:
  1. Make sure **WINGHOME** is set to the full path of the Wing installation from which you copied **wingdbstub.py**. This may already be done, since it is usually set automatically during installation.
  2. Change the value of **kEmbedded** to **1**. This tells Wing's debugger that you are working with an embedded copy of Python, which affects some aspects of how code is debugged.
- Add **wingdbstub** to the modules list in your **pyxll.cfg** file:

```
[PYXLL]
modules =
    wingdbstub
    ...
```

- Make sure the **Debugger > Listening > Accept Debug Connections** preference is enabled on in Wing, to allow debug connections from the Excel process. This can also be enabled by clicking on the bug icon in the lower left of Wing's window.

Now hovering your mouse over the bug icon should show that Wing is listening for externally initiated debug connections on the local host.



If Wing is not listening, it may be that it has not been allowed to do so by Windows. In that case, try restarting Wing so that Windows will prompt you to allow network connections.

- Set any required breakpoints in your Python source code by clicking on the leftmost margin next to the code in Wing's editor, or with the breakpoint items in the **Debug** menu.
- Restart Excel or reload the PyXLL add-in so that the **wingdbstub** module is imported. You should see the status indicator in the lower left of Wing's window change to yellow, red, or green, as described in [Debugger Status](#).
- Call a Python function from Excel that will reach a breakpoint.

When a breakpoint is reached, Wing will come to the front and show the file where the debugger has stopped. If no breakpoint or exception is reached, the program will run to completion, or you can use the **Pause** command in the **Debug** menu.

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

If this doesn't work at first, try using **wingdbstub.Ensure()** to force **wingdbstub** to make the connection to the debugger. The following code creates an Excel worksheet function that, when called, ensures the debugger is connected:

```
from pyxll import xl_func
import wingdbstub

@xl_func
def debug_test():
    wingdbstub.Ensure()
    return "Connected Ok!"
```

If this code can't connect then check that the Wing application is allowed to make network connections in your Windows Firewall settings. To do this, go to the Windows **Start** menu and type "Allow an app through Windows firewall", select "Change Settings" and then "Allow another app...". Navigate to the Wing installation folder and select the Wing executable from the **bin** folder. Restart Wing and Excel and now the two should be able to connect.

If you still have problems making this work, try setting the **kLogFile** variable in **wingdbstub.py** to log additional diagnostic information. This diagnostic output can be emailed to [support@wingware.com](mailto:support@wingware.com) for help.

### ***Related Documents***

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

- [PyXLL website](#).
- [Debugging Externally Launched Code](#).
- [Quickstart Guide](#) contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#) documents Wing in detail.

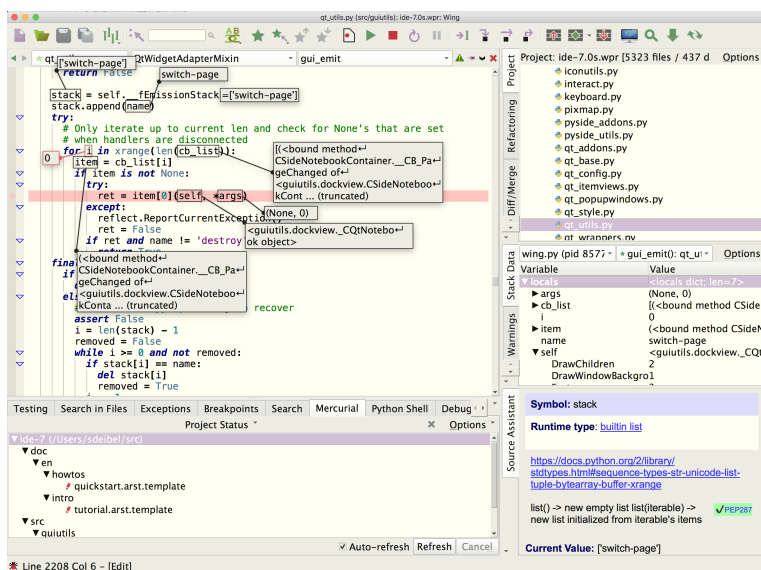
## **How-Tos for Web Development**

The following How-Tos explain how to get started using Wing with a number of popular web development frameworks.

### 3.1. 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. The debugger works with Django's auto-reload feature and can step through and debug Python code and Django templates. Wing Pro also automates some aspects of the creation and management of Django projects and applications.

Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional developers, 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](#).

#### Automated Configuration

The fastest way to get started working with a local installation of Django is to use Wing Pro's Django extensions. Skip ahead to the **Manual Configuration** section below if you have Wing Personal or if you need to work with Django running on a remote host.

### **Existing Django Project**

To set up a Wing Pro project for an *existing* Django project, create a new project from the **Project** menu using project type **Django**. You will be prompted to select the Python executable to use with Django and the directory for your existing Django project, where your **manage.py** is located.

If you are using Django with a named virtualenv or Anaconda environment, set **Python Executable** to **Activated Env** and enter the command that activates the environment (for example **activate django2** or **conda activate django3**). The drop down menu to the right of this field lists recently used and automatically found environments. If you are not using a named environment, select **Command Line** instead and enter the full path to **python.exe** or **python**. This value can be found by running Python outside of Wing and executing the following:

```
import sys
print(sys.executable)
```

Once the project is created, this will display a dialog that confirms the configuration, with a detailed list of the settings that were made.

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.

In some cases, the project creation process may prompt you to take additional steps manually:

- If **django-admin.py** could not be found from the specified **Python Executable** then you will be prompted to change this value in **Project Properties**, from the **Project** menu. Wing looks for **django-admin.py** in the same directory as the selected Python's **python.exe** or **python**.
- If **settings** is a package in your project (instead of a **settings.py** file), you will need to manually enable template debugging in Django. This is done by setting **debug** to **True** under **OPTIONS** in the **TEMPLATES** section of your **settings.py** or **settings** package. If you are using Django 1.8 or earlier, instead set **TEMPLATE\_DEBUG** to **True**.

### **New Django Project**

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

- Select **Start Django Project** from the **Extensions** sub-menu of the **Project** menu, fill in the requested values, and press **OK**.
- Wing will display a confirmation dialog, with a detailed list of actions taken. This may include a command that needs to be run manually to set up the superuser account for your new Django project. If so, copy and paste to run it in a command shell.
- Finally, press **Create Wing Project** to create and configure a new Wing project for the new Django project. This confirms the project configuration, as described above for existing Django projects.

### ***Automated Django Tasks***

The **Django** menu, shown in Wing when the current project is configured for Django, contains items for common tasks such as creating a new app, generating SQL for a selected app, migrating an app or database, running validation checks or unit tests, and restarting the integrated **Python Shell** with the Django environment.

Wing's Django extensions are open source and can be found in **scripts/django.py** in the install directory listed in Wing's **About** box. For detailed information on writing extensions for Wing, see [Scripting and Extending Wing](#).

### ***Remote Development***

Wing Pro can work with Django running on a remote host. See [Remote Python Development](#) for instructions on setting up remote access. Then use the following manual configuration instructions. You'll be able to use either of the described methods for debugging.

### ***Manual Configuration***

This section describes manual configuration of Wing projects for use with Django. Manual configuration is necessary when Django is running on a remote host or when using Wing Personal. If you are using Wing Pro with a local installation of Django, see **Automated Configuration** above instead.

### ***Configuring the Project***

To get started, create a new project from the **Project** menu using the **Generic Python** project type for a local installation and **Connect to Remote Host via SSH** for a remote installation of Django. If you are using a named virtualenv or Anaconda environment, set the **Python Executable** to **Activated Env** and enter the command that activates the environment (for example **activate django-test** or **conda activate env2**). The drop down menu to the right of this field lists recently used and automatically found environments. If you are not using a named environment, select **Command Line** instead and enter the full path of the Python used for Django. This path can be found by running Python outside of Wing and typing the following:

```
import sys
print(sys.executable)
```

Then add your project files with **Add Existing Directory** in the **Project** menu.

You may also need to set the **DJANGO\_SITENAME** and **DJANGO\_SETTINGS\_MODULE** environment variables and add the project directory and its parent directory to the **Python Path** under **Environment** tab in **Project Properties**, from the **Project** menu.

In Django 1.7 and later, set **PYTHONSTARTUP\_CODE** in the **Environment** in Project Properties to **import django; django.setup()** so Django will be initialized in Wing's integrated **Python Shell**.

For unit testing in Wing Pro, set the **Default Test Framework** under the **Testing** tab of **Project Properties** to **Django Tests** and then add **manage.py** as a test file with **Add Single File** in the **Testing** menu.

### ***Configuring the Debugger***

There are two ways to debug Django code: Either (1) configure Django so it can be launched by Wing's debugger, or (2) cause Django to attach to Wing from the outside as code that you wish to debug is executed.

### ***Launching from Wing***

To start Django from Wing, right-click on **manage.py** in the **Project** tool and select **Set as Main Entry Point**. This causes execution and debugging to start here.

Next, configure the command line arguments sent to **manage.py** by opening the file in the editor or finding it in the **Project** tool and right-clicking to select **File Properties**. Then set the run arguments under the **Debug/Execute** tab to your desired launch arguments. For example:

```
runserver 8000
```

Other command line arguments can be added here as necessary for your application.

If you are using Wing Pro, enable **Debug Child Processes** under the **Debug/Execute** tab of **Project Properties**. This allows Django to load changes you make to code without restarting.

Child process debugging is not available in Wing Personal, so you instead need to add **--noreload** to the run arguments for **manage.py**:

```
runserver --noreload 8000
```

In this case, you will need to restart Django each time you make a change, or use the debugging method described below.

### ***Launching Outside of Wing***

Another method of debugging Django is to use **wingdbstub.py** to initiate debugging after 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** into the top of the Django directory, where **manage.py** is located. This file can be found in the install directory listed in Wing's **About** box. Make sure that **WINGHOME** in your copy of **wingdbstub.py** is set to the full path of the install directory. On OS X, use the full path of Wing's **.app** folder (without the **Contents/Resources** part).

If you are developing on a remote host, instead use the copy of **wingdbstub.py** that is located in the remote agent's installation directory, on the remote host. This is preconfigured to work correctly with your remote project, as described in [Remote Web Development](#).

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

```
import wingdbstub
```

Then make sure that the **Debugger > Listening > Accept Debug Connections** preference is enabled in Wing and start Django. The Django process should connect to Wing and stop at any breakpoints reached after **wingdbstub** has been imported.

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

### ***Debugging Django Templates***

To enable debugging of Django templates, you need to:

1. Enable template debug in Django. This is done by setting **debug** to **True** under **OPTIONS** in the **TEMPLATES** section of your **settings.py** or **settings** package. If you are using Django 1.8 or earlier, instead set **TEMPLATE\_DEBUG** to **True**.
2. Turn on **Enable Django Template Debugging** under the **Options** tab of **Project Properties**, from the **Project** menu. When you change this property, you will need to restart your Django debug process, if one is already running.

### ***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 system):

```
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 **Debugger > Exceptions > Report Exceptions** preference is left set to its default value **When Printed**.

### ***Template Debugging***

If you enabled Django template debugging as described above, you should be able to set breakpoints in any file that contains `{%%}` or `{}` tags, and the debugger will stop at them.

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.

When template debugging is enabled, you won't be able to step into Django internals during a template invocation. To work around that, temporarily uncheck **Enable Django Template Debugging** under the **Extension** tab of Project Properties in Wing, and then restart your debug process.

### ***Better Auto-Completion***

Wing provides auto-completion on Python code and Django templates. The completion information is based on static analysis of the files and runtime introspection if the debugger is active and paused. 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***

Wing Pro includes a unit testing integration capable of running and debugging Django unit tests. For Django projects, the **Default Testing Framework** under the **Testing** tab of **Project Properties** is set to **Django Tests** so that the **Testing** tool runs `manage.py test` and displays the results. Individual tests can be run or debugged by selecting them and pressing **Run Tests** or **Debug Tests** in the **Testing** tool.

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.

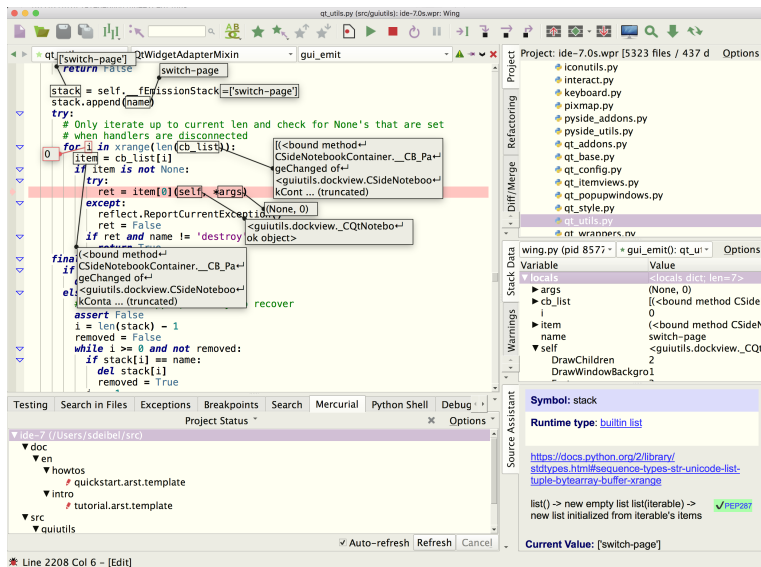
### ***Related Documents***

For more information see:

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



### 3.2. 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 developers, 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](#).

#### Project Configuration

To create a new project, use **New Project** in Wing's **Project** menu. Select the project type **Flask** and enter the **Python Executable** to use. If you are using Flask in a virtualenv or Anaconda environment, select **Activated Env** and enter the command that activates the environment (for example, `/path/to/venv/Scripts/activate`, `/path/to/env/bin/activate`, or Anaconda's `activate env`). The drop down menu to the right of this field lists recently used and automatically found environments. If you are not using a named environment, select **Command Line** instead and then enter the full path of the Python you plan to use with Flask. You can determine the correct value to use by executing the following commands in Python:

```
import sys
print(sys.executable)
```

Press **OK** and then add the directory with your source code to the new project with **Add Existing Directory** in the **Project** menu.

### ***Remote Development***

Wing Pro can work with Flask code that is running on a remote host, VM, or container. To do this, you need to be able to connect to the remote system with SSH. Then you can create your project in the same way as above, using the **Connect to Remote Host via SSH** project type. See [Remote Hosts](#) for more information on remote development with Wing Pro.

### ***Debugging Flask 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 Current as Main Entry Point** in the **Debug** menu to set this file as your main entry point. 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.

### ***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***

For more information see:

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

### 3.3. 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 developers, 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](#).

#### Creating a Wing Project

To create a new project, use **New Project** in Wing's **Project** menu. If Pyramid was not installed into your default Python, use **Python Executable** to select the Python interpreter to use. If you are using a virtualenv or Anaconda environment, set this to **Activated Env** and enter the command that activates the environment. The drop down menu to the right of this field lists recently used and automatically found environments. Otherwise, set this to **Command Line** and enter the full path to **python.exe** or **python**. The full path to use can be found by running Python outside of Wing and executing the following:

```
import sys
print(sys.executable)
```

Press **OK** and then add the directory with your source code to the new project with **Add Existing Directory** in the **Project** menu.

## **Debugging**

### **Launching from Wing**

The easiest way to debug Pyramid is just to launch it from Wing. To do this, find and open **pserve** from Pyramid and select **Set Current as Main Entry Point** from the **Debug** menu.

Then right-click on **pserve** and under **Environment** enter your run arguments, for example:

```
development.ini
```

Then go into **Project Properties** in the **Project** menu and set **Initial Directory** under the **Debug/Execute** tab to the full path of the directory that contains your **.ini** files.

Now you can start debugging with **Start/Continue** in the **Debug** menu or from the toolbar. You can load <http://localhost:6543/> or other page, or initiate an AJAX request, and Wing will stop on any breakpoints or exceptions. This works with any Python code, including any View Callable, Pyramid internals, or any other library or package used by your code.

From here, you can step through code or inspect the program state with **Stack Data** and other tools. In Wing Pro, the **Debug Console** provides a command line that allows you to interact with the current stack frame in your debug process. All the debugging tools are available from the **Tools** menu.

### **Auto-reloading Changes**

With the above configuration, you'll need to restart Pyramid every time you make a change. If you have Wing Pro you can cause Pyramid to auto-reload changes. To do this, add the **--reload** option to the run arguments you set for **pserve**, for example:

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

Then enable **Debug Child Processes** in **Project Properties**, from the **Project** menu, so that Pyramid's reloaded processes will also be debugged.

This option is only available in Wing Pro. In Wing Personal you'll need to use **wingdbstub** for reloading, as described below.

### **Launching Outside of Wing**

Wing can also debug code that is launched from outside of the IDE, for example from the command line. To do this with Pyramid, copy **wingdbstub.py** from the **Install Directory** listed in Wing's **About** box into the directory that contains your Pyramid **.ini** files. You may need to set the value of **WINGHOME** inside your copy of this file to the full path of the install directory you copied it from, or on macOS to the full path of the **.app**.

Next place the following line into your source, on a line before the code you wish to debug:

```
import wingdbstub
```

Then click on the bug icon in the lower left of Wing's window and make sure that **Accept Debug Connections** is checked.

Now you can start your Pyramid server as you usually would, for example:

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

Using **--reload** is not necessary but it is supported by Wing's debugger and makes testing of changes much easier.

### ***Notes on Auto-Completion***

Wing provides auto-completion on Python code and in other files, including the various templating languages that can be used with Pyramid.

The autocomplete information available to Wing is based on static analysis of your project files and any files Wing can find through imports, by searching on your Python Path.

When the debugger is active and paused, Wing also uses introspection of the live runtime 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 Jinja2 Templates***

The Jinja2 template engine sets up stack frames in a way that makes it possible to set breakpoints directly in **.jinja2** template files and step through them, viewing data in **Stack Data** and other tools in the same way as for Python code.

Debugging support in the Jinja engine is imperfect in that not all tags are reached and some tags cause lines to be visited multiple times. However, this capability can still be useful to stop Wing's debugger when a particular template is being invoked.

### ***Debugging Mako Templates***

Another good choice of templating engine for Pyramid is **Mako**, because it allows the full syntax of Python in expression substitutions and control structures. However, Mako templates cannot be directly stepped through using the debugger. Instead, you can set breakpoints in the **.py** files produced by Mako for templates.

To debug Mako templates with Wing you will need to modify your **.ini** file to add the following line in the **[app:main]** section:

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

You may need to change the path to match your project. Without this setting, mako templates are compiled in memory and not cached to disk, so you won't be able to debug them. With this setting, Mako will write **.mako.py** files for each template to the specified directory, whenever the template changes. You can set breakpoints within these generated files.

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.

If you are starting Pyramid outside of Wing and need to use **wingdbstub** to initiate debugging, as described earlier, and want to do this from a Mako template, then you can add the following to the template:

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

### ***Remote Development***

Wing Pro can work with Pyramid code that is running on a remote host, VM, or container. To do this, you need to be able to connect to the remote system with SSH. Then you can create your project in the same way as above, using the **Connect to Remote Host via SSH** project type. See [Remote Hosts](#) for more information on remote development with Wing Pro.

### ***Related Documents***

For more information see:

- [Pyramid documentation](#)
- [Quickstart Guide](#) contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#) documents Wing in detail.

### 3.4. Using Wing with web2py



**Wing** is a Python IDE that can be used to develop, test, and debug Python code and templates written for **web2py**, a powerful 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 developers, 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 Console** 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

The best way to install web2py is to clone the [git repository](#). Be sure to follow the instructions in the readme so you clone all the dependencies recursively.

To create a new project, use **New Project** in Wing's **Project** menu and select the **project type** **web2py**. If your default Python is not the one into which you installed **web2py**, use **Python Executable** to select the Python to use. If using a virtualenv or Anaconda environment, set



this to **Activated Env** and enter the command that activates the environment. The drop down menu to the right of this field lists recently used and automatically found environments. Otherwise, select **Command Line** and then enter the full path of the Python want to use. You can determine the correct value to use by executing the following in Python:

```
import sys
print(sys.executable)
```

Press **OK** and then add the directory with your source code to the new project with **Add Existing Directory** in the **Project** menu.

After the **Project** tool populates, find and right click on the file **web2py.py** and select **Set As Main Entry Point**.

### ***Remote Development***

Wing Pro can work with web2py code that is running on a remote host, VM, or container. To do this, you need to be able to connect to the remote system with SSH. Then you can create your project in the same way as above, using the **Connect to Remote Host via SSH** project type. See [Remote Hosts](#) for more information on remote development with Wing Pro.

### ***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 Console** (in Wing Pro) to look around.

An example is to set a breakpoint in **applications/examples/views/default/index.html**, which is loaded when you go to the URL **http://127.0.0.1:8000/examples/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 Console** (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.

## **Usage Tips**

### **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 '<recycles>'** 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 **Debug Environment** in the **Debug** menu.

### **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 Auto-completion**

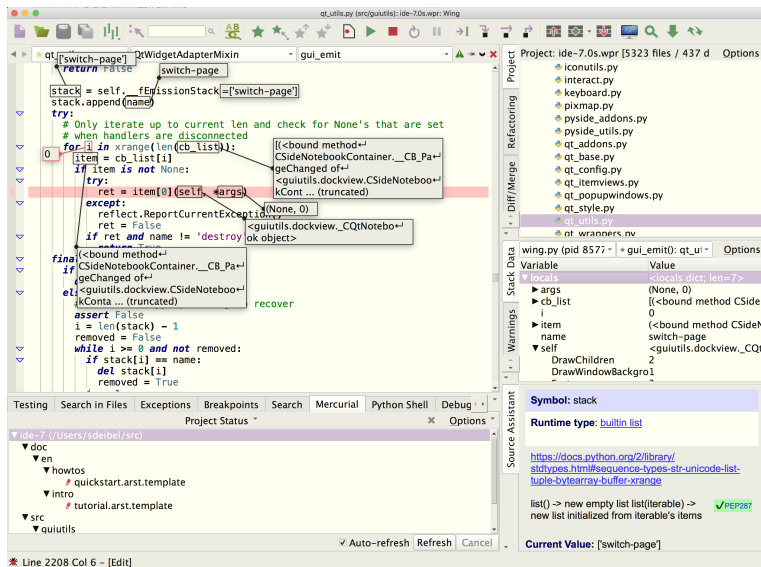
Because of the way web2py is designed, Wing's static analysis engine can fail to find the types of commonly used values like **db**. To work around this, run to a breakpoint in your code before editing it. This causes Wing to use runtime analysis as well as static analysis to drive auto-completion and other IDE features.

### **Related Documents**

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

- [web2py website](#) provides documentation and downloads.
- [Remote Web Development](#) describes how to set up development on a remote host, VM, or container.
- **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.

### 3.5. 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 developers, 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 [Debugging Externally Launched Code](#).

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
```

### ***Remote Development***

Wing Pro can work with `mod_wsgi` code that is running on a remote host, VM, or container. To do this, you need to be able to connect to the remote system with SSH. Then you can create your project in the same way as above, using the **Connect to Remote Host via SSH** project type. See [Remote Hosts](#) for more information on remote development with Wing Pro.

### ***Related Documents***

For more information see:

- [mod\\_wsgi website](#) for downloads and documentation.
- [Quickstart Guide](#) contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#) documents Wing in detail.

## **How-Tos for GUI Development**

The following How-Tos explain how to get started using Wing with a number of popular GUI development frameworks.

### 4.1. 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 developers, 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 desktop application developers that want to use Python. It is available for Windows, Linux, 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 advanced editing, debugging, testing, and code inspection capabilities for Python, 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 wxPython as described on the [wxPython Downloads](#) page. Be sure to install also the wxPython demo and samples.
- Install **Wing** if you don't already have it.
- Start Wing from the Start menu on Windows, the Finder or OS X, or by typing **wing-personal7.2** on the command line on Linux.

- Select **Show Python Environment** from the **Source** menu. 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 set **Python Executable** to select the Python interpreter into which you installed wxPython. If you are using a virtualenv or Anaconda environment, set this to **Activated Env** and enter the command that activates the environment. The drop down menu to the right of this field lists recently used and automatically found environments. Otherwise, set this to **Command Line** and enter the full path to **python.exe** or **python**. The full path to use can be found by running Python outside of Wing and executing the following:

```
import sys
print(sys.executable)
```

- Locate and open wxPython's **demo.py** into Wing and then select **Add Current File** from the **Project** menu to add it to your project. If you can't find **demo.py** but have other wxPython code that works, you can just use that. However, you'll need to adapt the instructions in the rest accordingly.
- Set **demo.py** as main entry point for debugging using the **Set Current as Main Entry Point** 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:

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.

Next open **ImageBrowser.py**, located in the same directory as **demo.py**. 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.

From here, you can step through code or inspect the program state with **Stack Data** and other tools. In Wing Pro, the **Debug Console** provides a command line that allows you to interact with the current stack frame in your debug process. All the debugging tools are available from the **Tools** menu.

See the [Wing Tutorial](#) and [Quick start](#) for more information.

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

Wing doesn't include a GUI builder for wxPython but it can be used with other tools, such as wxGlade or wxFormBuilder. Wing will automatically reload files that are generated by the GUI builder.

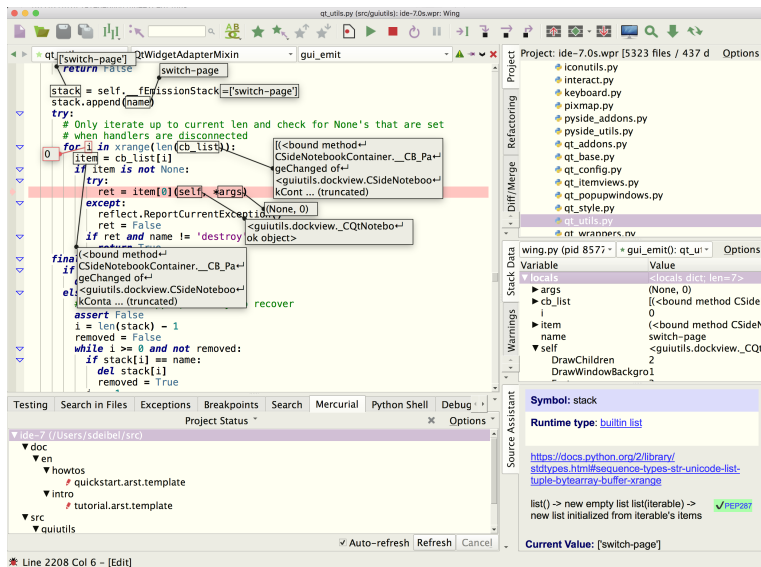
### ***Related Documents***

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

- [wxPython website](#)
- [Quickstart Guide](#) contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#) documents Wing in detail.



### 4.2. 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 developers, 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, Mac OS, and mobile devices.

While Wing does not provide a GUI builder for PyQt, it does provide advanced editing, debugging, testing, and code inspection capabilities for Python, and it can be used with other available GUI builders, as described below.

These instructions should also work with PySide, which are roughly comparable non-commercial open source bindings for Qt.

#### Installation and Configuration

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

- Install PyQt as described in [Installing PyQt5](#). Be sure to install also the **qtdemo**.
- Install **Wing** if you don't already have it.

- Start Wing from the Start menu on Windows, the Finder or OS X, or by typing **wing-personal7.2** on the command line on Linux.
- Select **Show Python Environment** from the **Source** menu. 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 set **Python Executable** to select the Python interpreter into which you installed PyQt. If you are using a virtualenv or Anaconda environment, set this to **Activated Env** and enter the command that activates the environment. The drop down menu to the right of this field lists recently used and automatically found environments. Otherwise, set this to **Command Line** and enter the full path to **python.exe** or **python**. The full path to use can be found by running Python outside of Wing and executing the following:

```
import sys
print(sys.executable)
```

- Locate and open **qtdemo.py** into Wing and then select **Add Current File** from the **Project** menu to add it to your project. If you can't find **qtdemo.py** but have other PyQt code that works, you can just use that. However, you'll need to adapt the instructions in the rest accordingly.
- Set **qtdemo.py** as main entry point for debugging with **Set Current as Main Entry Point** 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:

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.

Next locate and open **menumanager.py** in the **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.

From here, you can step through code or inspect the program state with **Stack Data** and other tools. In Wing Pro, the **Debug Console** provides a command line that allows you to interact with the current stack frame in your debug process. All the debugging tools are available from the **Tools** menu.

See the [Wing Tutorial](#) and [Quick start](#) for more information.

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

Wing doesn't include a GUI builder for PyQt but it can be used with an external GUI builder like Qt Designer. Wing will automatically reload files that are generated by the GUI builder.

### ***Related Documents***

For more information see:

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

### 4.3. 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 developers, 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](#).

#### Introduction

PyGObject implements Python bindings for GTK, an open source GUI development toolkit.

While Wing does not provide a GUI builder for GTK, it does provide advanced editing, debugging, testing, and code inspection capabilities for Python, 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 PyGObject:

- Install PyGObject as described in the [PyGObject documentation](#).
- Install **Wing** if you don't already have it.
- Start Wing from the Start menu on Windows, the Finder or OS X, or by typing **wing-personal7.2** on the command line on Linux.
- Select **Show Python Environment** from the **Source** menu. If the Python version reported there doesn't match the one you're using with PyGObject, then select **Project Properties**

from the **Project** menu and set **Python Executable** to select the Python interpreter into which you installed PyGObject. If you are using a virtualenv or Anaconda environment, set this to **Activated Env** and enter the command that activates the environment. The drop down menu to the right of this field lists recently used and automatically found environments. Otherwise, set this to **Command Line** and enter the full path to **python.exe** or **python**. The full path to use can be found by running Python outside of Wing and executing the following:

```
import sys
print(sys.executable)
```

- Locate and open the Python main entry point for your PyGObject-based application and then select **Add Current File** from the **Project** menu to add it to your project.
- Set your Python main entry point for debugging with **Set Current as Main Entry Point** 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:

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**.

Your application should start up. If its main window doesn't come to front, bring it to front from your task bar or window manager.

Next locate and open Python source code that you know will be reached when you use your application and set a breakpoint by clicking on the margin to the left of the code. Then trigger the breakpoint by performing an action in your application that results in execution of the code at that line.

From here, you can step through code or inspect the program state with **Stack Data** and other tools. In Wing Pro, the **Debug Console** provides a command line that allows you to interact with the current stack frame in your debug process. All the debugging tools are available from the **Tools** menu.

See the [Wing Tutorial](#) and [Quick start](#) for more information.

### ***Improving 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** preference.

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.

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

Wing doesn't include a GUI builder for PyGObject but it can be used with an external GUI builder like Glade. Wing will automatically reload files that are generated by the GUI builder.

### ***Related Documents***

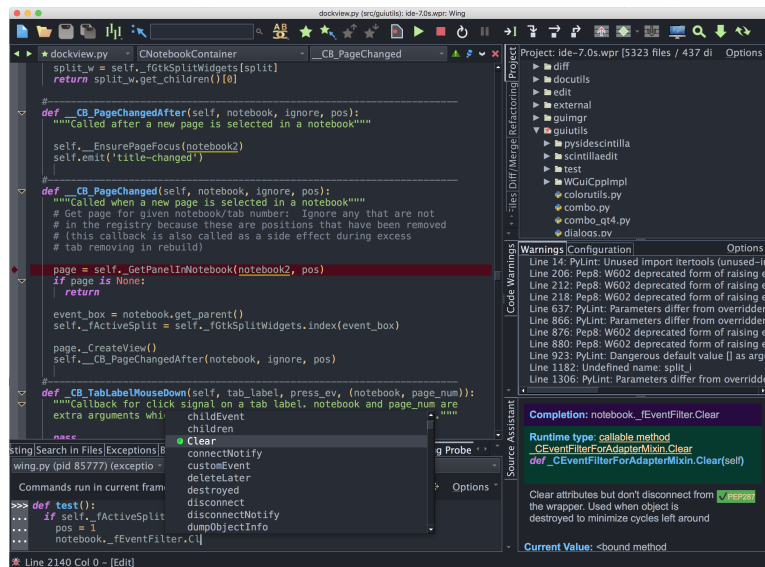
For more information see:

- [GTK](#) using [PyGObject](#) websites.
- [Quickstart Guide](#) contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#) documents Wing in detail.

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

The following How-Tos explain how to get started using Wing with a number of modeling, rendering, and compositing systems that use Python.

## 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 developers, 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 Pro 7.2'
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.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 blender stub file in blender. This is done using the **Run Script** button on the bottom toolbar or by pressing **Alt-P**, although note that **Alt-P** is sensitive to how the focus is set.

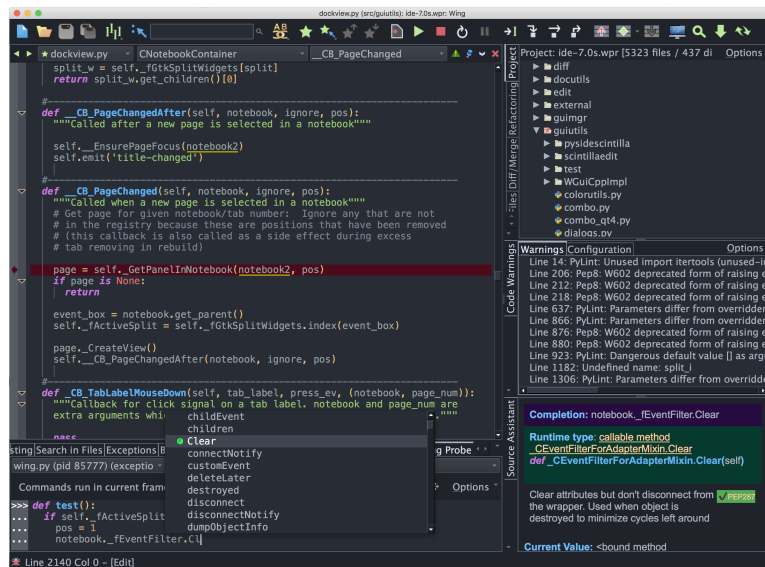
Note that you will need to turn on listening for externally initiated debug connections in Wing, by clicking on the bug icon in the lower left of the main window and selecting **Accept Debug Connections** in the popup menu that appears.

### ***Related Documents***

For more information see:

- [Blender website](#)
- [Quickstart Guide](#) contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#) documents Wing in detail.

## 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 developers, 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***

You can use the **mayapy** executable found in the **Maya** application directory to run Wing's **Python Shell** tool and to debug standalone Python scripts.

To do this, select **Command Line** for **Python Executable** in **Project Properties**, accessed from the **Project** menu, and then enter the full path of the **mayapy** file (**mayapy.exe** on Windows).

### ***Better Static 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.

At least in some versions of Maya, Wing cannot statically analyze the files in the Python API without some additional configuration. As a result, it will fail to offer auto-completion for the API. 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" add **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 **Command Line** and then enter the full path 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.

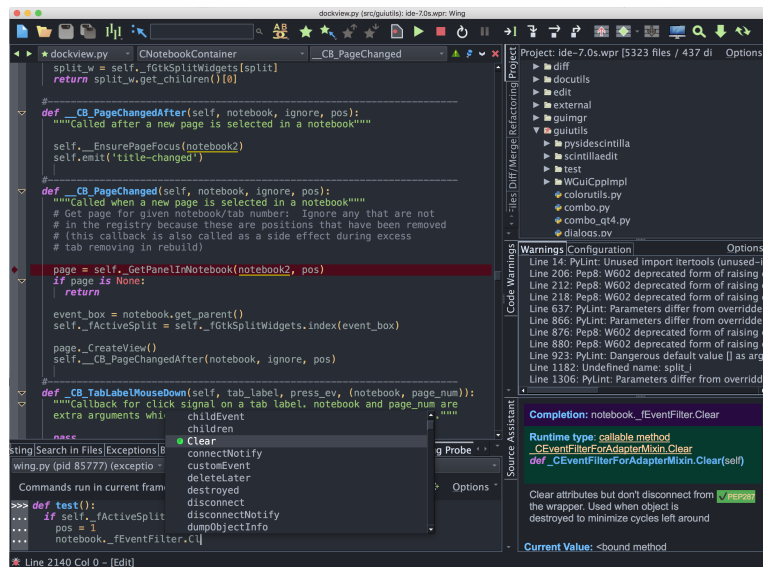
See also the section **Using Wing with Maya** in [Autodesk Maya Online Help: Tips and tricks for scripters new to Python](#).

### **Related Documents**

For more information see:

- [Autodesk Maya website](#)
- [Quickstart Guide](#) contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#) documents Wing in detail.

### 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 developers, 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.

#### 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 **Command Line** and then enter the full path 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 set **Python Executable** in **Project Properties**, from the **Project** menu in Wing, to the full path to the Python interpreter. The correct value to use can be determined by running Python outside of Wing and executing the following:

```
import sys
print(sys.executable)
```

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** 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 Console** with **Evaluate Selection in Debug Console** from the **Source** menu.

Both the **Debug Console** 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 Console** 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 Console** (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 Console** 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.

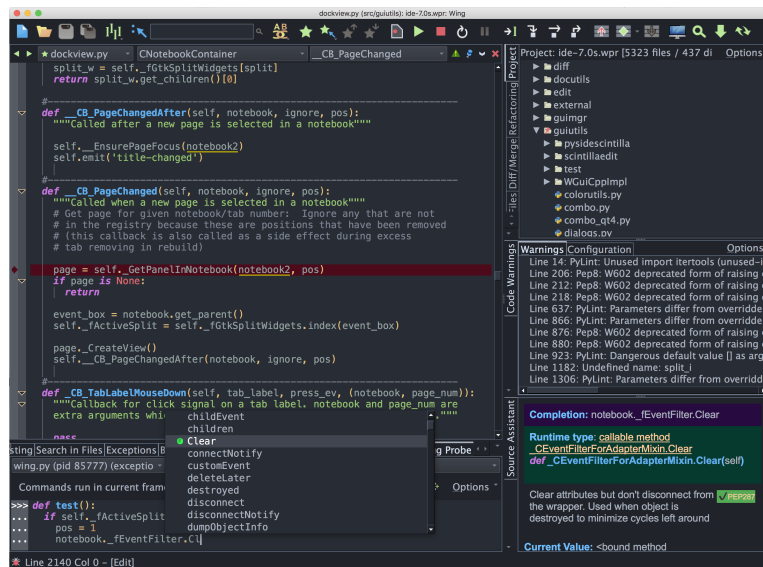
These problems should only occur when Wing's debugger is attached directly to NUKE, and can be avoided by working through **NukeExternalControl** instead, as described in the first part of this document.

### ***Related Documents***

For more information see:

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

## 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 developers, 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 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**.

Some versions of **SFM** comes with **wingdbstub.py** in the site-packages directory in its Python installation. However, this file must match the version of Wing you are using so you may need to 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
```

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

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 under **Command Line** in the **Python Executable** field of the **Project Properties** dialog.

### ***Related Documents***

For more information see:

- [Source Filmmaker website](#)
- [Quickstart Guide](#) contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#) documents Wing in detail.

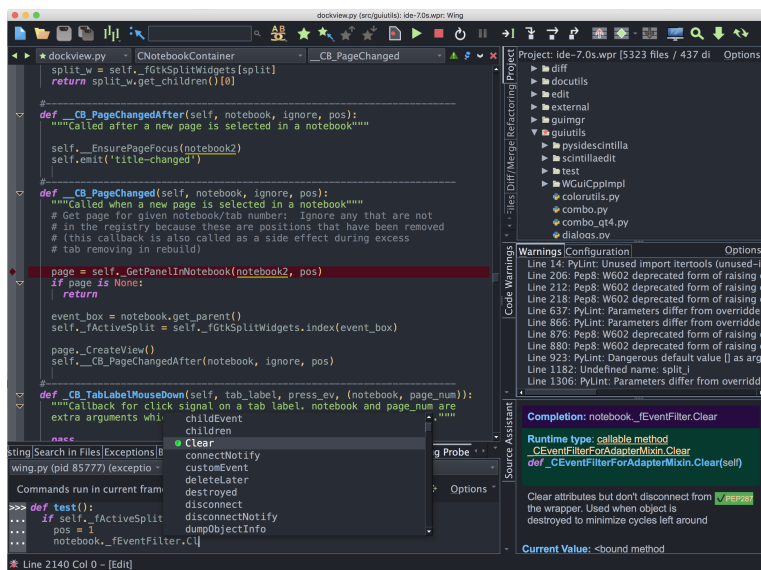
## **How-Tos for Educational Tools**

The following How-Tos explain how to get started 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 developers, 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 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). See [SSH Setup Details](#) for step-by-step instructions.
- 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**.

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 install the remote agent and then establish a connection. 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 resultin dialog, press **Browse** to choice 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 xjf wing-debugger-linux-arm32-7.2.9.0.tar.bz2**. This creates a directory named **wing-debugger-linux-arm32-7.2.9.0**.
- 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.
- 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.

- Next create a project in Wing from the **Project** menu using the **Empty Python Project** project type, and add all your source directories to the project. This allows Wing to automatically discover a mapping between where files are located on the Raspberry Pi and the local host. See [File Location Maps](#) for details.

### *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-linux-arm32-7.2.9.0/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 Console** 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-linux-arm32-7.2.9.0/wingdbstub.py** and change the line **WINGHOME = None** to **WINGHOME = /home/pi/wing-debugger-linux-arm32-7.2.9.0** 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-linux-arm32-7.2.9.0**.

## Access Control

The first time you initiate debug from a Raspberry Pi, Wing will reject the debug connection and prompt you to accept a new security token. After accepting the token, future debug connections should be accepted.

To preauthorize the debug connection, copy **wingdebugpw** from [Settings Directory](#) on the machine where you have Wing installed to the directory **wing-debugger-linux-arm32-7.2.9** on the Raspberry Pi.

## 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, and make sure 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 that all your source files have been added to your project, or if you manually configured a file mapping then 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. This can be used to correct any manually configured location map.

### ***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.

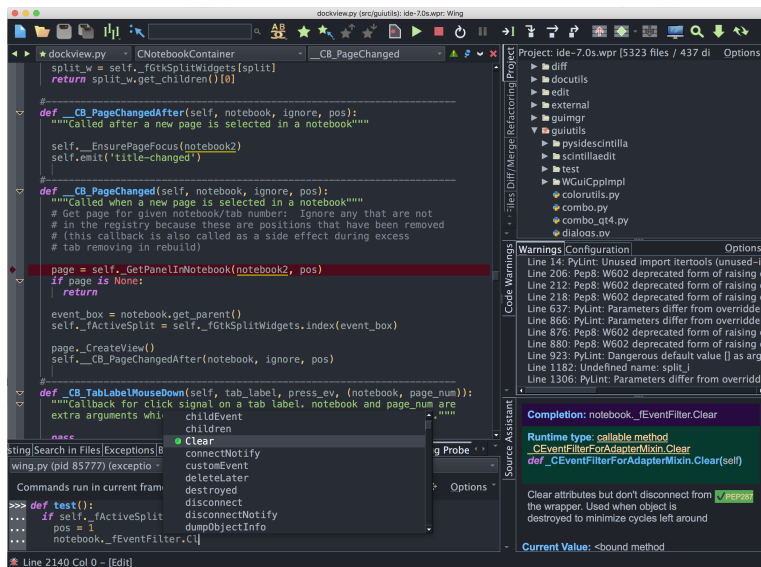
- Restart your Raspberry Pi again and wifi should work.

### ***Related Documents***

For more information see:

- [Raspberry Pi home page](#) for documentation and downloads.
- [Quickstart Guide](#) contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#) documents Wing in detail.

## 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 developers, 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](#).

### Project Configuration

Pygame works just fine with Wing without any special configuration. You'll need to first install pygame according to the instructions on the [pygame](#) website.

To create a new project, use **New Project** in Wing's **Project** menu. **Select the project type ``Pygame**. If your default Python does not have Pygame installed into it, you will need to choose the **Python Executable** to use with Pygame. This is either set to set to **Activated Env** to enter the command that activates a virtualenv or Anaconda environment, or **Command Line** to enter the full path of **python.exe** or **python**. You can determine the correct value to use by executing the following in Python:

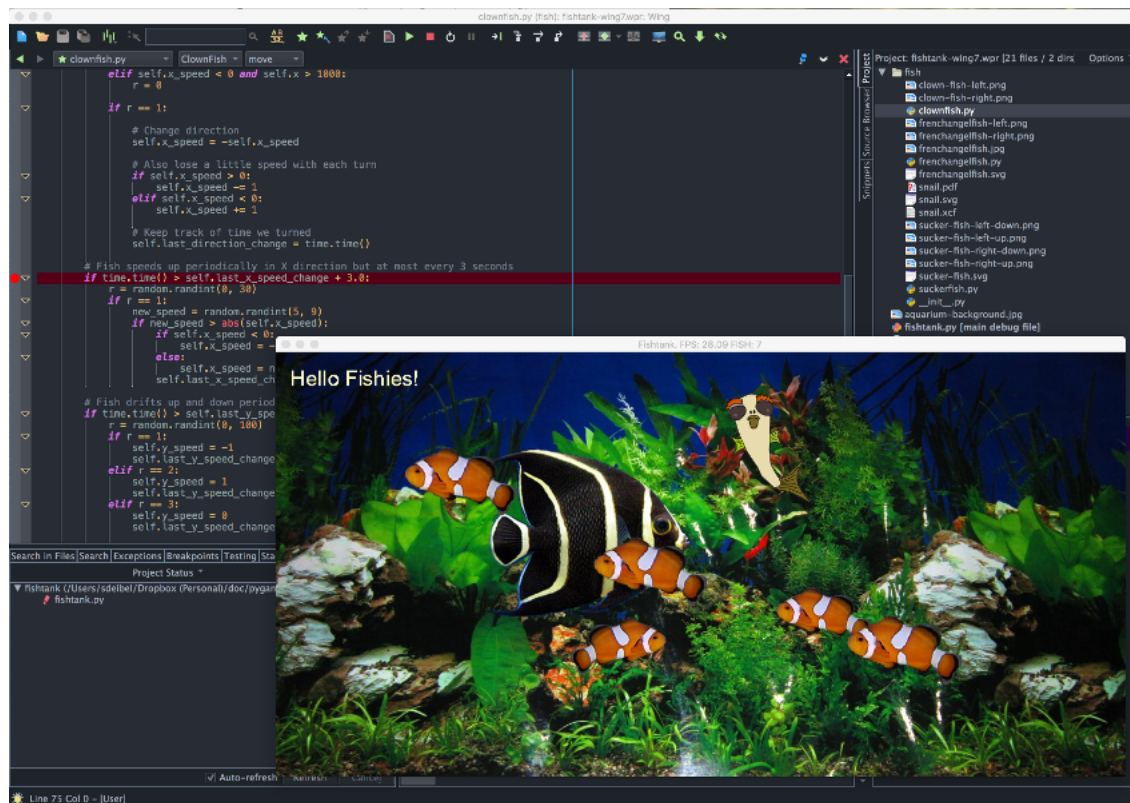
```
import sys
print(sys.executable)
```

Press **OK** and then add the directory with your source code to the new project with **Add Existing Directory** in the **Project** menu.

Next find your main entry point, open it into Wing, and select **Set Current as Main Entry Point** in the **Debug** menu.

### Debugging

Now you can launch your game from Wing with **Start/Continue** in the **Debug** menu. Wing will stop on any exceptions or breakpoints reached while running your game, and you can use the debugger to step through code, inspect the value of variables, and try out new code interactively.



To learn more about Wing's features, please refer to the tutorial in Wing's **Help** menu or read the [Quickstart Guide](#).

### Related Documents

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

- [pygame home page](#) provides downloads and documentation.
- [Quickstart Guide](#) contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#) documents Wing in detail.

## **Unmaintained How-Tos**

This section contains unmaintained How-Tos for using Wing with older and less commonly used frameworks, tools, and alternate Python implementations.

## 7.1. 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 developers, 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](#).

**Note:** This document is not being maintained and was last tested with Twisted version 8.

### Project Configuration

To create a new project, use **New Project** in Wing's **Project** menu. Select the project type **Twisted** and under **Python Executable** select **Custom** and then enter the full path of the Python you plan to use with Twisted. You can determine the correct value to use by executing the following commands interactively in Python. If you are using virtualenv, this will be the virtualenv's Python executable:

```
import sys
sys.executable
```

Press **OK** and then add the directory with your source code to the new project with **Add Existing Directory** in the **Project** menu.

### **Remote Development**

Wing Pro can work with Twisted code that is running on a remote host, VM, or container. To do this, you need to be able to connect to the remote system with SSH. Then you can create your project in the same way as above, using the **Connect to Remote Host via SSH** project type. See [Remote Hosts](#) for more information on remote development with Wing Pro.

### **Debug Configuration**

To debug Twisted code launched from within Wing, create a file with the following contents and set it as your main entry point by adding it to your project and then using the **Set Main Entry Point** 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** as follows:

```
-n -y filename.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 **filename.tac** in the above example with the correct name of your file.

You can also launch Twisted code from outside of Wing as described in [Debugging Externally Launched Code](#) in the manual.

### **Related Documents**

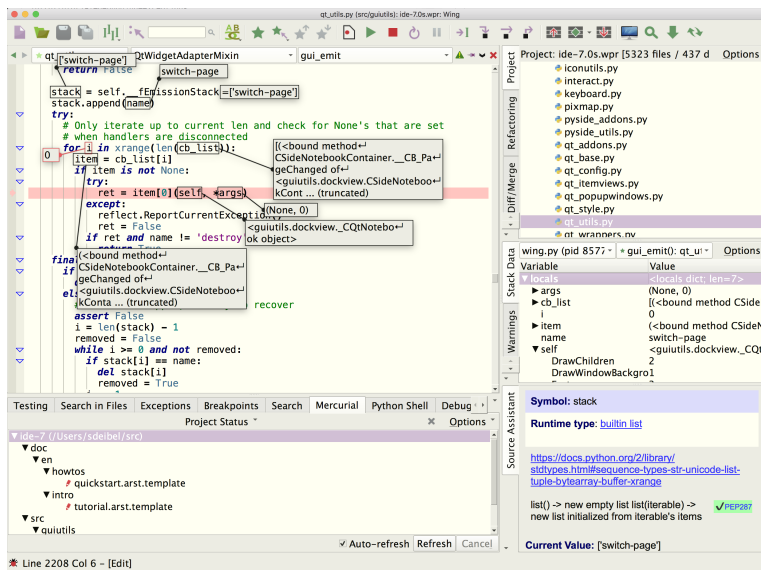
For more information see:

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

## 7.2. 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 developers, 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](#).

**Note:** This document is not being maintained and was last tested with Plone 4.

### Introduction

These instructions are for the Plone 4+ unified installer. If you are using an older version of Plone or 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](#).



### ***Configuring your Project***

To create a new project, use **New Project** in Wing's **Project** menu. Select the project type **Plone** and under **Python Executable** select **Custom** and then enter the full path of the Python you plan to use with Plone. The full path can be found by looking at the top of many of the scripts in **zinstance/bin** or **zeocluster/bin**. You can also determine the correct value to use by executing the following commands interactively in Python. If you are using virtualenv, this will be the virtualenv's Python executable:

```
import sys
sys.executable
```

Press **OK** and then add the directory with your source code to the new project with **Add Existing Directory** in the **Project** menu.

Next find and open the file **zinstance/bin/instance** and select **Set Current as Main Entry Point** in **Project** menu. If you have a ZEO cluster, instead use **zeocluster/bin/client1** or whatever name is given in the **.cfg** file. Wing reads the **sys.path** updates from that file so that it can find your Plone modules.

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***

If you have followed the instructions above, you should be able to start debug from the toolbar or **Debug** menu. The debugger will stop on breakpoints and any exceptions that are printed, and debug data can be viewed in the **Stack Data** tool, by hovering over values, and in Wing Pro by pressing **Shift-Space** or with the interactive **Debug Console**.

### ***Debugging with WingDBG***

WingDBG is a legacy product originally designed for use with old versions of Zope running on much slower machines with older versions of Wing's debugger, where starting all of Zope or Plone in the debugger would take too much time. It packages code to start and stop the debugger in a way that allows debugging only on a particular port.

We recommend against using WingDBG to debug Plone 4+. If you want to defer starting debug until after Plone initialization is complete, this is possible using **wingdbstub** as described in [Debugging Externally Launched Code](#).

If you do need to use WingDBG, it is located in **zope/WingDBG-7.2.9.tar** in the **Install Directory** listed in Wing's **About** box. Unpack it into **zinstance/products**, or **zeocluster/products** in a ZEO cluster.

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 cluster this file is located at **zeocluster/parts/client1/etc/zope.conf**.

Finally, click on the bug icon in the lower left of Wing's 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. Subsequently, any page loads through <http://localhost:50080/> (port 50080) will be debugged and will reach breakpoints and report exceptions. This port and other debugger options are configurable from the WingDBG control panel. Note that requests made through the regular port (8080 by default) will run without debug.

Once you reach a breakpoint or exception, you can step through code and use the **Stack Data** and other debug tools in the **Tools** menu to interact with the debug process. In Wing Pro, the **Debug Console** provides an interactive shell that works in the context of the current debug stack frame.

### ***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
```

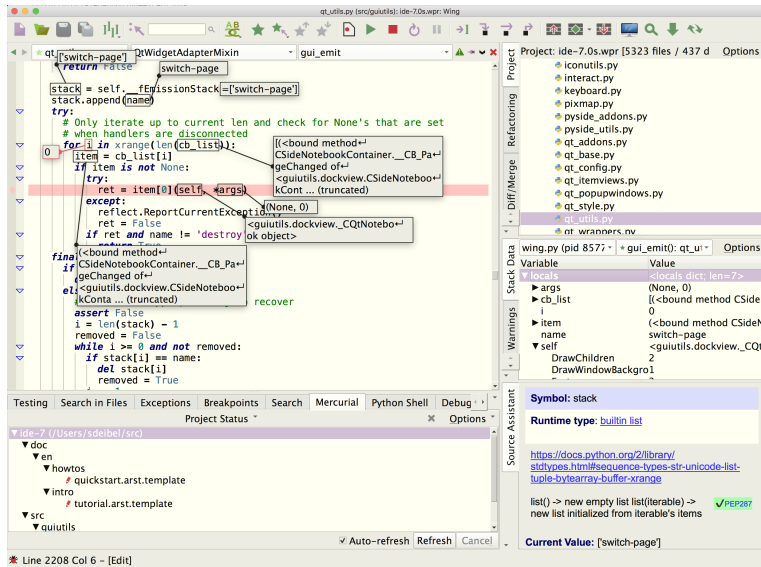
You will need to add the specified products directory manually, and then unpack WingDBG into it.

### ***Related Documents***

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

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

## 7.3. 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 developers, 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:** This document is not being maintained and was last tested with Turbogears 2.

### Project Configuration

This section assumes your Turbogears 2 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 entry point 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**.

### ***Debugging***

To debug Turbogears 2 from Wing:

- 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 or **Debug** menu. If Wing shows 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 steps to keep debugging working 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. From here, you can step through code or inspect the program state with **Stack Data** and other tools. In Wing Pro, the **Debug Console** provides a command line that allows you to interact with the current stack frame in your debug process. All the debugging tools are available from the **Tools** menu.

### ***Remote Development***

Wing Pro can work with Pyramid code that is running on a remote host, VM, or container. To do this, you need to be able to connect to the remote system with SSH. Then you can create your project in the same way as above, using the **Connect to Remote Host via SSH** project type. See [Remote Hosts](#) for more information on remote development with Wing Pro.

### ***Related Documents***

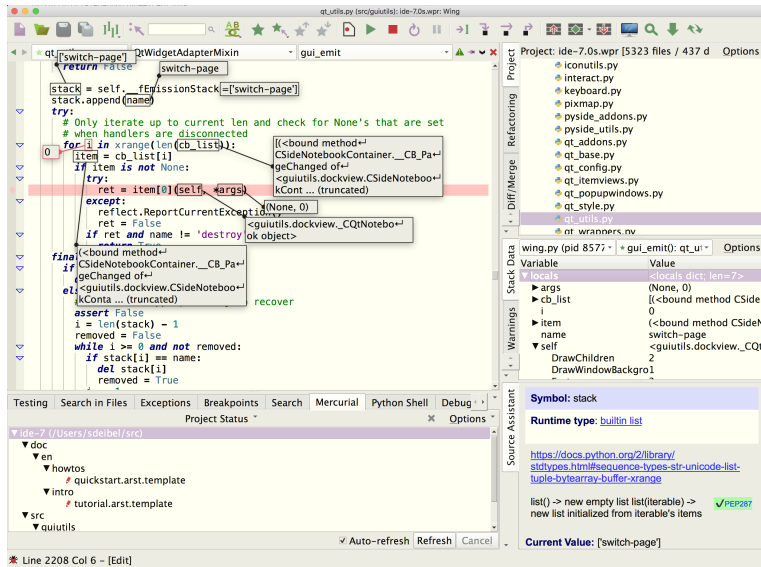
For more information see:

- [Turbogears home page](#) for downloads and documentation.
- [Quickstart Guide](#) contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#) documents Wing in detail.

## 7.4. 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 Zope.

Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional developers, 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](#).

**Note:** This document is not being maintained and was last tested with Zope2.

If you are using Plone, see also [Using Wing with Plone](#). If you are using Zope3, now known as BlueBream, see [Debugging Externally Launched Code](#) or try `z3wingdbg` by Martijn Pieters.

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

### 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).

- **Install Wing** -- Download and install [Wing](#). 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 entry point 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 Entry Point** 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 Console** (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 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



**Debugger > Network > Location Map** preference setting that maps **192.168.1.1** to a list containing **/home/myuser/Zope** and **e:/Zope**. For more information on this, see [File Location Maps](#) and [Location Map Examples](#) in the Wing 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.

### ***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.

### ***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 [Settings Directory](#)). It contains additional logging information from the Wing process.

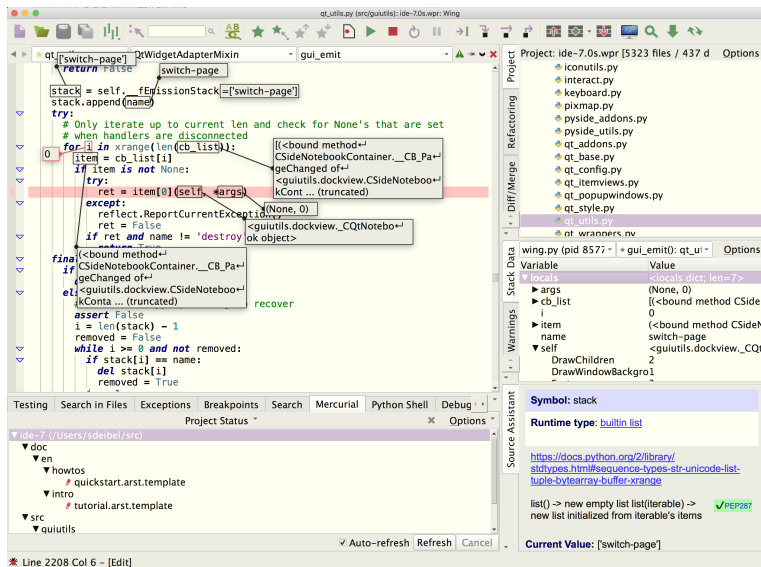
For additional help, send these errors logs to [support at wingware.com](mailto:support@wingware.com).

### ***Related Documents***

For more information see:

- [Zope home page](#) contains much additional information for Zope programmers.
- [Quickstart Guide](#) contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#) documents Wing in detail.

## 7.5. 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 developers, 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](#).

**Note:** This document is not being maintained and was last tested in ancient times.

### 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 [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 **Debugger > Listening > 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.

### Remote Development

Wing Pro can work with mod\_python code that is running on a remote host, VM, or container. To do this, you need to be able to connect to the remote system with SSH. Then you can create your project in the same way as above, using the **Connect to Remote Host via SSH** project type. See [Remote Hosts](#) for more information on remote development with Wing Pro.

### Related Documents

For more information see:

- [Mod\\_python website](#) for downloads and documentation.

## Unmaintained How-Tos

- [Quickstart Guide](#) contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#) documents Wing in detail.

## 7.6. Debugging Code Running Under Py2exe



**Wing** is a Python IDE that can be used to debug Python code running in an application packaged by **py2exe**. This is useful to solve a problem seen only when the code is running from the package, or so that users of the packaged application can debug Python scripts that they write for the app.

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

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

This document just describes how to configure Wing for debugging Python code running under **py2exe**. 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:** This document is not maintained and was last tested in 2007.

### Configuring the Debugger

To debug code running under **py2exe** you will need to use **wingdbstub** to initiate debug from outside of Wing, as described in [Debugging Externally Launched Code](#), along with some additional configuration described below.

There are two important ways in which the environment differs when code runs under **py2exe**:

1. When **py2exe** produces the \*.exe, it strips out all but the modules it thinks will be needed by the application. This will remove modules needed by Wing's debugger.
2. **py2exe** runs in a slightly modified environment and it ignores the **PYTHONPATH** environment.

As a result, some custom code is needed so the debugger can find and load the modules that it needs:

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

# Start debugging
import wingdbstub

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

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

```
EXTRA_PYTHONPATH=\\Python25\\Lib\\site-packages\\py2exe\\samples\\simple\\dist;\\Python25\\lib;\\Python25\\dlls
WINGDB_EXITONFAILURE=1
```

In this example, **\\Python25\\Lib\\site-packages\\py2exe\\samples\\simple\\dist** contains the source for the packaged application and also the copy of **wingdbstub.py** used to initiate debug.

The other added path entries point at a Python installation that matches the one being used by **py2exe**. This is how the debugger will load missing standard library modules from outside of the **py2exe** package.

Setting **WINGDB\_EXITONFAILURE** causes the debugger to print an exception and exit if it fails to load. Without this it will fail silently and continue to run without debug.

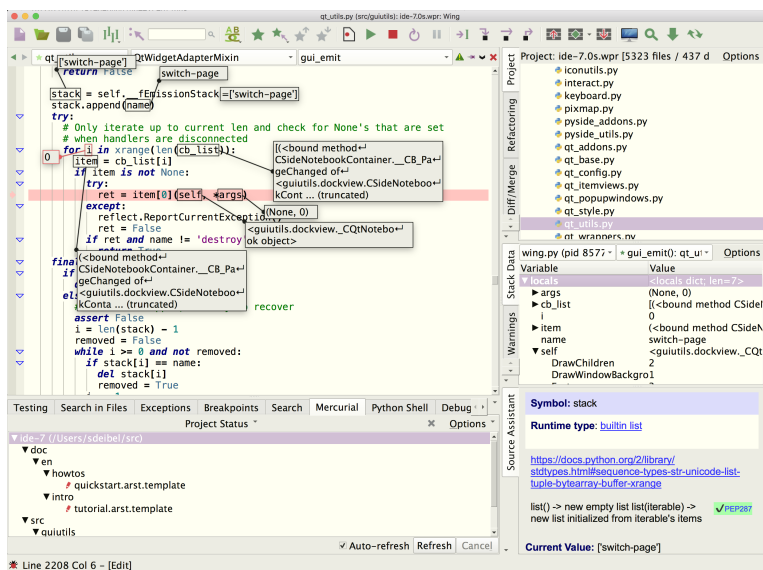
The above was tested with Python 2.5 using **py2exe** run with **-q** and **-b2** options.

### **Related Documents**

For more information see:

- [py2exe home page](#) provides downloads and documentation.
- [Quickstart Guide](#) contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#) documents Wing in detail.

## 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 developers, 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](#).

**Note:** This document is not maintained and was last tested in 2012.

### Debugging IDA Python in Wing

IDA embeds a Python interpreter that can be used to write scripts for 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, this is the full path of Wing's **.app** folder.

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 will be made as soon as the above code is executed.



## Unmaintained How-Tos

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

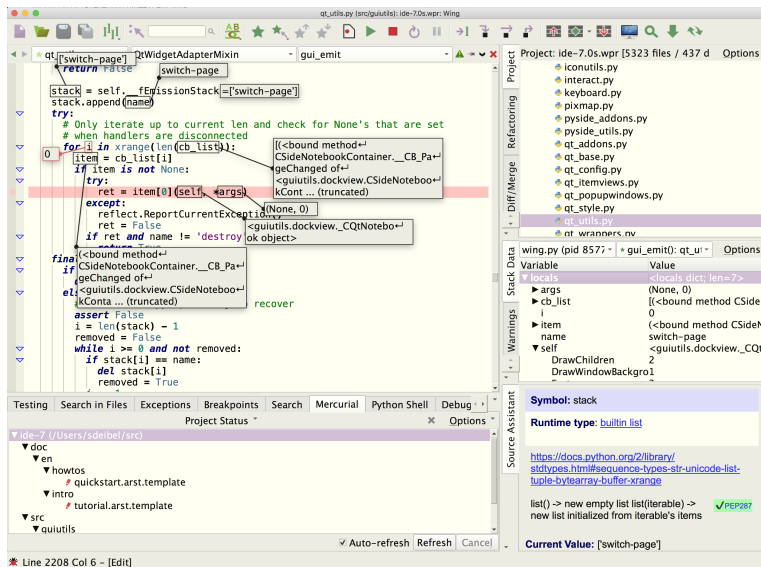
For more information on this configuration, see [Debugging Externally Launched Code](#).

### ***Related Documents***

For more information see:

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

## 7.8. Using Wing with IronPython



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

Two versions of Wing are appropriate for use with this document: Wing Pro is the full-featured Python IDE for professional developers, 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](#).

**Note:** This document is not maintained and was last reviewed in 2011.

### Project Configuration

For instructions on setting up Wing with IronPython, see [IronPython and Wing: Using 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, in `src\wingutils\generate_pi.py` inside the Wing install directory, which is listed in Wing's **About** box.

### Related Documents

For more information see:

- [IronPython home page](#) provides downloads and documentation.
- [Quickstart Guide](#) contains additional basic information about getting started with Wing.
- [Tutorial](#) provides a gentler introduction to Wing's features.
- [Wing Reference Manual](#) documents Wing in detail.