Multiprocess System for Virtual Instruments in Python

Brian D'Urso (dursobr@pitt.edu) – *University of Pittsburgh, Department of Physics and Astronomy, 3941 O'Hara St., Pittsburgh, PA 15260* US

Programs written for controlling laboratory equipment and interfacing numerical calculations share the need for a simple graphical user interface (GUI) frontend and a multithreaded or multiprocess structure to allow control and data display to remain usable while other actions are performed. We introduce Pythics, a system for running "virtual instruments", which are simple programs typically used for data acquisition and analysis. Pythics provides a simple means of creating a virtual instrument and customizing its appearance and functionality without the need for toolkit specific knowledge. It utilizes a robust, multiprocess structure which separates the GUI and the back end of each instrument to allow for effective usage of system resources without sacrificing functionality.

Python is an attractive language for scientific programming because of the simplicity of expressing mathematical ideas and algorithms within it. However, it is the broad range of open source libraries available that enables scientific computing within Python. With the capabilities of libraries such as Numpy [numpy] and SciPy [scipy] for numerical processing, SymPy [sympy] for symbolic manipulation, and Sage [sage] integrating them together, Python is in a strong position to handle a wide range of scientific computational problems.

However, in experimental sciences, where computer-based data acquisition and control is dominated by a small number of proprietary programming systems, Python is not such an obvious choice. These proprietary systems often make the task of simple experiment control easy, however they often don't scale well to complex experiments because they lack a well-designed, general purpose programming language. We present a system starting with a complete programming language, Python, rather than trying to develop another special purpose language. There are existing, mature Python libraries for supporting data acquisition and experiment control; perhaps most critically PyVISA [pyvisa], which provides a robust bridge to commercially-supported VISA libraries and the experiment hardware they can control. In practice we also make use of the Python Imaging Library (PIL) [pil] for image manipulation, wxPython [wxpython] as a user interface toolkit, and matplotlib [matplotlib] for plotting.

In addition to being able to communicate with instruments, modern software for data acquisition and control must be able to present a graphical user interface (GUI) for display of data as well as providing a means for the experimenter to interact with an experiment in progress. GUI toolkits such as wxPython provide a means to create a GUI, but are not tailored to the requirements of data acquisition, where communication with instruments often must be proceeding in parallel with GUI operation. Furthermore, students working on experiments may have little or no programming experience, so programming a multithreaded or multiprocess application may be beyond what they can or want to pursue.

Here we introduce Pythics (PYTHon Instrument Control System), a multiprocess system designed to make it straightforward to write software for data acquisition and control with Python. There are several important features which expedience has taught us are needed to produce a successful system. First, the system must be cross platform, at least supporting Linux and Microsoft Windows XP. While many developers prefer Linux, Windows XP is presently often the simplest choice for interfacing instruments because of the support provided by the instrument manufacturers. Second, we want to avoid excessive dependencies that could make it difficult to install the system and lead to bloated code; instead we prefer to make modular, optional, orthogonal features that can be used if the supporting libraries are available. The system must provide a simple means of specifying a GUI which does not require knowledge of the underlying GUI toolkit and must be easy to understand, even for an inexperienced programmer. Critically, it must be possible for the GUI to continue to function even when data acquisition or any communication with instruments is in progress. In most cases, this requires a multithreaded or multiprocess system. Yet, we do not want to require the user-programmer to have a thorough understanding of multithreaded programming. So, the system must handle the multithreaded or multiprocess structure transparently.

Similar to some commercial software, we call the code that runs within Pythics a virtual instrument (VI), and in general multiple VIs may be running within a single Pythics application. We require that there be a mechanism for sharing objects between VIs, for example for sharing data between VIs or synchronizing some operation. An additional requirement is that the system must be robust. It should be possible to terminate one VI without affecting the others, or at least to continue without having to restart Pythics.

Features

We are willing to accept some loss in GUI design flexibility for the simplicity of programming Pythics that we require. We looked for a means of specifying a GUI that would be simple, would layout in a usable manner across a wide variety of screen and window sizes, and

would grow to handle a VI GUI which might gradually increase in complexity over time as a VI evolves. We found inspiration for a solution in the layout of hypertext markup language (HTML) used in web browsers, which has proven to be remarkably flexible over time. While HTML is primarily oriented towards the layout of text, we primarily require the layout of GUI elements. Fortunately, we found adequate flexibility in extensible hypertext markup language (XHTML), a markup language similar to HTML which also conforms to extensible markup language (XML) syntax requirements. By following the stricter XHTML syntax, the implementation of the Pythics layout engine can be less forgiving and thus simpler than a modern web browser. Furthermore, we only support a minimal subset of XHTML as needed for layout of simple lines or tables of GUI elements, and a small number of cascading style sheets (CSS) attributes to customize the appearance of the GUI. In practice, we make extensive use of the `object` XHTML element to insert GUI elements.

Using an XHTML-based system for GUI layout already constrains many features of our layout engine. Elements (e.g. a button) are generally fixed in height and may or may not expand in width as the enclosing window is resized. As more elements are added, the GUI will generally have to be scrolled vertically for access to all the GUI elements, while horizontal scrolling is avoided unless necessary to fit in the minimum horizontal size of the GUI elements. It may seem surprising that a layout system based on a system designed for text (HTML) would be general enough to specify a GUI, but the adaptability of the world wide web to new functionality demonstrates the flexibility of HTML.

In a large and evolving experiment, it is typical to have an ever-growing number of VIs for data acquisition and control. To organize multiple VIs running within Pythics, we again borrow a feature from web browsers: tabbed browsing. In Pythics, each VI runs within its own tab, all contained within the same Pythics window. This help avoid confusion if many windows are open, and in particular if multiple instances of Pythics are running, a scenario which is both possible and, in some cases, desirable.

Pythics maintains a strict separation between the GUI layout specification and the VI functionality by separating the code for a VI into two or more files. The first file contains the XHTML specification of the GUI and additional objects which trigger loading of the remaining files. These files are pure python, loading any additional libraries which may be needed and defining the functions triggered by GUI callbacks.

Multiprocess Structure

The multiprocess structure of Pythics is dictated by the GUI structure and the requirements of the VIs. First, many GUI toolkits (including wxPython) place

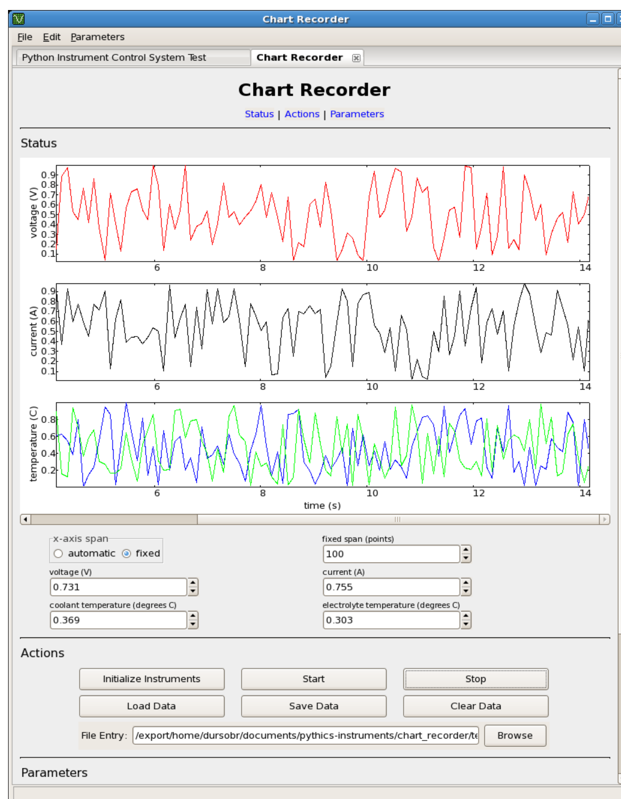

Figure 1: Screenshot of a Pythics session, showing a chart recorder VI.

the GUI for all VIs within a single Pythics application to reside in the same thread of the same process. Since each VI may have its own tasks to work on while the GUI is to remain functional, each VI must also have its own worker thread or process. An early version of Pythics used a separate thread for each VI, but we found that the system was excessively fragile, with a fault in a single VI sometimes ending in a abandoned, but often still running thread, because of the limited methods to safely kill a thread in Python. The need for a more robust system and the appearance of the multiprocessing module in Python 2.6 lead us to a multiprocess design, where each VI has its own worker process which handles data acquisition and data processing. Furthermore, the use of multiple processes avoids the Python global interpreter lock (GIL), which could limit the benefits of using multiple threads alone.

For simplicity, we provide only a single thread in a single process for the work of each VI, although each VI is in principle free to use and manage multiple processes or threads as needed. Since wxPython restricts the GUI to a single process and thread, we are pushed towards one particular process structure: each VI has its own worker process which communicates with a single shared GUI process. If multiple GUI processes are desired, multiple instances of Pythics can be running simultaneously, although there is no support for communication between VIs in different Pythics instances. For the rest of the description here, we assume there is only a single running Pythics instance.

The multiprocess structure of Pythics is illustrated in

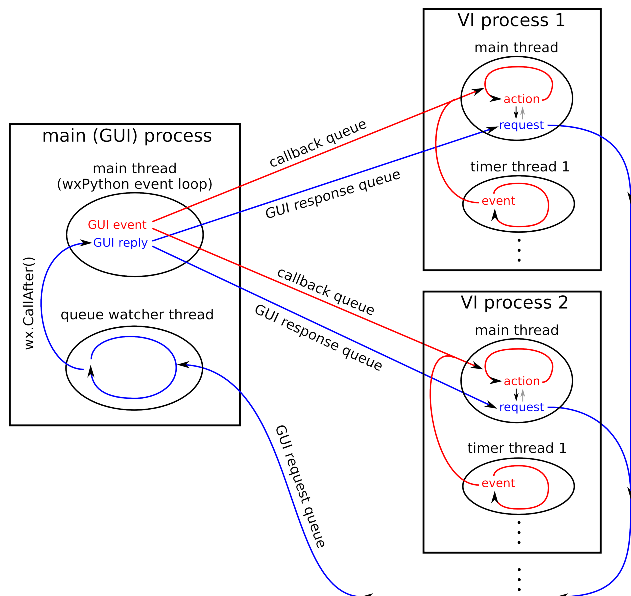

Figure 2: *The multiprocess structure of Pythics.*

Figure 2. Each VI worker process communicates back and forth with the single GUI process by placing messages in queues. There are three routes for communication between the worker processes and the GUI process. First, the GUI may want to trigger a callback function in response to an action, for example a button being pressed. Each worker process has a queue for callback requests, and each worker process waits for a request on this queue when it is idle. Next, a worker process, typically within a callback function, may want the GUI to respond in some way or may want to get some other information from the GUI, for example reading the value of a parameter from within a box in the GUI panel. The GUI process has a thread which watches a single queue where requests from all worker processes are funneled. This thread waits for a request to be appear in the queue, and when one arrives, the thread transfers the request to the wxPython main thread with a call to `wx.CallAfter`. The GUI process then answers the request by placing the response in the response queue for the requesting worker process. Finally, the worker process receives the response and continues with its task.

In practice, the Pythics user-programmer does not have to worry about the inter-process message passing and queue structure. The execution of callback functions is handled automatically. Requests to and responses from the GUI process are handled by GUI control proxy objects, essentially encapsulating all of the multiprocess and queue structure. We will go into further details in later sections.

GUI Specification

The GUI layout engine, which allows the arrangement of GUI elements within a VI GUI tab, was inspired by, and originally based on, the wxPython HtmlWindow. This wxPython widget can layout HTML text mixed with GUI elements such as buttons. We eventually wrote a replacement XHTML renderer for Pythics so it could gracefully handle exceptions in the process of creating controls. If a Python exception is encountered while creating a GUI element in our renderer, the element is replaced by a red box which displays the exception, making debugging much easier.

The GUI specification or markup language is XML, and is essentially a subset of XHTML. The style of the text elements, the background color, etc. are specified with a simplified cascading style sheets (CSS) system, also similar to typical XHTML or HTML files. Our specification must have many GUI elements that are not common for XHTML, so we make extensive use of the XHTML `object` element. Within the object element, we use the `classid` attribute to specify the GUI element class (e.g. `Button`), and the `id` attribute to specify a name for the GUI element which becomes the name of the object in Python. Other parameters for the GUI elements are specified with `param` elements within the object element.

An example GUI, here for a “Hello World” VI in Pythics, illustrates how a GUI is specified:

```
<html>
    <head><title>Hello World</title></head>
    <body>

        <h1>Hello World</h1>

        <object classid='Button' width='200'>
            <param name='label' value='Run'/>
            <param name='action' value='hello_world.run'/>
        </object>
        <br/>

        <object classid='TextBox' id='result' width='200'>
        </object>
        <br/>

        <object classid='ScriptLoader' width='100%'>
            <param name='filename' value='hello_world'/>
        </object>

    </body>
</html>
```

In the `<head>` element, the `<title>` element gives a title to the VI which appears on the tab containing the VI in the Pythics GUI. There is then a printed title within the `<h1>` element, followed by the primary GUI elements, a button and a text box. Note the `param` element with `name='action'` within the button object. This specifies the callback function which is executed when the button is pressed. We have also given the text box a name, through its `id` attribute. We will use this to access the text box for displaying a message later.

Without the final `object` element, Pythics would not know where to find the callback function specified by the button object. The `ScriptLoader` object functions similar to the Python `import` statement, loading a file which typically contains callback functions. This object can be used multiple times to import multiple files. This object also shows up in the GUI as a line of

©2009, B. D'Urso

text to show that it is there. The resulting GUI, which appears within a tab in Pythics, is shown in Figure 3.

Figure 3: *Screenshot of example GUI window for the "Hello World" example.*

Other GUI elements that are available within Pythics include images, many kinds of buttons, file dialogs, numeric input/output boxes, spreadsheets, sliders, embedded Python shells, and plots (using wxPython or matplotlib).

Callback Functions

Our "Hello World" example doesn't yet have any functionality. To make it respond to pressing the button, we need to introduce callback functions within Pythics. We want the structure of callback functions in Pythics to be as unconstrained as possible, but they do need some way to access the GUI elements. We do not introduce a formal event object, since this adds complexity that is unnecessary in most VIs. If a callback function needs information about the event that triggered it, it should get that information by addressing the appropriate GUI element.

In order give callback functions access to the GUI elements, we introduce a simple calling convention. The callback function is called with all of the GUI elements which have an `id` attribute in the XML specification file, which we will call named elements, as keyword arguments. There are no other arguments passed to callback functions. Thus, a callback function could receive all the named elements as a dictionary using the Python `**kwargs` syntax, or it can separate out the named elements it will use as individual arguments and group the unused named elements together with the `**kwargs` syntax.

The completion of our "Hello World" example clarifies the use of callback functions. Here is the entire Python file for our example:

```
def run(result, **kwargs):
    result.value = "Hello, world!"
```

Note that the callback function receives the one named element, `result`, as a keyword argument, and any other named elements (there are none in this case) would be grouped into `kwargs` as a dictionary. Thus, more GUI elements can be added without breaking this callback function. We display the message "Hello, world!" within the text box in the GUI with a simple call to to the GUI element proxy, by setting its `value` attribute. Clearly, no knowledge of the multiprocess structure and message passing within Pythics is needed. In most cases, these proxy objects have a fairly high level interface within Pythics, so most exchanges with the GUI within the callback functions require only a small number of commands using the GUI element proxies.

Other than our callback function calling convention, the files that contain the callback functions are standard, pure Python files. Additional functions can be defined, and packages can be imported as needed. Pythics itself does not require any imports within the callback function files, making it easy to maintain a clean namespace.

Additional Features

There are several other features of Pythics which make it more useful for making functional VIs. Most of these are implemented as additional or modified GUI controls, so they fit easily into the framework described above. These include:

- The values entered into a VI's GUI controls can be stored in a parameters file. This includes support for default parameters, which are automatically loaded when a VI is started. It also allows for alternative sets of parameters which can be saved and recalled.
- Global namespaces are available for sharing data between VIs. This uses the Python multiprocessing `Namespace` object, so almost arbitrary objects can be shared.
- Optional display of images in the GUI with shared memory instead of message passing. With one option in the GUI specification, the image display proxy will pass data to its control using shared memory to allow for much more rapid updates. In the current implementation, the maximum size of the image must be given in the GUI specification if shared memory is used.
- Timer elements are available for executing a function at regular intervals in time. To the user-programmer, a timer looks like any other GUI element, such as a button, which calls its action callback at regular intervals. Timers actually operate completely in the worker process through their proxy object, but within a separate thread from the callback function execution. This structure eliminates the overhead of message passing for frequently executed callbacks when no GUI interaction is needed. Timers are implemented with Python threading Event objects, so they can be interrupted at any time. A single VI may have many timers.

- There is a SubWindow element which can embed an entire second VI within a box inside the GUI of the first VI, allowing for modularization of GUIs as well as functions. There are methods to pass data between the fist and second VI, although both run in the same thread and process.

Future

Pythics is still evolving very quickly, and we anticipate continuing to add features, primarily driven by the needs of our laboratory or the needs of other groups that may start to use Pythics. The highest priority item is documentation, which will be necessary for the Pythics community to grow. As a more technically challenging direction, we may pursue operating Pythics over a network, with the GUI and worker processes on different machines communicating over the network. There is already support for this in the Python multiprocessing module.

We plan continued expansion and improvement in the graphics display controls available within Pythics. One significant required feature is the ability to plot data in near real time, as it is being acquired. Simple, fast, line plots are available now in Pythics, but more complex plots require matplotlib, which is often too slow for keeping up with incoming data when most of the available processor time is used acquiring and processing data. We are investigating alternative plotting libraries which could be easily used within wxPython. Another graphics feature we would like to add to Pythics is simple three-dimensional display capability, similar to VPython [vpython]. This could make Pythics an attractive system to use for teaching introductory computational physics classes, both for data acquisition and for simple models and calculations.

We also plan to improve the GUI layout design tools available for Pythics, to make it easier to write the XML GUI specification. One possibility would be to have a graphical editor for the GUI interfaces, and perhaps a graphical HTML or XML editor could be adapted to this purpose. An alternative possibility, which is inspired by the reStructuredText format [rest] commonly used for Python documentation, would be to have an "ASCII art" layout description. In this case, we would develop a more readable text format for specifying and placing GUI elements, then translate that format to the XML format actually used by Pythics. In principle, there is no reason not to support both of these options.

We currently release Pythics under the GNU General Public License (GPL), and it is free to download [pythics]. We hope to attract the interest of other researchers and students, and that they will contribute to the success of Pythics. Ultimately, Pythics could become a center for free exchange of code, instrument drivers, and data analysis tools.

References

[numpy] T. Oliphant et al., *NumPy*, http://numpy.scipy.org/

[scipy] E. Jones, T. Oliphant, P. Peterson, et al. SciPy http://www.scipy.org/

[sympy] Development Team (2008). SymPy: Python library for symbolic mathematics http://code.google.com/p/sympy/

[sage] W. Stein et al., *Sage Mathematics Software*, http://www.sagemath.org/

[pyvisa] http://pyvisa.sourceforge.net/

[pil] http://www.pythonware.com/products/pil/

[wxpython] http://www.wxpython.org/

[matplotlib] J.D. Hunter. *Matplotlib: A 2D graphics environment.* Computing in Science and Engineering. (2007) 9: 90-95. http://matplotlib.sourceforge.net

[vpython] http://vpython.org/

[rest] http://docutils.sourceforge.net/rst.html

[pythics] http://code.google.com/p/pythics/

©2009, B. D'Urso