# PyCafe

## **Show and Tell**: Fri. June 17, 10:00, WBGB/021

- Establishing connections to EPICS Process Variables (PVs)
- Simple single channel operations
- Waveforms and arrays (memoryview data types)
- Multiple scalar operations, i.e., simultaneous operations on several PVs with scalar values
- Multiple compound operations, i.e., simultaneous operations on several PVs with either scalar values or arrays
- Synchronous group operations (user or externally defined)
- Control system parameters, i.e., operating limits, engineering units
- Monitors, either with or without user supplied callbacks
- Asynchronous interactions and retrieving data from cache
- Special methods, e.g., match, setAndMatch, wishlist …
- Error Handling and Reporting to Message Logger

# Preliminaries

```
#Python 3.5
export PYTHONPATH =
  $PYTHONPATH:/opt/gfa/cafe/python/python-3.5/<version>/lib


#Python 2.7
export PYTHONPATH =
  $PYTHONPATH:/opt/gfa/cafe/python/python-2.7/<version>/lib


#Python 2.6
export PYTHONPATH =
  $PYTHONPATH:/opt/gfa/cafe/python/python-2.6/<version>/lib

<version> =  {default, latest, pycafe}
```

**Examples:**

**/opt/gfa/cafe/python/python/python-<x.y>/examples/**

# Preliminaries

```
%import PyCafe and instantiate CAFE

cimport PyCafe

cafe = PyCafe.CyCafe()

cyca = PyCafe.CyCa()
```

CA enumerated types status codes:

```
cyca.ICAFE_NORMAL
```

Data types:

```
cyca.CY_DBF_DOUBLE
```

# Preliminaries

```
%import PyCafe and instantiate CAFE
cimport PyCafe
cafe = PyCafe.CyCafe()
cyca = PyCafe.CyCa()

% Exception handling
try:
    handle = cafe.open    ('pvName')
    val    = cafe.get     (⟨handlePV⟩, dt=⟨datatype⟩)
except Exception as inst:
    if isinstance(inst.args[0], PyCafe.CyCafeException):
      cafeEx=inst.args[0]
      cafeEx.show() %or
      print (cafeEx.errorCode,`:',cafeEx.errorText)

% Tidy up: stop any monitors, close channels,
% release all CA resources
cafe.terminate()
```

# Preliminaries

The ⟨**datatype**⟩ argument (of type 'str'), appearing in data retrieval methods is a placeholder for any one of the following set of supported Python data types: 'float', 'int', 'str'. Alternatively 'native' may be entered, as is the default, which specifies the data is to be presented in the Python equivalent native data type. Data transmitted over the network is (by default) in the native data type, but may be presented to user in any meaningful types.

The ⟨**handlePV**⟩ is a placeholder for either the Process Variable (PV) name, 'pvName', or handle (of type 'int'), which is the reference to the PV as returned from the channel `cafe.open` method.

# Basic Single Channel Operation

Data retrieval methods returning a **scalar** value

```
% Scalar returned as Python equivalent of native type.
% If waveform, only first element is returned
val = cafe.get (⟨handlePV⟩)

% Scalar returned as given Python ⟨datatype⟩, i.e.
% 'float','int','str'
% dt='native' returns the Python equivalent of the
% underlying native type
val = cafe.get (⟨handlePV⟩, dt=⟨datatype⟩)

% or equivalently with wrapper methods getFloat, getInt,
% getStr:
val = cafe.getStr (⟨handlePV⟩)
```

# Basic Single Channel Operation

Data retrieval methods returning a **Python List or Array**

```
valList  = cafe.getList (⟨handlePV⟩, dt=⟨datatype⟩)

valArray =
  cafe.getArray(⟨handlePV⟩, art=<arrayType>, dt=⟨datatype⟩)

% where ⟨arraytype⟩ may be one from
% 'memoryview','numpy.ndarray','array.array'


% memoryview returned by default
mv = cafe.getArray (⟨handlePV⟩, dt=⟨datatype⟩)
```

# Basic Single Channel Operation

Data retrieval methods returning **structured** data, i.e., timestamps, alarm status/severity values

```
pvdata  = cafe.getPV (⟨handlePV⟩, dt=⟨datatype⟩)

% pvdata has the following fields
pvdata.nelem              'int'    % No. of elements in value[] field
pvdata.value[]            ⟨datatype⟩
pvdata.alarmStatus        'int'
pvdata.alarmSeverity      'int'
pvdata.ts[]               'int'    % EpicsTimeStamp: List of 2 elements
pvdata.tsDate[]           'int'    % EpicsTimestamp as date: List of 7 elem
pvdata.status             'int'
% and methods
pvdata.show()                      % print all field values
pvdata.showMax(int n)              % print first n value[] elements of waveform
```

# Basic Single Channel Operation

**set/get** methods with **user supplied callbacks**

```
cafe.setCallbackGet (⟨handlePV⟩, cb=py_callback_get)
cafe.setCallbackPut (⟨handlePV⟩, cb=py_callback_put)
```

```
% A user supplied callback function for put operations
def py_callback_put (handle)
        print ("Put handler completed")
        %do stuff - whatever is wanted
        return

% A user supplied callback function for get operations
def py_callback_get (handle)
        cafe.getPVNameFromHandle(handle)
        value=cafe.getCache(handle)
        …
        return
```

# Basic Single Channel Operation

## **Asynchronous** data retrieval methods

**The asynchronous get** is a non-blocking interaction, i.e., it does not wait to retrieve the data from the control system. The subsequent `getCache` methods will, if necessary, wait until the underlying callback function is complete or a timeout is reached.

```
% Non-blocking data retrieval operation
cafe.getAsyn(⟨handlePV⟩)
  ...
% Data may be subsequently retrieved from cache. Some examples are:
val      = cafe.getCache      (⟨handlePV⟩, dt=⟨datatype⟩)
valList  = cafe.getListCache  (⟨handlePV⟩, dt=⟨datatype⟩)
valArray = cafe.getArrayCache (⟨handlePV⟩, art=⟨arrayType⟩,
                                           dt=⟨datatype⟩)
pvdata   = cafe.getPVCache    (⟨handlePV⟩, dt=⟨datatype⟩)
```

# Basic Single Channel Operation

## Setting value(s)

The **`set`** method is able to interpret all Python data types and caters for scalars, lists, and the various array types.

```
try:
  % Input data may be in any Python data type
  status = cafe.set (⟨handlePV⟩, data)
except Exception as inst:
  if isinstance(inst.args[0], PyCafe.CyCafeException):
    cafeEx=inst.args[0]
    cafeEx.show() %or
    print (cafeEx.errorCode,`:',cafeEx.errorText)
  else:
    print inst
```

# Opening and Closing Channels

## Opening channels

The `cafe.open` method creates the Channel Access (CA) virtual circuit and returns a handle (or handles) to the given PV(s). The following methods will wait for a default time period to establish a connection.

```
try:
  % Open (i.e. connect to) a single channel; returns an 'int'
  handle    = cafe.open ('pvname')
  % Open (i.e. connect to) a Python List of PV names; returns a Python List of 'int's
  handle[] = cafe.open (pvnames[])
except Exception as inst:
  print inst
```

An exception is not invoked if the channel is disconnected. In such cases the handle will be automatically activated on the channel's eventual connection.
It is not essential to explicitly invoke these cafe.open methods. They will otherwise be called internally by cafe at the time of the first data access operation.

# Opening and Closing Channels

## Opening channels

If there are multiple invocations of `cafe.open`, it would be more efficient and less of a time sink, to first prepare the messages before flushing the message buffer once at the end.

```
try:
  cafe.openPrepare ()
  % Open (i.e. connect to) a single channel; returns an 'int'
  handle    = cafe.open ('pvname')
  % Open (i.e. connect to) a Python List of PV names; returns a Python List of 'int's
  handle[] = cafe.open (pvnames[])
  % Now flush the message buffer and wait for waitTime (float) seconds
  cafe.openNowAndWait (⟨timeout⟩)
except Exception as inst:
  print inst
```

# Opening and Closing Channels

## Checking on the connection state of channels

```
% Connection state of ⟨handlePV⟩
isConnected  = cafe.isConnected (⟨handlePV⟩)
% Connection state of all channels
allConnected = cafe.allConnected()
% Display information on handle(s)
cafe.printHandle (⟨handlePV⟩)
cafe.printHandles()
cafe.printDisconnectedHandles()
cafe.printConnectedHandles()
```

# Opening and Closing Channels

## Closing channels

```
% Close connection to a single channel, referenced by either handle or PV name

handle    = cafe.close (⟨handlePV⟩)

% Close connection – all channels

handle[] = cafe.closeChannels()
```

# Multiple Scalar Operations

Operations to **`set/get`** a Python List of n scalar values to/from n PVs. Should one of the PVs be a waveform (or other multi-element record), then the get operation retrieves the first element only.

The input Python List in the **`set`** operation may contain varied data types.

The **`get`** operation will likewise, by default, return a Python List with each element of the List being presented in its native data type, unless otherwise specified by the dt=⟨datatype⟩, in which case the List is populated homogeneously with the specified data type.

# Multiple Scalar Operations

Simultaneous operations on multiple channels with scalar values

`if status != cyca.ICAFE_NORMAL` then loop thru `statusList[]`

```
% get
values[], status, statusList[]
        = cafe.getScalarList (⟨handlePVList⟩, dt=⟨datatype⟩)
% set
status, statusList[]
        = cafe.setScalarList (⟨handlePVList⟩, values[])
```

# Multiple Compound Operations

Operations to set/get a Python List of n compound values to/from n PVs.
A waveform is represented as a List within the List.

`if status != cyca.ICAFE_NORMAL` then loop thru `statusList[]`

```
values[], status, statusList[]
        = cafe.getCompoundList  (⟨handlePVList⟩,  dt=⟨datatype⟩)
status  = cafe.setCompoundList  (⟨handlePVList⟩,  values[])
% Example for PVs with various record types
PVNames = ['PV:AI','PV:MBBI','PV:WF']
values  = [1.23,'On',[1,2,3,4,5,6,7]]
status, statusList[]  = cafe.setCompoundList  (PVNames,  values)
values[], status, statusList[]  = cafe.getCompoundList (PVNames, dt='native')
% Above returns Values  = [1.23,'On',[1,2,3,4,5,6,7]]
values[], status, statusList[]  = cafe.getCompoundList  PVNames, dt='int')
% Above returns Values  = [1,     1, [1,2,3,4,5,6,7]]
values[], status, statusList[]  = cafe.getCompoundList  PVNames, dt='float')
% Above returns Values  = [1.23,  1.,[1.,2.,3.,4.,5.,6.,7.]]
values[], status, statusList[]  = cafe.getCompoundList (PVNames ,dt='str')
% Above returns Values  = ['1.23','On',['1','2','3','4','5','6','7']]
```

# Synchronous Group Operations

Defining and Opening Groups

(1) Groups may be defined on the fly.
(2) Pre-defined groups may also be read from an XML configuration file

```
cafe.loadGroupsFromXML(filename)
```

(3) or (for SwissFEL) loaded from memory (M. Aiba):

```
cafe.loadSFGroups
cafe.loadSFGroupsWithBase ('VA')
```

The members of the group do not necessarily have to be related
(i.e., be all of the same record and data type)

```
% Example for PVs with various record types
PVNames = ['PV:AI','PV:MBBI','PV:WF']
status  = cafe.defineGroup ('groupName',  PVNames)
try:
  groupHandle = cafe.openGroup ('groupName')
except Exception as inst:
  print inst
```

# Synchronous Group Operations

Groups may be identified in ⟨groupHandleName⟩ either by their 'groupName' or groupHandle.

`if status != cyca.ICAFE_NORMAL` then loop thru `statusList[]`

```
% Retrieving List of values; a waveform is represented as a List within the List.
values[], status, statusList[]
       = getGroup (⟨groupHandleName⟩, dt='native')

% Retrieves a pvgroup object
pvgroup  = getPVGroup (⟨groupHandleName⟩, dt='native')

% pvgroup has the following fields
pvgroup.npv              'int' % No. of PVs in group
pvgroup.pvdata[]          ⟨datatype⟩   % A List of pvdata types
pvgroup.name             'str' % Name of group
pvgroup.groupHandle      'int' % Group handle
pvgroup.groupStatus      'int' % Overall status of group operation
pvgroup.show()                 % Print all field values
pvgroup.showMax(int n)         % Print the first n group members
```

# Synchronous Group Operations

Groups may be identified in ⟨groupHandleName⟩ either by their 'groupName' or groupHandle.

`if status != cyca.ICAFE_NORMAL` then loop thru `statusList[]`

```
% set values to group members
PVNames = ['PV:AI','PV:MBBI','PV:WF']
setList  =[  23.3,      'on', [1,2,3]]

status, statusList[]
       = setGroup (⟨groupHandleName⟩, setList)

% Retrieves a pvgroup object
pvgroup  = getPVGroup (⟨groupHandleName⟩, dt='native')

setList = []
setList = cafe.PVGroupValuesToList(pvgroup)
setList[0]=setList[0]+1
status, statusList[]
       = setGroup (⟨groupHandleName⟩, setList)
```

# (Asynchronous) Group Operations

Groups may be identified in ⟨groupHandleName⟩ either by their 'groupName' or groupHandle.

`if pvgroup.groupStatus != cyca.ICAFE_NORMAL` then loop thru `pvgroup.pvdata[].status`

`cafe.getCompoundPVGroup` does a collective asynchronous get on all group members and waits until all channels have reported or a timeout is reached

```
% Retrieves a pvgroup object.
pvgroup  = cafe.getCompoundPVGroup (⟨groupHandleName⟩,
                                    dt='native')

% pvgroup has the following fields
pvgroup.npv              'int' % No. of PVs in group
pvgroup.pvdata[]          ⟨datatype⟩   % A List of pvdata types
pvgroup.name             'str' % Name of group
pvgroup.groupHandle      'int' % Group handle
pvgroup.groupStatus      'int' % Overall status of group operation
pvgroup.show()                 % Print all field values
pvgroup.showMax(int n)         % Print the first n group members
```

# Retrieving Control Parameters

```
pvctrl = cafe.getCtrl (⟨handlePV⟩, dt=⟨datatype⟩)
pvctrl = cafe.getCtrlCache (⟨handlePV⟩, dt=⟨datatype⟩)

% pvctrl has the following fields
pvctrl.nelem              'int' %No. of elements in value[] field
pvctrl.value[]            ⟨datatype⟩
pvctrl.alarmStatus        'int'
pvctrl.alarmSeverity      'int'
pvctrl.precision          'int' %The precision to which the value(s) is given
pvctrl.units              'str' %Engineering units
pvctrl.noEnumStrings      'int' %The number of enumerated types in mbbi/o records
pvctrl.enumStrings        list['str'] %List of enumerated types in mbbi/o records
pvctrl.upperDisplayLimit  'float‘  pvctrl.lowerDisplayLimit  'float'
pvctrl.upperAlarmLimit    'float‘  pvctrl.lowerAlarmLimit    'float'
pvctrl.upperWarningLimit  'float'  pvctrl.lowerWarningLimit  'float'
pvctrl.upperControlLimit  'float'  pvctrl.lowerControlLimit  'float'
% and methods
pvctrl.show()             %print field values
pvctrl.showMax(int n)     %print first n value[] elements of waveform
```

# Monitors

## Initiating a monitor

Monitors may be started with a number of optional input parameters, including a user-supplied callback function.

```
monitorID = cafe.monitorStart (⟨handlePV⟩, cb=py_cb,
dbr=⟨CY_DBR_TYPE⟩, mask=⟨CY_DBE_TYPE⟩)
```

```
%Simplest case
monitorID = cafe.monitorStart (⟨handlePV⟩)

% Data may be subsequently retrieved from cache, e.g.:
val      = cafe.getCache      (⟨handlePV⟩, dt=⟨datatype⟩)
valList  = cafe.getListCache  (⟨handlePV⟩, dt=⟨datatype⟩)
valArray = cafe.getArrayCache (⟨handlePV⟩, art=⟨arrayType⟩,
                                           dt=⟨datatype⟩)
pvdata   = cafe.getPVCache    (⟨handlePV⟩, dt=⟨datatype⟩)
```

# Monitors

```
monitorID = cafe.monitorStart (⟨handlePV⟩, cb=py_cb,
dbr=⟨CY_DBR_TYPE⟩, mask=⟨CY_DBE_TYPE⟩)
```

***dbr=⟨CY_DBR_TYPE⟩*** determines what parameter values are returned in addition to the value. Monitors are always started with the native data type, though the updated value may be selected in any meaningful type.

```
%dbr=⟨CY_DBR_TYPE⟩
dbr=cyca.CY_DBR_PLAIN    % monitor returns value only
dbr=cyca.CY_DBR_STS      % monitor returns value, alarm
                           status and alarm severity
dbr=cyca.CY_DBR_TIME     % (default) monitor returns
                           value, alarm status/sev,
                           timestamp
```

# Monitors

```
monitorID = cafe.monitorStart (⟨handlePV⟩, cb=py_cb,
dbr=⟨CY_DBR_TYPE⟩, mask=⟨CY_DBE_TYPE⟩)
```

***mask* = *⟨CY_DBE_TYPE⟩*** determines the trigger for the monitor callback:

| | |
|---|---|
| cyca.CY_DBE_VALUE | on value change |
| cyca.CY_DBE_ALARM | on alarm change |
| cyca.CY_DBE_LOG | on logging to the archiver |
| cyca.CY_DBE_PROPERTY | on change in enum property values of mbbi records as from EPICS v.3.14.11 |

```
% mask=⟨CY_DBE_TYPE⟩ is a logical OR of cyca.CY_DBE_xxx types
% default:
mask=cyca.CY_DBE_VALUE|cyca.CY_DBE_LOG|cyca.CY_DBE_ALARM
```

# Monitors and Callbacks

## Initiating a monitor with a user supplied callback

```
monitorID = cafe.monitorStart (⟨handlePV⟩, cb=py_cb)
```

```
def py_cb(handle):
    try: %Invoke any Cache operation to obtain updated value
        p1=cafe.getPVCache(handle)
        p1.show()
%  Not all CA operations are permitted, e.g..
% 'get', 'monitorStart' operations within the callback are
disallowed by CA
% 'set' operations, including those on other handles, are,
however, permitted.
%  Invoke Qt signals here to update Qt Widget
         …
    except Exception as inst:
        print "ERROR IN CALLBACK HANDLER"
        print inst
```

# Monitors and Callbacks

## Initiating several monitors at once

```
cafe.openMonitorPrepare ()

mID1 = cafe.monitorStart (⟨handlePV⟩, cb=py_cb)
…
mIDn = cafe.monitorStart (⟨handlePV⟩, …)

%just a little trick to speed things up slightly
cafe.setGetCacheWaitPolicy((⟨handlePV⟩,
                                cyca.GET_CACHE_NO_CHECK)

cafe.openMonitorNowAndWait (timeout)
```

# Monitors and Callbacks

## Initiating several monitors at once

```
cafe.groupMonitorStart (⟨groupHandleName⟩)

% a single user supplied callback for all monitors
 cafe.groupMonitorStart (⟨groupHandleName⟩, cb=py_cb, ...)

% separate user supplied callback for each group member
 cafe.groupMonitorStartWithCBList(⟨groupHandleName⟩,
cb=py_cb[], ...)
```

# Stopping a Monitor

```
% Stop all monitors belonging to a given handle
status = cafe.monitorStop (⟨handlePV⟩)

% Stop a specific monitor belonging to a given handle
status = cafe.monitorStop (⟨handlePV⟩, monitorID)

% Stop all monitors (for all handles)
status = cafe.monitorStopAll ()
```

# Timeouts

```
% Timeouts are self-regulating

% otherwise

cafe.setTimeout (float)

cafe.setSGTimeout (float)
```

# Specialized Methods

Match: set Channel1 and readback Channel 2

Method does not do a set but merely waits until the desired value, valueToReach, for the specified channel, <handlePV1> is reached within the given tolerance and timeout; method returns cyca.ICAFE_NORMAL as soon as a match is reached

```
status= cafe.match (valueToReach, <handlePV1>,
                                  tolerance, timeout, bint)
```

```
                                  printFlag = True or False
```

# Specialized Methods

setAndMatch: set Channel1 and readback Channel 2

Method verifies whether or not the two values agree within the given tolerance and timeout; method returns cyca.ICAFE_NORMAL as soon as a match is reached

```
status= cafe.setAndMatch (<handlePV1>, val1,
                          <handlePV2>, tolerance, timeout, bint)
                                                  printFlag=True or False
```

# Specialized Methods

setAndMatchMany:
set Channel_Set[M] and readback Channel_Read[M]

Method verifies whether or not the set/readback values agree within the given tolerance and timeout; method returns cyca.ICAFE_NORMAL as soon as a match is reached

```
status= cafe.setAndMatchMany (
                    <handlePVSetList>, valInputList,
        <handlePVReadBackList>, tolerance, timeout, bint)
                                                  printFlag=True or False
```

# Specialized Methods

setAndMatchMany:
set Channel_Set[M] and readback Channel_Read[M]

Method verifies whether or not the set/readback values agree within the given tolerance and timeout; method returns cyca.ICAFE_NORMAL as soon as a match is reached

```
status= cafe.setAndMatchMany (
                    <handlePVSetList>, valInputList,
        <handlePVReadBackList>, tolerance, timeout, bint)
                                                  printFlag = True or False
```

# Specialized Methods

Anything missing?

Wish list?

Would be routine to provide other specialized methods, for e.g. motor, undulator

# Monitors and Qt Widgets

[M. Aiba]

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *
import sys
import PyCafe
cafe=PyCafe.CyCafe()
cyca=PyCafe.CyCa()

class Form(QDialog):
    def __init__(self, parent=None):
        super(Form, self).__init__(parent)
        layout=QGridLayout()
        layout.addWidget(QLabel('Status:'), 0, 0)
        self.StatusLabel=QLabel('Open')
        layout.addWidget(self.StatusLabel, 0, 1)
        self.setLayout(layout)
        self.launchMonitor()
        self.connect(self.StatusLabel,SIGNAL("MonitorCallback"),self.updatePanel)
    def updatePanel(self,Stat):
        self.StatusLabel.setText(Stat)
    def launchMonitor(self):
        def cbmonitor(h1): # Monitor callback function
            c=cafe.getPVCache(h1)
            Value=c.value[0]
            self.StatusLabel.emit(SIGNAL('MonitorCallback'),Value)
        h1=cafe.open('VA-SINSS-LPSA:SHUTTER')
        m0=cafe.monitorStart(h1, cb=cbmonitor, dbr=cyca.CY_DBR_PLAIN,
mask=cyca.CY_DBE_VALUE)
app=QApplication(sys.argv)
form=Form()
form.show()
app.exec_()
```

# Terminating CAFE

```
cafe.terminate ()

% Stops all monitors,
% closes all channel access virtual circuits,
% and release all CA resources
```

# PyCafe Summary

PyCafe highlights:

- Synchronous, asynchronous interaction for individual, collection/groups of channels (optional XML configuration)
- Monitors made easy
- Error messages caught and reported with integrity
- Compilations for Python 2.6, 2.7 and 3.5
- Stable and robust API

**Examples:**
**/opt/gfa/cafe/python/python/python-<x.y>/examples/**
**<x.y>= 2.6, 2.7, 3.5**