

ERC CISST Software

User Guide for cisstInteractive

Peter Kazanzides

Copyright © 2006 Johns Hopkins University (JHU). All rights reserved.

Contents

1 Introduction	1
2 The cisstInteractive Library	2
2.1 C++ Files	3
2.2 Python Files	4
3 pythonEmbeddedIRE Example Program	4
4 Standalone IRE: Launching from Python	5
5 Embedded IRE: Launching from C++	5
5.1 The IRE Thread	6
5.2 Caveat for Macintosh OS X	6
6 Accessing C++ Objects from Python	7
7 IRE User Interface	7
8 Troubleshooting	8

1 Introduction

This document describes the *Interactive Research Environment (IRE)* provided by the cisstInteractive library. The IRE is a graphical user interface that includes an interpreter (shell) for the Python programming language. It is primarily intended to be embedded in a C++ program that uses the cisst libraries, but it can also be used in standalone mode, where it is started (launched) directly from Python.

The IRE requires that the Python interpreter, and the wxPython package (Version 2.5 or greater), be installed on the system. Note that although wxPython is based on the wxWidgets C++ package, it is not necessary to install wxWidgets – the wxPython installation includes a wrapped version of wxWidgets. For more details, consult the build instructions (ADD LINK HERE).

The IRE provides the following main features:

- A Python shell that displays pop-up help information as the user types, such the documentation string or a list of all class attributes (methods and data).
- A command history that persists between different runs of the application, along with menu options to clear, load and truncate/save the history. One or more commands can also be dragged from the history list into the shell for immediate execution.
- A list of all defined shell variables and menu options to save this workspace (to a file) and restore it. The user can drag one or more variable names to the shell to display their values.
- A list of all objects in the Object Registry (see 6). This enables display and modification of C++ objects from Python. The Object Registry is initially disabled when the IRE is run in standalone mode, but can be enabled later.
- A built-in editor that can be used to create and modify Python source files, as well as a menu option to run these files in the shell.
- A *Logger Output* window that displays log messages generated by the C++ software via the `cmnLogger` class and related functions/macros. The window includes a control to set the level of detail that determines whether a particular message is displayed. The logger window can only be enabled when the IRE is run in embedded mode.

The `cisst` examples directory includes `pythonEmbeddedIRE`, which demonstrates use of the IRE with a C++ program. Note that there is also a `pythonEmbedded` example that demonstrates sharing of objects between a C++ program and its embedded (text-mode) Python interpreter. Because this document focuses on the IRE, this latter example will not be described in detail.

2 The `cisstInteractive` Library

The `cisstInteractive` library consists of C++ and Python source code. The C++ code is provided by the file `ireFramework.cpp` and its associated header files, `ireFramework.h` and `ireExport.h`. The Python code consists of several Python source files (`*.py`) grouped into the `irepy` module. Note that the `irepy` is organized as a subdirectory in the `cisst` source tree:

```
.../libs/code/cisstInteractive/irepy
```

During compilation of the `cisst` source code, a special build command is executed to create “compiled” versions (`*.pyc`) of the `irepy` module in the build tree; specifically in an `irepy` subdirectory of the `bin` directory. Note that `.../bin/irepy` is used for Windows as well, even though C++ libraries and executables are placed in `.../bin/release` and `.../bin/debug`, because the Python files are not dependent on the C++ compiler configuration.

2.1 C++ Files

The `ireFramework.h` file declares the `ireFramework` class. C++ programs can launch the IRE by invoking the `ireFramework::launchIREShell` method, as described in Section 5. The `ireFramework.cpp` file provides the implementation of the `ireFramework` class. The procedure for launching the IRE depends on whether the `new_thread` parameter is `true` or `false`. (MAYBE MOVE THE DISCUSSION OF THIS HERE).

The `ireFramework.cpp` file also provides the implementation of the `ireLogger` module, which provides an interface between the C++ log functions (see `cmnLogger.h`) and the *Logger Output* window of the IRE. The `ireLogger` module provides the following four methods to the Python interpreter:

`void SetTextOutput(callback, strip, LoD)` This method sets the callback function (`callback`) that is used to log text to the IRE window. The optional `strip` parameter specifies whether new-lines should be stripped from the text before it is passed to the callback function (default is `False`, which leaves newlines intact). The optional `LoD` parameter specifies the level of detail that should be used to filter messages sent to the *Logger Output* window. If `LoD` is not specified, a default value (initially `CMN_LOG_DEFAULT_LOD`) is used. This function creates a new `streambuf` object (based on `cmnCallbackStreambuf`) and adds it as a new channel to the `cmnLogger` multiplexer.

`void ClearTextOutput()` This method disables logging of text to the IRE window. It clears the callback function, removes the channel from the `cmnLogger` channel multiplexer, and deletes the `streambuf` object that was created by `SetTextOutput`.

`int GetLoD()` This method returns the level of detail currently associated with the *Logger Output* window.

`void SetLoD(LoD)` This method sets the level of detail used by the *Logger Output* window.

One interesting feature of the `ireLogger` callback is that it can bypass the Python interpreter if a callable C function (as determined by `PyCFunction_Check`) is passed via `SetTextOutput`.

In this case, the `ireLogger` callback function, `PrintLog` directly calls the C function via `PyCFunction_Call`. Besides improving efficiency, this has the added advantage that it does not require multi-threaded access to the Python interpreter, so the overhead of acquiring and releasing the global interpreter lock can be avoided.

The Python wrappers for the `ireLogger` methods were manually created, rather than being automatically generated by Swig. It may be possible to use Swig and still obtain the unique features of this implementation, such as direct calls to wrapped C functions via `PyCFunction_Call`, but that was not investigated.

2.2 Python Files

The `irepy` module includes the following Python files:

`__init__.py` Initialize the module.

`ireMain.py` Set everything up...

`ireEditorNotebook.py`

`ireListCtrlPanel.py`

`ireShell.py`

`ireLogCtrl.py`

`ireWorkspace.py`

`ireDragAndDrop.py`

`ireImages.py` Images encoded as Python strings. Note that `ArtProvider` could have been used instead.

`ireInputBox.py` Not really part of the IRE.

3 pythonEmbeddedIRE Example Program

Describe the example program.

4 Standalone IRE: Launching from Python

The IRE can be launched from Python by simply importing the “irepy” module and then invoking the “launch” method, as follows:

```
import irepy
irepy.launch()
```

These two commands can be specified on the command line, allowing the user to create convenient “shortcuts”. For example, an IRE desktop shortcut can be created on Microsoft Windows by specifying the following target:

```
C:\Python24\python.exe -i -c "import irepy; irepy.launch()"
```

Note that the Python installation on the Microsoft Windows and Macintosh OS X operating systems includes `python` and `pythonw` executables. The `pythonw` executable must be used on the Macintosh because otherwise OS X will not allow the IRE to access the display. Either executable can be used on Microsoft Windows. The difference between them is that `python` runs in a terminal window, whereas `pythonw` does not. Although `pythonw` is more visually appealing (no terminal window lying in the background), the `python` executable is recommended for the IRE because output from the IRE Python modules (e.g., error or debug info) is displayed in the terminal window. Note that if `pythonw` is used, the `-i` option should not be specified.

5 Embedded IRE: Launching from C++

The `cisstInteractive` library has been designed to streamline the launching of the IRE. The C++ program must include the `ireFramework.h` file, for example as follows:

```
#include <cisstInteractive/ireFramework.h>
```

This file defines the `ireFramework` class, which is a Singleton class (i.e., only one instance can be created). The IRE can be started by the following command:

```
ireFramework::LaunchIREShell();
```

This launches the IRE in the current thread and does not return until the IRE is exited. The `LaunchIREShell` command takes the following optional parameters:

- startup** A character string (`char *`) that specifies a command string to be passed to the Python interpreter. This can be used for any application-specific initialization, such as importing an application module. The default is an empty string.
- new_thread** A boolean that specifies whether or not to use the Python *threading* module to create a new thread for the IRE. The default value is `false`. If `new_thread` is `true`, the `LaunchIREShell` method returns after the IRE is initialized. See Section 5.1 for more information about starting the IRE thread.

The `LaunchIREShell` method may throw the `std::runtime_error` exception, so it is recommended that `ireFramework::LaunchIREShell` be invoked within a `try...catch` block.

Finally, the C++ program should call `ireFramework::FinalizeShell()` when the IRE is exited.

5.1 The IRE Thread

Obviously, the IRE must execute in a separate thread if the user wishes to run the C++ program and the IRE concurrently. There are two options for creating the IRE thread:

1. Create the IRE thread in the C++ software and then call `LaunchIREShell`, specifying `false` (the default value) for the `new_thread` parameter.
2. Create the IRE thread in Python by calling `LaunchIREShell` with the `new_thread` parameter set to `true`.

The first option is recommended because the IRE thread will be managed in the same manner as any thread created within the C++ program (i.e., multithreading will work as expected). In the second case, the Python thread will not get any execution time unless the C++ program periodically calls the `ireFramework::JoinIREShell` method with a non-zero `timeout` parameter.

Note, however, that the first method (starting a thread in C++) is not as portable as the second method (starting a thread in Python), unless an operating system abstraction library, such as `cisstOSAbstraction`, is used. Because `cisstOSAbstraction` is not yet available as open source software, the *pythonEmbeddedIRE* example program uses the second method.

5.2 Caveat for Macintosh OS X

OS X does not generally allow terminal programs, such as *pythonEmbeddedIRE*, to access the display. Therefore, the *pythonEmbeddedIRE* example program employs a workaround, where it “borrows” the identity of `pythonw` to get access to the display. This workaround can be used for other terminal programs. For details, see the file `pythonEmbeddedIRE.cpp`.

6 Accessing C++ Objects from Python

The `cisstCommon` library provides an Object Registry (`cmnObjectRegister` class) that can allow C++ objects to be made available to Python code. An object can only be added to the Registry if it is derived from the `cmnGenericObject` base class.

Because the Object Registry is part of the `cisstCommon` library, it can be used with or without the IRE (`cisstInteractive` library). The *pythonEmbedded* example program demonstrates use of the Object Registry with a simple embedded Python shell that is invoked by repeated calls to `PyRun.SimpleString`. The *pythonEmbeddedIRE* example program demonstrates use of the Object Registry with the IRE.

In general, the process is as follows:

1. Create (or use) a C++ class that is derived from `cmnGenericObject`. In the *pythonEmbeddedIRE* example, this is the `SineGenerator` class.
2. Wrap the C++ class (at least the methods of interest) so that it can be accessed from Python. The `cisst` package uses Swig (www.swig.org) to automate the wrapping process and it is highly recommended for wrapping any user-defined classes. The *pythonEmbeddedIRE* example includes `SineGeneratorPython.i` as the Swig input file.
3. In the C++ program, create an instance of the class and store it in the Object Registry using the `cmnObjectRegister::Register` method. This method takes two parameters: a string (the object name) and a reference to the object being registered.
4. In the Python program, retrieve the object from the registry by calling the `cmnObjectRegister.FindObject` method. This method requires a string parameter (the object name) and returns a reference to the object, assuming that it is found in the registry.
5. The Python code can now manipulate the object by invoking any wrapped method. For example, the *pythonEmbeddedIRE* example wraps `SineGenerator` methods such as `GetAmplitude`, `SetAmplitude`, `SetFrequency`, and `GetFrequency`.

Obviously, the Object Registry is only useful when the user is working with Python and C++ code. For this reason, it is initially disabled (and the Register Contents window is not displayed) when the IRE is started directly from Python (i.e., in standalone mode). It is automatically enabled when the `cisstCommon` library (actually, `cisstCommonPython`) is imported into the IRE. The Register Contents window is not automatically shown, however, and must be selected via the *View* menu.

7 IRE User Interface

Describe all the menu items, windows, etc...

8 Troubleshooting

What to do when it doesn't work. Make sure wxPython is installed. Make sure that PYTHONPATH is defined. Make sure that LD_LIBRARY_PATH (or DYLD_LIBRARY_PATH) is defined. And so on...