PyEpics: Epics Channel Access clients with Python

Matthew Newville

Consortium for Advanced Radiation Sciences
University of Chicago

24-Oct-2014

A copy of these slides and instructions for the hands-on exercises are at:
http://cars9.uchicago.edu/epicsclass/
PyEpics Documentation: http://pyepics.github.io/pyepics/
Python offers several convenient features for scientific and engineering programming:

- **Clean Syntax**: Easy to read, learn, remember, share code.
- **Cross Platform**: Code portable to Unix, Windows, Mac.
- **High Level**: No pointers or explicit memory management.
- **Object Oriented**: Full object model, name spaces.
- **Easily Extensible**: With C, C++, Fortran, ...
- **Many Libraries**: GUIs, Databases, Web, Image Processing, ...
- **Free**: Python and all these tools are Free (BSD, LGPL, GPL).

**Main Scientific Libraries:**

- **numpy**: Fast multi-dimensional arrays.
- **matplotlib**: Excellent 2-D Plotting library.
- **scipy**: Numerical Algorithms (FFT, lapack, fitting, ...)

Scientists in many domains are using Python.
PyEpics has several levels of access to Epics Channel Access (CA):

**Procedural**: `caget()`, `caput()`, `cainfo()`, `camonitor()` functions. Simplest interface to use. Built with PV objects.

**PV Objects**: has methods and attributes for setting/getting values of Epics Process Variables. Automatically handles connection management and event handling, extended data types. Built with `ca` and `dbr` modules.

**C-like API**: `ca` and `dbr` modules, giving a nearly complete and 1-1 wrapping of C library for Channel Access. Does automatically determine data types, conversion of C structures, supports all CA callbacks. Built with `libca` and `ctypes`.

**libca**: direct access to the CA library using Python ctypes library: much too hard to use.

Higher-level Python objects (Devices, Alarms, GUI controls) are built from PV objects.
Procedural Interface: caget() and caput()

Easy-to-use interface, similar to command-line tools, EZCA library, or Spec interface:

```python
>>> from epics import caget, caput
>>> m1 = caget('XXX:m1.VAL')
>>> print m1
-1.2001

>>> print caget('XXX:m1.DIR')
1

>>> print caget('XXX:m1.DIR', as_string=True)
'Pos'
```

Options:

```
caget(pvname, as_string=True)
returns the String Representation of value (Enum State Name, formatted floating point numbers, . . . )

caput(pvname, wait=True)
waits until processing completes. Also support a timeout option (wait no longer than specified time).
```

Pretty simple, but does most of what you need for many scripts.

A global cache of Process Variables is maintained. Each caget(), etc function does **not** create a new connection.
Procedural Interface: `cainfo()` and `camonitor()`

`cainfo()` shows a status information and control fields for a PV:

```python
>>> cainfo('XXX.m1.VAL')
== XXX:m1.VAL (native_double) ==
value = 0.49998
char_value = '0.5000'
count = 1
nelm = 1
type = double
units = mm
precision = 4
host = iocXXX.cars.aps.anl.gov:5064
access = read/write
status = 0
severity = 0
timestamp = 1413889163.083 (2014-10-21 05:59:23.08320)
upper_ctrl_limit = 3.70062825
lower_ctrl_limit = -4.89937175
upper_disp_limit = 3.70062825
lower_disp_limit = -4.89937175
upper_alarm_limit = 0.0
lower_alarm_limit = 0.0
upper_warning_limit = 0.0
lower_warning_limit = 0.0
PV is internally monitored, with 0 user-defined callbacks:
```

`camonitor()` prints a message every time a PV’s value changes:

```python
>>> camonitor('XXX:DMM1Ch2_calc.VAL')
XXX:DMM1Ch2_calc.VAL 2014-10-21 09:02:09.94155 -197.8471
XXX:DMM1Ch2_calc.VAL 2014-10-21 09:02:11.96659 -197.8154
XXX:DMM1Ch2_calc.VAL 2014-10-21 09:02:13.94147 -197.8455
XXX:DMM1Ch2_calc.VAL 2014-10-21 09:02:15.94144 -197.8972
XXX:DMM1Ch2_calc.VAL 2014-10-21 09:02:17.94412 -197.8003
XXX:DMM1Ch2_calc.VAL 2014-10-21 09:02:20.00106 -197.8555
```

It runs until `camonitor_clear()` is called.

You can give a callback function to `camonitor()` to change the formatting or do something other than print the value to the screen.
PV objects: Easy to use, richly featured.

PV objects are the recommended way to interact with Process Variables:

Creating a PV, PV.get()

```python
>>> from epics import PV

>>> pv1 = PV('XXX:m1.VAL')
>>> print pv1
<PV 'XXX:m1.VAL', count=1, type=double, access=read/write>

>>> print pv1.count, pv1.type, pv1.host, pv1.upper_ctrl_limit
(1, 'double', 'ioc13XXX.cars.aps.anl.gov:5064', 19.5))

>>> pv1.get()
-2.3400000000000003

>>> pv1.value
-2.3400000000000003

>>> # get values as strings
>>> pv1.get(as_string=True)
'2.340'
```

PVs have:

- Automatic connection management.
- `get()` and `put()` methods.
- ...read/write to `PV.value` attribute.
- `as_string` to get string representations of values.
- Attributes for many properties (count, type, host, upper_crtl_limit, ...)

PVs are set up to be monitored internally and asynchronously: new values arrive, and `PV.get()` usually simply returns the latest value the PV received.
PV use callbacks to automatically monitor changes in PV value, and in changes in connection status.

**PV.get() for Enum PV**

```python
>>> pv2 = PV('XXX:m1.DIR')
>>> print pv2
<PV 'XXX:m1.DIR', count=1, type=enum, access=read/write>

>>> print pv2.count, pv2.type, pv2.enum_strs
(1, 'enum', ('Pos', 'Neg'))

>>> pv2.get(), pv2.get(as_string=True)
0 'Pos'
```

**Enum PVs have a enum_strs attribute that holds the names for enum states.**

**PV.put()**

```python
>>> from epics import PV
>>> pv1 = PV('XXX:m1.VAL')

>>> pv1.put(2.0)  # returns immediately

>>> pv1.value = 3.0  # = pv1.put(3.0)

>>> pv1.put(1.0, wait=True)  # waits for completion
```

**put() can wait for completion, or run a callback function when complete.**
Event Callback: User-defined function called when a PV changes.
This function must have a pre-defined call signature, using keyword arguments:

**Code Example**

```python
import epics
import time

def onChanges(pvname=None, value=None, char_value=None, **kws):
    print 'PV Changed! ', pvname, \char_value, time.ctime()

mypv = epics.PV(pvname)

# Add a callback
mypv.add_callback(onChanges)

print 'Now watch for changes for a minute'

t0 = time.time()
while time.time() - t0 < 60.0:
    time.sleep(1.e-2)

mypv.clear_callbacks()
print 'Done.'
```

**Callback Function Arguments:**

- `pvname`  name of PV
- `value`   new value
- `char_value`  String representation of value
- `count`   element count
- `ftype`   field type (DBR integer)
- `type`   python data type
- `status`  ca status (1 == OK)
- `precision`  PV precision

And many CTRL values (limits, units, . . . ).
Use **kws to allow for all of them!!
More on Callbacks: Warnings and Recommendations

Event Callbacks happen asynchronously – the CA library is running an event-loop and will call your Python functions when a value changes.

Your function runs inside a CA pend_event() call. This restricts what you can do in a callback function:

You can read values for other PVs. You cannot do much I/O, – including putting new values! – inside a callback.

Callback Recommendations:

- Keep callback functions small and fast.
- Set a flag that a value has changed, react to that change when after
  Keep callback functions small and fast.

Callbacks and GUI Programs: A GUI C++ event-loop can conflict disastrously with Epics event-loop (GUI Button-push sets PV, PV event should update widget value, ...).

- PyQt/PySide/Tk: seem to handle this issue itself.
- wxPython: pyepics has decorators to help handle this.
Epics Waveform records for array data are important for experimental data.

**double waveform**

```python
>>> p1vals = numpy.linspace(3, 4, 101)
>>> scan_p1 = PV('XXX:scan1.P1PA')
>>> scan_p1.put(p1vals)

>>> print scan_p1.get()[:101]
[3. , 3.01, 3.02, ..., 3.99, 3.99, 4.]
```

**character waveform**

```python
>>> folder = PV('XXX:directory')
>>> print folder
<PV 'XXX:directory', count=21/128, type=char, access=read/write>

>>> folder.get()
array([ 84, 58, 92, 120, 97, 115, 95, 117, 115, 101, 114, 92, 77, 97, 114, 99, 104, 50, 48, 49, 48])

>>> folder.get(as_string=True)
'T:\xas.user\March2010'

>>> folder.put('T:\xas.user\April2010')
```

You can set an EPICS waveform with a Python list or numpy array. Reading a waveform will return a numpy array (a list if numpy is not installed).

Also: For recent versions of Epics base (3.14.12.2), sub-arrays are supported.

Epics Strings are limited in length. Character waveforms can be used for strings longer than 40 characters.

Setting an Epics Character waveform with a Python string will automatically convert the string to a list of bytes (and append a trailing NULL).
The ca module uses ctypes to wrap almost all functions in the Channel Access library:

```python
from epics import ca
chid = ca.create_channel('XXX:m1.VAL')
count = ca.element_count(chid)
ftype = ca.field_type(chid)
value = ca.get()
print "Channel ", chid, value, count, ftype

# put value
cia.put(chid, 1.0)
cia.put(chid, 0.0, wait=True)

# user defined callback
def onChanges(pvname=None, value=None, **kw):
    fmt = 'New Value for %s value=%s
    print fmt % (pvname, str(value))

# subscribe for changes
eventID = ca.create_subscription(chid,
    userfcn=onChanges)

while True:
    time.sleep(0.001)
```

### Enhancements for Python:

- **Python namespaces, exceptions used.**
  - `ca_fcn → ca.fcn`
  - `DBR_XXXX → dbr.XXXX`
  - `SEVCHK → Python exceptions`

- **OK to forget many tedious chores:**
  - initialize CA.
  - create a context (unless explicitly using Python threads).
  - wait for connections.
  - clean up at exit.

- **No need to worry about data types – the C data structures for Epics data types are mapped to Python classes.**

Python decorators are used to lightly wrap CA functions so that:

- CA is initialized, finalized.
- Channel IDs are valid, and connected before being used.
Essentially all CA functions work “Just like C”. A few notes:

- Preemptive Callbacks are used by default. Events happen asynchronously. This can be turned off, but only before CA is initialized.

- DBR_CTRL and DBR_TIME data types supported, but not DBR_STS or DBR_GR.

- Waveform records will be converted to numpy arrays if possible.

- Some functions are not needed: ca_set_puser(), ca_add_exception_event().

- EPICS_CA_MAX_ARRAY_BYTES is set to 16777216 (16Mb) unless already set.

- Connection and Event callbacks are (almost always) used internally. User-defined callback functions are called by the internal callback.

- Event Callbacks are used internally except for large arrays, as defined by ca.AUTOMONITOR_LENGTH (default = 16K).

- Event subscriptions use mask = (EVENT | LOG | ALARM) by default.
The standard Python ctypes library that gives access to C data structures and functions in dynamic libraries at Python run-time.

ctypes gives a "just like C" interface to any dynamic library. For Channel Access, this means:

- Load library
- Create Channel IDs
- Use Channel IDs with library functions, being careful about data types for arguments.
Why ctypes and libca->ca->PV matters

Using ctypes makes several parts of PyEpics very easy:

1. Complete CA interface easy to implement, debug, all in Python (no C code).
2. Install on all system with `python setup.py install`.
3. Best thread support possible, with Python Global Interpreter Lock.
4. Python 2 and Python 3 supported, with little code change.
Threading and multiprocessing

Python Threads and multiprocessing are both supported.

This will run two separate threads, each creating and monitoring PVs.

**Note:** Python does not really execute 2 instructions at once. It runs only 1 interpreter in 1 process (on 1 core).

It *does* release a global interpreter lock at each call to C code (I/O or ctypes!) to allow code in another thread to run while I/O or C code is processing.

Multiprocessing is bit more complicated, but can be done as well.
Several Devices and GUI Components are built into pyepics
A PyEpics Device is a collection of PVs, usually sharing a Prefix. Similar to an Epics Record, but relying on PV names, not Record definition.

Epics Analog Input as Python epics.Device

```python
import epics

class ai(epics.Device):
    "Simple analog input device"
    _fields = ('VAL', 'EGU', 'HOPR', 'LOPR', 'PREC',
               'NAME', 'DESC', 'DTYP', 'INP', 'LINR', 'RVAL',
               'ROFF', 'EGUF', 'EGUL', 'AOFF', 'ASLO', 'ESLO',
               'EOFF', 'SMOO', 'HIHI', 'LOLO', 'HIGH', 'LOW',
               'HHSV', 'LLSV', 'HSV', 'LSV', 'HYST')

    def __init__(self, prefix, delim='. '):
        epics.Device.__init__(self, prefix, delim=delim,
                                      self._fields)
```

A Device maps a set of PV “fields” (name “suffixes”) to object attributes, holding all the associated PVs.

Can save / restore full state.

Using an ai device

```python
>>> from epics.devices import ai
>>> Pump1 = ai('XXX:ip2:PRES')
>>> print "%s = %s %s" % (Pump1.DESC,
                         Pump1.get('VAL',as_string=True),
                         Pump1.EGU )
Ion pump 1 Pressure = 4.1e-07 Torr
>>> print Pump1.get('DTYP', as_string=True)
asyn MPC
>>> Pump1.PV('VAL') # Get underlying PV
<PV 'XXX:ip1:PRES.VAL', count=1, type=double, access=read/write>
```

Can use get() / put() methods or attribute names on any of the defined fields.
And, of course, a *Device* can have methods added:

**Scaler device**

```python
import epics
class Scaler(epics.Device):
    "SynApps Scaler Record"
    ...
    def OneShotMode(self):
        "set to one shot mode"
        self.CONT = 0
    def CountTime(self, ctime):
        "set count time"
        self.TP = ctime
    ...
```

Add Methods to a Device to turn it into a high-level Objects.

Can also include complex functionality – dynamically, and from client (beamline).

Long calculations, database lookups, etc.

**Using a Scaler:**

```python
s1 = Scaler('XXX:scaler1')
s1.setCalc(2, '(B-2000*A/10000000.)')
s1.enableCalcs()
s1.OneShotMode()
s1.Count(t=5.0)
print 'Names: ', s1.getNames()
print 'Raw values: ', s1.Read(use_calcs=False)
print 'Calc values: ', s1.Read(use_calcs=True)
```

**Simple Example:** Read Ion Chamber current, amplifier settings, x-ray energy, compute photon flux, post to a PV.

Needs table of coefficients (∼16kBytes of data), but then ∼100 lines of Python.
A **Motor** Device has ~100 fields, and several methods to move motors in User, Dial, or Raw units, check limits, etc.

**Using a Motor**

```python
>>> from epics import Motor
>>> m = Motor('XXX:m1')
>>> print 'Motor: ', m.DESC, ' Currently at ', m.RBV

>>> m1.tweak_val = 0.10 # or m1.TWV = 0.10

>>> m1.move(0.0, dial=True, wait=True)

>>> for i in range(10):
    >>>    m1.tweak(dir='forward', wait=True)

>>> print 'Motor: ', m.DESC, ' at ', m.RBV

>>> print m.drive, m.description, m.slew_speed
1.030 Fine X 5.0

>>> print m.get('device_type', as_string=True)
'asynMotor'
```

**Motor features:**

- `get()` and `put()` methods for all attributes
- `check_limits()` method.
- `tweak()` and `move()` methods.
- Can use Field suffix (.VELO, .MRES) or English description (slew_speed, resolution).

Other devices included in the main distribution:
- ao, ai, bi, bo, mca, scaler, struck (multi-channel scaler), transform.
Alarms: react to PV values

An alarm defines user-supplied code to run when a PV’s value changes to some condition. Examples might be:

- send email, or some other alert message
- turn off some system (non-safety-critical, please!)

```
from epics import Alarm, poll

def alertMe(pvname=None, char_value=None, **kw):
    print "Soup is on! %s = %s" % (pvname, char_value)

my_alarm = Alarm(pvname = 'WaterTemperature.VAL',
                 comparison = '>',
                 callback = alertMe,
                 trip_point = 100.0,
                 alert_delay = 600)

while True:
    poll()
```

When a PV’s value matches the `comparison` with the `trip_point`, the supplied `callback` is run. A delay is used to prevent multiple calls for values that “bounce around”.

Epics Data Archiver – Epics+Python+MySQL+Apache

Main features:

- Web interface to current PV values.
- $\gtrsim 5000$ PVs monitored
- Data archived to MySQL tables.
- templates for status web pages
- plots of historical data
- web-definable email alerts

PV values displayed as html links to Plot of Data
Epics Archiver: Plotting Historical Data

Plots:
- default to past day
- using Gnuplot (currently)
- Plot “From now” or with “Date Range”
- Plot up to 2 PVs
- “Related PVs” list for common pair plots
- pop-up javascript Calendar for Date Range
- String labels for Enum PVs
GUI Controls with wxPython

Many PV data types (Double, Float, String, Enum) have wxPython widgets, which automatically tie to the PV.

**Sample wx widget Code**

```python
from epics import PV
from epics.wx import wxlib

# text controls: read-only or read/write.
txt_wid = wxlib.pvText(parent, pv=PV('SomePV'),
                      size=(100,-1))
text_ctrl_wid = wxlib.pvTextCtrl(parent, pv=PV('SomePV'))

# enum controls: drop-down or toggle-buttons
ddown_wid = pvEnumChoice(parent, pv=PV('EnumPV.VAL'))
btns_wid = pvEnumButtons(parent, pv=PV('EnumPV.VAL'),
                         orientation=wx.HORIZONTAL)

# float control: values restricted by PV limits.
flt_wid = wxlib.pvFloatCtrl(parent, size=(100, -1),
                         precision=4)
flt_wid.set_pv(PV('XXX.VAL'))
```

- **pvText** read-only text for Strings
- **pvTextCtrl** editable text for Strings
- **pvEnumChoice** drop-down list for ENUMs.
- **pvEnumButtons** button sets for ENUMs.
- **pvAlarm** pop-up message window.
- **pvFloatCtrl** editable text for Floats, can enter only numbers that obey PV limits.
- **Others**: Bitmap, Checkboxes, Buttons, Shapes, etc

Mixin classes help extending other wx Widgets (Many from Angus Gratton, ANU).
Function Decorators help write code that is safe against mixing GUI and CA threads.
Simple Area Detector Display: A $1360 \times 1024$ RGB image (4Mb) from Prosilica GigE camera.

Works with any areaDetector Image.

Includes basic controls:
- trigger mode
- exposure time
- gain
- color mode
- start /stop acquisition.

Click-and-Drag on Image Zooms in, using ROI plugin.

Displays at about 15 Hz: slightly slower than ImageJ.
wx Motor Controls

MEDM-like Motor Display, except
much easier to use.

Entry Values can only be valid number. Values outside soft limits are highlighted.

Tweak Values are generated from precision and range.

Cursor Focus is more modern than Motif.

More Button leads to Detail Panel . . .
A custom GUI for controlling a six-motor Sample Stage:

- **Named Positions** Positions can be saved by named and restored.
- **Sample Image (JPEG)** captured at each saved position (either Web Camera or area Detector image).
- **Simple Configuration** with Windows-style .INI file.

Useful for my station, but definitely beamline specific.
A GUI for controlling XRF detectors and viewing XRF data.
Live Plots of Time-Dependent PVs:

- Interactive Graphics, Zooming.
- Set Time Range, Y-range
- Set log plot
- Save Plot to PNG, Save Data to ASCII files.
Epics Instruments is a GUI application that lets any user:

- Organize PVs into **Instruments**: a named collection of PVs
- Manage Instruments with “modern” tab interface.
- Save Positions for any Instrument by name.
- Restore Positions for any Instrument by name.
- Remember Settings all definitions fit into a single file that can be loaded later.

Typing a name in the box will save the current position, and add it to the list of positions.

Multiple Users can be using multiple instrument files at any one time.

Epics Instruments: A few more screenshots

Save/Restore for motor or regular PVs.

At startup, any recently used database files can be selected.

On “Go To”, settings can be compared with current values, and selectively restored:

Server Mode: An application can listen to a simple Epics Record.

This allows other processes (IDL, Spec, . . .) to restore instruments to named positions by writing to a couple PVs.

Edit screen to set PVs that make up and Instrument.
The stepscan module provides tools needed for step scans, similar to Spec or Epics SScan Record.

**Simple Scan in Python**

```python
import numpy
from epicsapps.stepscan import (StepScan, Positioner,
                                Counter, ScalerDetector)

scan = StepScan()
x = Positioner('13IDE:m1')
x.array = numpy.linspace(0, 1, 21)

scan.add_positioner(x)

scan.add_detector(ScalerDetector('13IDE:scaler1'))

scan.add_counter(Counter('13IDE:m3.RBV'))

scan.add_extra_pvs(((0, 'Ring Current', 'S:SRcurrentAI.VAL'),
                    (0, 'Ring Lifetime', 'S:SRlifeTimeHrsCC.VAL')))

scan.run(filename = 'simple_scan.001')
```

- **Borrows many concepts from Epics SScan Record, but is pure Python.**
- **Positioners:** any Epics PV.
- **Detectors:** Epics Scaler, Mca, Multi-Element Mca Record, AreaDetector so far.
- **Writes ASCII column file, expects AreaDetector to write its own data.**
Borrowing as much from Epics Scan Record as from Spec:

A scan is a loop that contains:

- a list of **Positioners** to set at each point.
- a list of **Triggers** to start detectors.
- a list of **Counters** to read.
- A list of **Extra_PVs** to record at beginning of scan.
- **pre_scan()** method to run before scan.
- **post_scan()** method to run after scan.
- **breakpoints** a list of indices to pause and write data collected so far to disk.
- **at_break()** method to at each breakpoint.

```python
# set positioners, triggers, counters, # and array(s) of positions
read_extra_pvs()
run_pre_scan()

for i in range(scan_npts):
    for p in positioners:
        p.move_to_pos(i)  # use put-callback
        while not all([p.done for p in positioners]):
            epics.poll()

    for t in detectors.triggers:
        t.start()  # use put-callback
        while not all([t.done for t in detectors.triggers]):
            epics.poll()

    for counter in detectors.counters:
        counter.read()

    if i in breakpoints:
        write_data()
        run_at_break()

# loop done, write file
write_data()
run_post_scan()
```
Positioner and Detector Classes

Positioners:

- unlimited number
- uses **full array** of points for each Positioner. *not start/stop/npts.*
- multidimensional scans are not **inner** and **outer**. This allows non-rectangular scans.
- can alter integration time per point: dwell time for each detector is a Positioner.
- can put in attenuators, filters, etc for certain points, etc.
- use **breakpoints** to write out portions of multi-dimensional scans.

Detectors:

- unlimited number
- a Counter can store any Scalar PV. **No waveform PVs, yet.**
- a Trigger starts a detector.
- **ScalerDetector** sets trigger, dwelltime PV, add counters for *all* named scalers channels.
  - Option to use calc record.
- **McaDetector** sets trigger, dwelltime PV, adds all named ROIs as counters.
- **AreaDetector** sets trigger, dwelltime PV, saves `.ArrayCounter RBV`. Expects detector to save data, saves
- Add **pre_scan()**/**post_scan()** methods to put detectors in correct modes.
Spec-like Scans

A simple, Spec-like scan:

```python
from epicsapps.stepscan import SpecScan

scan = SpecScan()
scan.add_motors(x='13IDE:m9', y='13IDE:m10', ...)
scan.add_detector('13IDE:scaler1', kind='scaler')
scan.add_counter('13IDE:ao1', label='A_Counter')
scan.filename = 'simple_scan.001'

scan.ascan('x', 10, 11.0, 41, 0.25)

# scan.lup('y', -0.1, 0.1, 21, 0.5)
# scan.mesh('x', 10, 10.5, 11, 'y', 0, 0.5, 11, 0.5)
# scan.d2scan('x', -1, 1, 'y', 0, 1, 41, 0.25)
```

- `ascan()`, `a2scan()`, `a3scan()`, `dscan()`, `d2scan()`, `d3scan()`, `lup()`, and `mesh()`,
- **Motors**: any Epics PV.
- **Detectors**: Epics Scaler, Mca, Multi-Element Mca Record, AreaDetector so far.
- currently writes ASCII column file, but not exactly a “Spec File”.
- no live plotting, yet . . . (in progress).
- built on top of more general Step Scanning module.

Can use an INI file to save / load configuration of Positioners and Detectors.
Scan GUIs for defining and running scans

Building on-top of the StepScan code:

Many beamlines want GUI forms for users to set up and remember scan parameters, and do different sorts of scans (ascan, mesh, EXAFS, etc).

NOTE: This is a work in progress... needs more testing, but most things are working.
Overall Scan Architecture

How the Scan GUI actually runs the scan:

The GUI puts a JSON-encoded strings of scan parameters into a Postgres database.

A server process monitors the database, running scans when requested. This server saves the file, and posts real-time data back into the database.

Clients can read this data from the database to display it.

Data Collection Scheme

Relational Database Management Systems (like Postgres) are designed with

- multiple simultaneous clients
- data integrity
- access security layers
- multiple programming languages

in mind. SQL becomes the communication layer between Clients and Server.
The Scanning GUI is a work in progress

Current Status / To-Do list:

- GUI to configure/store scan parameters works.
- Running and displaying simple scans works.
- Postgres DB used for client/server communication.
- Real-time plot works. Peak analysis, “move to cursor position”, etc in progress.
- Embedded “macro language” exists, needs more testing.

Suggestion, ideas, collaboration would be welcome!
PyEpics: Epics Channel Access for Python

- near complete interface to CA, threads, preemptive callbacks.
- works on linux-x86, linux-x86_64, darwin-x86, win32-x86 (base 3.14.12.1) with Python 2.6, 2.7, 3.3.
- reasonably well-documented and unit-tested.
- easy installation and deployment.
- high-level PV class, Devices.
- GUI support (wxPython mostly, some PyQt support).
- A few general-purpose applications, more in progress.
- [http://github.com/pyepics](http://github.com/pyepics)
- [http://github.com/pyepics/epicsapps](http://github.com/pyepics/epicsapps)

**Acknowledgments:** co-developer: Angus Gratton, ANU.

Suggestions, bug reports and fixes from Michael Abbott, Marco Cammarata, Craig Haskins, Pete Jemian, Andrew Johnson, Janko Kolar, Irina Kosheleva, Tim Mooney, Eric Norum, Mark Rivers, Friedrich Schotte, Mark Vigder, Steve Wasserman, and Glen Wright.
There are 36 sets of PVs: PREFIX = id01, id02, ..., id36.

<table>
<thead>
<tr>
<th>NAME</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PREFIX:shutter</td>
<td>binary</td>
</tr>
<tr>
<td>PREFIX:dval1</td>
<td>double</td>
</tr>
<tr>
<td>PREFIX:dval2</td>
<td>double</td>
</tr>
<tr>
<td>PREFIX:dval3</td>
<td>double</td>
</tr>
<tr>
<td>PREFIX:ival1</td>
<td>integer</td>
</tr>
<tr>
<td>PREFIX:ival2</td>
<td>integer</td>
</tr>
<tr>
<td>PREFIX:ival3</td>
<td>integer</td>
</tr>
<tr>
<td>PREFIX:string1</td>
<td>epics string (max 40 characters)</td>
</tr>
<tr>
<td>PREFIX:string2</td>
<td>epics string (max 40 characters)</td>
</tr>
<tr>
<td>PREFIX:bo1</td>
<td>binary (0 / 1)</td>
</tr>
<tr>
<td>PREFIX:mbbo1</td>
<td>multi-bit binary (enum)</td>
</tr>
<tr>
<td>PREFIX:filename</td>
<td>byte waveform (128 characters) – long filename</td>
</tr>
<tr>
<td>PREFIX:dwave</td>
<td>double waveform (256 elements)</td>
</tr>
<tr>
<td>PREFIX:iwave</td>
<td>integer waveform (256 elements)</td>
</tr>
<tr>
<td>PREFIX:image</td>
<td>byte waveform (65536 elements)</td>
</tr>
</tbody>
</table>
These slides and instructions: http://cars9.uchicago.edu/epicsclass/

**Data Logger #1** There are 3 double and 3 integer PVs are updated somewhat randomly. Write a program to print out these values every second.

**Data Logger #2** Modify your program to print out all 6 values when any of them changes.

**Shutter Watcher #1** The 36 shutter PVs are opening and closing periodically, somewhat randomly. Write a program to print out when any shutter opens or closes.

**Shutter Watcher #2** Measure how long each shutter is open during a 5 minute interval and print out a table of result.

**Create a Device** Create an Epics Device from the 3 double PVs, 3 integer PVs, the mmbo PV, a regular string PV, and the filename PV.

**Working with waveforms** Read one of the updating dwave, iwave, and image PVs into a numpy ndarray and find the mean, sum, and standard deviation, and plot the array.

I’ll show a solution to these (in order) every 20 minutes.

To use Anaconda python (more recent version) on the APS Share, type

```
source /local/newville/anaconda.sh
```