

# Python and Epics: Channel Access Interface to Python

Matthew Newville

Consortium for Advanced Radiation Sciences  
University of Chicago

October 12, 2010

<http://cars9.uchicago.edu/software/python/pyepics3/>

# Why Python? The Standard Answers

Clean Syntax	Easy to learn, remember, and read
High Level Language	No pointers, dynamic memory, automatic memory
Cross Platform	code portable to Unix, Windows, Mac.
Object Oriented	full object model, name spaces. Also: procedural!
Extensible	with C, C++, Fortran, Java, .NET
Many Libraries	GUIs, Databases, Web, Image Processing, Array math
Free	Both senses of the word. No, really: completely free.

Scientists use Python.

# Why Do Scientists Use Python?

Python is great. The tools are even better:

<code>numpy</code>	Fast arrays.
<code>matplotlib</code>	Excellent Plotting library
<code>scipy</code>	Numerical Algorithms (FFT, lapack, fitting, ...)
<code>f2py</code>	Wrapping Fortran for Python
<code>sage</code>	Symbolic math (ala Maple, Mathematica)
<b>GUI Choices</b>	Tk, wxWidgets, Qt, ...
<b>Free</b>	Python is Free. All these tools are Free (BSD).

# Why Do Scientists Use Python?

Python is great. The tools are even better:

<code>numpy</code>	Fast arrays.
<code>matplotlib</code>	Excellent Plotting library
<code>scipy</code>	Numerical Algorithms (FFT, lapack, fitting, ...)
<code>f2py</code>	Wrapping Fortran for Python
<code>sage</code>	Symbolic math (ala Maple, Mathematica)
<b>GUI Choices</b>	Tk, wxWidgets, Qt, ...
<b>Free</b>	Python is Free. All these tools are Free (BSD).

All of these tools use the C implementation of Python.

NOT Jython (Python in Java) or IronPython (Python in .NET):

I am not talking about Jython.

It's easy to wrap C for Python.

There have been several wrappings of Epics CA:

- G Savage (FNAL) ~1999. Very low-level wrapping, much like C interface: pass around `chids` and specifying `DBR.XXXX` types all the time. Uses SWIG.
- MN started with EZCA, then moved to custom mapping of CA (~2002), with emphasis on a Python PV Object, not low-level interface. Used SWIG.
- X. Wang at PSI, N. Yamamoto at KEK built upon Savage's wrapping, updating for better 3.14 support, and using C-Python API (not SWIG).
- M. Abbot at Diamond wrote an interface with `ctypes` (~2008) but used a non-standard third-party psuedo-threading library.

It's easy to wrap C for Python.

There have been several wrappings of Epics CA:

- G Savage (FNAL) ~1999. Very low-level wrapping, much like C interface: pass around `chids` and specifying `DBR.XXXX` types all the time. Uses SWIG.
- MN started with EZCA, then moved to custom mapping of CA (~2002), with emphasis on a Python PV Object, not low-level interface. Used SWIG.
- X. Wang at PSI, N. Yamamoto at KEK built upon Savage's wrapping, updating for better 3.14 support, and using C-Python API (not SWIG).
- M. Abbot at Diamond wrote an interface with `ctypes` (~2008) but used a non-standard third-party psuedo-threading library.

Sept 2009, A discussion on Tech-Talk asked "Can we combine forces?"

I was growing unhappy with my own library (Windows build, no connection callbacks)

... So I rewrote from scratch.

### Starting Wish List:

- complete (nearly?) exposure of low-level CA.
- high-level PV class built on this.
- thread support (as well as Python can).
- preemptive callbacks: connection, event, put.
- documentation and unit-testing.
- easy installation, including Windows.
- Support for Python 2 and Python 3.

The key decision: *Use Python's ctypes module. NO C Code!*

## Starting Wish List:

- complete (nearly?) exposure of low-level CA.
- high-level PV class built on this.
- thread support (as well as Python can).
- preemptive callbacks: connection, event, put.
- documentation and unit-testing.
- easy installation, including Windows.
- Support for Python 2 and Python 3.

The key decision: *Use Python's ctypes module. NO C Code!*

### ctypes for libca.so

```
import ctypes
libca = ctypes.cdll.LoadLibrary('libca.so')
libca.ca_context_create(1)
chid = ctypes.c_long()
libca.ca_create_channel('MyPV', 0,0,0, ctypes.byref(chid))
libca.ca_pend_event.argtypes = [ctypes.c_double]
libca.ca_pend_event(1.0e-3)

print 'Connected: ', libca.ca_state(chid) == 2 # (CS.CONN)
print 'Host Name: ', libca.ca_host_name(chid)
```

Makes several goals trivial:

- 1 Easy install on all systems:  
python setup.py install
- 2 best thread support possible.
- 3 Python 2 and Python 3.

PyEpics3 contains 3 levels of access to CA:

**Lowest level:** ca and dbr modules. C-like API, nearly complete.

**High level:** PV object. Built on ca module.

**Functional:** `caget()`, `caput()`, `cainfo()`, `camonitor()`, Built on PV.

PyEpics3 contains 3 levels of access to CA:

**Lowest level:** ca and dbr modules. C-like API, nearly complete.

**High level:** PV object. Built on ca module.

**Functional:** caget(), caput(), cainfo(), camonitor(), Built on PV.

## caget() / caput()

```
>>> from epics import caget, caput, cainfo

>>> m1 = caget('XXX:m1.VAL')
>>> print m1
-1.2001

>>> caput('XXX:m1.VAL', 0)

>>> caput('XXX:m1.VAL', 2.30, wait=True)

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

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

This is probably too easy, huh?

## cainfo()

```
>>> cainfo('XXX:m1.VAL')
== XXX:m1.VAL (double) ==
value          = 2.3
char_value     = 2.3000
count         = 1
units         = mm
precision     = 4
host          = xxx.aps.anl.gov:5064
access       = read/write
status       = 1
severity     = 0
timestamp    = 1265996455.417 (2010-Feb-12 11:40:55.417)
upper_ctrl_limit = 200.0
lower_ctrl_limit = -200.0
upper_disp_limit = 200.0
lower_disp_limit = -200.0
upper_alarm_limit = 0.0
lower_alarm_limit = 0.0
upper_warning_limit = 0.0
lower_warning    = 0.0
PV is monitored internally
no user callbacks defined.
=====
```

Python namespaces used `ca_fcn` → `ca.fcn`, `DBR_XXXX` → `dbr.XXXX`.

## The ca interface

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

ca.put(chid, 1.0)
ca.put(chid, 0.0, wait=True)

# user defined callback
def onChanges(pvname=None, value=None, **kw):
    fmt = 'New Value for %s value=%s\n'
    print fmt % (pvname, str(value))
# subscribe for changes
eventID = ca.create_subscription(chid,
                                userfcn=onChanges)

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

## *Enhancements:*

- OK to forget to initialize CA.
- OK to forget to create a context (expect in Python threads).
- OK to not explicitly wait for connection.
- OK to not clean up at exit.
- `get()` returns value.
- SEVCHK → Python exceptions

Python namespaces used `ca_fcn` → `ca.fcn`, `DBR_XXXX` → `dbr.XXXX`.

## The ca interface

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

ca.put(chid, 1.0)
ca.put(chid, 0.0, wait=True)

# user defined callback
def onChanges(pvname=None, value=None, **kw):
    fmt = 'New Value for %s value=%s\n'
    print fmt % (pvname, str(value))
# subscribe for changes
eventID = ca.create_subscription(chid,
                                userfcn=onChanges)

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

## Enhancements:

- OK to forget to initialize CA.
- OK to forget to create a context (expect in Python threads).
- OK to not explicitly wait for connection.
- OK to not clean up at exit.
- `get()` returns value.
- SEVCHK → Python exceptions

The CA functions are lightly wrapped (Python decorators) to ensure

- CA is initialized and finalized.
- chid values are reasonable (C longs)
- channels are connected when needed.

- Preemptive Callbacks used by default. Must be set before using CA.
- Connection and Event callbacks are used internally.
- DBR\_CTRL and DBR\_TIME variants supported, not DBR\_STS or DBR\_GR.

These other “design choices” were made, but are configurable:

- Event subscriptions use mask = (EVENT | LOG | ALARM.)
- EPICS\_CA\_MAX\_ARRAY\_BYTES set to 16777216 (16Mb) unless already set.
- Event Callbacks are used internally except for large arrays (as defined by ca.AUTOMONITOR\_LENGTH (default = 16K).
- Small Arrays will be converted to numpy arrays (if available).
- Small CHAR waveforms can be converted to strings.

# PV objects: Easier to use, Full-featured.

Most python programmers will want to use PV objects:

## Using PV objects

```
>>> from epics import PV
>>> pv1 = PV('XXX:mi.VAL')
>>> print pv1.count, pv1.type
(1, 'double')

>>> print pv1.get()
-2.3456700000000001

>>> pv1.value = 3.0 # = pv1.put(3.0)
>>> pv1.value # = pv1.get()
3.0
>>> print pv.get(as_string=True)
'3.0000'

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

>>> # subscribe for changes
>>> pv1.add_callback(onChanges)
>>> while True:
...     time.sleep(0.001)
```

- Automatic connection management.
- Attributes for many properties (count, type, host, upper\_ctrl\_limit, ...)
- Can use get() / put() methods
- ... or PV.value attribute.
- as\_string uses ENUM labels or Precision.
- put() can wait or run user callback when complete.
- connection callbacks.
- multiple event callbacks.

A *device* is a collection of PVs, usually sharing a Prefix.

# Devices: collections of PVs

A *device* is a collection of PVs, usually sharing a Prefix.

## Epics Analog Input as Python epics.Device

```
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', 'SMOD', 'HIHI', 'LOLO', 'HIGH', 'LOW',
              'HHSV', 'LLSV', 'HSV', 'LSV', 'HYST')

    def __init__(self, prefix):
        if not prefix.endswith('.'):
            prefix = "%s." % prefix
        epics.Device.__init__(self, prefix, self._fields)
```

An `epics.Device` has `get` and `put` methods and a `PV` method to return the underlying PV.

That is really the entire definition.

## Using an ai device

```
>>> Pump1 = ai('XXX:ip1:PRES')
>>> print "%s = %s %s\n" % (Pump1.get('DESC'),
                           Pump1.get('VAL', as_string=True),
                           Pump1.get('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>
```

Yes, it is that easy.

# Subclassing Devices

Of course, a *device* can be subclassed, to add functionality.

## Scaler device

```
import epics
class Scaler(epics.Device):
    "SynApps Scaler Record"

    ...
    def OneShotMode(self):
        "set to one shot mode"
        self.put('.CONT', 0)

    def CountTime(self, ctime):
        "set count time"
        self.put('.TP', ctime)
```

Simply add Methods to turn a device into a full Object.

Can also complex functionality, from dynamic code at the client level.

Long calculations, DB lookups, etc.

## Use ai device

```
s1 = Scaler('XXX:scaler1')
s1.setCalc(2, '(B-2000*A/10000000.0)')
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)
```

# Subclassing Devices

Of course, a *device* can be subclassed, to add functionality.

## Scaler device

```
import epics
class Scaler(epics.Device):
    "SynApps Scaler Record"

    ...
    def OneShotMode(self):
        "set to one shot mode"
        self.put('.CONT', 0)

    def CountTime(self, ctime):
        "set count time"
        self.put('.TP', ctime)
```

Simply add Methods to turn a device into a full Object.

Can also complex functionality, from dynamic code at the client level.

Long calculations, DB lookups, etc.

## Use ai device

```
s1 = Scaler('XXX:scaler1')
s1.setCalc(2, '(B-2000*A/10000000.0)')
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)
```

**Example Program:** Read Ion Chamber currents, amplifier settings, x-ray energy, compute photon flux, post to PVs.

Needs table of coefficients (~16kBytes of data), but then ~100 lines of Python.

# Subclassing Devices

Of course, a *device* can be subclassed, to add functionality.

## Scaler device

```
import epics
class Scaler(epics.Device):
    "SynApps Scaler Record"

    ...
    def OneShotMode(self):
        "set to one shot mode"
        self.put('.CONT', 0)

    def CountTime(self, ctime):
        "set count time"
        self.put('.TP', ctime)
```

Simply add Methods to turn a device into a full Object.

Can also complex functionality, from dynamic code at the client level.

Long calculations, DB lookups, etc.

## Use ai device

```
s1 = Scaler('XXX:scaler1')
s1.setCalc(2, '(B-2000*A/10000000.0)')
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)
```

**Example Program:** Read Ion Chamber currents, amplifier settings, x-ray energy, compute photon flux, post to PVs.

Needs table of coefficients (~16kBytes of data), but then ~100 lines of Python.

A Motor Device has ~100 fields, and methods to move motors in User, Dial, Raw units, check limits, etc.

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

## Sample wx widget Code

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

txt_wid = wxlib.pvText(Parent, pv=PV('SomePV'),
                      size=(100,-1))

txtCtrl_wid = wxlib.pvTextCtrl(Parent, pv=PV('SomePV'))

dropdown_wid = pvEnumChoice(Parent, pv=PV('EnumPV.VAL'))

buttons_wid = pvEnumButtons(Parent, pv=PV('EnumPV.VAL'),
                            orientation=wx.HORIZONTAL)

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 ENUM states.
- **pvEnumButtons** Button sets for ENUM states.
- **pvAlarm** Pop-up message window.
- **pvFloatCtrl** editable text for Floats, only valid numbers that obey limits.

A common mixin class allows extensions to other widgets.

Function Decorators help write code that is safe against mixing GUI and CA threads.

## Some Epics wxPython Apps:

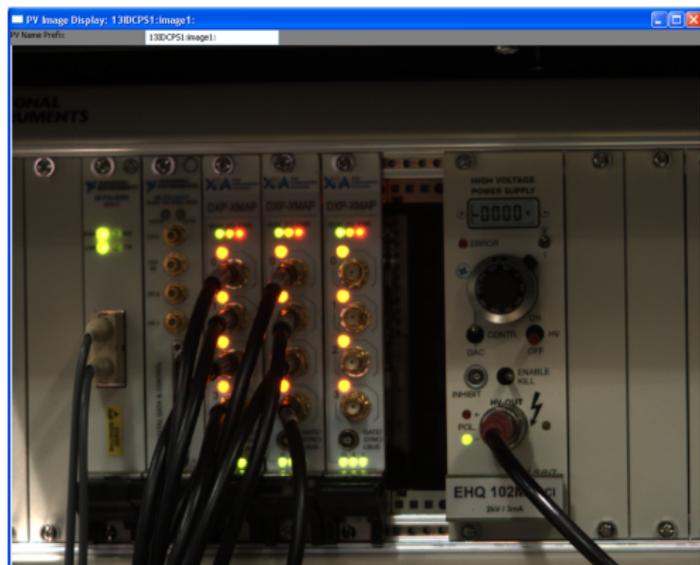


Area Detector Display.

A  $1360 \times 1024$  RGB image (4Mb)  
from Prosilica GigE camera.

Displays at  $\sim 10$ Hz.

## Some Epics wxPython Apps:



Area Detector Display.

A  $1360 \times 1024$  RGB image (4Mb)  
from Prosilica GigE camera.

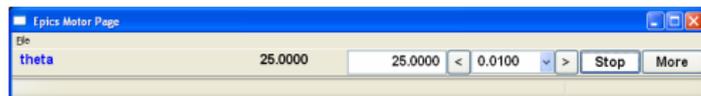
Displays at  $\sim 10$ Hz.

fineX	1.00000	1.00000	< 0.00100 >	Stop	More
fineY	-0.45000	-0.45000	< 0.20000 >	Stop	More
theta	25.0000	25.0000	< 0.0100 >	Stop	More
stageX	-0.4000	-0.4000	< 0.2000 >	Stop	More

MEDM-like Motor Display.

*Much easier to use.*

Built on Motor device.



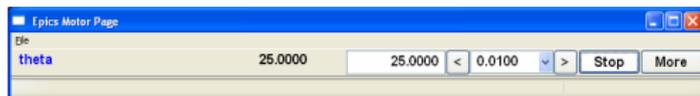
**Entry Values** can only be valid number.

**Entry Values** outside of limits are highlighted. On "Return", the nearest limit is displayed.

**Tweak Values** are auto-generated from precision and range.

**Cursor Focus** is sane (unlike MEDM).

**More Button** leads to Detail Panel.



**Entry Values** can only be valid number.

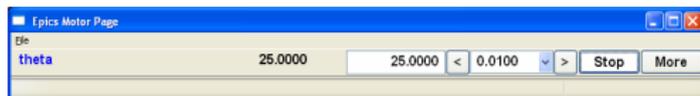
**Entry Values** outside of limits are highlighted. On "Return", the nearest limit is displayed.

**Tweak Values** are auto-generated from precision and range.

**Cursor Focus** is sane (unlike MEDM).

**More Button** leads to Detail Panel.





**Entry Values** can only be valid number.

**Entry Values** outside of limits are highlighted. On "Return", the nearest limit is displayed.

**Tweak Values** are auto-generated from precision and range.

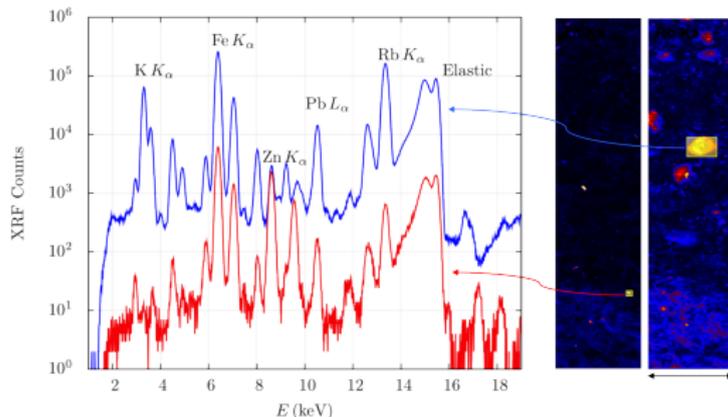
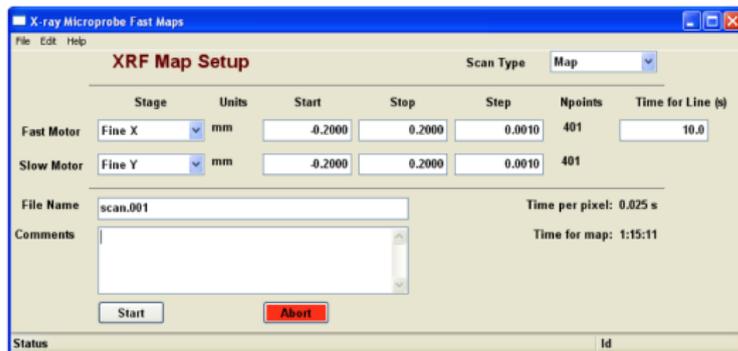
**Cursor Focus** is sane (unlike MEDM).

**More Button** leads to Detail Panel.

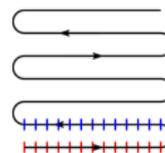
Multiple Motor Panels can be combined into **Instruments** definitions which can save/restore positions by name (in progress).



# Fast x-ray Fluorescence Mapping with an x-ray microprobe



Continuously scan a pair of Motor (at  $\sim 100 \mu\text{m}/\text{sec}$ ), triggering a detector at 100 Hz.



Scan and Motor Records have marginal support for this.

Using Newport XPS controller (and Python!) to define trajectories that triggers multi-element, multi-channel XRF detector, and AreaDetector to save data.

Client GUI coordinates data collection. Perfection in progress. ;)

### Summary:

- near complete low-level interface to CA.
- preemptive callbacks on by default.
- thread support.
- high-level PV class.
- GUI support (only wxPython so far).
- tested: linux-x86, linux-x86\_64, darwin-x86, darwin-ppc, win32-x86 (base 3.14.11).
- tested: Python 2.5, 2.6, 2.7, 3.1.
- Easy installation, including Windows.
- documented and some unit-testing (~70% coverage of core)
- Core routines (ca, dbr, PV, caget()/caput()): ~2000 lines of code.
- Devices, wxlib (Motor displays): ~1800 lines of code.
- <http://github.com/newville/pyepics>

Suggestions, contributions, collaborations welcome.