Wing IDE Reference Manual
for version 1.1b7-2

Archaeopteryx Software, Inc.
www.archaeopteryx.com

September 17, 2001

Archaeopteryx Software, Inc.
P.O. Box 1937
Brookline, MA 02446
United States of America

Contents

Chapter 1

Introduction

Thanks for choosing Archaeopteryx Software's Wing IDE! This manual will help you get started and serves as a reference for the entire feature set of Wing IDE.

The manual is organized by major functional area of Wing IDE, including the project manager, text editor, source code browser, and debugger. Several appendices document the entire command set, provide pointers to resources and tips for Wing and Python users, and list the full software license.

The rest of this chapter describes how to install and start using Wing IDE. If you hate reading manuals, you should be able to get started by reading this chapter only.

★ Throughout this manual, key concepts and important notes are highlighted in the same way as this paragraph. If you are skimming only, look for these marks.

1.1 Licenses

Every copy of Wing requires either an evaluation (demo) or a paid license to run, in addition to an installation made from the download or CD media. Demo licenses can be obtained from our web site at `http://wingide.com/wingide/demo` and permanent licenses can be purchased in the online store at `http://wingide.com/order`. When Wing is first started, it will ask to copy your license file into place.

Evaluation licenses may be used for 30 days to evaluate Wing, and allow all features to be used. Paid licenses may be used with any version in the 1.x series and allow access to the source code via `wingide.com`. Licenses may be single user licenses, which allow one named user to use Wing, or floating multi-user licenses, which allow up to a set number of users to use Wing at the same time.

1.2 Installing

★ **Quick start on Linux:**

To install Wing for individual use on an RPM-based system, obtain `wingide-1.1b7-2.rpm` and a license file from `wingide.com`, then run `rpm -i wingide-1.1b7-2.rpm` as root.

For single-user installation on systems without RPM, obtain `wingide-1.1b7-2.tar.gz` and a license file from `wingide.com`, unpack the tar file with `tar -zxvf wingide-1.1b7-2.tar.gz`, cd to the `wingide-1.1b7-2` directory, type `wing-install.py`, and answer the questions to specify where the program files should go.

After installing, the command `wing` should start the IDE. You may have to modify your `path` if you've installed the executables in a directory that isn't already on your path. You may also have to open a new shell or request that your shell rescan the disk for executables (for example, with `rehash` under tcsh).

Once Wing starts, follow the instructions to locate your license file (Wing will be copy it into place), accept the licensing agreement, and set up your initial preferences.

Quick start on Windows

Ensure that Python 1.5.2, 2.0, or 2.1 is installed on your system. Obtain the `wingide-1.1b7-2.exe` installer package from `wingide.com` and execute it on your system. After installing, run wing by selecting `Wing` from the `Programs:Wing IDE` section of the Start menu.

Once Wing starts, follow the instructions to locate your license file Wing will be copy it into place), accept the licensing agreement, and set up your initial preferences.

1.2.1 Supported Platforms

This version of Wing is available for Microsoft Windows and Linux.

The Windows product has been tested on Windows 98se, NT4 SP3, and 2000 SP1.

The Linux product has been tested on RedHat 6.0, 6.1, 6.2, and 7.1; Suse 6.2, 6.4, and 7.0; Caldera 2.4; Mandrake 6.1, 7.1, 7.2, and 8.0; and Debian 2.2. On RedHat 6.0, you must install Python 1.5.2, 2.0, or 2.1 and use that to run your debug program and the tar file installer (if RPM is not used), instead of the default Python 1.5.1 installation that comes with RedHat 6.0. On Suse, you may need to install the gmp and python packages, or install Python from source, since Python is not installed by default here.

RedHat 5.2 is known not to work and there are no plans for supporting it.

1.2.2 Prerequisites

In order to use Wing, you need to have the following third party materials installed on your system:

For Linux

- Python 1.5.2, 2.0, or 2.1
- enscript version 1.1.6 or later (for printing only)
- Adobe Acrobat Reader version 4.0.5 or later (for printing only)

Most users already have these available. The RPM distribution will check for most of these dependencies. If you are installing from tar, you must verify manually that you have these installed already, as there is no automatic dependency checker in the tar file installation.

By default, Wing uses Adobe Acrobat Reader and Netscape Navigator for viewing the manual and other items in the Help menu. Other PDF viewers and web browsers can be used instead; see section 2.2.7 for information on configuring these.

Gnome users should note that Wing comes with its own copy of GTK 1.2.8 that has disabled use of themes. As a result, Wing ignores your theme settings and always runs as if the `Default` theme were selected. This avoids problems with with some versions of GTK and some themes. You can get Wing to run against your native installed version of GTK, and to use themes, in one of two ways: (a) specify `--system-gtk` as the first command line argument for the `wing` startup script, or (b) rename or move `WINGHOME/bin/gtk-bin` (be sure to retain a copy in case your system's version of GTK is one of those that causes frequent crashing in Wing).

For Windows

- Python 1.5.2, 2.0, or 2.1

If you have ordered the CD, most of the above are available in the `3rdparty` directory. They are also available in the download area of our website at `ftp://archaeopteryx.com/pub/wingide`.

In order to view the PDF versions of the Wing IDE manual, you must also have a PDF viewer such as Adobe Acrobat Reader 4.0.5 installed and configured as your default PDF viewer. Similarly, to view website materials and HTML formatted documentation, you must have Internet Explorer or Netscape installed and configured as your default web browser.

1.2.3 Python versions

Wing contains its own subset of Python 1.5.2 that is used to run the IDE itself (but should not be used for running your debug programs). The debugger, which runs in a seperate process, can run under Python 1.5.2, 2.0, or 2.1.

In most cases, Wing will work out of the box, will find your default python installation, and will use it for debugging. You can, however alter the version of Python that is used to debug your program by using Project Properties and per-file Debug Properties dialogs. See section 6.2 for details.

The interpreter that is used for debugging is also used to determine which version of the Python manual is displayed from the Help menu, and it affects which Python standard libraries are used for source code analysis. See section 5.6 for more information.

1.2.4 Linux Installation with RPM

Wing can be installed from an RPM package on RPM-based systems, such as RedHat and Mandrake. RPM packages for Wing are available from `wingide.com`. After downloading the package file, run `rpm -i wingide-1.1b7-2.rpm` as root or use your favorite RPM administration tool to install the RPM. The RPM places most files for Wing under the `/usr/lib/wingide` directory and creates links for the `wing` and `wingdb` commands in the `/usr/bin` directory.

RPM installation also creates a directory called `floating-locks` in `/usr/lib/wingide`. This is set up to be world writable by default. If you are worried about the security of this location, you may change its permissions or remove it entirely, as long as you don't plan a multi-user installation (described in section 1.7).

In the text that follows, the installation location of Wing is referred to as `WINGHOME`. If you installed from RPM, this will always be `/usr/lib/wing`.

1.2.5 Linux Installation from Tar Archive

Wing may also be installed from a tar archive. This can be used on systems that do not use RPM, or if you wish to install Wing into a directory other than `/usr/lib/wingide`. The tar archive is available from `wingide.com`. Unpacking this archive with `tar -zxvf wingide-1.1b7-2.tar.gz` will create a `wing-1.1b7-2` directory that contains the `wing-install.py` script and a `binary-package.tar` file.

Running the `wing-install.py` script will configure Wing for use with individual licenses (multi-user installation is described in section 1.7). The install script will prompt for the location to install support files for Wing (`WINGHOME`), and the location in which to create symbolic links to `wing` and `wingdb`. These locations default to

`/usr/local/lib/wingide` and `/usr/local/bin`, respectively. The install program must have read/write access to both of these directories, and all users running Wing must have read access to both.

In the text the follows, the installation location of Wing is referred to as `WINGHOME`. If you installed from tar, this will be the location you chose when you ran the installer.

1.2.6 Windows Installation

On Windows, Wing is installed by running the installer executable. The installer relies on the Microsoft installer engine, which will be installed automatically if it is not already on your system. Wing's files are installed by default in `C:\Program Files\Wing IDE`, but this location may be modified during installation. All users of Wing must have `Create File` and `Write` privileges to `C:\Program Files\Wing IDE\profiles`. This is where per-user preferences and other information is stored on Windows. Except for a shortcut in the `Start` menu heirarchy, all files are installed under`C:\Program Files\Wing IDE`.

In the text the follows, the installation location of Wing is referred to as `WINGHOME`. If you installed to the default location, this is `C:\Program Files\Wing IDE`.

1.3 Adding Wing to your Path (for Linux)

In many cases, the Wing executable will already be on the user path. If it is not, you may type in the full path to Wing, or add the location where Wing's executable was installed to your path. This location will be `/usr/bin` if the IDE was installed from RPM. When installed from tar file, it will be the value supplied to the `wing-install.py` script.

Setting the path may be done system-wide or individually. How this is done will differ according to your exact OS version, the shell you are running and, in some cases, the preferences of your system administrator. Typically, per-user setup is in `~/.profile`, `~/.login`, `~/.bashrc`, `~/.cshrc`, or equivalent shell configuration file, and system-wide setup is accomplished with `/etc/profile`, `/etc/cshrc`, or `/etc/csh.login`.

After installing or altering a user's path, you may need to open a new shell for that user or request that the shell rescan the disk for executables (for example, with `rehash` under tcsh).

If for some reason you cannot set the path, typing the full path to the `wing` executable, creating a small shell script containing the full path, or setting an alias will all work as alternatives.

1.4 Running the IDE

You are now ready to use Wing IDE!

On Windows, start Wing IDE from the Program group of the Start menu. To start it on Linux, just type `wing` (or the full path to the executable as described in the previous section).

The first time you run, Wing will create your personal settings directory, `~/.wingide` on Linux or `WINGHOME\profiles\[username]` on Windows. If no user is logged in on Windows, "Default User" is used as the username. The personal settings directory is used to store your license, preferences, and other files used by Wing. If the directory cannot be created, Wing will exit.

Once the personal settings directory has been created, Wing will ask you to locate your license file. This is the file that was emailed to you when you signed up to try the demo, or purchased a permanent license. Once you locate the file, Wing will copy it into place in `~/.wingide/license.dat` (on Linux) or `WINGHOME\profiles\[username]` on Windows. You will then be asked to accept the license terms.

At this time, Wing will also take the opportunity to ask you to specify a few major options, such as your preferred editor personality, print paper size (Linux only), and whether or not to auto-save project files. You may either select values or ask to always use the system defaults (as defined in `WINGHOME/preferences`). Once this is done, Wing will place a file called `~/.wingide/preferences` (on Linux) or `WINGHOME\profiles\[username]` (on Windows), with contents according to your choices. This will include at most only a few of all the available preferences, and you can add to or alter these values at any later time. Please refer to `WINGHOME/preferences` and the rest of this manual for more information about all the supported options.

Whenever you run `wing` from the command line, you may specify a list of files to open. These can be arbitrary text files and a project file. For example, the following will open project file `myproject.wpr` and also the three source files `mysource.py`, `README`, and `Makefile`:

```
wing mysource.py README Makefile myproject.wpr
```

(on Windows, the executable is called `wing.exe`)

Wing determines file type by extension, so position of the project file name (if any) on the command line is not important.

1.5 Installing a Permanent License

The Wing IDE trial license is a temporary unlimited-user license that will expire one month from issue. If you have decided to purchase the product from the `wingide.com` website, you will receive your permanent license file, called `license.dat`, via email.

To install a new license, make sure that no copies of Wing are running on your old demo license (they will cease to function when the license file is changed). Then remove or rename the demo license `~/.wingide/license.dat` (on Linux) or `WINGHOME\profiles\[username]\license.dat` (on Windows). Next, run Wing and follow instructions as it prompts you to locate your license file. The new file will be moved into place by Wing. If your trial license has already expired, you can skip the step of removing the old file because Wing will automatically prompt you for a new license.

You are now ready to use the new license.

1.6 Installing Extra Documentation

The Help menu in Wing IDE provides quick access to the online versions of the Wing IDE manual, the Python documentation collection, and some useful web resources.

The HTML and PDF A4 and US Letter versions of the Wing manual are shipped by default with Wing's binary distribution. Additional manual formats are available from `http://wingide.com/support/manual`.

If you are using Linux, the Python manual is not included in most installations, so you may also wish to download and install local copies of these pages. Place the top-level of the HTML formatted Python manual (where `index.html` is found) into `WINGHOME/python-manual/#.#` in your Wing IDE installation. Substitute for `#.#` the major and minor version of the corresponding Python interpreter (for example, `1.5` or `2.0`). Once this is done, Wing will use the local disk copy rather than going to the web when the Python Manual item is selected from the Help menu.

1.7 Multi-user Installations (Linux only)

A multi-user installation requires that you have the ability to share a common disk area among all the machines that will participate in the installation. This is usually done via NFS under Linux, although other file sharing techniques (such as Samba) will also work.

The primary difference between multi-user installation and single-user installation is the location and nature of the license file. In single-user installations, licenses are in `~/.wingide/license.dat` and license lock files are placed in `/var/tmp`. In multi-user installations, licenses and license lock files are both in `WINGHOME/floating-locks`.

There are two approaches to setting up a multi-user installation:

- You can share the entire Wing installation directory to multiple machines. In this case, the `floating-locks` directory must be writable by all users of Wing on all of the machines, and `WINGHOME` must either be on the users' `path`, symbolic links from a location on the users' `path` must be made to `WINGHOME/wing`, or users must set up aliases or type the full path to `WINGHOME/wing` in order to start Wing.
- You can install one Wing installation on each machine and replace the floating-locks directory on each machine with a symbolic link to a directory accessible via NFS or other file sharing mechanism. This must be done on each machine where Wing IDE is installed, using commands similar to the following:

  ```
  cd /var/lib/wingide
  rmdir floating-locks
  ln -s <path-of-shared-directory> floating-locks
  ```

If you have installed from RPM, the `floating-locks` directory should already exist and be world-writable on each installation, and the IDE should be configured to use it. You only need to alter the installation if you want to change the permissions on the lock file directory or if you want to replace it with a symbolic link, as described above.

If you installed from tar file, you need to perform the installation in a way that indicates that you want it to work in multi-user mode. This is done by running `wing-install.py --multi-user`. At the end of the installation process, this will ask for the name of the directory that should contain license lock files and will create the directory if it does not exist. The script will also ask for the name of the group that all users of the floating license must belong to. Use `<everyone>` to specify that all users can use the floating license. Any group specified must exist prior to running the install script.

In either case, once installation is complete, you must copy the multi-user license file that you have purchased into `WINGHOME/floating-locks` and make sure it is readable by all users of the installation.

★ If copies of Wing crash or are terminated from outside, the license lock files located in `/var/tmp` (or in `WINGHOME/floating-locks` in multi-user installations) may be left in place, consuming one user license each. To fix this, remove those files for which no Wing instance is running. The file name combines the license number, host name, and process ID of Wing so it is possible to determine whether a file is a lost file or an active file:

```
BD12-A63A-690C-A517-pangolin-982
```

Be careful not to remove an active license lock file, as the Wing instance that is using it will cease to function until it is restarted.

1.8 Source Installation

All non-evaluation licenses allow the source code for Wing to be obtained from `wingide.com`. The source is available either as an RPM package or as a tar archive. You should use the RPM if you used the RPM to install the binary, and the tar file if you used the tar file to install the binary.

1.8.1 For Linux

The RPM package installs the source into `/usr/lib/wingide` and is installed by running `rpm -i wingide-source-1.1b7-2.rpm` as root or using your favorite RPM administration tool. The tar archive contains a `source-package.tar` file and a copy of the `wing-install.py` script.

To install from the tar file, obtain the wingide-source-1.1b7-2.tar.gz tar archive as available from `wingide.com`. Unpacking this archive with `tar -zxvf wingide-source-1.1b7-2.tar.gz` will create a `wing-source-1.1b7-2` directory that contains the `wing-install.py` script and a `source-package.tar` file. Running the `wing-install.py` script will ask for the location of your binary installation and then will overlay the source files on top of it.

If you are planning to modify the Wing IDE source, it is strongly recommended that you either install the source as a non-root user from the tar archive, or copy it from `/usr/lib/wingide` as a non-root user. Otherwise only root will be able to modify the source.

See the file `src/README.LINUX` for more information on getting started developing on Linux.

1.8.2 For Windows

To install source on Windows, obtain the wingide-source-1.1b7-2.tar.gz tar archive as available from `wingide.com`. Copy the contents of your Wing IDE binary installation to your development location and then unpack the file `source-package.tar` (found in the source distribution) manually into your development location.

Setting up for development on Windows is complicated and requires a number of additional tools and downloads. See the file `src/README.WIN32` for details on getting started.

1.9 Removing an Installation

On Windows, use the Add/Remove Programs control panel, select Wing IDE and remove it.

To remove an RPM installation on Linux, type `rpm -e wingide`. If you have also installed the source distribution or other support RPMs, these must be removed also. Source must be removed before the binary rpm or at the same time as the binary rpm. Use the `rpm -e wingide wingide-source` command to remove both rpm's at the same time.

To remove a tar archive installation on Linux, invoke the `wing-uninstall` script in

`WINGHOME`. This will automatically remove all binary and source files in the installation that appear not to have been changed since installation, including source files. It will ask whether it should remove any files that appear to be changed.

Chapter 2

Customization

This chapter describes how you can customize Wing IDE according to your needs.

★ Wing may be customized in the following ways:

- Through the preferences files
- Keyboard shortcuts can be defined for any Wing command
- The editor can run with different personalities, including a generic editor personality and an emacs-like mode
- The menubar and toolbar can be altered

2.1 Preferences

Wing has a set of preferences that control the basic layout of the user interface and affect features of the editor, debugger, source browser, and project manager. Although Wing will prompt you to select a few options the first time you run the IDE, it is likely that you will want to refine your preferences settings subsequently.

Default values for all preferences are set in the file `WINGHOME/preferences`. Individual users can override these by placing a preferences file in the `.wingide` subdirectory of their home directory (on Linux) or `WINGHOME\profiles\[username]` (on Windows), which are created the first time Wing is run. The values given in the user-specific preferences file take precedence over any values in the default `WINGHOME/preferences` file.

It is also possible to specify additional preferences files on the command line though the `--prefs-file` option. For example:

```
wing --prefs-file /path/to/myprefs
```

The individual preferences are documented in this chapter and the chapters that follow for each IDE subsystem.

★ Note that setting a preference currently requires quitting and restarting Wing before the preference takes effect.

2.1.1 Preferences File Format

The preferences file format consists of a sequence of lines, each of which is a `name=value` pair.

The name is in *domain.preference* form, where *domain* is the IDE subsystem affected and *preference* is the name of the specific preference (for example, `edit.personality` defines the source editor's runtime personality).

Preference values can be any Python expression that will evaluate to a number, string, tuple, list, or dictionary. Additionally, the constants `true` and `false` are defined and bound to 1 and 0, respectively. Long lines may be continued by placing a backslash (\\) at the end of a line and comments may be placed anywhere on a line by starting them with #.

For security's sake values in a preferences file are evaluated in the context of a restricted execution space and cannot access the disk or network. See the Python Language Manual's RExec module for more information on these restrictions.

2.2 Configuring the User Interface

We ship Wing configured in a way we think will be most usable for the widest range of users. However, a range of options exist to users who want to alter the user interface to suit their needs.

2.2.1 GUI Modes

The following preference can be used to alter windowing style in the graphical user interface:

- **gui.embed-command-bar** - Set to `true` to embed the command bar within each source or manager window. Otherwise, a floating window containing a single shared menu bar and toolbar is shown at the top of the allocated screen area. Default=`true`

You can also change the screen area that will be used by Wing and how windows will be presented initially:

- **gui.work-area-rect** - Specifies the screen rectangle within with Wing will display all its windows. The rectangle is in (left, top, right, bottom) format, for example `(10, 10, 1250, 930)`. When set to `None`, the full screen size is used. Default=`None`
- **gui.initial-window-size** - Defines initial window size for document windows in (width, height) format, for example `(650, 1000)`. When set to `None`, a best guess based on screen size is made. Default = `None`
- **gui.initial-window-position** - Defines (x, y) pixel offset of the first window within the work rectangle. Default=`(10, 10)`
- **gui.stagger-window-position** - Set to `true` to stagger windows rather than bringing up all new windows at the same location. Default=`true`
- **gui.remember-window-positions** - Set to `true` to remember window positions so that documents and other windows are opened at using their last position and size. Default=`true`

- **gui.show-toolbar** - Set to `true` to include the toolbar in windows displayed by Wing or `false` to always hide the toolbar. Default=`true`
- **gui.show-toolbar-in-browser** - Set to `true` to include the toolbar in the source browser window or `false` to exclude it from this window. Default=`false`
- **gui.enable-tooltips** - Set to `true` to display tooltips when the mouse cursor is above an enabled toolbar icon, or `false` never to show tooltips. *Caution: On Linux, tooltips have bugs that cause the GUI to hang up under gtk versions less than 1.2.8. Do not enable this if you've set Wing up to run with an older version of gtk.* Default=`true`.

It is also possible to control which manager windows are visible at IDE startup, and whether or not the most recently open project file is reopened automatically:

- **main.startup-show-project** - Set to `true` to show the project manager window at startup. Default=`true`
- **main.startup-show-browser** - Set to `true` to show the source code browser window at startup. Default=`true`
- **main.startup-show-debugger** - Set to `true` to show the debugger window at startup. Default=`false`
- **main.auto-reopen-last-project** - Set to `true` to reopen the most recently open project file at startup, if no other project file is given on the command line. Default=`true`

2.2.2 Editor Personalities

Wing ships with a default editor personality that acts like a simple graphical text editor.

★ The first thing any emacs user will want to do is to set the editor personality to emulate emacs! This is done with the editor's personality preference:

- **edit.personality** - Set to `'normal'` for vanilla key bindings and actions, and `'emacs'` for emacs-like operation. Default=`'normal'`

Additional information about the editor personalities can be found in sections 4.10 and 4.11 of the Source Code Editor chapter.

2.2.3 Key Equivalents

Wing ships with two key equivalency maps, both found in `WINGHOME`: `keymap.normal` and `keymap.emacs`. These are used as default key maps for the corresponding editor personalities.

However, it is possible to copy these maps and customize them. If you do this, it is best to start with the mapping that most closely matches the editor personality you plan to use.

Key binding definitions in these files are in the form:

```
'key-sequence': 'ide-command'
```

The command portion of the key equivalency definition may be any of the commands listed in this manuals in appendix A. The key sequence is built from key names defined in `WINGHOME/pygtk-0.6.5/GDK.py` starting at line 300; the names are matched without regard to case. A single unmodified key is specified by its name alone (for example, `'Down'` for the down arrow key). Modified keys are specified by hyphenating the key names (for example, `'shift-Down'` for the down arrow key pushed while shift is held down). Multiple modifiers may also be specified, as in `'ctrl-shift-Down'`.

It is also possible to build multi-key combinations by listing multiple key names separated by a space. For example, to define a key equivalent that consists of first pushing **ctrl-x** and then pushing the **a** key by itself, use `'ctrl-x a'` as the key sequence.

Wing may be set to use the customized key map file through the following preference:

- **gui.keymap** - The full path to the alternate keymap to use, if any. Default=`''`

Note that key bindings defined in your mapping will be shown in any menu items that implement the same command. This makes it easier to learn key bindings while using Wing.

2.2.4 Menu Bar

It is also possible to customize the menu bar, although the way in which this is done requires that you have access to the source code, affects all users of a given Wing installation and will change in future versions of Wing.

To alter your menu bar, edit the file `WINGHOME/guimgr/constants.py` and change the definition of the kMenuBarDefn constant. This constant is described in a comment in the file.

Changes take effect when you restart Wing and caution is required as inserting a syntax error will prevent Wing from starting or may prevent menus from being available. Making a copy of the original file is strongly recommended.

2.2.5 Tool Bar

The toolbar can also be altered using `WINGHOME/guimgr/constants.py` by editing the `kToolbarDefn` constant. This also affects all users of a given Wing installation and will change in future versions of Wing.

The toolbar icons are located in `WINGHOME/guimgr/icons`. Making a copy of the original `constants.py` file is strongly recommended.

To enable or disable tooltips over the toolbar, set the `gui.enable-tooltips` preference. The default is to disable tooltips on Linux because of problems with gtk versions less than 1.2.8. On new Windows installations, the default is to enable tooltips (because Wing is shipped with its own copy of gtk). If you have a user preferences file on Windows from installation of version 1.1b3-3 or earlier, you need to set this manually.

2.2.6 Debug output for Wing

It is possible to have Wing's internal status messages appear in the terminal window Wing was launched from by setting the `main.print-wing-debug-output` preference to true. The messages include the location of Wing's library directory and any exceptions that may occur in Wing's Python code. This option is useful when tracking down bugs in Wing itself.

2.2.7 Other Preferences

Some additional preferences are also available for controlling top-level behaviors:

- **gui.display-font-name** - Map of font names to use by default on each supported platform. This affects all areas of the user interface other than the source code editor. A value of None causes Wing to use the externally configured default for that platform. Default=`{ 'linux': None, 'win32': None }`
- **gui.display-font-size** - Map of font sizes (in int form) to use by default on each supported platform. This affects all areas of the user interface other than the source code editor. A value of None causes Wing to use the externally configured default for that platform. Default=`{ 'linux': None, 'win32': None }`
- **gui.open-projects-as-text** - When set to `true`, Wing will open project files as text when they are opened from the File menu. Otherwise, any file with `.wpr` or `.wpu` extension is opened as a project file regardless from where it is opened. Default=`false`
- **gui.max-recent-files** - Maximum number of files to show in the recent files lists in the File and Project menus. Default`10`
- **gui.source-title-style** - Title format used for source files: One of "prepend-relative", which prepends the partial relative path from the project file location to the source file, "append-relative", appends the partial relative path after the source file name, "prepend-fullpath" to always prepend the full path, "append-fullpath" to always append the full path, or "name-only" to always use only the file name without showing any path information. Default=`"append-relative"`
- **gui.file-display-cmds** - Linux only: The commands used to display local disk files from the Help menu, or project files selected for external display. This is a map from mime type to a list of display commands; each display command is tried in order of the list until one works. The mime type `"*"` can be used to set a generic viewer, such as a web browser. Use `%s` to place the file name on the command lines. On Windows, the system-wide configured default viewer for the file type is used instead so this preference is ignored. Default= `{ 'application/pdf': [ 'acroread %s &', 'ghostview %s &' ], '*': [ "netscape -remote 'openFile(%s)'", "netscape %s &" ], }`
- **gui.url-display-cmds** - Linux only: The commands used to display URLs. This is a map from protocol type to a list of display commands; each display command is tried in order of the list until one works. The protocol `"*"` can be

used to set a generic viewer, such as a multi-protocol web browser. Use `%s` to place the URL on the command lines. On Windows, the system-wide configured default web browser is used instead so this preference is ignored. Default= `{ 'http': [ "netscape -remote 'openURL(%s)'", "netscape %s &" ], '*': [ "netscape -remote 'openURL(%s)'", "netscape %s &" ], }`

- **gui.auto-save-before-action** - Set this to control whether or not altered source files and the project are auto-saved before taking actions like starting a debug run, executing a file, or running a build command. When set to `false` the user is instead prompted for action. The user will be prompted for file names for untitled files in all cases, even when this preference is set to `true`. Default=`false`

- **main.extra-mime-types** - This is a map from file extension to mime type. It can be used to add additional mime types to those built into Wing IDE and those found in standard system-wide configuration files. Keys should be the file extension in lower case, and values should be the mime type in lower case, for example `{ 'gif': 'image/gif' }`. Any value defined here will override the Wing-defined defaults and any system-wide defaults. Default=`{}`

- **main.extra-mime-type-names** - This is a map from mime type to human readable names for mime types. One entry should be added for each new mime type added with extra-mime-types (but this is not needed if just adding a new extension to an already-supported mime type). An example entry is `{ 'image/mpmp': 'Multi-pixel moon phase image' }`. Names given here can also be used to override the Wing-defined default names for files. Default=`{}`

- **main.use-native-file-selector** - Set this to `true` to use the file selector that's native to your platform (for example, the Windows file selector instead of the GTK selector). Default=`true`

Chapter 3

Project Manager

This chapter describes how to use the Wing IDE project manager.

★ The project manager is designed to act as a convenient index into the files of your software project, without tying you to Wing or requiring other developers that you work with to use Wing.

It is possible to run Wing without using the project manager. However, doing so will prevent the source browser and other source analysis-based capabilities from easily discovering the extent of your source code base. For this reason, we recommend setting up a project file for your work.

3.1 Creating a Project

Before creating a project, make sure that your project manager window is visible. If it isn't, choose the Project Manager item in the Windows menu to display it.

3.1.1 New Project

To create a new project, use the New Project item in the Project menu. This will prompt you to save any changes to your currently open project and will create a new blank project. If Wing is started without any command line arguments, a blank new project is opened by default.

3.1.2 Adding Files and Packages

To add files to your project, use the Add File, Add Package, and Add Directory Tree menu items in the Project menu. These can also be accessed from the popup menu that appears when right-clicking your mouse on the surface of the project manager window.

- **Add Current File** will add the frontmost current open file to the project if it is not already there.
- **Add File** will prompt you to select a single file to add to the project view. Note that this also may result in adding a new directory to the project manager window, if that file is the first to be added for a directory.
- **Add Package** can be used to add more than one file at once. Select a directory with your left mouse button so that the directory name is shown in the area at the bottom of the file selection window. Then click OK. You will be prompted with a list of files within the selected directory. Highlight any that you wish to add, using shift-click to highlight a contiguous range or ctrl-click to select an arbitrary set. Then click Yes to add all those files to your project.
- **Add Directory Tree** can be used to add many files in a directory structure in one operation. Select a directory with your left mouse button so that the directory name is shown in the area at the bottom of the file selection window. Then click OK to add files recursively from that directory and all its children.

Note that Add Package and Add Directory Tree operate only on files that Wing considers likely to be relevant for adding to the project. If the list is missing files you wish to add, you will need to change the `package-file-types` `package-omit-types` preferences and restart the IDE. See the section 3.8 below for details.

3.1.3 Setting a Main Debug File

Normally, Wing will start debugging in whatever file you have as your front- most window. Depending on the nature of your project, you may wish to specify a file as the default debug entry point.

To do this, right-click on one of your Python files in the project manager window and choose the Set As Main Debug File option from the popup menu, or use the Set Current as Main Debug File item in the Project menu. This file is subsequently run whenever you start the debugger, except if you use the Debug Selected File popup menu item on a specific file or the Debug Current File item in the Run menu.

Note that the path to the main debug file is highlighted in red in the project window. You may clear the default debug entry point with the popup menu's Clear Main Debug File item or the Clear Main Debug file in the Project menu.

3.1.4 Removing Files and Packages

To remove a specific file, select it and use the Remove From Project menu item in the right-click popup menu from the surface of the Project Manager window, or from the Remove Selected Entry item in the Project menu. You can also remove a whole package directory and all the files that it contains in this way.

3.1.5 Saving the Project

You can save your project at any time with the Save Project item in the Project Menu. You will also be prompted to save if you try to close the project, open another project, or exit Wing and you have an unsaved, altered project.

It is possible to set up automatic saving of project files, which is often convenient as they are altered by many actions taken from the user interface. To do this, set the `proj.save-without-asking` preference to `true`.

You can also save a copy of your project to another location or name with Save Project As in the Project menu.

★ Using Save Project As is recommended if you need to alter the location of your project file in relation to your source files because it will update the partial relative paths that the project manager uses to locate files in the project. Otherwise, Wing may not be able to find all of the files in the project.

3.2 Sorting the View

The project window can be set to show your files in one of two modes:

- **By Directory** - This view (the default) shows files organized according to their location on disk. The path names shown are partial relative paths based on the location of the project file. If you alter the location of the project file with Save Project As, these paths will be updated accordingly.
- **By File Type** - This view organizes your files by MIME type.

3.3 Project Types

★ There are two related file formats in which you can save your project. One supports sharing the project file via a revision control system.

The default project type is 'Normal', which results in all project data being stored in a single file. This file usually will end in `'.wpr'` but does not have to.

If you use a revision control system to share code with multiple developers, you should change your project to 'Shared' type using the Project Type item in the Project menu. After making this change, save your project to obtain the two separate project files on disk. The main project file (usually ending in `'.wpr'`) can be checked into revision control and the user-specific file (ending in `'.wpu'`) should not be checked into revision control.

If you subsequently change from a Shared project back to Normal, the user-specific data file, ending in `'.wpu'`, will be removed from disk and its data will be merged back into the main project file.

Note that both the combined 'Normal' file and two split 'Shared' files use the same textual file format that is used for the preferences file. See section 2.1.1 for more information on the format itself.

3.3.1 Shared Project Data

This section enumerates the values stored in the shared project area:

- **file_list** - A list of all the files in the project, using partial relative path names based on the location of the project file.
- **main_debug_file** - The file that you have selected as the main entry point for execution.
- **version** - The version of Wing that wrote this file.
- **build_cmd** - A build command that is executed before debugging any file in the project.
- **file_type** - Either 'shared' or 'normal', indicating the project type as described above.
- **shared_file_attributes** - Values associated with specific files in the project that are relevant to all developers. This currently includes per-file debug options, such as the PYTHONPATH to use, the Python executable to run under, arguments to pass on the command line, the initial run directory, and a build command to execute before debugging that file. Also included here are per-file positioning or view configuration options.

3.3.2 User-specific Project Data

This section enumerates the values stored in the user-specific project area:

- **version** - The version of Wing that wrote this file.
- **user_file_attributes** - Values associated with specific files in the project that are specific to an individual developer. This includes breakpoints, various debugger run options, and window position information.

3.4 Project-wide Properties

Each project has a set of top-level properties that can be accessed and edited via the Properties item in the Project menu. Because these are all debugger-related properties, they are described in the Debugger chapter in section 6.2.1. Note however that these values also inform the source code analysis engine, as described in section 5.6.

3.5 Per-file Properties

Properties similar to those available for the project as a whole can also be set on a per-file basis. This is done by right-clicking on a Python source file and selecting the Set Debug Properties menu item in the popup. Values entered here will override any project-wide values.

Since these all control the debugger, per-file properties are documented in the Debugger chapter in section 6.2.2. See also section 5.6 for information on how these values affect the source code analysis engine.

3.6 Viewing File Information

The bottom of the project manager window contains a file information area that will display file name, file type, and file-level documentation string (when available) for the current selection in the file selection area of the project manager window.

The documentation string contains the file-level docstring for Python files only and currently does not support any other programming languages.

The size of this area can be altered by dragging the divider handle between the file information area and the rest of the project manager window.

3.7 Navigating to Source

Files can be opened from the project manager window by double clicking on the file name, middle-clicking on the file name, or right-clicking and using the Open in Wing IDE menu item.

Files may also be opened using an external viewer or editor by right-clicking on the file and using the Open in External Viewer item. On Windows, this opens the file as if you had double clicked on it in Windows Explorer. On Linux, the preferences `gui.file-display-cmds` and `main.extra-mime-types` can be used to configure how files are opened. See section 2.2.7 for details.

It is possible to debug specific files (even if you have set a main debug file for your project) by right-clicking on the file in the project manager window and choosing the Debug Selected File item from the popup menu. A debug session for the selected file is launched, but the definition of the main debug file is not altered.

You can also execute Makefiles, python source, and any executable files by selecting the Execute Selected File item from the popup menu. This executes outside of the debugger with any input/output occurring in the window from which Wing was launched.

3.8 Preferences

The following preferences affect the project manager:

- **proj.package-file-types** - The file types that will be included when you select the Add Package or Add Directory Tree menu items from the Project menu and choose a directory for loading into the project. Default= `("*.*",)`
- **proj.package-omit-types** - The file types that will be omitted when you select the Add Package or Add Directory Tree menu items, even if a wildcard found in `proj.package-file-types` matches the given file. Default= `("*.o", "*.a", "*.so", "*.pyc", "*.pyo", "core", "*~", "#*#")`
- **proj.file-panel-percent** - The percentage of the total project manager window height that will be used initially by the file area. Set this to `100` if you don't want to see the file information area. Default=`60`

- **proj.file-type** - The default file type for newly created project files, either `'normal'` or `'shared'`. Default=`'normal'`
- **proj.save-without-asking** - Controls whether the project files will be saved without asking the user. Default=`true`
- **proj.reopen-windows** - Selects whether the list of open windows should be saved and reopened later when a project is opened. Use `false` to open only the project when a project is opened and `true` to open the project and also any windows that were open the last time the project was closed. Default=`true`
- **proj.close-also-windows** - When `true`, all document windows open at time of project close will also be closed. When `false` only the project will be closed and other windows will remain untouched. Default=`true`

Changing these values requires you to restart Wing before they take effect.

Chapter 4

Source Code Editor

Wing IDE's source code editor is designed to make it easier to adopt the IDE even if you are used to using other editors. If you are frustrated by the editor or key combinations, please review information in this chapter and in the Customization chapter; you have a fair amount of control over how the editor acts.

★ For those in a rush, key things to know about the editor are:

- The editor has personalities, including one similar to basic editors on Windows and another similar to emacs.
- Key mappings are configurable.
- The editor supports a wide variety of file types for syntax colorization.
- Auto-completion is supported for Python source (but can be turned off if desired).

4.1 Syntax Colorization

The editor will attempt to colorize documents according to their MIME type, which is determined by the file extension, or content. For example, any file ending in `'.py'` will be colorized as a Python source code document. Any file whose MIME type cannot be determined will display all text in black normal font by default.

4.1.1 Supported file types

The editor supports the following file types for syntax colorization:

Mime Type	Description	Allowable Extensions
text/x-python	Python Source	py, cgi, or files starting with a #! specifier that invokes Python
text/html	HTML Source	html, htm, asp, shtml
text/x-c-source	C Language Source	c, h
text/x-cpp-source	C++ Source	cc, cpp, cxx, hh, hpp, hxx
text/x-java-source	Java Source	java
text/x-vb-source	Visual Basic Source	cls, bas, ctl, frm, vbs
text/x-dos-batch	DOS Batch File	bat
text/x-properties	DOS Style INI File	properties, ini, inf, reg
text/x-makefile	Makefile	makefile, mak, and any file named 'makefile' or 'Makefile'
text/x-errorlist	Compilation Error List	err
text/x-sql	SQL Source	sql
text/x-plsql	PL SQL Source	spec, body, sps, spb, sf, sp
text/x-xml	XML Source	xml, xul, glade
application/x-tex	LaTeX Source	tex, sty
text/x-lua-source	Lua Source	lua
text/x-idl	CORBA IDL	idl, odl
text/x-javascript	Javascript	js
text/x-rc	DOS RC File	rc, rc2, dlg
text/x-php-source	PHP Source	php, php3, phtml, inc
text/x-perl	Perl Source	pl, pm, pod
text/x-diff	Diff/CDiff or Patch File	diff, patch
text/x-conf	Conf Files	conf
text/x-pascal	Pascal Source	pas, inc
text/x-ave	Ave document	ave
text/x-ada	Ada Source	abs, adb
text/x-ave	Eiffel Source	e
text/x-ave	Lisp Source	lsp, lisp
text/x-ave	Ruby Source	rb
text/x-ave	Bash File	sh, bsh
text/plain	Plain Text (no highlighting)	all others

If you have a file that is not being recognized, you can use the Source Menu's Syntax

Highlighting section to alter the way the file is being displayed. Your selections from this menu are stored in your project file, so changes made here are permanent in the context of that project.

If you have many files with an unrecognized extension, you can alter the `main.extra-mime-types` preference to add your extension. See section 2.2.7 for details on setting this value. Note however that adding a new MIME type not already in the list above will not work without more extensive modifications to the IDE source code.

4.1.2 Colorization Options

Although the specific colors and other values associated with syntax colorization are not yet user-configurable, you can alter the text and font that is used throughout a file or files. To do this, select the Set Display Font/Size item from the Source menu and make your selection from the dialog's popup menus.

You can make changes for individual files on a per-file basis, or for all files that you open. Either way, your selection is saved in the project file for subsequent work sessions.

Changes are shown immediately on your source, but are only permanent after the font/size selection dialog is closed if you use the Apply and Exit button. Otherwise, values are restored to those in use before the selector dialog was displayed.

4.2 Navigating your Source

★ When you right-click on the surface of the editor, all editor windows will bring up a popup menu with commonly used commands such as Copy, Paste, Undo, and Redo. When the file is a Python file, this menu also includes a construct-by-construct breakdown of the source file.

In order to navigate your source, right-click on the editor surface and select one of the constructs in the popup menu. This will scroll the source code to the position at which that construct is defined and select its point of definition.

Python language constructs in the popup menu are colorized as follows:

- Classes are shown in red
- Variables, object attributes, and imported names are shown in green
- Methods and functions are shown in black

You can also use the Goto Definition menu item in the right-click popup menu to click on a construct in your source and zoom to its point of definition.

The editor popup menu is also available from the popup button at the lower right corner of the editor window.

4.3 Structural Folding

The editor supports optional structural folding for Python, C, C++, Java, Javascript, HTML, files, Eiffel, Lisp, and Ruby files. This allows you to visually collapse logical hierarchical sections of your code as you are working in other parts of the file.

Because this feature adds overhead, it is turned off by default. You can turn it on from the Structural Folding section of the Source menu, or by setting the `edit.enable-folding` preference to `true` (this value is used for new projects also). The preference `edit.fold-line-mode` can be used to determine whether or not a horizontal line is drawn at fold points, whether it is drawn above or below the fold point, and whether it is shown when the fold point is collapsed or expanded. See section 4.14 for allowed values for this preference.

Once folding is turned on, an additional margin appears to the left of source files, where fold points are by default indicated with blue minus signs. Left mouse click on one of these signs to collapse that fold point. Once collapsed, the fold point is by default indicated by a blue plus sign. Clicking again will re-expand it. Preference `edit.fold-indicator-style` can be used to change the style of indicators used at fold points to arrows, or tree style views. Details for changing this preference are in section 4.14

You can also hold down the following key modifiers while clicking to modify the folding behavior:

- **Shift** - Clicking on any fold point while holding down the shift key will expand that point and all its children recursively so that the maximum level of expansion is increased by one.

- **Ctrl** - Clicking on any fold point while holding down the shift key will expand that point and all its children recursively so that the maximum level of expansion is decreased by one.
- **Ctrl+Shift** - On a currently expanded fold point, this will collapse all child fold points recursively to maximum depth, as well as just the outer one. When the fold point is subsequently re-expanded with a regular click, its children will appear collapsed. Ctrl-shift-click on a collapsed fold point will force re-expansion of all children recursively to maximum depth.

Fold commands are also available in the Structural Folding section of the Source menu and by the indicated key equivalents:

- **Toggle Current Fold** - Like clicking on the fold margin, this operates on the first fold point found in the current selection or on current line.
- **Collapse Current More** - Like shift-clicking, this collapses the current fold point one more level than it is now.
- **Expand Current More** - Like ctrl-clicking, this expands the current fold point one more level than it is now.
- **Collapse Current Completely** - Like shift-ctrl-clicking on an expanded node, this collapses all children recursively to maximum depth.
- **Expand Current Completely** - Like shift-ctrl-clicking on a collapsed node, this ensures that all children are expanded recursively to maximum depth.
- **Collapse All** - Unconditionally collapse the entire file recursively.
- **Expand All** - Unconditionally expand the entire file recursively.

Since only a subset of file types supported by Wing IDE also support folding, the preference `edit.fold-mime-types` is used to turn on folding by mime type for only specific file types. This will generally remain unchanged from the defaults but can be used to turn off folding for specific file types, such as only C or only Python source, while keeping folding on for other files.

4.4 Brace Matching

Wing will highlight matching braces in green when the cursor is adjacent to a brace. Mismatched braces are highlighted in red.

You can also cause Wing to select the entire contents of the innermost brace pair from the current cursor position with the Match Braces item in the Source menu.

Parenthesis, square brackets, and curly braces are matched in all files. Angle brackets (< and >) are matched also in HTML and XML files.

4.5 Indentation

The editor provides a range of features for managing indentation in source code. The following preferences affect how the indentation features behave in newly created source files and non-Python files:

1. The preference `edit.tab-size` defines the default size of each tab character, in spaces.
2. The preference `edit.indent-size` defines the default size of each level of indent, in spaces.
3. The preference `edit.indent-style` defines the default indentation style, one of `'spaces-only'`, `'tabs-only'`, or `'mixed'`. Mixed indentation replaces each tab-size spaces with one tab character.
4. The preference `edit.auto-indent` controls whether or not each new line is automatically indented.
5. The preference `edit.show-indent-guides` controls whether or not to show indentation guides as light vertical lines. This value can be overridden on a file by file basis from the Indentation section of the Source menu.

When an existing Python file is opened, it is scanned to determine what type of indentation that is used in that file. If the file contains some indentation, this may override the tab size, indent size, and indent style values given in preferences and the file is subsequently indented in a way that matches its existing content rather than your configured defaults.

For non-Python files, this check does not occur and you can change indentation styles on the fly using the Use Spaces Only, Use Tabs Only, and Use Mixed Tabs and Spaces items in the Indentation portion of the Source menu.

For Python files, you may convert the entire file between different forms of indentation using the Indentation Manager available from the Indentation portion of the Source menu. This is described in section 4.5.5 below.

★ Note that tab size is automatically forced to 8 characters for all Python source files that contain some spaces in indentation, since the Python interpreter defines tabs as 8 characters in size in this case. This version of Wing does not recognize `vi` style tab size comments, but it does apply the configured tab-size when a file contains only tabs in indentation.

4.5.1 Auto-Indent

The IDE ships with auto-indent turned on. This causes leading white space to be added to each newly created line, as `return` or `enter` are pressed. Enough white space is inserted to match the indentation level of the previous line, possibly adding a level of indentation if this is indicated by context in the source (such as `if` or `while`).

Note that if preference `edit.auto-indent` is set to `false`, auto-indent does not occur until the tab key is pressed.

4.5.2 The Tab Key

By default, the tab key acts like auto-indent: the leading white space of the current line is adjusted to achieve a reasonable indentation level for that line.

Existing leading white space is replaced with white space containing either spaces only or tabs and spaces, as determined by the method described above. This behavior may also decrease indent level of a line, if it is deemed to be indented too far according to its context.

If multiple lines are selected at the time that the tab key is pushed, all those lines will be indented or outdented as a unit according to the amount of change necessary for the first line in the selected unit. This is very useful when moving around blocks of code.

To insert a real tab character regardless of the indentation mode or the position of the cursor on a line, type ctrl-tab.

4.5.3 Checking Indentation

Wing analyses existing indentation whenever it opens a Python source file, and will indicate a potentially problematic mix of indentation styles, allowing you to attempt to repair the file. Although files are checked each time they are opened, Wing will display the indentation warning dialog only once per file and will not ask twice about the same file if you do not repair it (this information is stored in the project file). Files can be inspected more closely or repaired at any time using the Indentation Manager described in section 4.5.5.

★ In general, mixing tab/space and space-only indentation in the same file can be confusing, especially if files are viewed with different editors and by different developers. We recommend using spaces only or tabs only as the best alternatives. To convert existing code containing a mix of tabs and spaces, use the Indentation Manager.

4.5.4 Changing Block Indentation

Wing provides Indent and Outdent commands in the Indentation portion of the Source menu, to support increasing or decreasing the level of indentation for selected blocks of text. All lines that are included in the current text selection are moved, even if the entire line isn't selected.

Indentation placed by these commands will contain either only spaces, only tabs, or a mixture of tabs and spaces, as determined by the method described at the start of section 4.5 above.

4.5.5 Indentation Manager

The indentation manager can be used to inspect and change indentation style in Python language source. To display the indentation manager for a given file, use the Indentation Manager item in the Indentation group of the Source menu.

The indentation manager has two parts to it: (1) The indentation report, and (2) the indentation converter.

A report on the nature of existing indentation found in your Python source file is given above the horizontal divider. This includes the number of spaces-only, tabs-only, and mixed tabs-and-space indents found, information about whether indentation in the file may be problematic to the Python interpreter, and the tab and indent size computed for that file. The manager also provides information about where the computed tab and indent size value come from (for example, an empty Python file results in use of the defaults configured in preferences).

Conversion options for your file are given below the horizontal divider. The three tabs are used to select the type of conversion desired, and each tab contains information about the availability and action of that conversion, and a button to do the conversion. Most of these conversions have no parameters that can be altered by the user. Only in the case of conversion from tabs-only to spaces-only indentation styles, the tab size value shown in the indentation report is made editable, so that the configured default can be overridden.

Once conversion is complete, the indentation manager updates to display the new status of the file, and action of any subsequent conversions. Because the indentation manager updates each time its source file is edited, typing in a source window may slow noticeably when its indentation manager window is visible.

4.6 Auto-completion

When editing a Python file, the source editor will attempt to identify construct names as they are typed and will display a popup list of possible matches. Pressing the tab key while the list is displayed will complete the current word with the selected item from the list. Up and down arrow keys or their equivalent (for example, ctrl-p and ctrl-n in the default emacs keymap) can be used to navigate the popup list while it is visible. Auto-completion will abort upon typing the escape, ctrl-g, right/left arrow keys, any character that cannot be contained in an identifier (such as space), or initiation of any other command key sequence or after a period where nothing is typed into the editor.

★ Auto-completion covers most but not all possible scenarios at this time. See the beginning of chapter 5 and section 5.7 for more information on current capabilities.

Until Python includes support for data typing, one way to get a lot more mileage out of the auto-completion facility is to use statements that assert variables as belonging to a specific class. An example is `assert isinstance(obj, CMyClass)`. The code analysis facility will pick up on these and present you with the correct auto-completion values when you type `obj.` subsequently. An added bonus is that your code will catch errors in expected variable types in more cases.

4.7 Auto-save

The source code editor auto-saves files to disk every few seconds. The autosave failes are placed in a subdirectory of your Wing user directory (`~/.wingide/autosave` on Linux or `WINGHOME\profiles\[username]\autosave` on Windows). If Wing ever crashes or is killed from the outside, you can use these files to recover any unsaved changes. It is usually safe to copy the autosave files to overwrite the older unsaved files, but you may want to do a comparison first to verify that the autosave file is what you want.

4.8 Notes on Copy/Paste

There are a number of ways to copy and paste text in the editor:

- Use the Edit menu items. This stores the copy/cut text in the system-wide clipboard and can be pasted into or copied from other applications.
- Use key equivalents as defined in the Edit menu.
- Right-click on the editor surface and use the items in the popup menu that appears.
- On Linux, select text anywhere on the display and then click with the middle mouse button to insert it at the point of click.

- On Windows, click with the middle mouse button to insert the current emacs kill buffer (if in emacs mode and the buffer is non-empty) or the contents of the system-wide clipboard (in all other cases).
- In emacs mode (described in section 4.11 below), ctrl-k (kill-line) will cut one line at a time into the kill buffer. This is kept seperate from the system-wide clipboard and is pasted using ctrl-y (yank-line). On Windows, ctrl-y will paste the contents of the system-wide clipboard only if the kill buffer is empty.
- Select a range of text and drag it using the drag and drop feature.

It's important to note which actions use the system-wide clipboard, which use the emacs kill buffer (emacs mode only), and which use the X windows selection (X Windows only). Otherwise, these commands are interchangeable in their effects.

4.9 Auto-reload of Externally Changed Files

Wing's editor detects when files have been changed outside of the IDE and can reload files automatically, or after prompting you for permission. This is useful when working with an external editor, or when using code generation tools that rewrite files.

Wing's default behavior is to automatically reload externally changed files that you have not yet been changed within Wing's source editor, and to prompt to reload files that have also been changed in the IDE.

You can change these behaviors by setting the value of the `cache.unchanged-reload-policy` and `changed-reload-policy` preferences, as described at the end of section 4.14.

On Windows, Wing uses the change signal capability of the OS to detect changes so notification or reload is usually instant. On Linux, Wing polls the disk by default every 3 seconds; this frequency can be changed with the `cache-reload-freq` preference.

4.10 Normal Editor Personality

★ Wing's source code editor can run with different personalities, either as a normal basic text editor, or as an emacs-like editor that is readily controlled with keyboard-driven commands.

The default personality for Wing is 'normal'. This uses only the graphical user interface for interacting with the editor and doesn't make use of any complex keyboard-driven command interaction.

The editor runs in normal mode when the `edit.personality` preference is set to `normal`. See section 2.1 of the Customization chapter for more information on how to alter your preferences.

In normal mode, the editor provides the following keyboard equivalents, which are defined in the default normal mode key mapping in file `WINGHOME/keymap.normal`.

4.10.1 Cursor Movement

Key Combination	**Command**	**Description**
shift + arrow keys	previous-line-extend, next-line-extend, backward-char-extend, forward-char-extend	Move cursor in indicated direction, extending the current text selection.
shift-page-up and shift-page-down	forward-page-extend, backward-page-extend	Move forward or backward a page, extending the current text selection.
ctrl-right-arrow and ctrl-left-arrow	backward-word, forward-word	Move cursor backward or forward a word. These can be combined with shift key for extension of selection.

home and end	beginning-of-line, end-of-line	Move cursor to beginning or end of line. Repeated home on an indented line will alternate between the absolute beginning and end of indentation. These can be combined with shift key for extension of selection.
ctrl-home and ctrl-end	start-of-document, end-of-document	Move cursor to beginning or end of document. These can be combined with the shift key for extension of selection.

4.10.2 Indentation

Key Combination	Command	Description
tab	indent-to-match	Indent the current line or selection to match the preceding non-blank line, adding or subtracting indentation as appropriate. See section 4.5 above for more information on indentation behavior.
ctrl-tab	forward-tab	Insert a forward tab character.
shift-tab	backward-tab	Insert a backward tab character.
ctrl-shift-greater	indent-region	Increase indent of the current line or selected region by one level of indentation.
ctrl-shift-less	outdent-region	Decrease indent of the current line or selected region by one level of indentation.

4.10.3 Commenting and Justification

Key Combination	Command	Description
ctrl-slash	comment-out-region	Comment out the selected region of code. This operates on whole lines, extending the current selection as necessary.
ctrl-shift-question	uncomment-out-region	Undo commenting out the selected region of code. This operates on whole lines, extending the current selection as necessary.
ctrl-j	fill-paragraph	Rejustify the paragraph of text at the current selection's starting position. This operates on whole lines, and is most useful for formatting comments, long strings, or documentation. Set the column at which to wrap with the `edit.text-wrap-column` preference.

4.10.4 Insertion and Deletion

Key Combination	Command	Description
delete	forward-delete-char	Delete the character in front of the cursor, or the current selection if not empty.
ctrl-delete	forward-delete-word	Delete the word in front of the cursor.
insert	toggle-overtype	Alternate between overtype and insert mode.
ctrl-backspace or alt-backspace or alt-delete	backward-delete-word	Delete the word behind the cursor.
ctrl + keypad plus and minus keys	zoom-in, zoom-out	Zoom text in and out by increasing and decreasing font sizes.

4.10.5 Undo and Clipboard

Key Combination	Command	Description
ctrl-z	undo	Undo most recent edit.
ctrl-y	redo	Redo most recent undone edit.
ctrl-x	cut	Cut the current selection to clipboard.
ctrl-c	copy	Copy the current selection to clipboard.
ctrl-v	paste	Paste from clipboard, replacing the current selection.
ctrl-a	select-all	Select the entire document.

4.10.6 File and Window Control

Key Combination	Command	Description
ctrl-o	open	Open a new document from disk.
ctrl-s	save	Save the current document to disk.
alt-d	delete-window	Close the current window.

4.10.7 Search and Replace

Key Combination	Command	Description
ctrl-f	search-forward	Initiate search in current document.
ctrl-r	query-replace	Initiate search/replace in current document.
ctrl-l	goto-line	Jump to a selected line number.
ctrl-e or ctrl-]	brace-match	Select the region between the nearest braces or defining the inner most code block reached from current cursor position.
ctrl-right-click	goto-selected-symbol-defn	Jump to point of definition of the symbol clicked on. This does not always succeed, as described at the beginning of chapter 5.

4.10.8 Macros

Key Combination	Command	Description
ctrl-shift-(	start-kbd-macro	Start recording a macro (will contain any keystrokes and commands issued to the editor).
ctrl-shift-)	stop-kbd-macro	Stop recording current macro and store it.
ctrl-m	execute-kbd-macro	Execute most recently defined keyboard macro relative to current cursor position.

See the section 2.2.3 for information on how to alter or replace this mapping.

4.11 Emacs Emulation

When preference `edit.personality` is set to 'emacs', the editor will run in a mode that emulates many emacs behaviors. In this mode, key strokes can be used to control most of the editor's functionality, using a textual interaction 'mini-buffer' at the bottom of the editor window where the current line number and other informational messages are normally displayed.

When in emacs mode, the editor supports the following key combinations for access to commands.

4.11.1 Cursor Movement

Key Combination	Command	Description
shift + arrow keys	previous-line-extend, next-line-extend, backward-char-extend, forward-char-extend	Move cursor in indicated direction, extending the current text selection.
ctrl + space bar	set-mark-command	Set selection start to current position and start automatically selecting from that point as the cursor is moved with arrow keys, page up/down, search commands, or in other ways.
ctrl-g	stop-mark-command	Cancel selecting a range, as initiated by set-mark-command, and set the cursor position to the end of the range.
home and end or ctrl-a and ctrl-e	beginning-of-line, end-of-line	Move cursor to beginning or end of line. Repeated home on an indented line will alternate between the absolute beginning and end of indentation. These can be combined with shift key for extension of selection.
ctrl-n and ctrl-p	next-line and previous-line	Move cursor to next or previous line.
ctrl-b and ctrl-f	backward-char and forward-char	Move cursor backward or forward one character.
ctrl-v and alt-v	forward-page and backward-page	Move forward or backward one page.

ctrl-home and ctrl-end	start-of-document, end-of-document	Move cursor to beginning or end of document. These can be combined with the shift key for extension of selection.
ctrl-l	center-cursor	Center the cursor on screen.

4.11.2 Indentation

Key Combination	**Command**	**Description**
tab	indent-to-match	Indent the current line or selection to match the preceding non-blank line, adding or subtracting indentation as appropriate. See section 4.5 for more information on indentation behavior.
ctrl-c shift->	indent-region	Increase indentation of the currently selected range of lines (or current line when selection is empty) by one indentation level.
ctrl-c shift-<	outdent-region	Reduce indentation of the selected range of lines (or current line when selection is empty) by one indentation level.
ctrl-tab	forward-tab	Insert a forward tab character.
shift-tab	backward-tab	Insert a backward tab character.

4.11.3 Commenting and Justification

Key Combination	**Command**	**Description**
ctrl-c c or ctrl-c shift-numbersign	comment-out-region	Comment out the selected region of code. This operates on whole lines, extending the current selection as necessary.
ctrl-c u	uncomment-out-region	Undo commenting out the selected region of code. This operates on whole lines, extending the current selection as necessary.

escape q or ctrl-j	fill-paragraph	Rejustify the paragraph of text at the current selection's starting position. This operates on whole lines, and is most useful for formatting comments, long strings, or documentation. Set the column at which to wrap with the `edit.text-wrap-column` preference.

4.11.4 Insertion and Deletion

Key Combination	**Command**	**Description**
ctrl-d	forward-delete-char	Delete the character in front of the cursor, or the current selection if not empty.
ctrl-delete or alt-d	forward-delete-word	Delete the word in front of the cursor.
delete	clear	Delete current selection, if any.
insert	toggle-overtype	Alternate between overtype and insert mode.
ctrl-backspace or alt-backspace or alt-delete	backward-delete-word	Delete the word behind the cursor.
ctrl + keypad plus and minus keys	zoom-in, zoom-out	Zoom text in and out by increasing and decreasing font sizes.

4.11.5 Undo and Clipboard

Key Combination	**Command**	**Description**
ctrl-/ or ctrl-x u	undo	Undo the most recent edit action.
ctrl-w or shift-delete	cut	Cut the current selection to system-wide clipboard.
alt-w or ctrl-insert	copy	Copy the current selection to system-wide clipboard.
ctrl-y or shift-insert	paste	Paste from system-wide clipboard, replacing any current selection.

ctrl-k	kill-line	Remove the current line after the cursor point and place it into the kill buffer. Groups of lines deleted this way can be pasted subsequently with yank-line (ctrl-y). This is kept seperate from the system-wide clipboard and (on X windows) from the current selection. This command removes the newline only if there is no visible character on the line after the cursor.
ctrl-y	yank-line	Paste the current contents of the kill buffer, created with one or more adjacent kill-line commands, replacing any current selection. If the kill buffer is empty, the contents of the system-wide clipboard is pasted instead.

4.11.6 File and Window Control

Key Combination	**Command**	**Description**
ctrl-x ctrl-c	quit	Quit the application, first prompting to save any unsaved documents.
ctrl-x ctrl-s	save	Save current file to disk.
ctrl-x k	kill-buffer	Close the current file, prompting to save if necessary.
ctrl-x ctrl-f	open	Open a file from disk. Enter the filename at prompt, optionally using the tab key to auto-complete the current entry. Two tabs in a row will display a list of possible matches; continue typing or click on a match as desired. You can also use the up/down arrow keys to scroll through recently-opened files. Hitting enter opens the selected file. Exit without opening with ctrl-g or esc.
ctrl-x i	insert-file	Insert a file at current cursor position. Interaction is the same as for the open command.

ctrl-x b	switch-document	Switch to another document. Hit enter to select the specified default, type file name fragments (tab and double-tab can be used for completion and viewing list of matches), or use the up/down arrow keys to scroll through a list of options. Enter accepts and switches; escape or ctrl-g aborts.
ctrl-x 5 0	delete-window	Close the current editor window.

4.11.7 Search and Replace

Key Combination	Command	Description
ctrl-s	search-forward	Initiate interactive incremental forward search. The search is case sensitive if you type any capital letters; otherwise case-insensitive. Exit search by using arrow keys or other commands, or abort (returning to original position) with ctrl-g or esc.
ctrl-r	search-backward	Initiate interactive incremental backward search. Acts like forward search, only proceeds backward in source.
alt-shift-percent or escape shift-percent	query-replace	Initiate interactive query/replace. First type search string, then replace string and respond to prompts with ’y’ and ’n’. You may use the tab key to auto-complete values you type into the search and replace strings. End the session with ctrl-g or escape.
alt-g or alt-l or escape g	goto-line	Go to a specific line; enter the line number at prompt.
ctrl-b	brace-match	Select all text within the innermost braces from current cursor position.

ctrl + mouse-button-3	goto-selected-symbol-defn	Jump to point of definition of the symbol clicked on. This does not always succeed, as described at the beginning of chapter 5.

4.11.8 Macros

Key Combination	**Command**	**Description**
ctrl-x shift-(	start-kbd-macro	Start recording a macro (will contain any keystrokes and commands issued to the editor).
ctrl-x shift-)	stop-kbd-macro	Stop recording current macro and store it.
ctrl-x m	execute-kbd-macro	Execute most recently defined keyboard macro relative to current cursor position.
escape + numbers	initiate-repeat-*	Start entering a repeat value which will cause the immediately following command or keystroke to be executed the specified number of times. Use esc or ctrl-g to abort. This is particularly useful to execute a keyboard macro multiple times in a row.

4.11.9 Debugging

Key Combination	**Command**	**Description**
ctrl-c ctrl-c	debug-continue	Start debugging the main project debug file, stopping at the first encountered breakpoint.
ctrl-c ctrl-s	debug-stop	Stop the current debug session at current run position, as if at a breakpoint.
ctrl-c ctrl-k	debug-kill	Stop and end the current debug session.
ctrl-x space	break-toggle	Add a regular breakpoint at current line of code if there isn't one already, or remove the existing breakpoint.

4.11.10 Other

Key Combination	Command	Description
alt-x or escape x	command-by-name	Enter a command by name for execution. Use the tab key for auto-completion or double-tab to see a list of possible matches. Up/down arrow keys also work for scrolling through recent commands. Hit enter to execute the command or esc or ctrl-g to abort.
ctrl-c ctrl-d	show-debug-window	Show the debugger window. This is not a usual emacs editor key binding.
ctrl-c ctrl-b	browse-project-modules	Show the source code browser window. This is not a usual emacs editor key binding.
ctrl-c ctrl-p	show-project-window	Show the project manager window. This is not a usual emacs editor key binding.
ctrl-c ctrl-e	show-error-list	Show the debugger error list window. This is not a usual emacs editor key binding.

Many of these commands are also available through menu items. See section 2.2.3 for information on how to alter or replace this mapping.

4.12 Search/Replace

All editor modes include a graphical search/replace manager, which can be brought up with the Search Manager item in the Edit menu. Commands in this window will apply to the current (most recently at front) source view.

To use this manager, enter search and optionally replace text, set search options, and control the search/replace process with the provided buttons.

Replace actions can always be undone, including Replace All.

Emacs mode additionally provides ctrl-s and ctrl-r incremental search, and query-replace facilities, as described in section 4.11.7 above.

4.13 Keyboard Macros

★ The Edit menu contains items for starting and completing definition of a keyboard or command sequence macro, and for executing the most recently defined macro. Once macro recording is started, any keystroke or editor command is recorded as part of that macro. Most commands listed in sections 4.10 and 4.11 above may be included in macros, as well as all character insertions and deletions.

The keyboard macro feature is currently most useful in emacs mode, where powerful cursor-relative macros can be built for repetitive reformatting tasks. This is done by combining incremental search with cursor movement and typing.

4.13.1 Example (in Emacs mode)

A common task when writing Python bindings for C/C++ libraries is copying lists of `#define` constants and converting them into Python variable assignments.

```
#define SC_MARK_CIRCLE 0
#define SC_MARK_ROUNDRECT 1
#define SC_MARK_ARROW 2
#define SC_MARK_SMALLRECT 3
#define SC_MARK_SHORTARROW 4
#define SC_MARK_EMPTY 5
#define SC_MARK_ARROWDOWN 6
#define SC_MARK_MINUS 7
#define SC_MARK_PLUS 8
```

In emacs mode, the above can be converted by positioning the cursor before the first `#define`, starting macro definition, and executing the following keystrokes:

```
escape 8 ctrl-d ctrl-s <space> <right arrow> = <space> ctrl-
a <down arrow>
```

This deletes the 8 characters '`#define `' (with trailing space) in front of the cursor, jumps to the space after the constant identifier, inserts '`= `', and moves to the beginning of the next line. When this is complete, stop macro recording and type the following to convert the remaining lines:

```
escape 8 ctrl-x e
```

This will execute the macro eight times resulting in the following reformatted source (the first line was reformatted during the creation of the macro):

```
SC_MARK_CIRCLE = 0
SC_MARK_ROUNDRECT = 1
SC_MARK_ARROW = 2
SC_MARK_SMALLRECT = 3
SC_MARK_SHORTARROW = 4
SC_MARK_EMPTY = 5
SC_MARK_ARROWDOWN = 6
SC_MARK_MINUS = 7
SC_MARK_PLUS = 8
```

Combine this technique with ctrl-space (set-mark-command) and copy/paste to alter the order of constructs within a line. Be creative... and don't forget that undo can be used to fix problems caused by incorrect macros or bad cursor position before executing a macro.

Macros will terminate if any command within the macro fails (for example, if an incremental search fails). This can be used to prevent edits when a macro is executed in a location where it doesn't make sense.

4.13.2 Repetition

★ In emacs mode, macros can be executed over and over again by using the escape + numbers key board interaction to set up a repetition count before executing the macro. For example, type 'escape 1 0 ctrl-x e' in emacs mode to execute a macro ten times in a row.

4.14 Preferences

The following preferences are defined for the editor:

- **edit.new-file-extension** - This defines the text, if any, to append to all newly created untitled documents. Default=`'.py'`
- **edit.new-file-eol-style** - This defines the end-of-line style that is used for a newly created file (otherwise, the style used matches existing file content). Select one value for each platform: One of 'lf', 'cr', or 'crlf' for each entry. Default`{'linux': 'lf', 'win32': 'crlf'}`
- **edit.personality** - Selects the editor personality. Default=`'normal'`
- **edit.tab-size** - Sets the tab size, in spaces. This is forced to `8` for all Python files, since the Python interpreter assumes tab size of `8`. Default=`8`
- **edit.indent-size** - Sets size of each indentation level, in spaces. Default=`2`
- **edit.indent-style** - Determines the style of indentation to use by default in new Python source files, and in all non-Python files. This can be one of `'spaces-only'` for spaces only in indentation, `'tabs-only'` to use tabs only, or `'mixed'` to use a mix of tabs and spaces (not recommended). Default=`'spaces-only'`.
- **edit.use-tabs-to-indent-default** - This is a deprecated preference and is no longer used. Use `edit.indent-style` instead.
- **edit.auto-indent** - Set to `true` to automatically indent each new line. Default=`true`
- **edit.show-whitespace** - Set to `true` to show tabs and spaces with visible characters by default. This value may then be overridden on a file by file basis from the Source menu. Default=`false`
- **edit.show-indent-guides** - Set to `true` to show light vertical lines at each indent level. This value may then be overridden on a file by file basis from indentation section of the Source menu. Default=`false`
- **edit.show-eol** - Set to `true` to show end of line characters with visible characters by default. This value may then be overridden on a file by file basis from the Source menu. Default=`false`

- **edit.display-font** - A map that defines the font to use for source code display on each supported platform. Default=`{ 'linux': 'lucidatypewriter', 'win32': 'Courier New' }`
- **edit.display-size** - A map that defines the font size to use for source code display on each supported platform. Default=`{ 'linux': 12, 'win32': 11 }`
- **edit.print-font** - (Linux only) Sets the font name used in printing Python files. One of `'Courier'`, `'Helvetica'`, or `'Times-Roman'`. Default=`{ 'linux': 'Courier', 'win32': 'unused' }`
- **edit.print-size** - (Linux only) Sets the font size used to print Python files. Default=`{ 'linux': 12, 'win32': 'unused' }`
- **edit.print-paper** - (Linux only) Sets the paper size used for printing. One of `'Letter'`, `'Legal'`, `'A3'`, `'A4'`, `'A5'`, `'B5'`, `'C6'`, etc. Default=`{ 'linux': 'Letter', 'win32': 'unused' }`
- **edit.print-python-as-text** - (Linux only) Set this to `true` to print Python files faster but without syntax highlighting. Otherwise the internal Python pretty printing facility is used. Default=`{ 'linux': false, 'win32': 'unused' }`
- **edit.text-print-cmd** - (Linux only) Sets the default command that is issued to print non-Python text files. The command is text with embedded %s to indicate where the printed file's name should be inserted. Default=`{ 'linux': 'enscript --pretty-print \%s', 'win32': 'unused' }`
- **edit.print-header-format** - (Windows only) Set the header format to use for printing. This can be ' any text with any of the following special fields mixed in: %basename% - base file name; %prepend-fullpath% - full path file name; %prepend-relative% - relative path with from project file; %append-relative% - file name with relative path appended; %append-fullpath% - file name with full path appended; %file-time% - file modification time; %file-date% - file modification date; %current-time% - current time; %current-date% - current date; %page% - current page being printed. Default=`{'linux': 'unused', 'win32': '\%prepend-fullpath\%' }`
- **edit.print-footer-format** - (Windows only) Set the footer format to use for printing. Same allowable values as for `edit.print-header-format` preference. Default=`{'linux': 'unused', 'win32': 'Page \%page\%, last modified \%file-date\% \%file-time\%'}`
- **edit.print-header-style** - (Windows only) Text style to use in print header, defined as comma delimited list of name:value pairs or style

names including: font:fontname, size:ptsize, bold, italics, underlined. Default=`{'linux': 'unused', 'win32': 'font:Arial,size:12,bold'}`

- **edit.print-footer-style** - (Windows only) Text style to use in print footer. Same allowable values as for `edit.print-header-style` preference. Default=`{'linux': 'unused', 'win32': 'font:Arial,size:10,italics'}`
- **edit.suspend-analysis-timeout** - The number of seconds between last key press and when the background code analysis is reenabled. Altering this value may make typing more or less responsive. When less than or equal to zero, analysis is never suspended during typing. Default=`3`
- **edit.typing-timeout-interval** - Timeout in seconds between last key press and when user is considered not be be typing, for purposes of hiding auto-completion and call tips popups. Default=`10`
- **edit.autocomplete-names** - Controls whether or not to display the auto-completion popup while editing Python source. Default=`true`
- **edit.autocomplete-attribs** - Controls whether or not to autocomplete attributes (items after '.' in identifiers) while editing Python source. Default=`true`
- **edit.autocomplete-keywords** - Controls whether or not to autocomplete Python keywords while editing Python source. If `None`, the list is obtained from the `keywords` module in the Wing source code. Default=`None`
- **edit.autocomplete-builtin-names** - Sequence of built-in names to be used in Python autocompletion lists. If `None`, the list is obtained from the `__builtins__` module in the Wing source code. Default=`None`
- **edit.autocomplete-names-to-always-del** - Sequence of names to always delete from initial autocompletion lists. If set to `None`, the list defaults to an internal value. Default=`None`
- **edit.autocomplete-names-to-del-if-line-start** - Sequence of names to always delete from initial autocompletion lists if the name is the first one on the line. If `None`, defaults to an internal value. Default=`None`
- **edit.autocomplete-names-to-del-if-not-line-start** - Sequence of names to always delete from initial autocompletion lists if the name is not the first one on the line. If `None`, defaults to an internal value. Default=`None`

- **edit.autocomplete-standard-exception-names** - Sequence of names of standard (built-in) exceptions to include in autocompletion lists. If `None`, defaults to an internal value. Default=`None`

- **edit.autocomplete-word-middle-action** - Whether or not to auto-complete when typing in the middle of a word. Default=`true`

- **edit.text-wrap-column** - The column at which text is wrapped by the fill-paragraph command. Default=`70`

- **edit.lineno-column-width** - The width of the line number display column, or `0` to hide it entirely. This is the default value; any changes from the GUI are recorded in the project file. Default=`0`

- **edit.enable-folding** - Set to `true` to enable structural folding in source editors by default, or `false` to disable. This is the default value; any changes made from the GUI are recorded in the project file and subsequently override this preference setting. Default=`true`

- **edit.fold-mime-types** - Set to a list of mime types for which folding should be enabled by default when preference enable-folding has been turned on to enable folding as a whole. Default=`[ ’text/x-python’, ’text/x-c-source’, ’text/x-cpp-source’, ’text/x-java-source’, ’text/x-javascript’, ’text/html’, ’text/x-eiffel’, ’text/x-lisp’, text/x-ruby ]`

- **edit.fold-line-mode** - Set to ″expanded-above″, ″expanded-below″, ″collapsed-above″, ″collapsed-below″, or ″none″ to indicate where fold lines are shown and whether they are above or below the line where the fold point is located. Default=`"above-collapsed"`

- **edit.fold-indicator-style** - Set to an integer to determine the style of fold indicators used with structural folding: `0` for arrow indicators, `1` for plus/minus indicators, `2` for rounded tree indicators, and `3` for square tree indicators.

- **edit.select-policy** - This is a map from actions to policy for leaving a range selected after the action takes place. Possible actions are ″indent-region″, ″outdent-region″, ″indent-to-match″, ″comment-out-region″, and ″uncomment-out-region″. Possible policies for each are ″always-select″, which always leaves a selection, ″retain-select″ which leaves a selection only if there was one to begin with, and ″never-select″ which never leaves a selection. Default=`{ ’indent-region’: ’retain-select’,`

```
'outdent-region': 'retain-select', 'indent-to-match': 'retain-
select', 'comment-out-region': 'never-select', 'uncomment-out-
region': 'never-select' }
```

The following additional preferences that control reloading of file caches are also relevant:

- **cache.unchanged-reload-policy** - Selects action to perform on files found to be externally changed but unaltered within the IDE. One of `"auto-reload"` to automatically reload these files, `"request-reload"` to ask via a dialog box upon detection, `"edit-reload"` to ask only if the unchanged file is edited within the IDE subsequently, or `"never-reload"` to ignore external changes (although you will still be warned if you try to save over an externally changed file). Default=`"request-reload"`
- **cache.changed-reload-policy** - Selects action to perform on files found to be externally changed and that also have been altered in the IDE. One of `"request-reload"` to ask via a dialog box upon detection, `"edit-reload"` to ask if the file is edited further, or `"never-reload"` to ignore external changes (although you will always be warned if you try to save over an externally changed file). Default=`"request-reload"`
- **cache.reload-check-freq** - Time in seconds indicating the frequency with which the IDE should check the disk for currently open files that have changed externally. Set to `0` to disable entirely. Default=`3.0`

You need to quit and restart Wing before any changes in these preferences take effect.

Chapter 5

Source Code Browser

This chapter describes the source code browser. The browser is intended to act as an index to your source code, supporting inspection of bodies of Python source code from either a module-oriented or class hierarchy-oriented standpoint.

The source code browser is divided into three panels: (1) a hierarchical list view, (2) a panel of display filter options, and (3) an embedded information display area. These panels can be resized by dragging on the seperator handles between them.

★ Wing IDE's source code browser, autocompletion, and other source navigation capabilities all make use of a central source code analyzer.

Because Python does not (yet) support data type definitions, source code analysis currently does not identify the type of all constructs found in your code. This will affect how often Wing manages to present the correct autocompletion list or correctly determines the point of definition of a construct.

For more information, read section 5.7 below.

5.1 Display Choices

The source code browser offers three ways in which to look at your body of source code. These are selected using the radio buttons along the top edge of the class browser window.

5.1.1 Viewing by Module

Viewing by module shows all files that you have placed into your project file within the top major panel of the source code browser window. This module list is a hierarchical expandable tree view that shows all modules, directories, and packages at the top level in alphabetical order. These major units are defined as follows:

- Packages are directories that contain a number of files and a special file `__init__.py`. This file contains a special variable `__all__` that lists the file-level modules Python should automatically import when the package as a whole is imported. See the Python documentation for additional information on creating packages, or look at the Wing source code for examples.
- Directories found in your project that do not contain the necessary `__init__.py` file are listed as 'directory' rather than 'package' in the source browser window.
- Python files found at any level are denoted as 'module'.

Within each top-level package, directory, or module, the browser will display all sub-modules, sub-directories, modules, and any Python constructs. These are all labeled by generic type, including the following types:

- **variable** - a variable defined at the top-level of a Python module
- **function** - a function defined at the top-level of a Python module
- **class** - an object class found in Python source
- **method** - a class method
- **attribute** - an class or instance attribute

5.1.2 Class Hierarchy

When viewing by class hierachy, the browser replaces the hierarchical tree view with a list of all top-level classes found in analyzed code, in alphabetical order.

In this display mode, the structure on disk of your packages, directories, and modules is completely hidden from view. Instead, the hierarchy of your classes is displayed, starting at base classes and working downward to derived classes.

Within each class, in addition to a list of derived classes, the methods and attributes for the class are shown.

5.1.3 All Classes

In order to find classes by name more easily, the browser can be asked to display a list that includes all found Python classes. In this case, all classes (and not just base classes) are displayed at the top-level of the hierarchical viewer.

This view is otherwise identical to the Class Hierarchy view.

★ The source code analyzer will run in the background from the time that you open a project until all files have been analyzed. You may notice this overhead during the first 5 to 30 seconds after you have opened your project, depending on the size of your source base. Until analysis is complete, the class-oriented view within the browser window will only include those classes that have been analyzed. This list is updated as more code is analyzed.

5.2 Display Filters

A number of options are available for filtering the constructs that are presented by the source code browser. These filters are organized into two major groups: (1) construct scope and source, and (2) construct type.

5.2.1 Construct Scope and Source

The following distinctions are made. Constructs in each category can be shown or hidden as a group:

- **Public** - Constructs for public use by any importer of a module or instantiator of an instance. These are names that have zero leading underscores, such as `Print()` or `kMaxListLength`.

- **Private** - Constructs intended to be private to a module or class. These are names that have two leading underscores, such as `__ConstructNameList()` or `__id_seed`. Python enforces local-only access to these constructs in class methods (see the Python documentation for details).
- **Semi-Private** - Constructs intended for use only within related modules or from related or derived classes. These are names that have one leading underscore, such as `_NotifyError()` or `_gMaxCount`. Python doesn't enforce usage of these constructs, but they are helpful in writing clean, well-structured code and are recommended in the Python language style guide.
- **Inherited** - Constructs inherited from a super-class.
- **Imported** - Constructs imported into a module with an import statement.

5.2.2 Construct Type

Constructs in the source code browser window can also be shown or hidden on the basis of their basic type within the language:

- **Classes** - Classes defined in Python source.
- **Methods** - Methods defined within classes.
- **Attributes** - Attributes (aka 'instance variables') of a class. Note that these can be either class-wide or per-instance, depending on whether they are defined within the class scope or only within methods of the class.
- **Functions** - Non-object functions defined in Python source (usually at the top-level of a module).
- **Variables** - Variables defined anywhere in a module, class, function, or method (but not including function or method parameters).
- **Derived Classes** - Classes which descend from another class; this controls whether or not a class' derived classes are shown within its scope.

5.3 Sorting

In all the display views, the ordering of constructs within a module or class can be controlled by the radio buttons labeled 'Sort'.

- **Alphabetically** - Displays all entries within each expanded portion of the tree in alphabetic order, regardless of type.
- **By Type** - Sorts each expanded portion of the tree first by construct type, and then alphabetically.

Sorting doesn't affect the top-level of the hierarchical list view, which is alphabetic in all cases.

5.4 Embedded Information Display

The tabbed area at the bottom of the source code browser can be used to view information about items selected in the hierarchical list view above.

5.4.1 Documentation

When the Documentation tab is selected, the source code browser will display any available Python 'doc string' information for the construct selected in the hierarchical list view above.

This only contains information when a doc string is defined and only works for packages, modules, classes, functions, and methods (and never for variables or attributes).

5.4.2 Source Display

When the Source tab is selected, the source code browser will display the source at point of definition of the selected construct.

The source view is a fully functional source code editor. If you make changes here, any additional view of that source will be updated.

★ You may be asked to save changes if you navigate to another source file after making an edit, since the source files are closed if there isn't another view to them open in another window.

You can disable tracking of your selection in the hierarchical list area by unchecking the 'Follow Selection' check box that is displayed under this tab. Otherwise, the source code view will scroll to position as you change your selection above.

5.5 Navigating to Source

A number of options are available for navigating around source from the browser window:

- In order to switch to a full editor window containing the source file for any construct in the hierachical list view, right-click on the construct in the list view and select the Goto Source menu item.
- For classes that have inherited classes, the right-click popup obtained in this way will also contain one item for each base class, allowing navigation within the hierarchical list view upwards to the location at which the base class is defined.

5.6 Python versions

★ In analysing your source, Wing will use the Python interpreter and `PYTHONPATH` that you have specified in your debug properties. If you have indicated a main debug file for your project, the values from that file's per-file debug properties are used; otherwise the project-wide values are used. Whenever any of these values changes, Wing will completely re-analyze your source code from scratch.

You can view the Python interpreter and `PYTHONPATH` that are being used by the source code analysis engine, by selecting the Show Analysis Stats item in the Source menu. The values shown in the resulting dialog window are read-only but may be changed by pushing the Settings button. See section 6.2 for details on changing these values.

Be aware that if you use multiple versions of the Python interpreter or different `PYTHONPATH` values for different source files in your project, Wing will analyse all files in the project using the one interpreter version and `PYTHONPATH` it finds through the main debug file or project-wide debug properties settings. This may lead to incorrect or incomplete analysis of some source, so it is best to use only one version of Python with each Wing IDE project file.

5.7 Limitations

The following are known bugs affecting source browsing generically (within the source code browser window, or using the editor's popup menu or auto-completion facility):

- Analysis sometimes fails to identify the type of a construct because Python code doesn't always provide clues to determine the data type. In these cases, you may use assertion of `isinstance` to inform the analyzer, as described below.
- Types of elements in lists, tuples, and dictionaries are not identified.
- Docstrings and other analysis information may be out of date if you edit a file externally with another editor and don't reload it in Wing. See section 4.9 for a discussion of the reload options.

- Additional problems are known. Please refer to the bug list at `http://wingide.com/support/issuetrak`

★ One way to inform the code analysis facility of the type of a variable is to add an `isinstance` call in your code. An example is `assert isinstance(obj, CMyClass)`. The code analyzer will pick up on these and present more complete information for the asserted values. This assertion may have the side-effect of catching type errors at the time `isinstance` is called rather then when an invalid operation is attempted.

5.8 Disk Cache

The source code analyzer writes information about files it has recently examined under `~/.wingide/cache` (on Linux) or `WINGHOME\profiles\[username]\cache` (on Windows).

Cache size may be controlled with the `pysource.max-disk-cache-size` preference. However, Wing does not perform well if the space available for the cache is smaller than the space needed for a single project's source analysis information. If you see excessive slowdowns, either increase the size of the cache or disable it entirely by setting its size to 0.

If the cache will be used by more than one computer, make sure the clocks of the two computers are synchronized. The caching mechanism uses time stamps, and may become confused if this is not done.

The analysis cache may be removed in its entirety with no ill effects (for example '`rm -rf ~/.wingide/cache`' on Linux or remove `WINGHOME\profiles\[username]\cache` on Windows). It will be rebuilt as needed subsequently unless it has been disabled.

5.9 Preferences

The following preferences are available for control source code analysis and the source browser:

- **pysource.builtin-interfaces-dir** - Name of the directory with interface files for built-in modules. If a partial path is given, it will be interpreted as relative to `WINGHOME`. Default=`'src/pysource/builtin-pi-files'`
- **pysource.analyze-in-background** - Whether or not Wing should should analyze python source in the background. Default=`true`
- **pysource.instance-attrib-scan-mode** - How Wing will scan for instance attributes. The scan either covers only the `__init__` method or all methods in each class. Valid values are `'init-only'` and `'all-methods'`. Default=`'all-methods'`.
- **pysource.max-disk-cache-size** - The maximum size of the disk cache in megabytes. This is the size of the actual data in the files, so actual disk usage will be somewhat greater because of sector sizes and other factors. Set this to 0 to disable writing any disk files. Default=`50`.
- **pysource.max-background-buffers** - The maximum number of analysis information buffers allowed in memory at once for files that are not being edited. This value is ignored if the disk cache is disabled. Default=`10`.

You need to quit and restart Wing before any changes in these preferences take effect.

Chapter 6

Debugger

This chapter describes how to use the Wing IDE debugger.

★ The Wing debugger consists of two parts: (1) the debug client, represented by the debugger window in Wing IDE, and (2) the debug server, which resides within another process space, possibly on another machine, and communicates with the client via TCP/IP.

The debugger supports launching your application not just from Wing itself but also externally, as with CGI scripts or code running in an embedded scripting facility within a larger application.

When externally launched, your code can attach to the Wing IDE debugger at any point in its execution, and may turn on and off debugging using a simple API.

Because the debugger core is written in optimized C, debug overhead is relatively low; however, you should expect your programs to run about 50% slower within the debugger.

6.1 Specifying Main Entry Point

Normally, Wing will start debugging in whatever file you have as your front- most window. Depending on the nature of your project, you may wish to specify a file as the default debug entry point.

To do this, right-click on one of your Python files in the Project Manager window and choose the Set As Main Debug File option from the popup menu. This file is subsequently executed whenever you start the debugger, except if you use the Debug Selection in the Project Manager's popup menu.

Note that the project manager will highlight the main entry point in red. You may clear the default main debug file subsequently by using the Clear Main Debug File item in the Project Manager window's popup menu.

Regardless of whether a main debug file is defined, you can also start debugging any open file with Debug Current File in the Debug menu, or by right-clicking on an entry in the project manager and choosing the Debug Selection popup menu item.

The main entry point defined for a project is also used by the source code analysis engine to determine the python interpreter version and python path to use for analysis. Thus, changing this value will cause all source files in your project to be reanalysed from scratch. See section 5.6 for details.

6.2 Debug Properties

In some cases, you will also need to set up some properties to use when debugging. These are set either on a project-wide basis or may be specified on a per-file basis using the main entry point file for the debug session. Project-wide debug properties apply in all cases where a per-file value was not given; otherwise the per-file value overrides the project-wide values.

Only per-file debug properties set on the *initially invoked file* are used in the debug session. Even if other files with set properties are used in the debug session, any values set for them will be ignored.

Debug properties may be set project-wide from the Properties item in the Project menu. Per-file debug properties may be set from the Current File Debug Properties item in the Run menu, or from the Set Debug Properties item in the right-click popup menu on the project manager.

6.2.1 Project-wide Properties

The project-wide properties dialog contains the following fields:

- **Python Executable** - When the *Custom setting* radio button is checked and the entered field is non-blank, this can be used to set the full path to the Python installation that should be used when debugging source code in this project. This can be used to control whether the debug program is executed under Python 1.5.2, 2.0, or 2.1. When *Use "python"* is selected, Wing tries to use the default Python obtained by typing `python` on the command line. If this fails, Wing will search for Python in `/usr/local` and `/usr` (on Linux) or in the registry (on Windows).
- **Python Path** - The `PYTHONPATH` is used by Python to locate modules that are imported at runtime with the `import` statement. When the *Use environment* checkbox in this area is checked, the inherited `PYTHONPATH` environment variable is used for debug sessions. Otherwise, is *Custom setting* is selected, the specified `PYTHONPATH` is used.
- **Run Directory** - When the *Use file's directory* radio button is checked, the initial working directory set for each debug session will be the location where the main entry point file is located. Otherwise, when *Custom setting* is selected, the specified directory is used or, when blank, the project file's directory is used.
- **Build Command** - This command will be executed before starting debug on any source in this project. This is useful to make sure that C/C++ extension modules are built, for example in conjunction with an external `Makefile`, before execution is started.
- **Environment** - This is used to specify values that should be added, modified, or removed from the environment that is inherited by debug processes started from Wing IDE. Each entry is in `var=value` form and must be specified one per line in the provided entry area. An entry in the form `var=` (without a value) will cause the given variable to be undefined. Note that you are operating on the environment inherited from the IDE and not defining a blank new environment space. When the *Use env* radio button is checked, any entered values are ignored and the inherited environment is used without changes.

All of these values are stored in the project file on a per-platform basis. This means that when a project is used both on Linux and on Windows (via file sharing or source code control), you must set values seperately for each platform. Wing shows and edits only the values for the platform you are currently running on.

If you are sharing a project file with other developers through a revision control system, it is important to know that any full or partial relative file and directory paths specified within the above values will be stored as a full path in the shared branch of the project file. This means that other developers must work in an environment that supports those paths.

6.2.2 Per-file Properties

The per-file debug properties dialog contains all the same fields as are available in the project-wide properties described above, with the following additions:

- **Run Arguments** - Enter any run arguments here, including the name of the Python source file as the first item in the list.
- **Environment** - This entry area contains some additional radio button options in the per-file properties area. Use 'add to project' to cause the values specified here to be applied to the inherited runtime environment after the project-wide values are applied, or 'add to env' to bypass the project-wide values and apply the per-file values directly to the inherited runtime environment.
- **Show this dialog before each run** - Check this checkbox if you want the debug options dialog to appear each time you start a debug session.

Values entered here will override any project-wide values.

6.2.3 Relation to Source Analysis

The per-file and project-wide debug properties are also used by the source code analysis engine in order to determine the Python interpreter version and `PYTHONPATH` to use in analysis. For this reason, changing either Python Executable or Python Path in the debug properties dialogs will cause Wing to reanalyse your entire source base from scratch. See section 5.6 for details.

6.3 Setting Breakpoints

A number of different styles of breakpoints are available for use with the Wing IDE debugger. These can be set on source code by opening the source file, placing the cursor on the line where a breakpoint is desired, and then either using items in the Breakpoints menu or pushing the breakpoints tool buttons (the rightmost group in the toolbar).

★ You can also set, clear, and edit breakpoints by clicking on the margin to the left of the source editor, where breakpoints are indicated. Plain clicks will toggle to insert a regular breakpoint or remove an existing breakpoint. You can also shift-click to insert a temporary (one-time) breakpoint, control-click to insert a breakpoint and set an ignore count for it, or shift-control-click to insert a conditional breakpoint. When a breakpoint is already found on the line, shift-click will disable or enable it, control-click will set its ignore count, and shift-control-click will set or edit the breakpoint conditional.

The following types of breakpoints are available:

- **Regular** - A regular breakpoint will always cause the debugger to stop on a given line of code, whenever that code is reached.
- **Temporary** - A one-time breakpoint will cause the debugger to stop the first time it is reached, but will be removed immediately thereafter.
- **Conditional** - A conditional breakpoint contains an expression that is evaluated each time the breakpoint is reached. The debugger will stop only if the conditional evaluates to `true` (any non-zero, non-empty, non-`None` value, as defined by Python). You may edit the conditional of any existing breakpoint with the Edit Breakpoint Condition item in the Breakpoints menu.

Once breakpoints have been defined, you can operate on them in a number of ways to alter their behavior. These operations are available as menu items in the Breakpoints menu, and toolbar icons:

- **Ignore** - It is possible to set an ignore count for a breakpoint. In this case, the breakpoint will be ignored the given number of times, and the debugger will only stop at the breakpoint if it is encountered more than the set number of times. The ignore count is reset to its original value with each new debug run.

- **Disable/Enable** - Breakpoints can be temporarily disabled and subsequently re-enabled. Any disabled breakpoint will be ignored until the user re-enables it.
- **Cleared** - Individual breakpoints can be selected and removed.
- **Clear All** - A menu item and toolbar icon also exists to clear all defined breakpoints at once.

6.4 Starting a Debug Session

There are several ways in which to start a debug session from within Wing:

- Choose Debug from the Run menu or push the Start debug tool bar icon. This will run the project's main debug file (described in section 6.1), if one has been defined, or otherwise the file that is open in the frontmost editor window. Execution will stop on the first line of the code.
- Choose Debug Current File from the Run menu or Debug Selection from the right-click popup menu on the project manager to run a specific file regardless of whether a main debug file has been specified for your project. This will stop on the first line of the code.
- Choose Continue from the Run menu or push the Run To Breakpoint icon in the toolbar. This will run the main debug file if one has been defined, or otherwise the file open in the frontmost editor window. Execution continues until a breakpoint or exception is encountered.
- Choose Run to Cursor from the Run menu or push the Run To Cursor icon in the toolbar. This will run the main debug file if one has been defined or otherwise the file open in the frontmost editor window. Execution continues until it reaches the line selected in the current source text window, or until a breakpoint or exception is encountered.

Note that additional options exist for initiating a debug session from outside of Wing and for attaching to an already-running process. These are described in sections 6.13 and 6.12 below.

★ If you are attempting to run your debug process against a non-standard version of Python, for example one that has been compiled with altered values for `Py_TRACE_REFS` or `WITH_CYCLE_GC`, or that has been altered in other ways, you may need to recompile the debugger core module. Please refer to section 6.17 for more information.

6.5 Flow Control

Once the debugger is running, the following commands are available for controlling further execution of the debug program from Wing. These are accessible from the middle panel of the tool bar or from the Run menu:

- At any time, a freely running debug program can be paused with the Pause item in the Run menu or with the pause tool bar button. This will stop at the current point of execution of the debug program.
- Continue and Run To Cursor are also available during the debug session, whenever the program is in paused (non-running) state. These act the same way they do if used to initiate the debug session in the first place.
- At any time during a debug session, the Kill menu item or tool can be used to force termination of the debug program. Note that this stops running the debug program without executing any further lines of code, so may result in problems with code that assumes it will get a chance to do cleanup tasks upon exit. This option is disabled if the current process was launched outside of Wing, so that only those processes spawned by Wing can be killed in this way.
- Attach and Detach may be used to change the debugger between different debug processes. This is for advanced users and is described in section 6.12.

When stopped on a given line of code, execution can be stepped line-by-line by using the Step Over, Step Into, and Step Out menu item or tool bar icons:

- **Step Over** will step over a single Python instruction, whether or not doing so will proceed to a new line of code.

- **Step Into** will attempt to step into the next executed function on the current line of code. If there is no function or method to step into, this command acts like Step Over.
- **Step Out** will complete execution of the current function or method and stop on the first instruction encountered after returning from the current function or method.

6.6 Viewing the Stack

Whenever the debug program is paused at a breakpoint or during manual stepping, the current stack is displayed as a list in the top panel of the debugger window. This list shows all stack frames encountered between invocation of the program and the current run position. Outermost stack frames are higher up on the list.

Note that the stack displayed is a concatenation of all Python stack frames seen, and may include discontinuities if your code calls C/C++ or other non-Python code, which in turn makes calls back into Python. In this case, the C/C++ stack frames will be missing but the overall order and flow of invocation should be apparent from those stack frames that are visible.

When the debugger steps or stops at a breakpoint or exception, it selects the innermost stack frame by default.

In order to visit other stack frames further up or down the stack, use the Up Stack and Down Stack items in the Run menu or the up/down tool bar icons. You can also click directly on the surface of the stack in the top panel of the debugger window.

When you change stack frames, the variables views will be changed accordingly, and the current line of code at that stack frame is presented in an editor window.

6.7 Viewing Variables

The Wing IDE debugger provides two ways in which to look at variables: (1) in a hierarchical tree view that can be expanded and collapsed, and (2) in a textual view that can be set to show a desired number of hierarchical levels. These two views are in the second and third debugger window panels, respectively. Note that the relative sizes of these panels can be adjusted by dragging the resize indicators on the dividers between them.

★ The variables data displayed by Wing is fetched from the debug server on the fly as you navigate. Because of this, you may experience a brief delay when a change in an expansion or stack frame results in a large data transfer.

6.7.1 Tree View

The tree view contains two top-level entries: (1) locals, which contains all the locals and parameters defined in the currently selected stack frame, and (2) globals, which are scoped to be accessible from all stack frames. Note that in the top-level scope, these two name spaces will be the same and will show the same data.

Simple values, such as strings and numbers, will be displayed in the value column of the tree view area. Strings are always contained in `""` (double quotes). Any value outside of quotes is a number or internally defined constant such as `None` or `Ellipsis`.

Complex values, such as instances, lists, and dictionaries, will be presented with an angle-bracketed type and memory address (for example, `<dict 0x80ce388>`) and can be expanded by clicking on the expansion indicator in the Variable column. The memory address uniquely identifies the construct. Thus, if you see the same address in two places, you are looking at two object references to the same instance.

Upon expansion of complex views, the position or name of each sub-entry will be displayed in the Variable column, and the value of each entry (possibly also complex values) will be displayed in the Value column. Nested complex values can be expanded indefinitely, even if this results in the traversal of cycles of object references.

Once you expand an entry, the debugger will continue to present that entry expanded, even after you step further or restart the debug session. Expansion state is saved for the duration of your Wing IDE session.

★ Values that have not yet been defined in the flow of control are shown with value `<undefined>`.

Some data types, such as those defined only within C/C++ code, or those containing certain Python language internals, cannot be transferred over the network. These are denoted with Value entries in the form `<opaque 0x80ce784>` and cannot be expanded further or inspected, although you may be able to use the expression evaluator described in section 6.8 to access them.

Values Class-scoped values seen within an instance are shown in italics.

When the debugger encounters a very long string, this is indicated in the Value column by prepending `<truncated>` to the start of the value. In these cases, the full string can be viewed by clicking on the value and using the Textual View panel at the bottom of the debugger window (explained in section 6.7.2).

Other indicators such as `<huge>`, `<error handling value>`, and `<network timeout during evaluate>` are also used. These are described in section 6.7.5.

Popup Menu Options

By right-clicking on the surface of the tree view of variables, it is possible to obtain a popup menu that contains several commands useful in navigating data structures. Most of the items in this popup menu act upon the specific data value that was right-clicked upon:

- **Zoom To Window** - This will open the selected data value in a seperate variable view window, using the default configured display and value tracking styles.
- **Expand More** - When a complex data value is clicked upon, and that value has parts that are not yet expanded, this menu item will be enabled. It will expand one additional level in the complex value. Note that since this expands a potentially large number of values, you may experience a delay before the operation completes.
- **Collapse More** - When a complex data value is clicked upon, and that value has any expanded parts, the value is collapsed by one additional level.
- **Zoom and Track by Symbolic Path** - This will open the selected data value in a seperate variable view window, tracking its value by the symbolic path that was selected.

- **Zoom and Track by Direct Reference** - This will open the selected data value a seperate variable view window, tracking its value by direct object reference to the value. This is available only if the value clicked upon is a mutable data type. Otherwise, this item is disabled.
- **Zoom and Track by Parent Slot** - This will open the selected data value in a seperate variable view window, tracking its value by a combination of object reference to the parent value and symbolic reference to the slot within the parent.
- **Default Zoom Style** - This is used to change the display style for newly created zoomed variable windows. Available styles are tree, textual (described in the next section), and a combination view containing both as in the main debugger window.
- **Default Zoom Tracking** - This is used to change the way in which variables in newly created zoomed out windows are tracked.

The variable value zooming and tracking features are described in more detail in section 6.7.3 below.

6.7.2 Textual View

The textual view is more useful in some cases than the tree view because (1) it can better display long strings, (2) whole structures are easily accessed without as much visual clutter, and (3) arbitrary sections of the view can be copied and pasted.

The layout textual view is very similar to the tree view, except that per-value expansion and collapse is not available. Instead, a hierarchy of values is displayed up to some selected depth of expansion for the entire set of values.

The Show More and Show Less buttons act like the Expand More and Expand Less popup menu items on the tree view surface, and are enabled and disabled in the same circumstances.

★ Using Expand More on the textual variable view, like use of Expand More on the tree view, may result in significant delays on the user interface if expanding large structures to a depth of more than 3 or 4 levels. Caution is advisable in cases where endless traversal of bushy structures is possible.

6.7.3 Tracking Individual Values

★ It is possible to zoom variable values out to seperate windows by right-clicking on tree style variable views, either in the main debugger window or in any other zoomed-out tree style view. The resulting window presents the value using a tree, textual, or combination display, according to the configured defaults. The value is tracked during execution and will change on the display according to either its symbolic path, a direct object reference to the value (only for mutable values), or according to a combination of object reference to the parent value and symbolic name for the parent slot.

The display style is controlled with the `debug.default-var-view-style` preference and can be overridden from the Default Zoom Style item in the popup menu. The selected display styles are the same as those described above: tree, textual, or a combination view containing both types of views, as is found in the main debugger window.

Before using this feature, it's a good idea to understand the subtleties of tracking values during execution. You will likely want to change how this is done during different debug tasks. The supported methods are:

- **By Symbolic Path** - The debugger looks at the symbolic path off `locals` or `globals` to the selected data value and tries to reevaluate that path whenever the value may have changed. For example, if you define a dictionary variable called `testdict` in a function and set a value `testdict[1] = ’test’`, the zoomed out view for `testdict[1]` would show any value for that slot of `testdict`, even if you delete testdict and recreate it. In other words, value tracking is independent of the life of any object instances in the data path.
- **By Direct Object Reference** - The debugger uses the object reference to the selected value to track it. If you use this mode with `testdict` as a whole, it would track the contents of that dictionary as long as it exists. If you were to reassign the variable `testdict` to another value, your zoomed out display would still show the contents of the original dictionary instance, rather than the new value of the variable `testdict`. In other words, the symbolic path to the value is completely disregarded and only instance identity is used to track the value. Because it's meaningless to track immutable types this way, this option is disabled or enabled according to the values you select to zoom out into a seperate window.
- **By Parent Reference and Slot** - The debugger uses the object reference to the parent of

the selected data slot and uses a symbolic representation of the slot within the parent where the value is to determined where to look for any value updates. This means that reassignment of the variable that points to the parent does not alter what is displayed in the zoomed-out view; only reassignment of the selected slot changes what is displayed by the debugger. Note that at the top level of `locals` or `globals`, this is the same as tracking the value by symbolic path, because the parent reference is to the `locals` or `globals` dictionary itself.

For any of these, if the value cannot be evaluated because it does not exist at any given point in time, the debugger displays `<undefined>`. This happens when the last object reference to a reference-tracked value is discarded, or if a selected symbolic path is undefined or unevaluable.

The type of tracking that is used is controlled by the `debug.default-var-track-style` preference and can be overridden from the Default Zoom Tracking item in the popup menu, or by selecting one of the specific Zoom and Track popup menu items (Zoom to Window simply uses the configured default).

6.7.4 Filtering Data Display

There are a number of ways in which variable display can be configured:

- Wing lets you prune the variable display area by omitting all values by type, and variables or dictionary keys by name.

  Currently, this can be done only by setting the two preferences, `debug.omit-types` and `debug.omit-names`. These are described in section 6.21.

- Wing provides control over size thresholds above which values are considered too large to move from the debug process into the variable display area. Values found to be too large are annotated as `huge` in the variable display area and cannot be expanded further. The data size thresholds are controlled with preferences `debug.huge-list-threshold` and `debug.huge-string-threshold`, described in section 6.21 and in section 6.7.5.

- By default Wing will display small items on a single line in the variable display areas, even if they are complex types like lists and maps. The size threshold used for this is controlled with preference `debug.line-threshold`. If you want all values to be shown uniformly, this preference should be set to `0`.

6.7.5 Problems Handling Debug Data

The Wing debugger tries to handle debug data as gently as possible to avoid entering into lengthy computations or triggering errors in the debug process while it is packaging debug data for transfer. Even so, not all debug data can be shown on the display. This section describes each of the reasons why this may happen:

- **Wing may time out handling a value** - Large data values may hang up the debug server process during packaging. Wing tries to avoid this by carefully probing an object's size before packing it up. In some cases, this does not work and Wing will wait for the data for the duration set by the `debug.network-timeout` preference and then will display the variable value as `<network timeout during evaluate>`.
- **Wing may encounter values too large to handle** - Wing will not package and transfer large sequences, arrays or strings exceed the size limits set by preferences `debug.huge-list-threshold` and `debug.huge-string-threshold` (described in section 6.21). On the debugger display, oversized sequences and arrays are annotated as `huge` and `<truncated>` is prepended to large truncated strings.

  To avoid this, increase the value of the threshold preferences, but be prepared for longer data transfer times. Note that setting these values too high will cause the debugger to time out if the `debug.network-timeout` value isn't also increased.

  An alternative for viewing large data values is to use the Expression Evaluator (described in section 6.8) or Interactive Debug Probe (described in section 6.9) to view sub-parts of the data rather than tranferring the whole top-level portion of the value.
- **Wing may encounter errors during data handling** - Because Wing makes assignments and comparisons during packaging of debug data, and because it converts debug data into string form, it may execute special methods such as `__cmp__` and `__str__` in your code. If this code has bugs in it, the debugger may reveal those bugs at times when you would otherwise not see them.

  The rare worst case scenario is crashing of the debug process if flawed C or C++ extension module code is invoked. In this case, the debug session is ended.

  Much more usual, but still rare, are cases where Wing encounters an unexpected Python exception while handling a debug data value. When this happens, Wing displays the value as `<error handling value>`. These errors are not reported as normal program errors in the Error List dialog. However, setting the preferences `main.print-wing-debug-output` and `debug.-verbose-debugging` to `true` will cause the exception that is being raised to appear in the window from which Wing was launched from the command line.

Wing remembers errors it encounters on debug values and stores these in the project file. These values will not be refetched during subsequent debugging, even if Wing is quit and restarted.

To override this behavior for an individual value, use the `Force Reload` item in the right-click popup menu of a tree view variable area.

To clear the list of all errors previously encountered so that all values are reloaded after subsequent stepping or restart of the debug session, use the `Clear Stored Value Errors` item in the `Run` menu or tree display right-click popup menu. This operates only on the list of errors known for the current debug file, if a debug session is active, or for the main debug file, if any, when no debug process is running.

6.8 Evaluating Expressions

Wing includes an interface for evaluating keyboard-entered expressions and viewing the result of the evaluation within a data display tree. Expressions are evaluated in the context of the current debug stack frame, so this feature is available only when a debug session is active and the debug program has been paused or has stopped at a breakpoint or exception. This also means that the value of the same typed expression may change as you move up and down the call stack in the main debugger window.

The Expression Evaluator item in the Windows menu will display the expression evaluator window. Expressions are typed into the panel at the top of the window and the Evaluate button is used to display the result of the evaluation below. Only expressions that evaluate to a value may be entered. Other statements, like variable assignments, import statements, and language constructs are rejected with an error. These may only be executed using the Interactive Debug Probe described in the next section.

A history list of expressions previously evaluated is also available via popup menu. Expressions selected from this menu are copied into the entry area and optionally evaluated immediately upon selection from the menu (only when the Evaluate Immediately checkbox is checked).

If an exception occurs during expression evaluation, this is reported with the Error List dialog and, if possible, the source location of the error is shown.

You may select from one of three display options when evaluating expressions: A single tree view, a textual view, or a tree/text combo view. These act exactly the same as those accessed from the main debugger window, as described in section 6.7. Once you have changed view modes, you must push the Evaluate button again before the view is filled with data.

In the rare cases where evaluating an expression results in changing the value of local or global variables, your debug program will continue in that changed context. Whenever a value is changed as a result of expression evaluation, the updated value will be propagated into any visible debugger variable display areas because Wing IDE refetches all displayed data values after the evaluation of each expression. However, since you may not notice these changes, caution is then required to avoid creating undesired side-effects in the running debug program. Otherwise, your program may not act as it would have during normal, uninterrupted execution.

Note that in this version of Wing, breakpoints are never reached as a result of expression evaluation, and any exceptions are reported only after the fact. This means that activity in the expression evaluation window has no effect on the debug run position or stack, even though an exception location in source code may in some cases by displayed.

6.9 Interactive Debug Probe

An interactive probe that acts like the Python shell is available for evaluating and executing arbitrary Python code in the context of a debug program. Like the expression evaluator, this acts on the current debug stack frame, and thus is available only when there is an active debug session and the debug program is paused.

This tool is displayed by selecting the Interactive Debug Probe menu item in the Windows menu. It acts much like the Python interpreter when it is invoked from the command line. You may use most of Wing's source editor commands and key bindings within the interactive debug probe window, and can use the up/down arrow keys to traverse a history of recently typed commands.

If commands you type change any local, instance, or global data values, cause modules to be loaded or unloaded, set environment variables, or otherwise alter the run environment, your debug program will continue within that altered state. All visible variable display views are also updated after each line entered in the interactive shell in order to reflect any changes caused by your commands. But since you may not notice these changes, caution is required to avoid creating undesired side-effects in the running debug program. Otherwise, your program may not act as it would have during normal, uninterrupted execution.

One limitation is that private instance variables prefixed by double underscore (such as `self.__my_var`) cannot be directly inspected or altered within the interactive shell. Python will report that the attribute is not defined because it internally prefixes the class name where the private variable is found. These can easily be viewed using the main debugger window or expression evaluator, or you can look at `self.__dict__` as a whole and then use the fully qualified `self._classname__my_var` to access or alter the value.

Note that in this version of Wing, breakpoints are never reached as a result of entries typed into the interactive shell, and any exceptions are reported only after the fact with a textual traceback. This means that activity in the interactive shell window has no effect on the debug run position or stack, even though an exception location in source code may in some cases be displayed.

Preference `debug.raise-from-interactive` can be used to determine whether source code windows will be raised when exceptions occur here. See section 6.21 for details.

6.10 Interactive Python Shell

Another shell tool is provided for execution of commands and evaluation of expressions outside of your debug program. This is the Interactive Python Shell, which may be accessed from the Windows menu.

Since this shell runs a seperate Python process that is independent of your debug process, it is always enabled and functions without regard to the state of any running debug process. As such, it acts very similarly to a regular command line Python shell.

The Interactive Python Shell always runs the same version of Python as is used for your debug process. This is described in more detail in section 6.2

To clear the state of the Python shell, press the New Session button. This will kill the external python process and restart it, thus clearing and resetting the state of the shell.

Note that in this version of Wing, breakpoints are never reached as a result of entries typed into the interactive shell, and any exceptions are reported only after the fact with a textual trace back. Preference `debug.raise-from-interactive` can be used to determine whether source code windows will be raised when exceptions occur here. See section 6.21 for details.

6.11 Managing Exceptions

During debug program execution, the debugger can be asked to stop on exceptions in the same way it would stop at breakpoints or in response to a Pause command.

A number of options for controlling when the debugger will stop on an exception are available from the Run menu's Exception Mode item:

- **Always Stop** - The debugger will stop at every single exception that is raised. Note that in some code this will be very often, since exceptions may be used internally to handle normal, acceptible runtime conditions.
- **Never Stop** - In this case, the debugger will never stop on any exceptions. Instead, the exception backtrace is printed to `stderr` and the program continues to execute, or exits, according to action taken by the Python interpreter. This behavior is the same as if the program were running outside of the debugger. You will, however, be given a post-mortem error and traceback by Wing to distinguish clearly from cases where a program has exited normally.
- **Stop on Unhandled** - This is the default. The debugger will only stop on exceptions for which no exception catching block is found. This check cannot take into account exceptions that are handled within C or C++ extension module code, so Wing may stop on more exceptions than are seen outside of the debugger. Use the "Don't show me this exception location again" option in the Error List dialog that appears to filter out these exceptions as they occur.

Whenever stopping on an exception, Wing presents an Error List window containing the exception information and backtrace, and stops the program at the point of exception (so that the stack frame and variables for that point are displayed in the debugger window).

Once stopped at an exception, execution can be continued in the same way as at any other time, using the run menu or tool bar, or by selecting Close & Continue in the Error List window.

6.11.1 Filtering Exceptions

The Error List window can be used to indicate to the debugger that the current exception should be omitted from those at which the debugger will stop in the future. To do this, check the checkbox labeled "Don't show me this exception location again" and push the "Close" or "Close & Continue" button.

This causes the debugger to ignore all exceptions on that line, regardless of type.

The list of ignored exceptions may be cleared to blank subsequently with the Clear Ignored Exceptions item in the Run menu.

6.12 Attaching to and Detaching from Debug Processes

Debug processes normally contact Wing IDE automatically during startup. However, Wing IDE can also attach to debug processes that are already running but not yet in contact with the IDE. There are two cases where this is useful:

- **(1)** When an externally launched process (one that uses `wingdbstub.py`, as described in section 6.13) cannot reach the IDE at the configured host and port during initial startup, for example because the IDE is not yet running or was not configured to accept debug connections.
- **(2)** When a process attached to the IDE is disconnected using Detach from Process in the Run menu or the detach icon in the toolbar.

In either case, the IDE can manage any number of detached processes, allowing you to attach to any one process at a time.

6.12.1 Access Control

Wing will not allow attach/detach functionality unless it has available to it a password that can be used to control access. This is very important because an unsecured debug server provides the client (Wing IDE) full control of the host machine via the Interactive Debug Probe tool. Any Python command can be executed in this way, including programs that compromise the security of your machine and network.

Because Wing sets up an access control password during installation, attach and detach will work out of the box as long as your debug processes are launched from Wing IDE, by you from the command line, or in the context of some service or program that is running under your user name on a machine that has access to your `~/.wingide` directory (on Linux) or `WINGHOME\profiles\[username]` (on Windows).

If you plan to debug remotely, please refer to the instructions in section 6.13.3 and 6.13.5. Setting up your remote debugging to work properly with encryption will also fullfill the requirements of the attach/detach facility. Note, however, that you can use encryption type `none` to enable attach/detach but disable channel encryption.

6.12.2 Detaching

The most common way of generating a process to which to attach is to first detach from a debug process. This is done by selecting the Detach from Process item in the Run menu or the detach icon in the toolbar.

Whenever a process is detached, it continues running as if outside of the debugger, without stopping at any breakpoints or exceptions. Even if a process is paused within the debugger at time of detaching from the IDE, the process will start running actively immediately after the IDE disconnects.

6.12.3 Attaching

The Attach to Process or attach toolbar icon are available whenever no other debug process is attached to the IDE. This brings up a dialog box that includes a list of available processes to attach to. The list is built from hard-wired host/port pairs given with the `debug.attach-defaults` preference (`('127.0.0.1', 50015)` by default), combined with known processes that were previously attached to Wing IDE.

Wing updates the list of available processes as debug sessions are killed from the IDE, as they are seen to exit from the outside while attached to Wing, or when the process cannot be contacted by Wing.

To attach to a process, select it from the list and push the Attach button. You may also type in a host/port value manually if your choice is not on the list (see the next subsection below).

Once you are attached to a process, it continues running until it reaches a breakpoint, unhandled exception, or you push the Pause button or use the Pause item in the Run menu.

6.12.4 Identifying Foreign Processes

If Wing is not running or not listening for connections because Passive Listen has been disabled, then additional processes may be available that are not listed in the Attach dialog.

This is usually the case when debugging externally launched code (that uses `wingdbstub.py` as described in section 6.13). You may use the `kAttachPort` constant in `wingdbstub.py` to set the port on which such processes will listen for attach requests. It is important to set unique values for this port for each concurrent, externally-launched process. If the set port is in use, a random port number will be used instead and it may be difficult to determine this number if the process cannot initially contact Wing IDE to register.

In any case, any unregistered debug process can be reached from Wing IDE by typing the host/port into the Attach dialog text areas. If you find yourself typing a host/port value often, it is best to add that value to the `debug.attach-defaults` preference.

Note that improperly configuring `wingdbstub.py` can also result in failure to reach the IDE during startup, since the configured `kWingHostPort` must match your Wing IDE preferences. See section 6.13 to work through your configuration before resigning to attaching manually.

6.12.5 Limitations

Currently, Wing supports attaching only to a single debug process at a time. Whenever you detach from a process, it begins free-running and will not stop at any breakpoints or non-fatal exceptions. This limits what can be done with detach/attach from a single copy of Wing. If you wish to actively debug two processes at once, simultaneously controlling stepping, breakpoint activation, and execution (as in a client/server network program), it is best to run two copies of Wing at once.

6.13 Debugging Externally Initiated Processes

This section describes how to start debugging from a process that is not launched by Wing. Examples of debug code that is launched externally include CGI scripts running under a web server, or embedded Python scripts running inside a larger application.

6.13.1 Importing the Debugger Stub

The following step-by-step instructions can be used to start debugging in externally launched code that is running on the same machine as Wing IDE:

1. Copy `wingdbstub.py` from the Wing IDE installation directory into the same directory as your debug program.
2. At the point where you want debugging to begin, insert the following source code:

   ```
   import wingdbstub
   ```

3. Make sure the Wing IDE preference `debug.passive-listen` is set to `true` and (re)launch the IDE, or use the Network Mode section of the Run menu to turn on passive listen.
4. Set any required breakpoints in your Python source code.
5. If you are using Windows, set an environment variable called WINGHOME that points to the location at which you installed Wing (usually `C:\Program Files\Wing IDE\`). You can also edit these values in wingdbstub.py if setting an env is inconvenient.
6. Initiate the debug program from outside Wing IDE, for example with a CGI invocation from your web browser, if the program is a CGI script. Make sure that you are running the Python interpreter without the `-O` option. The debugger cannot determine line numbers and thus cannot do anything meaningful when optimization is turned on.
7. The debugger should stop at the first breakpoint or exception found. If no breakpoint or exception is reached, the program will run to completion, or you can use the Pause command in the Run menu.

★ When an external process attaches to Wing IDE, the Kill item in the Run menu and toolbar icon are disabled, because Wing recognizes that it has not itself launched the debug process. To enable Kill in these cases also, set the `debug.enable-kill-external` preference to `true`.

If you have problems making this work, try setting `kSilent=0` variable in `wingdbstub.py` and launch the Python code from the command line where you can see its error output. Even though CGIs and similar code may not work properly this way, doing so can help to shed light on why a debug connection is not being initiated properly.

6.13.2 Server-side configuration

In some cases you may also need to alter other preset configuration values at the start of `wingdbstub.py`. These values completely replace any values set in Wing's project-wide or per-file debug properties, which are relevant only when the debug program is launched from within Wing. The following options are available:

- The debugger can be disabled entirely with `kWingDebugDisabled=1` This is equivalent to setting the `WINGDB_DISABLED` environment variable before launching the debug program.
- Set `kWingHostPort` to specify the network location of Wing IDE, so the debugger can connect to it when it starts. This is equivalent to setting the `WINGDB_HOSTPORT` environment variable before launching the debug program. The default value is `localhost:50005`. See section 6.13.3 below for details if you need to change this value.
- You can control whether or not the debugger's internal error messages are displayed by setting `kSilent`. This is equivalent to setting the `WINGDB_SILENT` environment variable before launching the debug program. When set to `1`, debugger internal messages are not printed to `stderr`, so they will not appear (for example, in the web server log file when running CGI scripts). However, to track down problems in launching the debugger in the first place, you may need to temporarily set this value to `0` to see the debugger's internal error messages (which can be quite verbose).
- Set `kEmbedded` to `1` when debugging embedded scripts. In this case, the debug connection will be maintained across script invocations instead of closing the debug connection when the script finishes. When this is set to `1`, you must call `wingdbstub.debugger.ProgramQuit()` before your program exits in order to cleanly close the debug connection to the IDE. This is equivalent to setting the environment variable `WINGDB_EMBEDDED`.
- Set `kAttachPort` to define the default port at which the debug process will listen for requests to attach. This is equivalent to setting the `WINGDB_ATTACHPORT` environment variable before launching the debug program. This value is used when the

debug process is running without being in contact with Wing IDE, as might happen if it initially fails to connect to the above-defined host and port, or if the IDE detaches from the process for a period of time. This is described in more detail in section 6.12 below.

- Optionally, set the location of the Wing IDE distribution's home directory. This is set up during installation, but may need to be altered if you are running Wing from source.

Setting any of the above-described environment variable equivalents will override the value given in the `wingdbstub.py` file.

★ Whenever the debugger cannot contact Wing IDE (for example, if the IDE is not running or is listening on a different port), the debug program will be run without any debugger. This is useful since debug-enabled CGIs and other programs should work normally when Wing is not present. In this case, you can attach to the process using Attach to Process from the Run menu, as described in section 6.12.

6.13.3 Remote Debugging

The description above covers only the case where you are running Wing and the debug code on the same machine. You can also ask the debugger to connect remotely over the network. This section describes how this is done with two machines of the same type. More information on working with machines of different type (e.g. Linux and Windows) can be found in section 6.17.

In order to do this, you need to check the following settings:

1. First set up Wing IDE to successfully accept connections from another process within the same machine, as described in section 6.13.1 above. You can use any Python script for testing this until you have values that work.
2. Set the `debug.network-server` preference to the name or IP address of the network interface on which the debugger listens for connections. This may be `None` or `""` to indicate that the debugger should listen on all the valid network interfaces on the host.

3. Set the `passive-hosts` preference list to include the host on which the debug process will be run.

4. Set the host and port to match the values set in Wing IDE with preferences `debug.network-server` and `debug.network-port`.

5. Next copy the debug server code out of your primary Wing installation over to the machine on which you wish to run your debug program. You will need the at least the files found in `bin/#.#.#/src/debug/server` and `bin/#.#.#/opensource/schannel` where `#.#.#` is your Python interpreter's version (for example `1.5.2`). If you're using a version of Python not found in the primary Wing installation, you can use the nearest lower release as long as the `major.minor` versions match. For example, the `2.1.0` binaries will work with `2.1.1`.

   Unless you are low on disk space, the easiest way to do this is to transfer the entire Wing IDE binary installation by copying over the contents of the `bin` directory inside your Wing installation.

   On Linux you might use this command to pack up the necessary files:

   ```
   cd WINGHOME
   tar cf dbgsvr.tar bin
   ```

   Move those files to the machine you want to debug on and unpack them into a directory to which your debug process will have access. This directory acts as `WINGHOME` on the debug server.

   Once done, you should have at least the following directories under `WINGHOME` on the machine where you wish to run your debug process, each filled with the `*.pyc` and `*.pyd` (Windows) or `*.so` (Linux) files that are in your primary Wing installation:

   ```
   bin/1.5.2/src/debug/server
   bin/1.5.2/opensource/schannel
   bin/2.0.0/src/debug/server
   bin/2.0.0/opensource/schannel
   bin/2.1.0/src/debug/server
   bin/2.1.0/opensource/schannel
   ```

   If you're only using one version of Python, you can omit the directories for the other versions that you are not using. If you're using a newer release of the same `major.minor` release of Python, for example `2.1.1`, you should copy out the newest available version in that `major.minor` series, for example `2.1.0`. Nothing outside of the the above directories is used, so may also be removed or omitted from your original copy.

6. Next, transfer copies of all your debug code so that the source files are available on the host where Wing IDE will be running and at least the `*.pyc` files are available on the debug server. You must also place a copy of `wingdbstub.py` with these files and import it as described in section 6.13. You will need to set at least the value of `kWingHostPort` and `kWingHome` inside of `wingdbstub.py` in order to tell the debugger where it should connect once your debug program has been started and where it can find the debug server code that you've transferred over from your primary Wing installation.

   Note that during debugging, the client and server copies of the files must match or the debugger will either fail to stop at breakpoints or stop at the wrong place, and stepping through code may not work properly. Since there is no mechanism in this version of Wing for transferring code, you need to use NFS, Samba, FTP or some other file transfer mechanism to keep the remote files up to date as you edit them in Wing.

7. If the real full path to the location of your files do not match on the client and server, use the `debug.location-map` preference to define one or more mappings from remote file location to local file location. This is described in more detail in the next section.

Then restart Wing, set a test breakpoint in the debug file, and try running the debug process. If you have problems, set `kSilent=0` in `wingdbstub.py` and check whether the debug stub reports that it could not connect to Wing IDE.

6.13.4 Defining file location maps

Wing currently requires access to debug files on disk both on the host where Wing IDE is running and where the debug process is running. Usually, this is facilitated by setting up file sharing between the two machines, using NFS, Samba, or other file sharing mechanism. Manual or semi-automatic FTP transfers will also work.

In cases where the full path to your source is not the same on both machines, you need to set up a mapping that tells Wing where it can find debug files. Note that making symbolic links to ’fake’ file locations on the client or server does not work because file locations are resolved to actual full path location by the debug server.

The debug file mapping is defined with the `debug.location-map` preference, which is a dictionary of remote host ip address (the host where the debug process is running) and a list of mapping entries.

In addition to standard dotted-quad IP addresses, you may specify a special ip address entry of `"*"` to define a default mapping for all hosts that are not otherwise specified in the location map.

The mapping entries themselves are arrays of tuples where each tuple is a `(remote_prefix, local_prefix)` pair. The remote file name will be a full path on the debug server’s file system. The local file name will be a URL, currently always starting with `file:`. Note that the local file URL should not contain backslashes (\) even if the local host is a Windows machine.

Examples

The default value for `debug.location-map` is `{’127.0.0.1’:None}` which indicates that all files at top level on the local host should be referred to with `file:` URLs. This is equivalent to the more verbose `{’127.0.0.1’:[(’/’,’file:’)]}`. It converts full paths on the debug server to the client-side URLs without altering any part of the full path.

Here is an example setting for `debug.location-map` that would be used if running Wing on `desktop1` and debugging some code on `server1` with IP address 192.168.1.1:

```
debug.location-map={ \
  ’127.0.0.1’:None, \
  ’192.168.1.1’:[(’/home/apache/cgi’, ’file:/svr1/home/apache/cgi’)] \
}
```

In this example, the same files are located in `/home/apache/cgi` on `server1` and in `/server1/home/apache/cgi` on `desktop1`, because the entire file system on `server1` is being NFS mounted on `desktop1` under `/svr1`.

Note that the trailing \ is required for line continuation by the preferences file format.

If you are debugging between Windows and Linux, some care is needed in specifying the conversion paths because of the different path name conventions on each platform. This entry would be used when running Wing IDE on a Linux host and the debug process on a Windows host with ip address 192.168.1.1:

```
debug.location-map={ \
  '127.0.0.1':None, \
  '192.168.1.1':[('c:\\src\\ide', 'file:/home/myuser/src/ide')], \
}
```

Note the double backslashes in the remote path and the use of forward slashes in the local URL specifier.

If running Wing IDE on a Windows host and the debug process on a Linux host with IP address 192.168.10, the following might be used instead for the same file locations:

```
debug.location-map={
  '127.0.0.1':None,
  '192.168.1.1':[('/home/myuser/src/ide', 'file:c:/src/ide')],
}
```

In the case where the Linux user `myuser` home directory is mounted via Samba to a Windows machine as the `e:` drive, the following similar configuration would be used (only the drive letter differs from the above):

```
debug.location-map={
  '127.0.0.1':None,
  '192.168.1.1':[('/home/myuser/src/ide', 'file:e:/src/ide')],
}
```

If you do not have NFS, Samba, or other file sharing, FTP or an FTP mirroring tool can be used to manually transfer files. Many FTP clients for Windows provide a mirroring capability.

6.13.5 Encrypting the Debug Channel

If you plan to debug over an unsecured network, you may wish to enable Wing's channel encrypter. Currently, the type of encryption used for this is weak, but it is enough to defeat casual packet sniffing and other abuses. Please don't use it for security-critical applications!

To turn on encryption, you need to create a file called `.wingdebugpw` which contains a single line of text that is used as the password at both ends of the debug channel. The line should be in the form `[encrypt]:[password]` where `[encrypt]` is one of `none` or `rotor`, and `[password]` is your password. If you specify only a password and no encryption type, `rotor` is used by default. The value `none` is used when a password is specified to control debug process attach/detach, as described further in section 6.12 later.

Place a copy of this file into the `.wingide` directory in your home directory (on Linux) or `WINGHOME\profiles\[username]` (on Windows). Whenever this file is present, Wing will enable encryption using the password in the file. You may need to restart Wing after placing, altering, or removing this file.

You will also need to place another copy of this file in the same directory as the remote debug program or within the `.wingide` directory of the user under which the debug program will run (or `WINGHOME\profiles\[username]` on Windows).

It is important that both ends of the channel match with respect to whether or not this file is present, what type of encryption is specified, and the password that is given. Passwords are case sensitive. If only one of the files is present or if they do not match, then the debug session will fail to initiate properly, possibly without clear notice of error.

The password is currently stored unencrypted, so it is important to make sure that the file is readable only by trusted users. Usually the best approach is to change the file so it is readable only by a single user (for example with `chmod 400` on Linux), and then change ownership of the file to match the user under which Wing or the debug program will run (for example, with `chown` on Linux).

Encryption will also be turned on for Wing-launched debug sessions as long as `~/.wingide/.wingdebugpw` (on Linux) or `WINGHOME\profiles\[username]\.wingdebugp` (on Windows) is present and specifies a non-`none` encryption type. Since an encrypted channel is noticeably slower than an unencrypted channel, you may wish to remove, rename, or alter this file unless you really need encryption.

6.13.6 Full Control via Debug API

A simple API can be used to control debugging more closely, once you have imported `wingdbstub.py` the first time, as was described in section 6.13.1 above.

This is useful in cases where you want to be able to start and stop debugging on the fly several times during a debug run, for example to avoid debug overhead except within a small sub-section of your code.

To use the API, take the following steps:

1. Configure and import `wingdbstub.py` as described in section 6.13.1.
2. Subsequently, use the instance variable `wingdbstub.debugger` to make any of the following calls:
    - **SuspendDebug()** - This will leave the connection to the debug client intact but disables the debugger so that connection overhead is avoided during subsequent execution.
    - **ResumeDebug()** - This will resume debugging using an existing connection to Wing.
    - **StopDebug()** - Stop debugging completely and disconnect from Wing IDE. The debug program continues executing in non-debug mode and must be restarted to resume debugging.
    - **ProgramQuit()** - This must be called before the debug program is exited if `kEmbedded` was set to `1` in `wingdbstub.py`. This makes sure the debug connection to the IDE is closed cleanly.

Here is a simple usage example:

```
import wingdbstub
a = 1 # This line is debugged
wingdbstub.debugger.SuspendDebug()
x = 1 # This is executed without debugging
wingdbstub.debugger.ResumeDebug()
y = 2 # This line is debugged again
```

Note that `SuspendDebug()` and `ResumeDebug()` can be called as many times as desired, and nested calls will be handled so that debugging is only resumed when the number of `ResumeDebug()` calls matches the number of `SuspendDebug()` calls.

6.14 Running Code Without Debug

It is possible to execute files outside of the debugger. This can be done with any Python code, Makefiles, and any other file that is marked as executable on disk. To do this, select the Execute Current File item in the Run menu to execute the currently at-front document, or use the project manager's popup menu item Execute Selected File.

Files executed in this way are run in a seperate process and any input or output occurs within the window from which Wing was launched (or is entirely hidden if Wing was launched from a desktop icon).

This is useful for triggering builds, executing utilities used in development, or even to launch a program that is normally launched outside of Wing and debugged using `wingdbstub.py`.

Note that files executed in this way are always invoked as if from their enclosing directory and without any parameters. There is currently no facility for specifying parameters or redirecting input/output.

6.15 Using the debugger and Python profiler together

★ Profiling under the debugger may yield inaccurate results since the debugger adds overhead that isn't always uniform across your code base.

The Python profiler makes some assumptions about how it is started that conflict with the way the Wing debugger works. Because the debugger can start debugging on the fly in the context of already-running code, it confuses the profiler into using the wrong top-level scope for its activities.

This means that a file like the following will not work:

```
import profile

def main():
  a = 1

profile.run("main()", "profile_tmp")
```

The profiler will raise an `AttributeError` on `main` because it is looking for it in the top-level file, which is not your code when running under Wing.

The way to solve this problem is to explicitely import and run the function you wish to profile, as follows:

```
import profile

def main():
  a = 1

profile.run("import mymodule; mymodule.main()", "profile_tmp")
```

6.16 Limitations

★ There are certain situations that the debugger cannot handle, because of the way the Python programming language works. If you are having problems getting the debugger to stop at breakpoints or to display source as you step through your code, please read through the following limitations:

- Running without saving will lead to incorrect display of breakpoints and run position because the IDE runs against the on-disk version of the source file.
- Filenames stored inside `*.pyc` files can be wrong if your move your `*.pyc` files around on disk. This may cause the IDE to fail to find source for these files, or to fail to stop at breakpoints set in source files whose `*.pyc` files are moved in this way. A similar problem may result from use of `compileall.py` and some other utilities that don't record a correct filename. It can also happen if running the same code using different paths to the same working directory, as is possible on Linux with symbolic links. However, the latter should only be a problem if you've removed the symbolic links subsequently. Hint: You can open `*.pyc` files in most text editors to inspect the stored file names. Or just remove these files so they can be regenerated with the correct file name information.
- You cannot run the debug program using the `-O` optimization option for the Python interpreter. This removes information about line numbers, making it impossible to stop at breakpoints or step through code.

- For code that spends much of its time in C/C++ without calling Python, for example as in a GUI main loop, the debugger may not reliably stop at breakpoints added during a run session. See section 6.18 for more information.

- You cannot use `pdb` in code that you are running within the Wing debugger. The two debuggers conflict because they attempt to use the same debugger hooks in the Python interpreter.

- If you override `__import__` in your code, you will break the debugger's ability to stop at breakpoints unless you call the original `__import__` as part of your code whenever a module is actually imported. To work around this, call `debug.server.netserver.NotifyImport()` (but only if you don't store and call the original `__import__`, which should really always be the case anyway!).

- When using the `wingdbstub`, if you set `sys.exitfunc` after debugging has been started, the IDE will time out in certain rare cases on a broken network connection after the debug program exits on an exception. This only happens for exceptions that look like they will be handled because a try/except block is present that might handle the exception, but where the exception is not in the end handled and the debug program exits without calling `StopDebug()`. Work-arounds include setting `sys.exitfunc` before importing `wingdbstub` or adding a top-level try/except clause that always calls `StopDebug()` before exiting the debug program.

6.17 Porting or Recompiling the Debug Server

If you have paid for a license, you have access to the source code for Wing IDE. The debug server is written in Python and standard C and can be recompiled and used on platforms other than Linux and Windows. When this is done, your debug process runs on a remote host of any type and Wing runs on either Linux or Windows. The remote host does not have to be one that is supported for Wing IDE. For example, the debugger is known to compile and run under Lynx OS, which is a proprietary real-time operating system.

Another reason to recompile the debug server is if you are running against an altered or experimental version of Python. For example, if the macros `Py_TRACE_REFS` or `WITH_CYCLE_GC` are altered from the defaults shipped with Python, the debug server running against that version of Python will need to be recompiled.

If you run into any problems with the instructions that follow, please send email to `bugs@archaeopteryx.com`. We'll help you get things working.

6.17.1 Getting started

How you port the debug server will vary slightly depending on whether or not your operating system supports shared libraries. If it does, you will compile and build a shared library that resides in the debug server source tree on your debug host. If it does not, you need to rebuild Python from source on your debug host with an added module for the debug server.

Both methods require the following two steps first:

- Follow the instructions in section 6.13.3 to set up your remote debug settings and to move the debugger binaries from your primary Wing installation over to the host where you plan to run your debug process.
- Obtain and install the Wing IDE source package, either from the CD if you purchased one, or from `http:www.wingide.com/support/downloads`. In order to gain access to the source files on our website, you must have your customer number and access password as emailed to you with your license files.

6.17.2 Building on hosts with shared libraries

If your debug host supports shared libraries, take these steps next. If it does not, skip to the next sub-section.

- Copy the following files out of your Wing IDE source base:

  ```
  src/debug/server/dbgtracermodule.c
  src/debug/server/dbgtracerhash.c
  src/debug/server/dbgtracerhash.h
  src/debug/server/Makefile
  src/debug/server/setup.pydist
  ```

  Place copies of these into the `WINGHOME/.bin/#.#.#` directory under on the host where you plan to run your debug program. `#.#.#` is the version of Python that you plan to use for debugging and `WINGHOME` is the directory where you copied the debug server binaries as described in section 6.13.3.

For example, if you created `/usr/lib/winghome` on your debug host and places `bin/1.5.2/src/debug/server` and `bin/1.5.2/opensource/schannel` there, you would place the source files in `bin/1.5.2/src/debug/server` (and not `src/debug/server`).

- Next, compile the source files within `bin/1.5.2/src/debug/server`. There are three ways to do this, either (a) with the Makefile, (b) with the distutils script (if you have distutils 1.0.1 or later on your machine), or (c) with manual compilation. There is no difference in the results of each of these; choose whichever is most convenient.

  If you use the Makefile, you need to change `PYPREFIX` and `PYINCLUDE` inside the Makefile to match your Python installation. Then type `make`.

  If you use distutils, you just need to type `python setup.pydist build_ext`.

  If you want to compile manually, the commands are as follows (but with the `-I` option altered to match the location of your Python's include directory):

  ```
  gcc -I/usr/local/include/python1.5 -c dbgtracermodule.c -
  o dbgtracermodule.o
  gcc -I/usr/local/include/python1.5 -c dbgtracerhash.c -
  o dbgtracerhash.o
  gcc -shared -o dbgtracermodule.so dbgtracermodule.o dbgtracerhash.o
  ```

If you plan to run with more than one version of Python, the above steps must be repeated for each of the versions, using each of `bin/1.5.2/src/debug/server`, `bin/2.0.0/src/debug/server`, and `bin/2.1.0/src/debug/server` and the build configuration for locating the appropriate header files.

6.17.3 Building on hosts with no shared libraries

When shared libraries are unavailable, you need to rebuild Python in order to statically link the Wing debug tracer into the executable. To do this:

- Copy the following files out of your Wing IDE source base:

  ```
  src/debug/server/dbgtracermodule.c
  src/debug/server/dbgtracerhash.c
  src/debug/server/dbgtracerhash.h
  ```

Place these into the `Modules` directory at the top-level of your Python source tree.

- Edit the `Modules/Setup.local` file and add the following line to it:

  ```
  dbgtracer dbgtracerhash.c dbgtracermodule.c
  ```

- Next go to the top-level of your Python installation. If you have not already done so in the past, type `./configure` to set up the build environment.
- Then type `make` to build your new Python executable.
- Follow this by `make install` if you're running Python from an installed location (such as `/usr/local/bin/python`).

Repeat this with each version of Python that you plan to run. Once completed, the command `import dbgtracer` should complete successfully in a Python script or when typed at the interactive Python prompt.

6.17.4 Using your setup

Once this is done, debugging happens in exactly the same way as any other remote debugging. See section `remotedebugging` for details.

Please drop us a note at `info@wingide.com` to let us know what OS and version you're using with your debug server!

6.18 Non-Python Mainloop Environments

Because of the way the Python interpreter supports debugging, the debug process may become unresponsive in environments where much time is spent running in non-Python code, such as C or C++. Whenever the Python interpreter is not called for long periods of time, messages from Wing IDE may be entirely ignored and the IDE may disconnect from the debug process as if it had crashed.

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

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

6.18.1 How it works

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

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

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

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

6.18.2 Writing a Debug Server Hook

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

Overview

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

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

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

Writing Your Own

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

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

Example

The following is a copy of the Python code that supports socket registration in Gtk. This file is also distributed as source with all copies of Wing, and can be foundin `src/debug/server` within `WINGHOME`.

```
#########################################################################
""" _gtkhooks.py -- Gtk socket management hooks for the Wing IDE debugger

Copyright (c) 1999-2001, Archaeopteryx Software, Inc.  All rights reserved.

Written by Stephan R.A. Deibel and John P. Ehresman

"""
#########################################################################

import _extensions

# The name of the module to watch for that indicates presence of this
# supported mainloop environment
kIndicatorModuleName = ’gtk’


#########################################################################
# GTK-specific support for managing the debug server sockets
#########################################################################
class _SocketHook(_extensions._SocketHook):
  """ Class for managing the debug server sockets:  This is used only
  when gtk is detected as being present in the debuggee’s code. """

  #-----------------------------------------------------------------------
  def __init__(self, err):
    """ Constructor """
    _extensions._SocketHook.__init__(self, err)
    self.__fGtkHandlers = {}
    self.__fGtkModule = None

  #-----------------------------------------------------------------------
  def _Setup(self, mod, s, cb_fct):
    """ Attempt to set up socket registration with the given module
    reference : This should be a reference to the indicator module
    for the supported environment.  The first socket is registered
    with given action callback via _RegisterSocket().  Returns the
    socket if succeeded or None if fails (for example, because the module is
    not yet fully loaded and we cannot yet use it to start registering
    sockets.  Note that the returned socket may be different than the
    socket passed in because some environments require a wrapper:  The
    returned socket is then used in place of the original in the
    debug server code. """

    # Check if module is fully loaded as far as the constructs we will need
    if mod.__name__ != 'gtk' or not hasattr(mod, 'GDK') \
       or not hasattr(mod, 'input_add') \
       or not hasattr(mod, 'input_remove') \
       or not hasattr(mod.GDK, 'INPUT_READ') \
       or not hasattr(mod.GDK, 'INPUT_EXCEPTION'):
      return None

    # Try to register the first socket
    self.__fGtkModule = mod
    new_sock = self._RegisterSocket(s, cb_fct)
    if new_sock == None:
      self.__fGtkModule = None
      return None

    # Success
    return new_sock

  #-----------------------------------------------------------------------
  def _RegisterSocket(self, s, cb_fct):
    """ Function to register a socket with a mainloop: Subsequently the given
    callback function is called whenever there is data to be read on the
    socket.  Returns the socket if succeeded; None if fails. As in _Setup(),
    the returned socket may differ from the one passed in, in which case
    the debug server will substitute the socket that is used in its code."""

    # Try to use given module to register the socket:  This may still fail
    # if module is in the process of being imported
    try:

      # Register with GTK
      cond = self.__fGtkModule.GDK.INPUT_READ | self.__fGtkModule.GDK.INPUT_EXCEPTION
      handler_id = self.__fGtkModule.input_add(s, cond, cb_fct)
      self.__fGtkHandlers[s] = handler_id

      # Done
      self.fErr.out("################## Socket registered with gtk: ", s)
      return s

    # Failed but will keep checking
    except:
      self.fErr.out("################## 'gtk' module not fully loaded")
      return None

  #-----------------------------------------------------------------------
```

```
def _UnregisterSocket(self, s):
  """ Function to unregister a socket with the supported environment.
  The socket passed in should be the one returned from _Setup() or
  _RegisterSocket(). """

  self.fErr.out("################ Deregistered socket with gtk: ", s)
  self.__fGtkModule.input_remove(self.__fGtkHandlers[s])
  del self.__fGtkHandlers[s]
```

Getting Help

If you are having difficulties writing your non-Python mainloop hooks, please contact our Technical Support group via our website at http://archaeopteryx.com/support. We will be happy to assist you, and welcome the contribution of any hooks you may write.

6.19 Hints for Debugging Web CGIs

Debugging CGI scripts can be annoying since any output from your program that is not understood by the web server will cause the server to write an error to its log files rather than displaying anything useful back to the browser.

Here are a few simple suggestions that may help you configure the debugger and verify that things are working correctly:

1. At the very start of your code, before importing `wingdbstub` or calling the debug API, insert the following temporary line of code:

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

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

2. Place a catch-all exception handler at the top level of your CGI code and print exception information to the browser. The following function is useful for inspecting the state of the CGI environment when an exception occurs:

   ```
   import sys
   import cgi
   import traceback
   ```

```
import string

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

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

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

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

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

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

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

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

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

7. Once you have the debugger working for one CGI script, you will have to set up the `wingdbstub` import in each and every other top-level CGI in the same way.

Because this can be somewhat tedious, and because the import needs to happen at the start of each file (in the `__main__` scope), it makes sense to develop your code so that all page loads for a site are through a single entry point CGI and page-specific behavior is obtained via dispatch within that CGI to other modules. With Python's flexible import and invocation features, this is relatively easy to do.

6.20 Using Wing with Zope

The easiest way to get started with Zope plus Wing is to install the combined Wing/Zope distribution provided on the CD in the `zope` directory or from our ftp site at `ftp://wingide.com/wingide/support/zope/`.

The `zope` directory on the CD also contains an HTML document called `zope-wing-howto.html` that describes how to use the combined distribution, and how to use Wing with an existing Zope installation. This document is also available on the web through our support library at `http://wingide.com/support/library`.

6.21 Preferences

The following preferences exist for controlling Wing IDE's debug facility:

- **debug.raise-window-on-break** - Controls when the window with the source file containing the current line is brought to front when the program being debugged reaches a breakpoint or is otherwise paused. One of `'raise-always'` to always raise the window, `'raise-never'` to never raise the window (except when explicitely opened from the debugger's stack view), or `'raise-new'` to raise the window only when stepping into a new file that isn't the same as the file in which the previous known run location was found. Default=`'raise-always'`
- **debug.raise-from-interactive** - Controls whether to raise windows showing exception location when working in the Interactive Debug Probe or Interactive Python Shell and a meaningful exception location is found. Set to one of `'raise-always'` or `'raise-never'`. Default=`'raise-always'`
- **debug.raise-from-evaluator** - Controls whether to raise windows showing exception location when working in the expression evaluator and a meaningful ex-

ception location is found. Set to one of `'raise-always'` or `'raise-never'`. Default=`'raise-always'`

- **debug.passive-listen** - Controls whether or not the debugger listens passively for connections from an externally launched program (`false` to disable; `true` to enable). This must be on when the debug program is not launched by Wing IDE (for example, as with a CGI script). This is the startup default and may be altered with the Network Mode section of the Run menu. Default=`false`

- **debug.passive-hosts** - Sets which hosts are allowed to connect to the debugger when it is listening passively for externally launched programs. This value is a tuple containing at least one host name, as a quote-delimited string. Default=`('127.0.0.1',)`

- **debug.attach-defaults** - A list of tuples containing (host, port). These are values that are automatically included in the Attach dialog box. Use it to avoid repeatedly typing manually entered host/port values. Default=`( ('127.0.0.1', '50015'), )`

- **debug.location-map** - Defines mappings between the debug server file locations and local file locations on disk. Each mapping key is the ip address of the remote location, or `"*"` to define a default that matches all otherwise unmatched hosts. The mapping values are arrays of tuples where each tuple is a `(remote_prefix, local_prefix)` pair. The remote file name will be a full path name on the debug server file system and the local file name will be a URL, currently always starting with `file:`. This should be used when files on the remote host are updated via ftp, NFS, Samba, or other method from master copies on the local host, but the full path file system locations on the local and remote hosts do not match. See section 6.13.3 for examples and details. Default= `{ '127.0.0.1':[('/','file:/'),]}`

- **debug.enable-kill-external** - Set this to `true` to allow Wing to terminate processes that it did not launch but that connected to its debugger from another locally running process. Whenever this is set to `false`, the Kill menu item and command are disabled when attached to a debug process launched outside of Wing. Default=`false`

- **debug.python-exec** - Set this to indicate the default Python executable used with the debug server. A `None` value uses `/usr/bin/env python` on Linux and the configured default on NT. Otherwise, give the full path of the Python executable, for example `/usr/local/bin/python` or `C:\dev\python`. This preference only affects programs that are launched from Wing and is overridden by any values specified in Wing's Project Properties or per-file Debug Properties. Default=`None`

- **debug.exception-mode** - Controls default behavior for stopping on exceptions in your debug program: `'never'` to never stop, `'always' to always stop, or \verb`'unhandled'! to stop only on unhandled exceptions. This is just the initial value that Wing will use; it can be changed after starting Wing using the Exception Mode item in the Run menu. Default=`'unhandled'`

- **debug.network-server** - Determines the network interface on which the debugger listens for connections. This can be a symbolic name, an IP address, or `None` to indicate that the debugger should listen on all valid network interfaces on the machine. Note that when a debug session is launched from within Wing IDE (with the Run button), it always connects from the loopback interface (127.0.0.1). Default=`None`

- **debug.network-port** - Determines the TCP/IP port on which the debug client will listen for the back-connection from the server. This needs to be unique for each developer using `wingdbstub` for debugging on a given host. The debug server needs to be told the value specified here (as described in sections 6.13.1 and 6.13.6). Note that this value is ignored if the debug program is launched from the application, in which case an available random port number is used instead. Default=`'50005'`

- **debug.network-timeout** - Controls the amount of time in seconds that the debug client will wait for the debug server to respond before it gives up. This protects Wing from freezing up if your program running within the debug server crashes (or if the server itself becomes unavailable). This can be a relatively low number unless you are debugging over a slow network or sending large data values (see the huge-list-threshold and huge-string-threshold preferences). Default=`5.0`

- **debug.stop-timeout** - The number of seconds the debugger will wait before stopping within its own code after a pause request is seen and no other Python code has been reached. Even when stopping within the debugger, the user is presented with the call stack to the host application's main loop only and step operations perform as if from there. Default=`3.0`

- **debug.use-xterm** - Controls default behavior for whether or not an extra terminal window is opened when a debug program is started: If `true` then all debug program input/output happens in a seperate new terminal window, if `false` then input/output happens in the window from which Wing was launched (if any; when Wing is launched from icon, the debug processes output may be invisible). On Windows systems, this value is currently always forced to `true`, and a dos shell is used. Important: On Linux systems where `xterm` runs with setuid, your environment variables are not propagated to the debug program unless explicitely listed in the `wingdb` script found in `WINGHOME`. The values of `LD_LIBRARY_PATH` and `PYTHONPATH` are propagated by the script that ships with Wing, but you may need to add any additional environment variables that your program depends on. Default=`false` (but always forced to `true` on Windows)

- **debug.persist-xterm** - Controls whether or not the debug terminal window, when it is enabled, will exit immediately after the debug run or only after enter or return are typed. Use `true` to allow inspection of debug output after exit of the debug program. This is only relevant when `debug.use-xterm` is set to `true`. Default=`true`
- **debug.omit-types** - Defines types for which values are never shown by the debugger. Default=`('function', 'builtin_function_or_method', 'class', 'instance method', 'type', 'module', 'ufunc')`
- **debug.omit-names** - Defines variable/key names for which values are never shown by the debugger. Default=`()`
- **debug.line-threshold** - Defines the character length threshold under which a value will always be shown on a single line, even if the value is a complex type like a list or map. Default=`55`
- **debug.huge-list-threshold** - Defines the length threshold over which a list, map, or other complex type will be considered too large to show in the normal debugger. If this is set too large, the debugger will time out (see network-timeout preference). Default=`2000`
- **debug.huge-string-threshold** - Defines the length over which a string is considered too large to fetch for display in the debugger. If this is set too large, the debugger will time out (see network-timeout preference). Default=`64000`
- **debug.use-members-attrib** - Set this to `true` in order to ask the debug server to use the `__members__` attribute on values that are otherwise opaque. This is useful in interpreting some values defined and set in C extension modules. However, there are sometimes bugs in these modules that cause crashing when this is used (for example, in pygtk-0.6.5). Default=`false`
- **debug.default-var-view-style** - This sets the default view style that is used when zooming variable data out into seperate windows. Choices are `'tree'` for a dynamic tree display, `'text'` for a textual display, and `'combo'` for a combination view like that found in the main debugger window. These values can be overridden from the right-click popup that appears over any tree formatted debugger variable display. Default=`'combo'`
- **debug.default-var-track-style** - This selects the default style of value tracking for variables that are zoomed out into seperate display windows. Choices are `'symbolic'`, `'parent-ref'`, and `'ref'`. These values can be overridden from the right-click popup that appears over any tree formatted debugger variable display. See section 6.7.3 for more information on these choices. Default=`'symbolic'`

- **debug.show-debug-data-warnings** - This controls whether or not Wing will display details about failure to evaluate a debug data value because of an error handling the value, network timeout, or oversized value error. When true, errors explaining each of these will be shown the first time they occur in each run of Wing; when false, they are never shown. Default=`true`
- **debug.run-marker-bg-color** - This is a tuple in the form (red, green, blue), with each value from `0` to `255`, that defines the color used behind the current run line during debugging. Default=`(255, 163, 163)`
- **debug.run-marker-fg-color** - This is a tuple in the form (red, green, blue), with each value from `0` to `255`, that defines the color used around the margin marker for the current run position during debugging. Default=`(255, 0, 0)`
- **debug.-verbose-debugging** - This is normally only used when developing the IDE itself, but can be set to `true` to obtain more detailed information about problems that come up with the Wing debugger itself (such as failure to start a debug session or unexpected termination of a debug session). Default=`false`

Note that currently Wing must be restarted before any altered values take effect.

Appendix A

Command Reference

This appendix enumerates the entire top-level Wing command set.

A.1 Overview

Any of the commands listed here may be used in custom menus, tools, or key equivalents that can be set up as described in chapter 2. In emacs emulation mode, commands may also be entered by name within the command entry area that is displayed with alt-x or esc-x.

Although many commands accept no parameters, some commands are defined to accept optional typed parameters. The type of a parameter can be any of those defined in Python's types module, or a file name string.

Some parameters defined have special meaning to Wing, according to the name of the parameter:

- **active_view** - This is either a given text or source document, or defaults to the active text or source document.
- **active_window** - This is either a given window reference, or defaults to the currently active window.

All other parameters on commands are considered to be user-entered values, such as the search and replace strings given to the `query-replace` command. Such parameters are collected from the user if they are missing when the command is invoked (as they would be when commands are used in menus, tools, or key equivalents).

Command parameters will also be used in future versions of Wing to support scripting the IDE. This reference will be expanded at that time to describe how commands are defined within Wing and how scripts can access command definition information.

Since commands act as the formal public API into the Wing source base, they will be maintained for compatibility in all future releases of Wing IDE. Note that more of the commands will take parameters in the future, but should continue to behave as they currently do when invoked without parameters.

A.2 Top Level

The following commands are defined by the `guimgr` subsystem in the class `guimanager.CTopLevelCommands`.

Command Name	**Description**	**Parameters**
quit	Quit Wing IDE, prompting to save any unsaved documents.	
about-application	Display the application about box.	
open	Prompt to open a document from disk. This may either prompt graphically or using the emacs mode interaction manager.	
open-gui	Prompt to open a document from disk. This always uses a graphical file open box.	
new-file	Create a new blank document.	
close	Close the current or given document.	*active_view*: The view reference
close-all	Close all the open document windows (but not non-document windows).	
save	Save the current or given document.	*active_view*: The view reference
save-as	Save the current or given document to another location.	*active_view*: The view reference

save-all	Save all unsaved altered documents to disk. This will prompt only for names of items that have not yet been assigned a name.	
print-view	Print the current or given view.	*active_view*: The view reference
switch-document	Switch to display one of the selected open documents	*document_name*: The name of document to open; either full path or last path component only
command-by-name	Prompt user to enter a command for execution by name	
delete-window	Delete current window, closing any views not found in any other window; only available in multi-view-per-window mode.	
search-manager	Display the graphical search manager window.	
show-font-size-dialog	Display the font/size selection dialog for altering the font and size used in a single text file or in all text files in the project.	
show-analysis-env	Display informational dialog that contains the current interpreter and python path information being used for source code analysis throughout the project.	
initiate-repeat-0	Start emacs-mode interaction for entering a repeat value for command or keystroke repetition; starts value entry with '0'	
initiate-repeat-1	Start emacs-mode interaction for entering a repeat value for command or keystroke repetition; starts value entry with '1'	

initiate-repeat-2	Start emacs-mode interaction for entering a repeat value for command or keystroke repetition; starts value entry with ’2’	
initiate-repeat-3	Start emacs-mode interaction for entering a repeat value for command or keystroke repetition; starts value entry with ’3’	
initiate-repeat-4	Start emacs-mode interaction for entering a repeat value for command or keystroke repetition; starts value entry with ’4’	
initiate-repeat-5	Start emacs-mode interaction for entering a repeat value for command or keystroke repetition; starts value entry with ’5’	
initiate-repeat-6	Start emacs-mode interaction for entering a repeat value for command or keystroke repetition; starts value entry with ’6’	
initiate-repeat-7	Start emacs-mode interaction for entering a repeat value for command or keystroke repetition; starts value entry with ’7’	
initiate-repeat-8	Start emacs-mode interaction for entering a repeat value for command or keystroke repetition; starts value entry with ’8’	
initiate-repeat-9	Start emacs-mode interaction for entering a repeat value for command or keystroke repetition; starts value entry with ’9’	

A.3 Project Manager

The following commands are defined in the `proj` subsystem in the class `managergui.CProjectCommands`.

Command Name	Description	Parameters
show-project-window	Bring the project manager window to front.	
new-project	Create a new project file; this causes the current open project file to be closed, and user is prompted to save the project if altered.	
open-project	Open a project file; the current project file is closed, and user is prompted to save the project if altered.	
close-project	Close the project file, prompting user to save if altered.	
save-project	Save the project file to disk.	
save-project-as	Save the project file to another location on disk; prompts user to enter the location.	
add-file-to-project	Prompt user to select a file to add to the open project.	
add-package-to-project	Prompt user to select a directory to add to the open project. All enclosed files of types configured with the `proj.package-file-types` and `proj.package-omit-types` preference will be presented to the user for optional individual selection.	
add-tree-to-project	Prompt user to select a directory to add to the open project. All enclosed files of types configured with the `proj.package-file-types` and `proj.package-omit-types` preference will be added in one operation.	
add-current-file-to-project	Add the file associated with the frontmost view to the project manager.	

remove-selection-from-project	Remove the currently selected file or package from the project.	
browse-project-modules	Switch to the source code browser, showing all modules and files in the currently open project.	
set-project-main-debug-file	Set the main debug file for open project. This is the file that is executed by default by the debugger.	
set-current-as-main-debug-file	Set the file associated with the current frontmost view as the main debug file for open project. This is the file that is executed by default by the debugger.	
clear-project-main-debug-file	Clear the main debug file for open project. The debugger will execute the frontmost window when this is cleared.	
open-selected-from-project	Open the currently selected item in the project manager window.	
browse-selected-from-project	Switch to the source code browser, showing the module view for the item currently selected in the project manager window.	
debug-selected-from-project	Launch a debug session for the item currently selected in the project manager window.	
execute-selected-from-project	Execute the item currently selected in the project manager window. Makefiles, Python code, and any executable file may be executed in this way outside of the debugger.	
choose-file-debug-args	Display the debug properties dialog box for the file currently selected in the project manager window.	
sort-project-by-directory	Group items in the project manager window by directory, sorting the top-level directory list alphabetically.	
sort-project-by-file-type	Group items in the project manager window by file type, soring the top-level mime type list alphabetically.	

use-normal-project	Switch the project disk storage format to a single file containing all project data.	
use-shared-project	Switch the project disk storage format to use two files, one containing user-specific data, and the other containing data that can be shared by all developers on a project.	
view-project-properties	Display the project-wide properties dialog.	

A.4 Source Code Editor

The following commands are defined in `edit` subsystem in the class `editor.CEditorCommands`.

Command Name	**Description**	**Parameters**
cut	Cut the current text selection to system-wide clipboard.	
copy	Copy the current text selection to system-wide clipboard.	
clear	Clear (delete) the current text selection.	
paste	Paste the current contents of the system-wide clipboard to replace the current text selection.	
undo	Undo the most recent command or typing action.	
redo	Redo the most recently undone action.	
kill-line	Remove text from the cursor to the end of the current line and place it into the kill buffer with any other contiguously removed lines. End-of-line is removed only if there is nothing between the cursor and the end of the line.	
yank-line	Paste the contents of the kill buffer into current position in the editor. This will paste the system-wide clipboard instead if the kill buffer is empty.	

beginning-of-line	Move the cursor to the beginning of the current line. Issuing this repeatedly will alternate between the very beginning of the line and the first non-blank character.	
beginning-of-line-extend	Move the cursor to the beginning of the current line and extend the current selection to that point.	
end-of-line	Move to the cursor to the end of the current line.	
end-of-line-extend	Move the cursor to the end of the current line and extend the current selection to that point.	
next-line	Move cursor to the next line.	
next-line-extend	Move cursor to the next line, extending the current selection to that point.	
previous-line	Move cursor to the previous line.	
previous-line-extend	Move cursor to the previous line, extending the current selection to that point.	
forward-char	Move the cursor forward one character.	
forward-char-extend	Move the cursor forward one character, extending the current selection to that point.	
backward-char	Move the cursor backward one character.	
backward-char-extend	Move the cursor backward one character, extending the current selection to that point.	
forward-word	Move the cursor forward one word.	
forward-word-extend	Move the cursor forward one word, extending the current selection to that point.	
backward-word	Move the cursor backward one word.	
backward-word-extend	Move the cursor backward one word, extending the current selection to that point.	
forward-page	Move the cursor forward one page.	

forward-page-extend	Move the cursor forward one page, extending the current selection to that point.	
backward-page	Move the cursor backward one page.	
backward-page-extend	Move the cursor backward one page, extending the current selection to that point.	
start-of-document	Move cursor to start of document.	
start-of-document-extend	Move cursor to start of document, extending the current selection to that point.	
end-of-document	Move cursor to end of document.	
end-of-document-extend	Move cursor to end of document, extending the current selection to that point.	
brace-match	Match brace at current cursor position, selecting all text between the two and highlighting the braces.	
set-mark-command	Set start of text marking for selection at current cursor position. Subsequently, all cursor move operations will automatically extend the text selection until stop-mark-command is issued.	
stop-mark-command	Stop text marking for selection at current cursor position, leaving the selection set as is. Subsequent cursor move operations will deselect the range and set selection to cursor position.	
forward-delete-char	Delete one character in front of the cursor.	
backward-delete-char	Delete one character behind the cursor.	
forward-delete-word	Delete one word in front of the cursor.	
backward-delete-word	Delete one word behind the cursor.	
select-all	Select the entire document.	

forward-tab	Place a forward tab at the current cursor position.	
backward-tab	Place a backward tab at the current cursor position.	
new-line	Insert a new line at the current cursor position.	
form-feed	Insert a form feed character at the cursor position.	
search-forward	Initiate interactive forward search from the cursor position.	
search-backward	Initiate interactive backward search from the cursor position.	
query-replace	Initiate interactive (prompted) query/replace from the cursor position.	
replace-string	Replace all occurrences of a string after current cursor position to the end of the file.	
zoom-in	Increase font sizes in display by one unit.	
zoom-out	Decrease font sizes in display by one unit.	
toggle-overtype	Toggle between overtype and insert mode.	
show-all-whitespace	Show all white space with visible whitespace characters, including spaces, tabs and end-of-line characters.	
hide-all-whitespace	Show white space without any visible whitespace characters.	
show-whitespace	Show white space with visible space and tab characters.	
hide-whitespace	Show white space without visible space and tab characters.	
show-eol	Show end of line characters with a visible character.	
hide-eol	Hide visible end of line characters.	
scroll-to-cursor	Scroll as necessary to make sure that the cursor is visible on screen.	
center-cursor	Center the cursor on the display.	

show-selection	Turn on display of the current text selection.	
hide-selection	Turn off display of the current text selection.	
save-buffer	Save the current text file to disk.	
kill-buffer	Close the current text file.	
cancel	Cancel current editor command.	
insert-file	Insert a file at given cursor position.	*filename*: The file to insert.
goto-line	Go to a selected line by number.	*lineno*: The line number.
start-kbd-macro	Start recording a keyboard and command macro.	
stop-kbd-macro	Stop recording a keyboard and command macro.	
execute-kbd-macro	Execute the most recently recorded keyboard macro.	
indent-region	Increase indentation of selected region by one level.	
outdent-region	Outdent indentation of selected region by one level.	
comment-out-region	Comment out the selected region. This operates on whole lines, extending the current selection if necessary.	
uncomment-out-region	Remove comments around lines in the selected region. This operates on whole lines, extending the current selection if necessary. Does nothing if commenting is not present.	
fill-paragraph	Rejustify the paragraph of text surrounding the current start of selection. This operates on whole lines, and is most useful in reformatting comments, long strings, or documentation.	

indent-to-match	Indent the current line or selection to match indentation of the preceding non-blank line, adding or subtracting indentation as needed for the current context.	
complete-autocompletion	Complete the current autocompletion, inserting text as appropriate at insertion point.	
show-popup-menu	Display the editor's popup menu.	
goto-selected-symbol-defn	Display the point of definition of the most recently clicked-upon symbol in the source.	
show-indent-manager	Display the indentation manager dialog for the current source window.	
convert-indents-to-spaces-only	Convert all indentation in the text file to use only spaces. Tabs are converted according to configured `edit.tab-size` and `edit.indent-size` preferences. Tab size is forced to 8 for all Python files unless the file previously contained only tabs in indents, in which case `edit.tab-size` is used in the conversion.	
convert-indents-to-tabs-only	Convert all indentation in a text file to use tabs only. Units of `edit.indent-size` spaces are converted into one tab each, and any remainder of space is converted into one tab. After conversion, the file is shown with tabs set to size `edit.tab-size` even for Python files (where tab size is otherwise forced to 8).	

convert-indents-to-mixed	Convert all indentation in a text file to use a mixture of tabs and spaces, using tab size as configured with preference `edit.tab-size` and indent size as configured with preference `edit.indent-size` (one tab is placed instead of each group of `edit.tab-size` spaces. Tab size is forced to 8 for Python files, since the Python interpreter expects this in mixed indentation.	
force-indent-style-to-spaces-only	Force future indentation in the text file to use spaces only in indentation, regardless of the current contents of the file. This is not available for Python files that don't already have inconsistent indentation.	
force-indent-style-to-tabs-only	Force future indentation in the text file to use tabs only in indentation, regardless of the current contents of the file. This is not available for Python files that don't already have inconsistent indentation.	
force-indent-style-to-mixed	Force future indentation in the text file to use mixed tab and space indentation, regardless of the current contents of the file. This is not available for Python files that don't already have inconsistent indentation.	
check-indent-consistency	Check current file for indentation consistency with respect to the use of tabs and spaces. If some areas are indented with tabs and others with spaces, then the user is prompted to convert to all spaces, convert to all tab/space indentation, or to leave the file as is.	

fold-toggle	Toggle the fold state of the first fold point found in the current selection or on the current line.	
fold-collapse-all-current	Collapse recursively the current fold point, folding up all children as well.	
fold-expand-all-current	Expand recursively the current fold point, ensuring that all children are visible as well.	
fold-collapse-all	Collapse the entire source file recursively.	
fold-expand-all	Expand the entire source file recursively.	
use-lexer-by-doctype	Determine syntax colorizing for the text file according to the probable mime type of the file, based on the file extension.	
use-lexer-none	Turn of syntax colorizing for the text file.	
use-lexer-python	Use the Python lexer to colorize the text file.	
use-lexer-cpp	Use the C++ lexer to colorize the text file.	
use-lexer-java	Use the Java lexer to colorize the text file.	
use-lexer-makefile	Use the Makefile lexer to colorize the text file.	
use-lexer-dos-batch	Use the DOS datch file lexer to colorize the text file.	
use-lexer-vb	Use the Visual Basic lexer to colorize the text file.	
use-lexer-html	Use the HTML lexer to colorize the text file.	
use-lexer-properties	Use the properties file lexer to colorize the text file.	
use-lexer-errlist	Use the error list lexer to colorize the text file.	
use-lexer-msidl	Use the MS IDL lexer to colorize the text file.	
use-lexer-sql	Use the SQL lexer to colorize the text file.	
use-lexer-xml	Use the XML lexer to colorize the text file.	
use-lexer-xcode	Use the xcode lexer to colorize the text file.	

use-lexer-latex	Use the LaTeX lexer to colorize the text file.	
use-lexer-lua	Use the Lua to colorize the text file.	
use-lexer-idl	Use the CORBA IDL lexer to colorize the text file.	
use-lexer-javascript	Use the Javascript lexer to colorize the text file.	
use-lexer-rc	Use the RC lexer to colorize the text file.	
use-lexer-plsql	Use the PLSQL lexer to colorize the text file.	
use-lexer-php	Use the PHP lexer to colorize the text file.	
use-lexer-perl	Use the Perl lexer to colorize the text file.	
use-lexer-diff	Use the diff/patch file lexer to colorize the text file.	
use-lexer-pascal	Use the Pascal lexer to colorize the text file.	
use-lexer-apache-conf	Use the Apache web server configuration lexer to colorize the text file.	
use-lexer-ave	Use the Ave lexer to colorize the text file.	
show-line-numbers	Show the line number display column on all source files. The width of the column is set with the `edit.lineno-column-width` preference.	
hide-line-numbers	Hide the line number display column on all source files.	

A.5 Debugger

The following commands are defined in `debug.client` subsystem in the class `cmdmanager.CDebuggerCommands`.

Command Name	**Description**	**Parameters**

show-debug-window	Bring the debugger window to front.	
show-error-list	Bring the runtime error window to front.	
enable-passive-listen	Enable passive listening on the debugger network port.	
disable-passive-listen	Disable passive listening on the debugger network port.	
exception-always-stop	Always stop on exceptions, even if they are handled by the debug program's code.	
debug-start or run	Start debug program execution, stopping on first line of code. This executes the main debug file as specified by the project manager, if it has been defined, or otherwise the current frontmost editor window.	
debug-file	Start debugging the current file (regardless of whether a main debug file has been defined in the project manager).	
execute-file	Execute the current file outside of the debugger. Makefiles, python code, and any executable file may be executed in this way.	
debug-continue	Continue the current debug session, or start a new one using the main debug file if one has been specified in the project manager, or otherwise the frontmost editor window. This will stop on the next breakpoint or exception (and not on the first line of code when debug is initiated).	
debug-attach	Display the attach dialog box, which allows attaching Wing IDE to debug processes that are already running.	

debug-detach	Detach from the current debug process. The process starts free-running right away and will not stop at breakpoints or exceptions as long as it is detached from Wing IDE.	
show-expression-evaluator	Display the expression evaluator window, bringing it to front if not already there.	
run-to-cursor	Start or continue running the debug program (either the main debug file, if specified, or otherwise the frontmost editor window) until the current cursor position is reached (or until an intervening breakpoint or exception is found).	
debug-kill	End the current debug session, exiting the debug program and terminating the debug server connecton.	
debug-stop	Stop (aka pause) the current debug session at current run location.	
choose-args	Display the debug argument dialog for the frontmost editor window.	
exception-never-stop	Never stop on execptions.	
exception-unhandled-stop	Only stop on exceptions that are not handled by the debug program's code.	
clear-exception-ignores-list	Clear all exception points previously designated as ignored during execution.	
step-over	Step over the current execution point.	
step-into	Step into function or method at current execution point.	
step-out	Step out of the current function.	
frame-up	Move up the current debug stack one frame.	
frame-down	Move down the current debug stack one frame.	
break-set	Set a new regular breakpoint at the cursor position.	

break-toggle	Set or clear a breakpoint at the cursor position.	
break-set-temp	Set a new temporary breakpoint at the cursor position.	
break-set-cond	Set a new conditional breakpoint at the cursor position, prompting the user to enter the conditional to use.	
break-ignore	Ignore the breakpoint at current cursor position for a given number of iterations, prompting user to enter the ignore value.	
break-edit-cond	Prompt user to edit the conditional for the conditional breakpoint at the current cursor position.	
break-enable	Enable the breakpoint at current cursor position.	
break-disable	Disable the breakpoint at current cursor position.	
break-clear	Clear the breakpoint at current cursor position.	
break-clear-all	Clear all set breakpoints of all types.	
show-var-defaults	Zoom the selected variable value into a seperate window, using the default configured view and value tracking styles.	
show-var-parent-ref-defaults	Zoom the selected variable value into a seperate window, tracking the value by object reference to the parent value and symbolic reference to the data slot. The display style is the default configured style.	
show-var-parent-ref-combo	Zoom the selected variable value into a seperate window, tracking the value by object reference to the parent value and symbolic reference to the data slot. The display style is a combo view containing both tree and textual areas.	

show-var-parent-ref-tree	Zoom the selected variable value into a seperate window, tracking the value by object reference to the parent value and symbolic reference to the data slot. The tree display style is used.	
show-var-parent-ref-text	Zoom the selected variable value into a seperate window, tracking the value by object reference to the parent value and symbolic reference to the data slot. The textual display style is used.	
show-var-value-ref-defaults	Zoom the selected variable value into a seperate window, tracking the value by direct object reference. The display style is the default configured style.	
show-var-value-ref-combo	Zoom the selected variable value into a seperate window, tracking the value by direct object reference. The display style is a combo view containing both tree and textual areas.	
show-var-value-ref-tree	Zoom the selected variable value into a seperate window, tracking the value by direct object reference. The tree display style is used.	
show-var-value-ref-text	Zoom the selected variable value into a seperate window, tracking the value by direct object reference. The textual display style is used.	
show-var-symbolic-defaults	Zoom the selected variable value into a seperate window, tracking the value by direct object reference. The display style is the default configured style.	
show-var-symbolic-combo	Zoom the selected variable value into a seperate window, tracking the value by its symbolic path. The display style is a combo view containing both tree and textual areas.	

show-var-symbolic-tree	Zoom the selected variable value into a seperate window, tracking the value by its symbolic path. The tree display style is used.	
show-var-symbolic-text	Zoom the selected variable value into a seperate window, tracking the value by its symbolic path. The textual display style is used.	
set-default-var-view-combo	Set the default zoom view style to combination view, containing both a tree and textual display area. This overrides the value set by the `debug.default-var-view-style` preference.	
set-default-var-view-tree	Set the default zoom view style to dynamic tree. This overrides the value set by the `debug.default-var-view-style` preference.	
set-default-var-view-text	Set the default zoom view style to textual. This overrides the value set by the `debug.default-var-view-style` preference.	
set-default-var-track-symbolic	Set the default zoom value tracking style to use the symbolic path to the data value. This overrides the value set by the `debug.default-var-track-style` preference.	
set-default-var-track-parent-ref	Set the default zoom value tracking style to use the object reference of the parent value and the symbolic name for the data slot. This overrides the value set by the `debug.default-var-track-style` preference.	

set-default-var-track-value-ref	Set the default zoom value tracking style to use a direct object reference to the value. This overrides the value set by the `debug.default-var-track-style` preference.	
expand-tree-more	Expand the currently selected variable in the tree formatted variable display by one additional level of depth.	
collapse-tree-more	Collapse the currently selected variable in the tree formatted variable display by one additional level of depth.	
force-var-reload	Force reload of the variable currently selected on the active tree variable view. This overrides previously stored errors causing the debugger to attempt to reevaluate the value.	
clear-var-errors	Clear the list of stored variables for which fatal errors were encountered during previous debug sessions. Values for these variables are not normally reloaded during subsequent debugging.	

Appendix B

Wing Tips

This appendix provides tips for usage and pointers to useful online resources for developers using Wing IDE.

B.1 Online Resources for Wing IDE

Archaeopteryx Software provides a number of support resources free of charge. All of these are available on the web at `http://wingide.com/support`.

- **FAQ** - A list of frequently asked questions is maintained on the support site.
- **Manual** - An online copy of the reference manual for the latest version of Wing IDE is available.
- **Updates** - Updated software versions and the product source code are made available to licensed users of Wing IDE.
- **Forum** - A Wing IDE user group mailing list acts as a forum for discussion among Wing users and a place to post questions you may have about Wing. Archaeopteryx staff respond to queries made here, and the list is archived and searchable.
- **IssueTrak** - This facility provides information about known bugs and requested features.

- **Product Announcements List** - This archived and searchable mailing list is used to announce new products and new product versions as they are released. See `http://wingide.com/announcelist`.

B.2 Python Language Reference

The Python Language Reference, maintained by the developers of Python, is available online at `www.python.org`. A copy is also included with each distribution of Wing IDE, in `WINGHOME/python/doc`.

This reference manual contains the following parts:

- **Tutorial** - Start here if you are new to Python
- **Library Reference** - This documents most of Python's functionality and all of the included support libraries; everything from the built-in types and string operations, to support for data serializing, encryption, sockets programming, and numerous internet protocols.
- **Language Reference** - This documents the language core only, which is relatively small since even basic services are provided in libraries described in the Library Reference. The Data Model chapter is the most useful for reference once you are up to speed with Python.
- **Extending and Embedding** - A tutorial to get you started writing a Python extension module in C or C++, or if you want to use Python as an embedded scripting language for an application.
- **Python/C API** - This documents the Python C language API, for use when writing Python extension modules, or to make use of Python as an embedded scripting language.

B.3 Useful Tools

This section describes some useful tools that are available to Python programmers but that have not yet been integrated into the Wing's graphical user interface.

B.3.1 Performance Profiling

Performance profiling is supported by the Python `profile` and `pstats` modules.

To create a profile file named `profile.tmp` for invocation of a function `main()` you would include the following code in your application:

```
import profile
profile.run('import mymodule; mymodule.main()', 'profile.tmp')
```

This will accumulate profile data while running the function `main()` in module `mymodule`. Note that importing and fully specifying the module scope is important if you plan to run the profiler under the Wing debugger. The profiler makes assumptions about scope that are violated by the debugger so just specifying `profile.run('main()', 'profile.tmp')` will not work.

Subsequently, the `pstats` module is used to inspect the contents of the profiler's output file. For example, the following command would sort the file by cumulative time spent in each function, and then print out the top 10 compute-intensive calls:

```
import pstats
p = pstats.Stats('profile.tmp')
p.sort_stats('cumulative').print_stats(10)
```

Detailed documentation for profiling is available in the Python Library Reference under The Python Profiler.

B.3.2 Busting Object Reference Cycles (Python 1.5.2)

Because Python versions 2.0 and later contain code to detect and break object reference cycles, you may not need to worry about cyclical references unless you are using Python 1.5.2.

A common problem in reference counted garbage-collected languages like Python is memory leakage due to cyclical object references. This occurs in cases where an object A has a reference to another object B that has a reference back to object A. The interpreter fails to discard memory held by these objects even if they become unused by the program because a non-zero reference count exists as a result of the cycle.

Cycles may be much longer than just two objects, for example `A -> B -> C -> D -> A` would result in failure to discard objects A through D.

In these cases, a long-running program will eventually run out of memory as more and more objects are left intact because of their participation in object reference cycles.

Cyclops

A Python module called Cyclops is available for monitoring your program as it runs, to determine when cyclical memory references are preventing the Python interpreter from discarding unused objects.

This module can be used to print information on existing cycles at any time, including at time of program exit.

The following function might be used to invoke a function called `main` and then output cycle information for modules and functions upon exit:

```
\#-----------------------------------------------------------------------
def RunWithCyclops():
  """Run the main program under Cyclops.  Require Cyclops.py found at
  http://www.python.org/ftp/python/contrib/System/Cyclops.py."""

  from cyclops import Cyclops
  import types

  def mod_refs(x):
    return x.__dict__.values()

  def mod_tag(x, i):
    return "." + x.__dict__.keys()[i]

  def func_refs(x):
    return x.func_globals, x.func_defaults

  def func_tag(x, i):
    return (".func_globals", ".func_defaults")[i]

  def instance_filter(cycle):
    for obj, index in cycle:
        if type(obj) is types.InstanceType:
            return 1
    return 0

  z = Cyclops.CycleFinder()
  z.chase_type(types.ModuleType, mod_refs, mod_tag)
  z.chase_type(types.FunctionType, func_refs, func_tag)
```

```
z.run(main)
z.find_cycles()
z.show_stats()
z.show_cycles()
```

Once cycles are found, the most effective method for fixing the resulting memory leaks is to introduce a `destroy` function that manually clears references that are causing a cycle (for example, by setting them to `None` or by calling the `clear()` method on Python dictionaries or sequences.

If a cycle is broken at a single point in this way, all objects in the cycle will subsequently be freed. For this reason, busting cycles tends to be relatively easy.

For more information on using Cyclops, please refer to documentation within the copy located in `WINGHOME/cyclops`.

B.3.3 Debugging C/C++ Modules (on Linux)

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

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

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

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

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

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

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

Then start Python as follows:

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

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

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

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

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

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

B.4 Wing IDE Source Code

Please note: This section was written for the Linux edition and still needs to be updated for Windows users.

The source code for Wing IDE is available to all users that have a permanent (non-evaluation) license to the product. This section contains information that may be useful to those wanting to delve into the source code to make changes or add custom features.

B.4.1 Setting up the Source

Before getting started, you should download and install the binary and source distributions, as described in sections 1.2 and 1.8, respectively. You may wish to install these somewhere other than the location of your main Wing installation, so that you can continue to run a working copy of the IDE regardless of what you do to the source code (but this is not a requirement).

Once the source is installed, go to the installation location and type 'make' to build the IDE modules from source (this is required because some of the modules are written in C or C++).

Whenever you wish to run the development version of Wing IDE, you must define an environment variable `DEVEL_WINGHOME` and set this to the full path of the location where you installed the Wing IDE source. You may also want to add the Wing development directory to your `path` so you can easily run the version of Wing that resides there.

Once this is done you can start Wing and open the `ide.wpr` project file, which is the project file that allows us to use Wing to develop itself. You may run either the copy of Wing that you just built (the development copy) or another copy installed from the binary distribution. Which you run depends on your `path` and the presence of the `DEVEL_WINGHOME` environment variable.

An alternative to this is to type `make run` from the development directory. This will first rebuild any altered C/C++ modules and then run the development copy of the IDE. Note that this requires that `DEVEL_WINGHOME` is already set.

B.4.2 Top-level Organization

The source code, relative to the top level of the Wing IDE installation, is organized into three major groups of functionality, according to its origin and the distribution licenses that apply to the code:

- Most Wing source code is in the `src` directory. This is the proprietary code to which you have certain rights if you own a license, but which you may not redistribute.

- The directory `opensource` contains the portions of the IDE that were written at Archaeopteryx Software and have been released under an open source license.
- The `external` directory contains additional open source items used within the IDE but written and maintained outside of Archaeopteryx Software Inc.

B.4.3 IDE Sub-systems

Distributed among the above directories are the following major sub-systems of the IDE:

- **src.browser** - The source code browser module, which implements the source browser window's functionality (but not the code analysis itself, which is in `src.pysource` and `opensource.parsetools`).
- **src.cache** - The central text file cache through which source files are read and managed. This also acts as the point of access to shared source analysis information, used by the browser, editor, and other parts of the IDE.
- **src.debug.client** - The debug client, which implements the debugger window and sub-windows.
- **src.debug.server** - The debug server and associated network protocol implementation.
- **src.edit** - The source code editor, which uses several other modules including `opensource.pyscintilla` and `external.scintilla` to do most of its work.
- **src.guimgr** - The top-level GUI manager for Wing IDE. This manages windows and views within windows, implements the command framework and keyboard equivalency support, and defines and manages the menubar and toolbar.
- **src.license** - Wing's license verification facility.
- **src.pref** - A generic properties file manager used for Wing's preferences files, the project files, and by the license manager.
- **src.proj** - The project manager, which implements the project window.
- **src.pysource** - The Python source code analyzer. Most of its work is delegated to `opensource.parsetools`.

- **src.util** - Miscellaneous modules and utilities used throughout Wing's source code.
- **opensource.parsetools** - The core of Wing's Python language source code analysis capabilities.
- **opensource.pyscintilla** - A Python language wrapper for the `external.scintilla` source code editor module.
- **opensource.schannel** - A semi-secure TCP/IP channel used by the debugger.
- **external.cyclops** - A tool for detecting cyclical object references in Python code, as described in section B.3.2.
- **external.py2pdf** - A tool for converting Python source files into PDF. This is used by Wing's printing facility.
- **external.pygtk-0.6.5-wing** - A modified version of `pygtk-0.6.5`, which is a Python language wrapper for the `gtk` GUI development library.
- **external.scintilla** - A powerful source code editing widget used by Wing for its editor windows.

B.4.4 Documentation

Throughout the source, Python documentation strings and comments are used to describe the code. As a result, most documentation is located within the code itself.

Each IDE module also has a `README` file containing at least some text identifying the module. This may also contain high level documentation for the module's source, so it's a good place to start when inspecting the code.

The top-level docstring (at start of each Python file) contains a CVS logging area, which lists all recent revision comments for that file.

B.4.5 Naming Convention

Three techniques are used to build names for language constructs in the Wing IDE source code: (1) leading underscores indicate the public, semi-private, and private nature of the construct, (2) prefix letters are used to indicate the type of some constructs, and (3) capitalization is used to indicate the type of some constructs. This naming style is used with all

Python language constructs, including classes, functions, methods, variables, attributes, and in some cases module names.

Underscoring

The presence and number of underscores before construct names is used to indicate the scope of access intended for that construct.

- No leading underscores (as in `CallMe()`) are used for publically accessible classes, methods, functions, or attributes. These may be accessed from anywhere.
- A single underscore (as in `_DoSomething()`) is used to indicate that a construct is semi-private, for access only from within the construct's scope and related scopes (such as in sub-classes, or other classes that are part of a logical sub-system).
- A double underscore (as in `__MyMethod()`) indicates that a construct is private. For object attributes, Python will enforce local-only access of these values by making them invisible from subclasses or outside of the body of the class.

Prefixing

A prefix-based naming convention is also applied to some language constructs, as follows:

- Names starting with 'g' are globals
- Names starting with 'f' are class instance attributes
- Names starting with 'k' are constants
- Names starting with 'C' are classes.

Capitalization

Capitalization of letters within construct names is also standardized:

- Names in the form `XxxXxxx` are used for classes, methods, instance attributes, functions, and constants.
- Names like `xxx_xxx` are used for parameters, locals, module names, and disk directories.

Underscoring, prefixing, and capitalization are combined as appropriate for all constructs, to build names like `_CMyClass` (a semi-private class), `kAValue` (a public constant), and `__fCount` (a private instance attribute).

Appendix C

Software License

This End User License Agreement (EULA) is a CONTRACT between you (either an individual or a single entity) and Archaeopteryx Software, Inc. (Archaeopteryx), which covers your use of either "Wing IDE for Linux" or "Wing IDE for Windows" and related software components. All such software is referred to herein as the "Software Product." A software license and a license key or serial number ("Software Product License"), issued to a designated user only by Archaeopteryx or its authorized agents, is required for each concurrent user of the Software Product. If you do not agree to the terms of this EULA, then do not install or use the Software Product or the Software Product License. By explicitly accepting this EULA you are acknowledging and agreeing to be bound by the following terms:

1. EVALUATION LICENSE WARNING

This Software Product can be used in conjunction with a free evaluation Software Product License. If you are using such an evaluation Software Product License, you may use the Software Product only to evaluate its suitability for purchase. Evaluation Software Product Licenses have an expiration date and most of the features of the software will be disabled after that date. ARCHAEOPTERYX BEARS NO LIABILITY FOR ANY DAMAGES RESULTING FROM USE (OR ATTEMPTED USE AFTER THE EXPIRATION DATE) OF THE SOFTWARE PRODUCT, AND HAS NO DUTY TO PROVIDE ANY SUPPORT BEFORE OR AFTER THE EXPIRATION DATE OF AN EVALUATION LICENSE.

2. GRANT OF NON-EXCLUSIVE LICENSE

Archaeopteryx grants the non-exclusive, non-transferable right for a single user to use this Software Product. Each additional concurrent user of the Software Product requires an additional Software Product License.

Archaeopteryx grants you the right to modify, alter, improve, or enhance the Software Product without limitation, except as described in this EULA.

Although rights to modification of the Software Product are granted by this EULA, you may not tamper with, alter, or use the Software Product in a way that disables, circumvents, or otherwise defeats its built-in licensing verification and enforcement capabilities. The right to modification of the Software Product also does not include the right to remove or alter any trademark, logo, copyright or other proprietary notice, legend, symbol or label in the Software Product.

You may at your discretion distribute patch files containing any modifications or improvements made to the Software Product, other than those that are aimed at disabling or circumventing its built-in license verification capabilities, that result in the removal or alteration of any trademark, logo, copyright, or other proprietary notice, legend, symbol or label in the Software Product. This right does not include the right to distribute substantial portions of the original source, where distribution rights are limited to contextual information normally existing in software patch files.

You may at your discretion designate license terms, open source or otherwise, for all modifications or improvements made by you. Archaeopteryx has no special rights to any such modifications or improvements.

You may make copies of the Software Product as reasonably necessary for its use. Each copy must reproduce all copyright and other proprietary rights notices on or in the Software Product.

You may install each Software Product License on a single computer system. A second installation of the same Software Product License may be made on one other computer system, so long as both copies of the same Software Product License never come into concurrent use. You may also make copies of the Software Product License as necessary for backup and/or archival purposes. Backup and archival copies may not come into active use, together with the Software Product, for any purpose. No other copies may be made. Each copy must reproduce all copyright and other proprietary rights notices on or in the Software Product License. You may not modify or create derivative copies of the Software Product License.

All rights not expressly granted to you are retained by Archaeopteryx.

3. INTELLECTUAL PROPERTY RIGHTS RESERVED BY ARCHAEOPTERYX

The Software Product is owned by Archaeopteryx and is protected by United States and international copyright laws and treaties, as well as other intellectual property laws and treaties. You must not remove or alter any copyright notices on any copies of the Software Product. This Software Product copy is licensed, not sold. You may not use, copy, or distribute the Software Product, except as granted by this EULA, without written authorization from Archaeopteryx or its designated agents. Furthermore, this EULA does not grant you any rights in connection with any trademarks or service marks of Archaeopteryx. Archaeopteryx reserves all intellectual property rights, including copyrights, and trademark rights.

4. NO RIGHT TO TRANSFER

You may not rent, lease, lend, or in any way distribute or transfer any rights in this EULA or the Software Product to third parties without Archaeopteryx's written approval, and subject to written agreement by the recipient of the terms of this EULA.

5. INDEMNIFICATION

You hereby agree to indemnify Archaeopteryx against and hold harmless Archaeopteryx from any claims, lawsuits or other losses that arise out of your breach of any provision of this EULA.

6. THIRD PARTY RIGHTS

Any software provided along with the Software Product that is associated with a separate license agreement is licensed to you under the terms of that license agreement. This license does not apply to those portions of the Software Product. Copies of these third party licenses are included in all copies of the Software Product.

7. SUPPORT SERVICES

Archaeopteryx may provide you with support services related to the Software Product. Use of any such support services is governed by Archaeopteryx policies and programs described in online documentation and/or other Archaeopteryx-provided materials.

As part of these support services, Archaeopteryx may make available bug lists, planned feature lists, and other supplemental informational materials. ARCHAEOPTERYX MAKES NO WARRANTY OF ANY KIND FOR THESE MATERIALS AND ASSUMES NO LIABILITY WHATSOEVER FOR DAMAGES RESULTING FROM ANY USE OF THESE MATERIALS. FURTHERMORE, YOU MAY NOT USE ANY MATERIALS PROVIDED IN THIS WAY TO SUPPORT ANY CLAIM MADE AGAINST ARCHAEOPTERYX.

Any supplemental software code or related materials that Archaeopteryx provides to you as part of the support services, in periodic updates to the Software Product or otherwise, is to be considered part of the Software Product and is subject to the terms and conditions of this EULA.

With respect to any technical information you provide to Archaeopteryx as part of the support services, Archaeopteryx may use such information for its business purposes without restriction, including for product support and development. Archaeopteryx will not use such technical information in a form that personally identifies you without first obtaining your permission.

9. TERMINATION WITHOUT PREJUDICE TO ANY OTHER RIGHTS

Archaeopteryx may terminate this EULA if you fail to comply with any term or condition of this EULA. In such event, you must destroy all copies of the Software Product and Software Product Licenses.

10. U.S. GOVERNMENT USE

If the Software Product is licensed under a U.S. Government contract, you acknowledge that the software and related documentation are "commercial items," as defined in 48 C.F.R 2.01, consisting of "commercial computer software" and "commercial computer software documentation," as such terms are used in 48 C.F.R. 12.212 and 48 C.F.R. 227.7202-1. You also acknowledge that the software is "commercial computer software" as defined in 48 C.F.R. 252.227-7014(a)(1). U.S. Government agencies and entities and others acquiring under a U.S. Government contract shall have only those rights, and shall be subject to all restrictions, set forth in this EULA. Contractor/manufacturer is Archaeopteryx Software, Inc., P.O. Box 1937, Brookline MA 02446-0016, USA.

11. EXPORT RESTRICTIONS

You will not download, export, or re-export the Software Product, any part thereof, or any software, tool, process, or service that is the direct product of the Software Product, to any country, person, or entity -- even to foreign units of your own company -- if such a transfer is in violation of U.S. export restrictions.

12. NO WARRANTIES

YOU ACCEPT THE SOFTWARE PRODUCT AND SOFTWARE PRODUCT LICENSE "AS IS," AND ARCHAEOPTERYX AND ITS THIRD PARTY SUPPLIERS AND LICENSORS MAKE NO WARRANTY AS TO ITS USE, PERFORMANCE, OR OTHERWISE. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, ARCHAEOPTERYX AND ITS THIRD PARTY SUPPLIERS AND LICENSORS DISCLAIM ALL OTHER REPRESENTATIONS, WARRANTIES, AND CONDITIONS, EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE, INCLUDING, BUT NOT LIMITED TO, IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, SATISFACTORY QUALITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE, AND NON-INFRINGEMENT. THE ENTIRE

RISK ARISING OUT OF USE OR PERFORMANCE OF THE SOFTWARE PRODUCT REMAINS WITH YOU.

13. LIMITATION OF LIABILITY

THIS LIMITATION OF LIABILITY IS TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW. IN NO EVENT SHALL ARCHAEOPTERYX OR ITS THIRD PARTY SUPPLIERS AND LICENSORS BE LIABLE FOR ANY COSTS OF SUBSTITUTE PRODUCTS OR SERVICES, OR FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, OR LOSS OF BUSINESS INFORMATION) ARISING OUT OF THIS EULA OR THE USE OF OR INABILITY TO USE THE SOFTWARE PRODUCT OR THE FAILURE TO PROVIDE SUPPORT SERVICES, EVEN IF ARCHAEOPTERYX HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN ANY CASE, ARCHAEOPTERYX’S, AND ITS THIRD PARTY SUPPLIERS’ AND LICENSORS’, ENTIRE LIABILITY ARISING OUT OF THIS EULA SHALL BE LIMITED TO THE LESSER OF THE AMOUNT ACTUALLY PAID BY YOU FOR THE SOFTWARE PRODUCT OR THE PRODUCT LIST PRICE; PROVIDED, HOWEVER, THAT IF YOU HAVE ENTERED INTO AN ARCHAEOPTERYX SUPPORT SERVICES AGREEMENT, ARCHAEOPTERYX’S ENTIRE LIABILITY REGARDING SUPPORT SERVICES SHALL BE GOVERNED BY THE TERMS OF THAT AGREEMENT.

14. HIGH RISK ACTIVITIES

The Software Product is not fault-tolerant and is not designed, manufactured or intended for use or resale as on-line control equipment in hazardous environments requiring fail-safe performance, such as in the operation of nuclear facilities, aircraft navigation or communication systems, air traffic control, direct life support machines, or weapons systems, in which the failure of the Software Product, or any software, tool, process, or service that was developed using the Software Product, could lead directly to death, personal injury, or severe physical or environmental damage ("High Risk Activities"). Accordingly, Archaeopteryx and its suppliers and licensors specifically disclaim any express or implied warranty of fitness for High Risk Activities. You agree that Archaeopteryx and its suppliers and licensors will not be liable for any claims or damages arising from the use of the Software Product, or any software, tool, process, or service that was developed using the Software Product, in such applications.

15. GOVERNING LAW; ENTIRE AGREEMENT ; DISPUTE RESOLUTION

This EULA is governed by the laws of the Commonwealth of Massachusetts, U.S.A., excluding the application of any conflict of law rules. The United Nations Convention on Contracts for the International Sale of Goods shall not apply.

This EULA is the entire agreement between Archaeopteryx and you, and supersedes any other communications or advertising with respect to the

Software Product; this EULA may be modified only by written agreement signed by authorized representatives of you and Archaeopteryx.

Unless otherwise agreed in writing, all disputes relating to this EULA (excepting any dispute relating to intellectual property rights) shall be subject to final and binding arbitration in the State of Massachusetts, in accordance with the Licensing Agreement Arbitration Rules of the American Arbitration Association, with the losing party paying all costs of arbitration. Arbitration must be by a member of the American Arbitration Association. If any dispute arises under this EULA, the prevailing party shall be reimbursed by the other party for any and all legal fees and costs associated therewith.

16. GENERAL

If any provision of this EULA is held invalid, the remainder of this EULA shall continue in full force and effect.

A waiver by either party of any term or condition of this EULA or any breach thereof, in any one instance, shall not waive such term or condition or any subsequent breach thereof.

17. OUTSIDE THE U.S.

If you are located outside the U.S., then the provisions of this Section shall apply. Les parties aux prsentes confirment leur volont que cette convention de mme que tous les documents y compris tout avis qui s’y rattache, soient redigs en langue anglaise. (translation: "The parties confirm that this EULA and all related documentation is and will be in the English language.") You are responsible for complying with any local laws in your jurisdiction which might impact your right to import, export or use the Software Product, and you represent that you have complied with any regulations or registration procedures required by applicable law to make this license enforceable.

18. TRADEMARKS

The following are trademarks or registered trademarks of Archaeopteryx:

Archaeopteryx, Wing IDE, Wing IDE Professional, Wing IDE Enterprise, Wing Debugger, and "Take Flight!".

19. CONTACT INFORMATION

If you have any questions about this EULA, or if you want to contact Archaeopteryx for any reason, please direct all correspondence to: Archaeopteryx Software, Inc., P.O. Box 1937, Brookline, MA 02446-0016, United States of America or send email to info@archaeopteryx.com.