CsPyController Programmers' Manual

Martin Tom Lichtman

2015 July 7

Contents

1 Introduction

The CsPyController software was written by MTL to run experiments and collect data for the AQuA(*Atomic Qubit Array*) project, however it was designed with extensibility in mind. This *Programmers' Manual* explains how to extend CsPyController by adding more **instruments** or **analyses**.

A separate *Users' Manual* explains the basic functionality of CsPyController. While successful use of CsPyController requires some knowledge of Python syntax, using the software as described in the *Users' Manual* requires little more than being able to write, for example, `a = 5`, or, `arange(10)`. This *Programmers' Manual* assumes familiarity with the use of CsPyController at least at the level of the *Users' Manual*. However, it also assumes at least a moderate degree of skill in object-oriented programming in Python.

Furthermore, while this manual explains the necessary features of a new **instrument** or **analysis**, it is left to the reader's creativity to invent new code that is powerful and useful.

2 GIT Version Control

As detailed in the *Users' Manual*, the CsPyController code is stored in, and can be cloned from, a GIT repository on the `hexagon`. You will need to be familiar at least with *branching, commiting*, *pushing* and *pulling* in GIT. A GIT primer is available on the Saffmanlab Wiki, and much more info is available on the web, particularly at `git-scm.com` and `stackexchange.com`. The main stable branch is called `master`. Always pull `master` before beginning your work to make sure your have the lastest version. You should never make edits in `master`. It is fast and resource-cheap to make new branches in GIT, so branch early and often. Make a new branch for every new idea that you try. Make frequent commits to your new branch, and push to the server to make sure your work is backed up.

The goal for all new branches should be to eventually merge them back into `master`, not to create a whole separate version of CsPyController for your project. New **instruments** and **analyses** that you program may eventually be useful to others, and they should be programmed with this in mind. Also consider that others may *not* want to use any particular **instrument** or **analysis**, and so they should always have `enable` flags that allow a particular piece of code to be ignored and consume no resources.

When your branch is mature enough to be both useful and stable, do not merge it into `master` yourself. Instead, make a *pull request* to whomever is in charge of CsPyController development (MTL until September 2015).

3 Tools

The author finds the *PyCharm* editor invaluable for programming in Python. A free *community edition* is available. It does a large amount of syntax and code flow checking for you on the fly, highlights potential errors, and it indexes the structure of your code so you can easily find the piece of code you are interested in.

4 Code Structure

4.1 Top-level Classes

The execution of CsPyController begins in `cs.py`, which does little more than create the GUI environment and assign an instance of `aqua.AQuA` to the GUI and vice-versa. `aqua.AQuA` is a subclass of `experiment.Experiment`. An instance of `experiment`, can be thought of as the master object that has complete knowledge of all the various components of the software apparatus. `experiment.Experiment` defines all the methods which control experiment flow and looping through experiments, iterations and measurements. `aqua.AQuA` catalogs the various instrument and analysis code that is available, and defines the evaluation and update order of those pieces. For example, `aqua.AQuA` has an instance of `andor.Andor` , an `Instrument` for the Andor camera. It also has an instance of `andor.AndorViewer`, which takes care of displaying new images from the Andor camera.

Once you program your **instrument** or **analysis** as a new class, you will usually need to add an instance of that class to `aqua.AQuA`. `Instruments` can be nested, and so in some cases you will not want to add your new **instrument** to `aqua.AQuA`, but instead to a lower class. For example, the `LabView.LabView` `TCPInstrument` handles communication for, and acts as a container for, several sub-instruments. So `LabView.LabView` has instances of `HSDIO.HSDIO`, `AnalogOutput.AnalogOutput`, and `AnalogInput.AnalogInput` (amongst others).

The appropriate places to add references to an `Instrument` are slightly different from an `Analysis`, and all the necessary references will be detailed in their respective sections of this document.

4.2 `Prop`

A `instrument_property.Prop` is the workhorse class that handles:

- evaluation with respect to the defined **constants**, **independent** and **dependent variables** namespace
- saving and loading settings

Most, if not all, classes that you define will inherit from `Prop`. For example, `Experiment`, `Instrument`, and `Analysis` are all subclasses of `Prop`. This allows their evaluation and save/load behavior to be standardized, called at the appropriate time. As long as you follow certain conventions, this allows you to program extensions to the code without worrying about these important processes.

Every `Prop` has at least the following instance variables:

- `name`: a name that gives it a unique path in the `property` tree (i.e. it does not have to be globally unique, just unique amongst its siblings)
- `description`: some helpful information about this particular `Prop` instance (such as why it was set to a particular value, or what units its value has)
- `experiment`: a reference to the top-level `Experiment` instance

Generally these three instance variables will be defined when the `Prop` is constructed, for example with a call to `super(Prop, myProp).__init__(name, experiment, description)`

Furthermore, every `Prop` also has at least the following instance variables:

- `properties`: a list of the instance variable names that should be evaluated (if they have such behavior) and saved. The order in which the `Prop`s are evaluated is determined by this list, which may be important. To add to the `properties` list, be sure to denote the variable names as strings, not as actual Python objects. Also, you will almost always want to append to the `properties`, instead of overwritting them, so that any `properties` from the parent class are preserved. For example:

  ```
  self.properties += ['clockRate', 'units']
  ```

- `doNotSendToHardware`: a list of items that are also in `properties`, but that you do not wish to be processed by the `Prop.toHardware()` method, which creates XML code for transmitting to some TCP instrument server. Adding items to this list follows the same syntax as `properties. For example:`

  ```
  self.doNotSendToHardware = ['description']
  ```

When a `Prop` is evaluated, CsPyController iterates through the `properties` list and attempts to evaluate each item. The items in `properties` do not have to be instances of `Prop`. However, if you do include `Props` in `properties`, you can created nested trees of `Props`. Furthermore, when the settings are saved to HDF5 files, these nested trees are preserved in the HDF5 hierarchy. This is how many of the CsPyController `Instruments` organize their settings.

4.2.1 `EvalProp`

A `Prop` knows how to save/load its `properties`, and when a `Prop` is evaluated it knows how to go through its `properties` and try to evaluate them. However, it still does not know *how* to actually evaluate itself with respect to equations or functions and the like. For this, we have the `instrument_properties.EvalProp` class, and several of its subclasses `StrProp`, `IntProp`, `RangeProp`, `IntRangeProp`, `FloatRangeProp`, `FloatProp`, `BoolProp`, `EnumProp`, `Numpy1DProp` and `Numpy2DProp`.

In addition to all the workings of a `Prop`, each of these has a `function`, and a `value`. The `function` is a string which holds Python syntax code that will be evaluated in the namespace of the **constants**, **independent** and **dependent variables**. The evaluation is checked to make sure it results in the correct `type`, and in some cases within the correct range, and then is stored in `value`.

The different subclasses of `EvalProp` are generally what you will use for `Instrument` and `Analysis` settings when you want users to be able to use variables there. More often than not you might as well enable this behavior, as opposed to static settings, as some future user might want to scan a setting in a way you did not expect.

4.3 GUI

4.3.1 Enaml

The CsPyController uses the `enaml` package to create the GUI. For reference on *Enaml*, be sure to refer to the *Nucleic* documentation at `http://nucleic.github.io/enaml/docs/` or the source code at `https://github.com/nucleic/enaml`, and not the older documentation or code from *Enthought*. The GUI is defined using a heirarchical (i.e. nested) syntax in `cs_GUI.enaml`. The syntax for the *Enaml* file is mostly Python syntax, with several added operators (and some restrictions). In order to make the settings on your new **instrument** or **analysis** accessible to the user, you will have to first define the appropriate GUI widgets, and then link them to the backend instance variables that represent your **instrument** or **analysis**.

4.3.2 Make a new window

Usually you will create a new window to display your **instrument** settings or **analysis** results or graphs. To do this, first define the new `Window` widget.

4.3.2.1 Instrument window

For example:

```
enamldef CameraWindow(Window):
    attr camera
    title = 'Groovy EMCCD Camera'
    Form:
        Label:
            text = 'enable'
        CheckBox:
            checked := camera.enable
        Label:
            text = 'scan mode'
        SpinBox:
            value := camera.scan_mode
            minimum = 1
            maximum = 3
    EvalProp:
        prop << camera.EM_gain
    EvalProp:
        prop << camera.cooling
    EvalProp:
        prop << camera.exposure_time
    PushButton:
        text = 'take a picture'
        clicked :: camera.take_one_picture()
```

In this example we see several new features. First the `enamldef` statement, which functions much like a `class` declaration, but signals the *Enaml* parser that this defines a new GUI object called `CameraWindow`. Here `CameraWindow` is defined as a subclass of `Window`. Merely defining `CameraWindow` does not actually create one, but we can create as many instances of it as we like, which will be explained below.

Next `attr camera` is how the instance variable `camera` must be declared. Here `camera` will store a reference to an instance of an `Instrument` which contains all the information and functions for controlling a camera.

The layout of the `Window` and its sub-widgets is controlled using a nested syntax. For example the `Window` contains `a Form` (an invisible container with two column layout), which in turn contains `Label` and `CheckBox`. These are base widgets from the `enaml` package. The default layout for *Enaml* widgets is usually adequate, but you may fine tune all the layout and behavior as described in the *Enaml* docs.

There are several assignment operators that are unique to *Enaml*. First we see `text = 'enable'` which is a simple one-time assignment of the string `'enable'` to the `text` field of the `Label`. Next we see `checked := camera.enable`, where the `:=` operator denotes a two-way synchronizaton. Any changes to `camera.enable` (a True/False boolean variable) will update the checked/unchecked state of the `CheckBox`. At the same time any time the user checks/unchecks the `CheckBox` causing its `checked` state to change, the value of `camera.enable` will change on the backend. This is one of the best features of *Enaml* that makes it easy to link up variables on the GUI and backend. `camera.enable` is an example of a setting that does not respond to equations (it is a `Bool`, not an `EvalProp`).

The Form also contains another Label and a SpinBox. The SpinBox has its value synced to the variable camera.scan_mode, which is another example of a setting that does not take equations (it is an Int, not an IntProp). The minimimum and maximum arguments define the available range of the SpinBox.

Then we see the EvalProp widget, which is a useful custom widget defined in cs_GUI.enaml, that links to an instrument_property.EvalProp, displays the EvalProp.name, gives a place to enter the EvalProp.description and EvalProp.function, and displays the evaluated EvalProp.value. This works with any kind of EvalProp, be it an IntProp, StrProp, or FloatProp, etc. Here we see how the CsPyController backend has made it easy for you to handle EM_gain (an IntProp), cooling (a BoolProp), and exposure_time (a FloatProp) all using the same code. The function field will highlight in red if it does not evaluate to the correct type, making it easy to find user errors. A placeholder in the function field shows the expected type or range if the field is left blank.

You may of course define your own custom widgets to handle your data structures in new ways.

Within the EvalProp widgets, we see the use of the subscription operator <<. This operator is a one-way subscription, so that whenever, for example, the camera.EMGain object changes identity (such as during a settings load), the GUI will update (but not vice-versa). The operator >> is also available which is a one-way broadcasting that will update the backend variable when the GUI updates, but not vice-versa.

Finally, we have clickable button defined using PushButton. We see the :: operator, which does not pass a value, but instead defines an action to be taken. In this case, when the clicked state of the PushButton changes, the method take_one_picture() is called.

4.3.2.2 Analysis Window A Window for an Analysis is created in much the same way as for an Instrument. For the Analysis you will usually want to have more ways to display data and statistics. For example:

```
enamldef PictureViewer(Window):
    attr viewer
    title = 'Picture Viewer'
    MPLCanvas:
        figure << viewer.fig
    Label:
        text << viewer.text
```

In this example the attribute viewer would be linked to an instance of, for example an analysis.AnalysisWithFigure. The MPLCanvas is a widget which allows the display of any matplotlib figure. The GUI display is only updated whenever the identity of viewer.fig is changed as specified by the << subscription operator. (A simple redraw is not enough, but these mechanics are handled for you if you use the AnalysisWithFigure class.) Finally the Label widget is used to display dynamic from viewer.text by using the << subscription operator, unlike in the CameraWindow example above where the Label text is static.

4.3.3 Add the Window to the list

In order so that the user can call up your new Window by selecting it on the combo box in the MainWindow (the one that opens when you launch cs.py), it must be added to the window_dictionary used in Main. Toward the bottom of cs_GUI.enaml you will find the definition of window_dictionary as a Python dict. Add your Window to the list with the following syntax:

```
'Groovy Camera Setup': 'CameraWindow(camera = main.experiment.groovy_camera)',
```

or

```
'Groovy Camera Display': 'PictureViewer(analysis = main.experiment.
    picture_viewer)',
```

The first element is a string key that is used as the display text in the combo box. The second element is a string, which when evaluated is a call to the constructor for your new `Window` subclass. The constructor is passed values for all the `attr` attributes defined in the `Window`. In `main.experiment.camera`, first `main` refers to the `MainWindow`, which knows about `experiment` which is your instance of `experiments.Experiment`, which finally has an instance of an `Instrument` named `camera`. (We will cover creating this backend instance below.) Similarly, the `PictureViewer` instance is passed a reference to an instance of an `Analysis` named `picture_viewer` on the backend. This list is automatically sorted alphabetically, so position is not important. However, be sure that each line except the last ends with a comma.

5 atom

Use of `enaml` for the GUI requires that we use the `atom` package to support the variable-to-GUI synchronization and event observation. This offers both advantages and additional headaches. Any class whose variables we would like to sync with the GUI, must descend from the class `atom.api.Atom`. To achieve this, we make `Prop` a subclass of `Atom` so that every `EvalProp`, `Instrument` and `Analysis` already has this inheritance.

One disadvantage (although it provides a performance boost) of using an `atom.api.Atom` is that you cannot declare instance-wide variables on the fly, they must be declared at the top of the class definition. What this means is that you cannot state:

```
from atom.api import Atom
class MyClass(Atom):
    def myMethod(self):
        self.x = 5
```

Instead you must declare `x` using, for example:

```
from atom.api import Atom, Int
class MyClass(Atom):
    x = Int()
    def myMethod(self):
        self.x = 5
```

Here the type used is `Int`, which is not the basic Python `int` but instead is an `atom` class which implements error checking to make sure that only integers are assigned to `x`. `atom` also has classes for `Bool`, `Float`, `String` and many other types as well as customizable wrappers. It is required to use these `atom` types instead of basic Python types. If you would like to synchronize one of these backend variables with the GUI, then the declared type of the variable must match the type expected by the GUI widget.

There are often variables that you would prefer not to have to both to declare, perhaps because they will never be used in the GUI, or they may have some unique type that is not supported easily by `atom`. For these, use the `Member` type which is the most general that `atom` allows:

```
from atom.api import Atom, Member
class MyClass(Atom):
    x = Member()
    def myMethod(self):
        self.x = some_weird_type()
```

The synchronization tools of `atom` are activated automatically by using the `:=`, `<<`, or `>>` operators in the `.enaml` file. There are further ways to leverage atom to perform actions on variable changes, such as the `@observe` decorator. Used here in an example from `AnalogInput.py`:

```
from atom.api import observe, Str
class MyClass(Atom):
    list_of_what_to_plot = Str()
    @observe('list_of_what_to_plot')
    def reload(self, change):
        self.updateFigure()
```

In this example, whenever the string `list_of_what_to_plot` is changed, then the `updateFigure()` method is called.

Info on the `atom` package is available at `https://github.com/nucleic/atom`, however the most complete information on `atom` is actually available in the `enaml` examples.

6 Instrument

6.1 Create a new Instrument

The class `cs_instrument.Instrument` is the base class to use to describe a new **instrument**. First, create a new `.py` file, in this case `groovy.py` to hold your class. At the top of the class, you will almost always want some fashion of the following import lines:

- Usually it is a good idea to set default divison to be floating point, not integer math (so that $1/2 = 0.5$ instead of $1/2 = 0$).

  ```
  from __future__ import division
  ```

- Don't use print statements, instead use `logger.info()`, `logger.debug()`, `logger.warning()`, `logger.error()`. These will handle time stamping and saving to the log.txt file.

  ```
  import logging logger = logging.getLogger(__name__)
  ```

- Your code should implement try/except error catching blocks, and give description errors sent to the `logger` commands. When the error is bad enough that the experiment execution should be paused, use `raise PauseError`.

  ```
  from cs_errors import PauseError
  ```

- You will probably need some `atom` types:

```
from atom.api import Member, Int, Bool, Str, Float
```

- Numerical functions are very often useful:

```
import numpy as np
```

- You will probably want some `EvalProp` types:

```
from instrument_property import BoolProp, IntProp, FloatProp, StrProp
```

- Finally, you will need the CsPyController base class for an **instrument**:

```
from cs_instruments import Instrument
```

Now define your new class. In this simple example we will create a new camera class that uses a `DLL` (dynamic link library) driver to send commands to the hardware. This is very hardware specific, and you might use some other means to communicate with the hardware. Use of the `TCP_Instrument` to communicate with a separate **instrument server** is shown later. The main points to absorb here are how the variables are set up to coordinate with the GUI frontend described above.

```
from ctypes import CDLL

class GroovyCamera(Instrument):
    EM_gain = Member()
    cooling = Member()
    exposure_time = Member()
    scan_mode = Int(1)

    dll = Member()
    current_picture = Member()

    def __init___(self, name, experiment, description='A great new camera'):
        # call Instrument.__init__ to setup the more general features, such as
            enable
        super(GroovyCamera, self).__init__(name, experiment, description)

        # create instances for the Prop properties
        self.EM_gain = IntProp('EM_gain', experiment, 'the electron multiplier
            gain (0-255)', '0')
        self.cooling = BoolProp('cooling', experiment, 'whether or not to turn
            on the TEC', 'True')
        self.exposure_time = FloatProp('exposure_time', experiment, 'how long to
             open the shutter [ms]', '50.0')

        # list all the properties that will be evaluated and saved
        properties += ['EM_gain', 'cooling', 'exposure_time', 'scan_mode']
```

```
    def initialize(self):
        """initialize the DLL"""
        self.dll = CDLL("camera_driver.dll")
        super(GroovyCamera, self).initialize()

    def take_one_picture(self):
        """Send a single shot command to the camera.
        Use a hardware command, which might be call to a DLL, for example
        This ficticious example returns the picture as an array, which is
            assigned to self.current_picture.
        """
        if not self.isInitialized:
            self.initialize()
        if self.enable:
            self.current_picture = self.dll.take_picture_now()

    def update(self):
        """Send the current settings to hardware."""
        self.dll.set_EM_gain(self.EM_gain.value)
        self.dll.set_cooling(self.cooling.value)
        self.dll.set_exposure_time(self.exposure_time.value)
        self.dll.scan_mode(self.scan_mode)

    def start(self):
        """Tell the camera to wait for a trigger and then capture an image to
            its buffer."""
        self.dll.wait_for_trigger()
        self.isDone = True

    def acquire_data(self):
        """Get the latest image from the buffer."""
        self.current_picture = self.dll.get_picture_from_buffer()

    def writeResults(self, hdf5):
        """Write the previously obtained results to the experiment hdf5 file."""
        try:
            hdf5['groovy_camera/data'] = self.current_picture()
        except Exception as e:
            logger.error('in GroovyCamera.writeResults() while attempting to save
                 camera data to hdf5\n{}'.format(e))
            raise PauseError
```

Let us go through the parts of this `Instrument`. First, we needed to declare all the instance variables, because `Instrument` is a `Prop` which is an `Atom`. The `EM_gain`, `cooling`, `exposure_time` and `scan_mode` variables all require this because they are synchronized with the GUI. The `dll` and `current_picture` variables don't have a purpose in being declared, but it is required to declare all instance variables within an `Atom`.

The `__init__` method is called when an instance of `GroovyCamera` is constructed, and takes in the `name`, a reference to the `Experiment`, and an optional `description` with a default description of `'A great new camera'`. We then immediately pass on most of this information to the parent class using a `super` command, so that `Instrument` can handle this info with the default behavior. Next we create instances for each `Prop` that this class contains. `IntProp`, `BoolProp` and `FloatProp` each take the same arguments of `(name, experiment, description, initial_function_string)` with the only difference being in the type that the function string will resolve to. It is necessary that the name parameter be a string that matches the actual variable name, and so self.EM_gain is given a name of 'EM_gain'. The reference to experiment will be passed in when we instantiate this class, which will be shown later. The description should contain useful text specific to this variable, such as the units, or why a particular value was chosen. Finally, the initial_function_string can contain variables and equations, but must evaluate to the correct type. Note that this is a string, and so we write '50.0' and not 50.0. If a StrProp were used, the string must evaluate to a string, so you could write '''hi''' or 'str(5)' for example. Finally we must indicate that all these variables should be evaluated and saved, by adding them to the properties string. Note that once again this is a list of strings, and so we write ['EM_gain', 'cooling', 'exposure_time', 'scan_mode'], and not [EM_gain, cooling, exposure_time, scan_mode]. Also note that we say properties += and not =, because we do not want to lose any properties from the list that were assigned in Instruments.__init__(), which in this case includes the enable variable.

Note how we used the enable variable in the GUI example, and yet it is not shown in the code above (except in take_one_picture()). That is because enable is set up in the parent class and inherited without modification here. It is necessary to specifically check the enable variable in take_one_picture() because that is a custom method for this example. However, we do not check enable in initialize(), update(), start(), acquire_data() or writeResults() because CsPyController takes care of checking that for all Instruments before these methods are called in Experiment.

The initialize() method is called for all Instruments before they update, but only once (or as long as self.isInitialized == False). This is a good place to do initial one-time setup of the instrument. In this example we use this method to setup the DLL. We end with a call to Instrument.initialize() via super, which in this case will just set self.isInitialized = True for us, so that initialize will not be called again.

The take_one_picture() method is something that we set up in this example to be called by a PushButton on the GUI. This method is therefore executed in the GUI thread. If the DLL call is very slow, it would be necessary to have this method spawn a different thread which would then make the DLL call, so as to not cause the GUI to hang. We check isInitialized before proceeding with this method because initialize() may not have been called yet, if this button is pressed before the first experiment is run.

The update() method is called at the beginning of every **iteration**, after everything has been evaluated with the newly iterated variables. The job of update is to send the updated settings to the hardware. This is very hardware specific, and in this example we do so with a series of calls to the DLL. Note how we pass EM_gain.value, cooling.value, and exposure_time.value, and not EM_gain, cooling and exposure_time, because we do not want to pass the whole EvalProp instance, just the relevant evaluated value. For scan_mode we can just pass scan_mode because it is a primitive type, not an EvalProp.

The start() method is called at the beginning of every **measurement**. If this instrument's timing is to be triggered by some other piece of hardware, like for example an HSDIO digital output channel, then start() should just set the instrument up to wait for the trigger, which is what we have done here. If this instrument will be internally timed, then just go ahead and tell it to proceed with a **measuremen**t. The **measurement** will not end until all the instruments have self.isDone == True (or if the experiment timeout is reached). You can delay setting isDone to True until you have confirmation that the instrument has fired, for example by creating a watchdog thread which keeps checking the camera status or buffer and updates isDone only once that status changes. Or you can take the easier route that we use here, and simply trust that the camera will do its job if it gets the trigger, and that the HSDIO will not report isDone until it finishes its sequence and all the output triggers. So here we just set self.isDone = True right away.

Next, acquire_data() is called after all the Instruments reach the isDone state. This may not be necessary for all instruments, if for example the data was returned right away in start(). This method should store the data in an instance variable, where it can be used directly or accessed later for saving to the results file.

Finally, we have the writeResults() method. This method should save any data to the HDF5 file. A reference to the HDF5 node within this particular measurement (i.e. f[/iterations/#/measurements/#/data]) is passed in as hdf5. The reason that this method exists separately from acquire_data is that in some cases you may need to get back data from other instruments before deciding exactly how to process and save the data from this instrument. This separation is not always necessary, and so you could do the work of both `acquire_data` and `writeResults` in just `writeResults` and avoid having to create the `current_picture` temporary storage variable.

6.2 TCP_Instrument

The example in Section 6.1 supposed that we have access to a DLL that can be called directly from Python. This is not always the case, either because the hardware is running on a different machine from the CsPyController **command center**, or because it is only accessible from another language (as with .NET assemblies), or because it is simply easier to access through another language. The prefered way of handling these situations is to create a separate **instrument server** program. The **instrument server** handles the direct control of the hardware, but does not have any user interaction. Instead, all the user interaction is still done through the CsPyController **command center**. Communication between the two programs is done using TCP/IP messaging. The **command center** acts as a TCP client, and the **instrument server** acts as a TCP server. The client sends settings updates and measurement requests to the server, and the server returns data to the client.

Since this is a common paradigm, the `TCP_Instrument` class exists as a standardized way to implement this behavior. It is a subclass of `Instrument`, so it has all the behavior of `Instrument` and more, and can be used in its stead. The creation of the server at the other end is left to the programmer, although you can base your new server off the many examples in the CsPyController package: in Python (`box_temperature_server.py` or `TCP.CsServerSock`), for C# (`PicomotorServer`), in C++ (`Picam`), or in LabView (`PXI_server` and `DDS_server`).

The CsPyController standard for TCP message formatting is:

1. Every message starts with 4 bytes 'MESG'.
2. Followed by a 4 byte big-endian unsigned long integer (the `'!L'` format in Python's `struct.pack`) which indicates the number of bytes in the rest of the message. This allows messages up to 4.2 GB in length.
3. Followed by the rest of the message, of the previously indicated length.

Both server and client should check for this formatting. If the message fails to satisfy this format in any way, the an error should be raised, and the TCP buffers should be cleared. The client may retry communications as many times as deemed fit, after which the connection should be closed. The server should wait for another message, and if the connection is closed, it should reset and wait for new connections.

The standard formatting for the message content is to use XML to describe the settings or commands. This will not work for all situations, but the `Prop` structure provides an easy method to generate the XML for the settings. Every `Prop` has a `toHardware()` method which returns XML for that `Prop` with nested XML for all of its `properties`, and so on recursively. The `doNotSendToHardware` list for each `Prop` excludes certain `properties` from the XML. Calling `TCP_Instrument.toHardware()` is then an easy way to generate an XML message that contains all the settings for the instrument and everything it contains. This message can then be sent to the server using `TCP_Instrument.send(toHardware())` as will be described further below.

Let us now go through the addition behavior that `TCP_Instrument` provides over `Instrument`: First, varibles for `IP`, `port` and `timeout`. Here `IP` is a string which gives the `IP` address of the server, and `port` is the TCP port number that the server listens on. Generally it is okay to pick any unused port number above 9000. So far we have used 9000 (`PXI_server`), 9001 (`DDS_server`), and 9002 (`box_temperature_server`). You may have to allow these ports through the firewall.

The `open()` method creates a connection between the client and server. The `connected` status is monitored and the method is called automatically by `initialize` if necessary. The `openThread()` method starts a new thread to run `open` in, and is useful for creating a GUI button to manually open the connection, if desired. The `close()` method can be used to manually close the connection.

The `send()` method takes a message as its only parameter, and then takes care of all the formatting described above and sends it to the server. `send()` then waits for a response, which must as always be correctly formatted as described above. The returned message is then processed using `TCP.CsClientSock.parsemsg()`, which assumes the remaining message has the following format:

1. 4 byte big-endian unsigned long interger indicating the number of bytes in a name
2. a *name*
3. 4 byte big-endian unsigned long integer indicating the number of bytes in some data
4. some *data*

Note that this format is *inside* the message portion of the TCP message format given above, it is not instead of that format. This sequence can be repeated any number of times within the remaining message, so long as the format is observed. Furthermore, you could nest this format inside *data*, and recursively call `parsemsg()`, to create a data hierarchy. When the whole message has been parsed, `parsemsg` returns a `dict` of `{name1: data1, name2: data2, name3: data3, ...}`. This response data might be quite short in the case of a settings update (for example `{'error': 0}`), or it could be quite long and involved containing experiment data as the response to a send('<measure/>') request for a measurement. We do not use XML for returned data because it is not possible to reliably send binary data through XML (there will at some point be <, >, or / bytes which will confuse the message). This format allows arbitrary binary data to be returned with ease. The returned dictionary is stored as `self.results`. It is up to the programmer to decide what to do with this dictionary and to cast the returned *data* into the correct `type`.

The `update` method is overridden from `Instrument`, and by default calls `send(toHardware())`. This will send XML formatted settings to the hardware, but of course can be overridden to implement some other behavior.

The `start()` method by default just sets `isDone = True`. However if the hardware needs to signaled to start the measurement it would be done here, for example with a send('<measure/>') command.

The `acquire_data()` method is not overridden by `TCP_Instrument`, but here is where the `parsemsg` behavior will be very useful. You could implement your `acquire_data` to send a request for recent data, for example with send('<get_new_data/>'). The message returned by the server should contain the results of the **measurement**, which will then be stored in `self.results`. You can then do some custom behavior with this data, or you can wait until `writeResults` to store it to HDF5.

The `writeResults()` method is overridden to give a default behavior to handle the `self.results` dictionary. It simply iterates through `self.results` and attempts to store each item to `hdf5[name] = data`. You will want to override `writeResults()` to give more instrument-specific behavior. Since you know what data types to expect from the returned elements, it is good practice to cast the returned data to those types before saving to the HDF5 file, so that the HDF5 file stores it with the correct metadata indicating that type. For example, if returned data is a string, then no conversion is necessary. However, if the returned data is binary data for a numerical array, then you should cast the data to a list using `struct.unpack` with the correct data type, and then cast the list to a `numpy` array, and then `reshape` the array to the correct dimensions, before storing it to the HDF5. For most data from the **PXI_server** we return not just the data, but also separately the dimensions of the data, so that it can be `reshaped` correctly. Taking the time to store the data properly in the HDF5 file will be beneficial in the long run, making data analysis much easier and eliminating possible confusion over the data type, as opposed to the many problems inherent in storing an unknown binary format, or storing numerical data in strings.

6.3 `aqua.AQuA`: Instantiate your `Instrument`

You have now created a beautiful new `Instrument` or `TCP_Instrument`, but it is not actually used anywhere yet. For this, we add it to the `aqua.py` file. Here `aqua.AQuA`, which is a subclass of `experiment.Experiment`, contains a list of all the `Instruments` and `Analysis`. The original intention for `AQuA` is that each project would define a similar file which lists the specific `Instruments` and `Analysis` needed on that project. That is no longer the desired paradigm, and the best practice now is that there should only be one `aqua.py` file for the CsPyController package, with all the various possible `Instruments` and `Analysis` known to the Saffmanlab, and that enable/disable behavior should be used to eliminate resources required for unused `Instruments` and `Analysis`.

Since `AQuA` is an `Prop` we will use the same behavior we use to add `properties` to a `Prop`, with some additions. First import your new file. Just use the filename without `.py`, so for `groovy.py` use:

```
import groovy
```

Then, you must declare your instance variable at the top of the class:

```
class AQuA(Experiment):
    groovy_camera = Member()
```

Then instantiate the `Instrument` in `__init__()` and be sure to make the `name` match:

```
    def __init__(self):
        super(AQuA, self).__init__()
        # instruments
        self.groovy_camera = groovy.GroovyCamera('groovy_camera', self, 'Our
            awesome camera')
```

Still in `__init__()`, add the camera to `self.instruments`, which is how the `Experiment` knows which objects to try to `initialize()`, `update()`, `start()`, etc.:

```
        self.instruments += [self.groovy_camera]
```

An important note, is that the order in which the instruments are called is defined by this list. So if it is important, for example, to `start` the HSDIO digital pulse output last after all other instruments are told to wait for a trigger, then the HSDIO instrument must come last in this list. And finally, still in `__init__`, add the camera name to `self.properties` so that its settings will be evaluated and saved:

```
        self.properties += ['groovy_camera']
```

Remember that the order of `properties` determines the order of evaluation. Note that in `instruments` we use a reference to the variable itself as `self.groovy_camera`, while in `properties` we use a string of the name as `'groovy_camera'`.

6.4 Summary

And that's it! Now you can run the CsPyController **command center** starting with `python cs.py`, use the combo box select *Groovy Camera Setup* to bring up the `CameraWindow`, enter the settings. Then run the experiment to automatically: cause all the `EvalProps` to resolve their `functions`, `initialize` and `update` the `GroovyCamera`, `start` a measurement, `acquire_data` and finally `writeResults` to the HDF5 file. If that's all you need, and analysis will be done off-line, then you are done. However, if you want to display some of the data in realtime, then continue to the next section to create an `Analysis`.

7 Analysis

An **analysis** is the place to do some calculations on returned data, filter dat based on some criteria, or to make useful plots that show during the experiment. The `analysis.Analysis` class is a subclass of `Prop`, just as `Instrument` was, so you are already familiar with much of the mechanics that you will need to work with.

7.1 Methods

An `Analysis` has several methods which are called the automatically at the appropriate time. Each of these methods is passed a reference to an HDF5 node. Depending on the method, it is either a reference to the data for the **experiment**, **iteration**, or **measurement**. By doing the analysis on the data in the HDF5 file, this allows the analysis to always have the data from the correct **measurement** at its disposal, rather than depending on the temporary instance variables for the various `Instrument`s to still be valid. In fact, the mechanics are available (although seldom used) to queue the analyses, so that a backlog of data can be processed while CsPyController moves to take more data. You don't need to worry about calling these methods or which HDF5 node to pass, that is all handled for you. All you need to do is override one or more of these methods so that they implement your calculations:

- `preExperiment(experimentResults)`: This is called before an **experiment**. The parameter experimentResults is a reference to the HDF5 file for this **experiment.** Override this in a subclass to prepare the analysis appropriately.
- `preIteration(iterationResults, experimentResults)`: This is called before an **iteration**. The parameter experimentResults is a reference to the HDF5 file for this **experiment**. The parameters (iterationResults, experimentResults) reference the HDF5 nodes for this coming **iteration** and its encapsulating **experiment**. Override this in a subclass to prepare the analysis appropriately.
- `analyzeMeasurement(measurementResults, iterationResults, experimentResults)`: This is called after each **measurement**. The parameters (measurementResults, iterationResults, experimentResults) reference the HDF5 nodes for this **measurement** and its encapsuling **iteration** and **experiment**. Override this in a subclass to update the analysis appropriately.
- `analyzeIteration(iterationResults, experimentResults)`: This is called after each **iteration**. The parameters (iterationResults, experimentResults) reference the HDF5 nodes for this **iteration** and its encapsulating **experiment**. Override this in a subclass to update the analysis appropriately.
- `analyzeExperiment(experimentResults)`: This is called at the end of an **experiment**. The parameter experimentResults is a reference to the HDF5 node for the **experiment**. Override this in a subclass to update the analysis appropriately.
- `finalize(hdf5)`: To be run after all optimization loops are complete, so as to close files and such. The parameter `hdf5` is a reference to the whole HDF5 file

7.2 Saving results

Having access to the HDF5 node paramters allows you to access data, but it also allows you to save the results of the analysis. For example, you might access camera data at:

```
measurementResults['data/Hamamatsu/shots']
```

and then write some integrated ROI counts to:

```
measurementResults['analysis/squareROIsums']
```

7.3 Queueing

As will be detailed below, the order of execution of the analyses is determined by the `analyses` list in `aqua.AQuA`. However, it is also possible to allow analyses to happen at the same time with multi-threading. This is not often used because in many cases one analysis will use the results of a previous analysis, but the mechanics are present and may be useful to you. Furthermore, when multi-threading you can choose to skip an analysis if the processing has fallen behind the data acquisition. This does not delete the raw data, which is still available, it just skips the analysis. All you need to do to enable these behaviors is set the following boolean flags:

- `queueAfterMeasurement`: Set to `True` to allow multi-threading on this analysis. Only do this if you are NOT filtering on this analysis, and if you do NOT depend on the results of this analysis later. Default is `False`.
- `dropMeasurementIfSlow`: Set to `True` to skip measurements when slow. Applies only to multi-threading. Raw data can still be used post-iteration and post-experiment. Default is `False`.
- `queueAfterIteration`: Set to `True` to allow multi-threading on this analysis. Only do this if you do NOT depend on the results of this analysis later. Default is `False`.
- `dropIterationIfSlow`: Set to `True` to skip iterations when slow. Applies only to multi-threading. Raw data can still be used in post-experiment. Default is `False`.

7.4 Filtering

Every `Analysis` has the ability to act as a filter, in order to drop a **measurement** that does not meet certain criteria. To filter, have `analyzeMesurement`() return a success code:

- `0` or `None`: good measurement, increment measurement total
- `1`: soft fail, continue with other analyses, but do not increment measurement total
- `2`: med fail, continue with other analyses, do not increment measurement total, and delete measurement data after all analyses
- `3`: hard fail, do not continue with other analyses, do not increment measurement total, delete measurement data

If more than one analysis returns a filtering code for a given **measurement**, then the highest returned code dominates the others. Using these filters, CsPyController can be made to continue an **iteration** until the number of *good* measurements reaches `experiment.measurementsPerIteration`. For example, you could filter on atom loading, so that **measurements** only count when all the sites of interest are loading.

7.5 `AnalysisWithFigure`

Of course you will want to graph some of your data and results. Generally this is done by creating a `matplotlib.pyplot.figure` and assigning it to an `MPLCanvas.figure` on the GUI. In order to get the `MPLCanvas` to update properly, the `MPLCanvas.figure` must change identity, not just redraw. To facilitate this, you can use the `AnalysisWithFigure` class, which is a subclass of `Analysis`. `AnalysisWithFigure` has two figures actually, and plotting is always done on the `backFigure` while the `figure` is the one that is visible. Once plotting is complete, call `swapFigures()` to bring the updated one to the screen. The `updateFigure()` method handles calling `swapFigures()` in a thread-safe manner. Generally you will override `updateFigure()` in your subclass, but be sure to end up a call to `super(myAnalysis, self).updateFigure()` to enable the GUI update. Where to call `updateFigure()` depends on what type of analysis this is. You will place this call inside either `analyzeMeasurement()`, `analyzeIteration()`, or `analyzeExperiment()`, depending on if you want the plot to update after each **measurement**, **iteration** or **experiment**, respectively.

7.6 Create a new `Analysis`

Let's run through an example, which will plot the pictures that we get from the `GroovyCamera`. To demonstrate a calculation, we'll calculate the level of the brightest pixel. We'll use an `AnalysisWithFigure` and update after each **measurement**. You can put this in the same `groovy.py` file with `GroovyCamera`.

```
class PictureViewerAnalysis(AnalysisWithFigure):
    """Plots the pictures from the GroovyCamera for the most recent measurement
        ."""
    data = Member() # the image data to be plotted
    max = Member()  # the brightness of the max pixel

    def analyzeMeasurement(self, measurementResults, iterationResults,
        experimentResults):
        # check to make sure camera data exists
        if 'data/groovy_camera/data' in measurementResults:
            # if camera data exists, grab it
            self.data = measurementResults['data/groovy_camera/data'].value
            # do a calculation, find the brightest pixel
            self.max = np.amax(self.data)
            self.updateFigure()

    def updateFigure(self):
        try:
            # plot to the off-screen figure
            fig = self.backFigure
            # clear figure
            fig.clf()
            # create one subplot
            ax = fig.add_subplot(111)
            # plot the image
            ax.imshow(self.data)
            ax.set_title('most recent shot, with the brightest pixel = {}'.format
                (self.max))
            # update the GUI
            super(PictureViewerAnalysis, self).updateFigure()
        except Exception as e:
            logger.warning('Problem in PictureViewerAnalysis.updateFigure():\n{}\
                n'.format(e))
```

This example starts by making a subclass of `AnalysisWithFigure`. To satisfy `Atom` we declare the instance variables `data = Member()` and `max = Member()`. Notice that we have left out the `__init__()` method. The reason for this is that we did not have anything we needed it to do that `AnalysisWithFigure.__init__()` doesn't already do. If we had some `EvalProp` that needed to be instantiated or any property that we wanted added to the `properties` list (which is certainly something you might want an `Analysis` to have), then you would override `__init__()` and do it there.

Next we define `analyzeMeasurement()`. Simply defining this method is enough to ensure that it will be called at the proper time, after each measurement. Remember how in `GroovyCamera.writeResults()` we stored the picture data to `hdf5['groovy_camera/data']` (where `hdf5` then referred `measurementResults['data']`)? We now access the data stored at that node with `measurementResults['data/groovy_camera/data'].value`. We assign this data temporarily to `self.data`. Then we do a little example calculation to find the brightest pixel with `np.amax(self.data)` and store that too at `self.max`. Then, we call `updateFigure()` so that the plot will update every **measurement**.

The plotting is done in `updateFigure()`. To make sure we plot to the off-screen figure, we use the reference that `AnalysisWithFigure` gives us to `backFigure`. The figure is cleared with `clf()`, and then we add one subplot that we can actually draw on with `add_subplot()`. The real plotting is done with `imshow(self.data)` which takes the picture data that we just pulled out of the HDF5 file and draws it. To show the results of our calculation, we create an axis title that will display the `self.max` value for the brightest pixel. Finally, and importantly, we use `super` to call `AnalysisWithFigure.updateFigure()` which swaps the front and back figures to get the GUI to update and show our latest picture.

7.7 Instantiate your `Analysis`

Similar to the situation with the `GroovyCamera Instrument`, now that we have defined this great new `Analysis`, we need to actually use it by instantiating it in `aqua.py`.

We have already imported `groovy.py`:

```
import groovy
```

Then, you must declare your instance variable at the top of the class:

```
class AQuA(Experiment):
    picture_viewer = Member()
```

Then instantiate the `Analysis` in `__init__()` and be sure to make the `name` match:

```
    def __init__(self):
        super(AQuA, self).__init__()
        # analyses
        self.picture_viewer = groovy.PictureViewerAnalysis('picture_viewer',
            self, 'A matplotlib plot of the most recent GroovyCamera picture')
```

Still in `__init__()`, add the camera to `self.analyses`, which is how the `Experiment` knows which objects to try to `analyzeMeasurement()`, `analyzeIteration()`, `analyzeExperiment()`, etc.:

```
        self.analyses += [self.picture_viewer]
```

An important note, is that the order in which the analyses are called is defined by this list. So if it is important, for example, to calculate the retention in one analysis before using that info in a separate graphing analysis, then the retention analysis must come before the retention graph in this list. And finally, still in `__init__`, add the analysis name to `self.properties` so that its settings will be evaluated and saved:

```
        self.properties += ['picture_viewer']
```

Remember that the order of `properties` determines the order of evaluation. Note that in `analyses` we use a reference to the variable itself as `self.picutre_viewer`, while in `properties` we use a string of the name as `'picture_viewer'`.

7.8 Summary

And that's it! You now have a `GroovyCamera Analysis` that collects data every **measurement**, and you have a `PictureViewerAnalysis` that plots that data every **measurement**. This is a toy example on an imaginary piece of camera hardware, but it illustrates all the major concepts that you will need to implement your own `Instrument` and `Analysis` on real hardware, and the steps necessary to add it to CsPyController.

8 Afterword

This guide is intended to explain the minimum necessary structure for adding an **instrument** or **analysis**. CsPyController is however a complicated package, with at the time of this writing 35519

lines of Python code, a great deal of auxiliary code in LabView, C, C++ and C#, totaling 821 MB for the repository, and 764 GIT commits on `master`. The ultimate way to understand the details of implementation, and to get ideas for how complicated structures have been implemented, is to look at the source code. A great deal of effort, to the best of the author's ability and time, has been put into making the source code well commented. Your contributions to making this code even better will be greatly appreciated.