CSOUND CONFERENCE 2011

Python Scripting in QuteCsound

Andrés Cabrera
mantaraya36 AT gmail.com

Introduction

This article will present the Python scripting facilities within QuteCsound, which expose both portions of the QuteCsound application and the running Csound engines within it.

Adding Python scripting to QuteCsound comes from the motivation of integrating a powerful interactive language like Python with QuteCsound to provide new possibilities for realtime control and scripting which can open the door to more interactive methodologies for algorithmic composition and to a broader range of applications and interfaces for arts and research through Python's extensive support of a broad range of technologies and infrastructures.

I. Using Python in QuteCsound

There are three ways of running Python code in QuteCsound:

1. As a whole Python file which is run in the interpreter or a separate shell using the Run and Run in Term actions from the menus or the icon bar.
2. On the interactive Python Console, which reports from the main interpreter, and shows the results of any Python code executed in the internal interpreter.
3. By evaluating code from editors or the Python scratch pad (using the Evaluate selection or Evaluate Section actions). The Python scratch pad is a disposable buffer to prepare constructs like function definitions or loops which are not practical or possible to type on the console directly. Section separators are marked using two number signs ##. These section separations are shown in the Inspector.

It is important to note that there is only one internal Python interpreter, so any variables, functions and objects will be available from any places where code can be evaluated in the interpreter.

Additionally, there is a menu on the interface called “Scripts” which lists the Python files found in the Scripts folder which can be configured in the Preferences. This simplifies execution of common Python tasks, and results in a sort of extension system, since it is possible for the script to act on the active content, for things like modifying indentation of a certain content, to performing text transformations of the score, or interacting with external applications like lilypond or a browser.

II. Architecture

The Python scripting capabilities in QuteCsound are implemented through PythonQt[1], a C++ library that automatically wraps C++ and Qt classes for exposure in Python. This greatly simplifies the work, reducing the amount of manual code wrapping that normally needs to be done. PythonQt simplifies the creation of Python interpreter instances and provides console widgets with code completion based on

1 http://pythonqt.sourceforge.net/

the active objects and variables of the interpreter instance.

An additional benefit of PythonQt is that it wraps most of Qt, giving the possibility of building GUIs in Qt directly from Python, without the need for PyQt[2] or PySide[3].

Elements of QuteCsound and Csound are exposed through a custom object called "q", which is available by default in the interpreter. This object wraps a lot of diverse functionality from the QuteCsound interface, Live Event Sheets, Code editors and running Csound instances, in an integrated and hopefully succinct way.

III. The QuteCsound Python Object

The QuteCsound object (called PyQcsObject in the sources) is the interface for scripting QuteCsound and exposes a set of functions for various elements of the QuteCsound application and any of the Csound instances associated with currently open documents. Here will be presented the different functions available, grouped according to the elements they affect.

As mentioned, by default, a PyQcsObject is already available in the interpreter called "q", so all the methods described below should be called as members of this object.

QuteCsound Interface

The QuteCsound object offers control of execution of any of the open documents, by index. Giving an index of -1 or omitting it, executes the order on the currently active document.

```
play(int index = -1, realtime = true)
pause(int index = -1)
stop(int index = -1)
stopAll()
```

There are also functions to find the index of a document by its name, and to set a particular index as active. Setting a document like this will make it visible, as if the tab for it had been clicked.

```
setDocument(index)
getDocument(name = "") # Returns document index. -1 if not current open
```

Editors

Operations on text for open files can be done through the API, for example:

```
insertText(text, index = -1, section = -1)
```

will insert the text on the current cursor position, for document *index*. Text for individual sections of the file can also be inserted using the functions:

```
setCsd(text, index = -1);
setFullText(text, index = -1)
setOrc(text, index = -1)
setSco(text, index = -1)
setWidgetsText(text, index = -1)
setPresetsText(text, index = -1)
setOptionsText(text, index = -1)
```

And text can also be queried with the functions:

2 http://www.riverbankcomputing.co.uk/pyqt/
3 http://www.pyside.org/

```
getSelectedText(index = -1, int section = -1)
getCsd(index = -1)
getFullText(index = -1)
getOrc(index = -1)
getSco(index = -1)
getSelectedWidgetsText(int index = -1);
getWidgetsText(index = -1)
getPresetsText(index = -1)
getOptionsText(index = -1)
```

Additionally information about the file being edited can be queried with:

```
getFileName(index = -1)
getFilePath(index = -1)
```

Widgets

Widget values and properties can be changed and queried through the API for any of the open documents with the functions:

```
setChannelValue(channel, value, index = -1)
getChannelValue(channel, index = -1)
setChannelString(channel, stringvalue, int index = -1)
getChannelString(channel, index = -1)
setWidgetProperty(channel, property, value, index= -1)
getWidgetProperty(channel, property, index= -1)
```

And widgets can be created and destroyed through the API. The functions to create widgets return a string with the UUID (unique id) of the widget, which can then be used in the destructor function:

```
createNewLabel(int x, int y, int index = -1)
createNewDisplay(int x, int y, int index = -1)
createNewScrollNumber(int x, int y, int index = -1)
createNewLineEdit(int x, int y, int index = -1)
createNewSpinBox(int x, int y, int index = -1)
createNewSlider(int x, int y, int index = -1)
createNewButton(int x, int y, int index = -1)
createNewKnob(int x, int y, int index = -1)
createNewCheckBox(int x, int y, int index = -1)
createNewMenu(int x, int y, int index = -1)
createNewMeter(int x, int y, int index = -1)
createNewConsole(int x, int y, int index = -1)
createNewGraph(int x, int y, int index = -1)
createNewScope(int x, int y, int index = -1)
destroyWidget(QString uuid)
```

Csound functions

Several functions can interact with the Csound engine, for example to query information about it:

```
getVersion() # QuteCsound API version
getSampleRate(int index)
getKsmps(int index)
getNumChannels(int index)
opcodeExists(QString opcodeName)
```

(Strictly speaking, the opcodeExists() function doesn't interact with the engine, but with the opcode list from the documentation, but they should be in sync, although they might not reflect details of a particular installation. The purpose of this function is more to work with text parsers so they can identify opcodes against the rest of the text)

There are objects to send score events to any running document, without having to switch to it:

```
sendEvent(int index, QString events)
sendEvent(QString events)
```

And there is a function which can register a Python function as a callback to be executed in between processing blocks for Csound. The first argument should be the text that should be called on every pass. It can include arguments or variables which will be evaluated every time. You can also set a number of periods to skip to avoid.

```
registerProcessCallback(QString func, int skipPeriods = 0)
```

You can register the python text to be executed on every Csound control block callback, so you can execute a block of code, or call any function which is already defined.

IV. Future Additions/ Work in progress

F-tables

An important functionality will be to connect f-tables from the running instance to Python. Although not implemented yet, it is hoped they will look like:

```
getTablePointer(fn, index = -1)
copyTableToList(fn, index = -1, offset = 0, number = -1)
copyListToTable(list, fn, index = -1, offset = 0)
```

The main issue complicating implementation of these is that tables must be accessed at a time when the Csound engine is idle, so these functions must work through a messaging system that sends requests to the performance thread, which will do the work when it should. The messages are stored in a lock-free ring buffer and the caller functions will wait for a done signal from the realtime thread, to read the data and return. This is what SuperCollider does in these cases and some of this has already implemented in the csPerfThread C++ class.

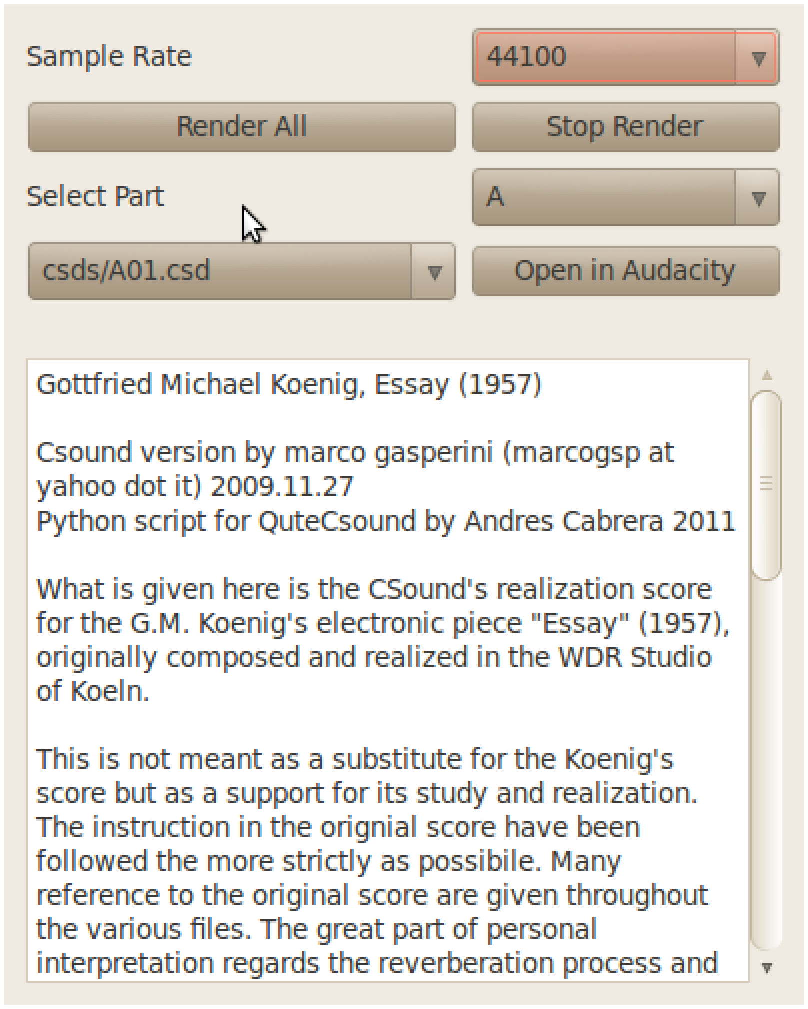
Live Event Sheet interaction

Although not yet implemented, it is hoped to implement a set of functions which can interact with Live Event Sheets, so they can become a sort of visual matrix, which can be used with the mouse and keyboard or programatically from the API. They will probably look like:

```
QuteSheet* getSheet(int index = -1, int sheetIndex = -1)
QuteSheet* getSheet(int index, QString sheetName)
```

And they should transparently connect the data so a change in one will affect the other. The QuteSheet object will implement the necessary functionality to query and modify cells.

Conclusions



Sample Rate 44100

Select Part A

csds/A01.csd

Gottfried Michael Koenig, Essay (1957)

Csound version by marco gasperini (marcogsp at yahoo dot it) 2009.11.27
Python script for QuteCsound by Andres Cabrera 2011

What is given here is the CSound's realization score for the G.M. Koenig's electronic piece "Essay" (1957), originally composed and realized in the WDR Studio of Koeln.

This is not meant as a substitute for the Koenig's score but as a support for its study and realization. The instruction in the orignial score have been followed the more strictly as possibile. Many reference to the original score are given throughout the various files. The great part of personal interpretation regards the reverberation process and

Illustration 1: Interface for rendering Marco Gasperini's realization of Keonig's Essay

It is expected that this work can open up new possibilities for control and interaction with Csound.

Some possible uses include:

- Graphical Ftable editor
- Sequencer and notation applications
- Interactive pieces (e.g. Koenig realization shown in Illustration 1)
- Automatic code generation/visualization/transformation
- Design of custom control GUIs and widgets
- Live Coding
- Remote control on mobile devices (e.g. Send the widgets and do the network connections automatically)