A CYTHON INTERFACE TO EPICS CHANNEL ACCESS FOR HIGH-LEVEL PYTHON APPLICATIONS

J. Chrin, Paul Scherrer Institut, 5232 Villigen PSI, Switzerland

Abstract

Through the capabilities of Cython (a Python-like programming language with the performance of C/C++), a Pythonic interface to an in-house C++ Channel Access (CA) library, CAFE, has been developed, thereby exposing CAFE's numerous multifaceted and user-friendly methods to Python application developers. A number of particularities of the PyCafe extension module are revealed. These include support for (i) memoryview and other data types that implement the new Python buffer protocol (allowing data to be shared without copying), (ii) native thread parallelism, and (iii) pointers to callback functions from wherein CAFE methods may be effortlessly executed in asynchronous interactions. A significant performance improvement is achieved when compared with conventional Pythonic CA libraries. The PyCafe interface has been realized within the context of high-level application development at SwissFEL, Switzerland's X-ray Free-Electron Laser facility.

INTRODUCTION

The Python programming language is enjoying an increasing profile within accelerator facilities, and is, furthermore, the topic of a number of tutorials and contributions to this workshop. It is both an interpreted and object oriented programming language, with dynamic typing and binding. Its straight-forward syntax is advocated for promoting conciseness and readability, thus enhancing rapid code development. The availability of third-party computational modules such as NumPy [1] and SciPy [2], coupled with modules for the visualization of multi-dimensional data [3], add to its appeal as a platform for beam dynamics applications at SwissFEL, Switzerland's X-ray Free-Electron Laser [4], and elsewhere, e.g., [5]. The very nature of Python's dynamic semantics, however, inevitably results in an adverse effect on performance, especially with respect to low-level computation involving mathematical operations and looping constructs. Faster processing time may be achieved by using Python modules written largely in C/C++ as is the case for NumPy and SciPy, else by wrapping pre-existing C/C++ code or libraries into Python. The latter may be accomplished in a number of ways. The more traditional approach is to use Python's C Application Programming Interface (Python/C API), which necessitates a greater expertise in C than in Python. Other methods include the use of specialized tools such as SWIG, SIP, CFFI, the Boost.Python C++ library or the `ctypes` package from Python's own standard library. While all of these possibilities have their own specific flavour and target, the focus of this work is to explore the emerging Cython [6] technology to provide a high-performing Python interface to EPICS (Experimental Physics and Industrial Control System) [7, 8], through a well-tested, in-house, C++ Channel Access (CA) client library, CAFE [9–12]. In this way, a full complement of CAFE's multifaceted CA methods are readily made available to application developers in Python.

THE CASE FOR CYTHON

Cython is a high-level, object-oriented, and dynamic programming language, whose principle merit is to provide a Python-like style of coding while maintaining the performance level of C. Cython is equipped with the `cython` compiler that translates Cython source code into optimised C/C++ code, which in turn is compiled into a Python extension module. Performance enhancement is achieved through Cython's declaration of static C (and certain Python) type variables and with its ability to interface to C/C++ libraries, thus bypassing execution of bytecodes by the Python Virtual Machine (VM). Its primary use cases can thus be anticipated. Speed critical, CPU-bound, segments of Python code may be shifted into the Cython domain, wherein the dynamic Python runtime semantics are replaced by the static C semantics, and existing code in external C/C++ libraries may be natively accessed [13].

Cython is a fully-fledged language that is a super-set of Python, making it enticingly simple to customize, simplify or otherwise Pythonize interfaces as they are wrapped. Cython keeps abreast with developments in the Python programming language, accommodating new features as they appear, as exemplified by its support for the new Python protocol buffer and the *typed memoryview* object. Cython's ability to generate highly optimized code, when compared with those of other wrapping tools, also places it in good stead. It is thus well suited for the task of exposing the many, well-tested, methods of the C++ CAFE library to Python.

THE PyCafe PYTHON MODULE

The PyCafe extension module exposes the Cython interface to the CAFE C++ library (CyCafe). CAFE provides a concise, complete, and clean interface with minimal details of the low-level CA implementation propagating to the user, and an abstract layer tailored for beam dynamics applications. The underlying code base ensures that CAFE clients are entirely decoupled from the CA servers, allowing, e.g., data aggregator daemons and Graphical User Interfaces (GUIs) to function in every eventuality, i.e., irrespective of changes to the connection state of the EPICS channel. CAFE's multifaceted interface further provides the flexibility that eases CAFE's use as CA host to C/C++ based scripting and domain-specific languages [11, 12]. In CyCafe, for instance, methods

ISBN 978-3-95450-189-2

with Python lists as arguments map directly onto CAFE's C++ vector counterparts.

The near-full complement of the Python language, with its standard and third-party libraries, together with the given Cython constructs, are at the developers disposal for wrapping C libraries in Cython. This includes support for data types that implement the new Python buffer protocol which allows for their data to be shared without the need for copying, vastly improving performance in large data transfers. In creating the CyCafe wrapper class, a proper handling of CPython's Global Interpretor Lock (GIL) is critical to achieving the thread based parallelism that is essential for establishing callbacks from within Python in asynchronous operations.

Memoryview and Typed Memoryview Objects

The introduction of a new buffer protocol in Python allows a number of Python built-in types, e.g., `bytes`, `bytearray`, and extension types, such as the `array.array` and `numpy.ndarray` arrays, and other objects that implement the protocol, to share their data without copying. The C level buffer interface may further be exposed as a Python `memoryview` object that, among other attributes, supports slicing and indexing to expose a subview of the underlying data. They are hence well suited to data arising from, and destined to, EPICS waveform and subArray record types.

Cython can similarly extend the buffer protocol to work with data arising from external libraries through the C-level *typed memoryview* object, which projects, and expands on, the Python `memoryview` interface. CyCafe consequently provides interfaces that recognize such `memoryview` and `memoryviewslice` data types.

Thread Based Parallelism

Cython uses the Python/C API to access C-level code. CPython's memory management is not, however, thread-safe, necessitating a dedicated mutex, the GIL, to ensure that only one native thread executes Python bytecodes at any given time. External C code that does not interact with Python objects can, however, be executed without the GIL in effect, thus achieving thread-based parallelism. All methods that access the low-level hardware through channel access are done so in a non-GIL context, i.e., with the GIL released. Without taking this necessary step of releasing the GIL, PyCafe will otherwise hang, particularly in cases where callbacks are involved.

C Function Pointers and Callbacks

Cython supports C function pointers allowing C functions, that take function pointer callbacks as input arguments, to be wrapped [14]. This feature allows users to pass a Python function, created at runtime, to control the behaviour of the underlying C function. Using this methodology, a Python callback function may be easily supplied for any asynchronous CA interaction. The requirement for the external CAFE library to call into the CyCafe code necessitate the inclusion of a Cython generated header file into the C++ CAFE library. The header contains the functions and methods purposely declared by Cython's `api` keyword. The Python import mechanism is then invoked within CAFE to dynamically expose the `api`-declared functions.

THE PyCafe INTERFACE

Listing 1 (lines 1-3) shows how the PyCafe extension module is imported from within a Python application and how the Cython CyCafe object is instantiated. The CyCa class exposes various CA and CAFE enumerated types and status codes. The notion to establish CA connections to PVs ahead of time, and in combination (lines 5-12), is optional but represents good practise, not least as network traffic and total wait time are reduced. Subsequent CA operations on PVs may be undertaken by making reference to the PV name or its handle (as returned by `cafe.open`), which is an object reference to CAFE's allotted cache for the associated PV. If the CA connection methods are not explicitly invoked, they will otherwise be made autonomously at the time of the first PV data operation. Helper methods (not shown) provide information on the association between a given PV and its handle. CyCafe defined exceptions (`PyCafe.CyCafeExpection`) are, by necessity, thrown under the disguise of the generic Python `RuntimeError` exception, but may nevertheless be explicitly tested for by the user (not shown). Alternatively, PyCafe may be configured with the exception handler disabled for single channel operations, such that status codes are instead returned. On application exit, the `cafe.terminate` method (line 60) closes all PV connections, deletes their handles, and releases all CA resources. A few selected PyCafe data read, write and monitor transactions are presented.

Read and Write Operations

Listing 1 shows the Python syntax for selected synchronous and asynchronous, single and multiple channel data transactions through PyCafe. Methods operating on scalar values support the two basic Python numeric types, namely `float` and `int` and the string type, `str`. The Python equivalent of the PV's native data type is the default type in data retrieval operations, unless otherwise specified by the `dt` argument keyword (line 16). Where timestamps and alarm conditions are required, methods that return the `pvData` struct are invoked (lines 21-22).

A number of data retrieval operations are accompanied by an equivalent 'cache' method that retrieves the last value written into CAFE's internal buffer. These are typically used in conjunction with asynchronous data access interactions and, if necessary, will wait until the asynchronous operation has provided a value or a timeout has been reached (cf. future class). Since the underlying CAFE C++ library separates data acquisition from data representation, data of any type may be requested from cache (lines 43-46).

Data from EPICS waveform and subArray records are retrieved using the `cafe.getArray` method (lines 25-26). In addition to the standard Python built-in array types, Cython array features provide support for NumPy arrays and *typed*

ISBN 978-3-95450-189-2

memoryviews. The array type may be indicated through the `art` argument keyword, which may take on one of a number of self-explanatory options (line 17). Cython's *typed memoryview* is converted to a regular Python `memoryview` by CyCafe when passed as a return value.

Control system parameters may be retrieved as shown (lines 27-28). Since these data are typically static it is usually sufficient to retrieve them from cache, which is always populated on (re-)connection, if not otherwise.

The `set` method interrogates the form of the data input argument and can accommodate any meaningful type (lines 29-30).

The aggregation of channels, whether related or unrelated, into a collection allows several requests to be delivered within a single method invocation, thereby minimizing network traffic and increasing efficiency. Such methods are shown for scalar (lines 36-38) and compound (lines 39-41) data sets. The data container here is itself a Python list, which may legitimately contain elements of different data types (as may arise, e.g., in data retrieval operations that render the data from unrelated channels in their native type). Compound operations differ from their scalar counterparts in that they may also contain a list within a list, e.g., to accommodate waveforms.

Channels may, alternatively, be gathered into a named group to profit from the EPICS synchronous group functionality. Here the group acts as a single logical software entity and subsequent transactions are invoked through intuitive methods that reference the group either by name or handle (lines 48-55). Where associated timestamps and alarm status and severity data are required, a `pvgroup` object may be returned that contains a sequence of `pvdata` objects. While synchronous groups are simple and inexpensive to create, they have the inherent disadvantage that a timeout on a group operation is unable to inform on the offending channel(s), consequently rendering the returned data unreliable. (The underlying CAFE library, incidentally, operates a self-regulating timeout policy to increment timeouts in cases where the setting has not been optimized for the local network capacity). To overcome this deficiency, where data from synchronous group operations cannot be verified due to a timeout, a multiple channel transaction, i.e., `cafe.getCompoundPVList` (lines 57-58), is autonomously invoked and the resulting data is repackaged into the `pvgroup` object. In this way, and oblivious to the end-user, data integrity and error reporting are ensured for each individual channel within the group. Indeed, use of this latter method may be preferred by some users over the synchronous group option.

The multiple channel methods presented do not throw CAFE exceptions requiring the user to first check on the overall status of the method invocation; only on error need the status code for the individual channels be examined.

Monitors

A Python callback function may be easily supplied to any asynchronous operation, whether this be a set, get, or

Listing 1: PyCafe Read/Write Examples

```python
1  import PyCafe
2  cafe = PyCafe.CyCafe()
3  cyca = PyCafe.CyCa() #ca enums, status codes
4
5  pvList=['pv1','pv2','pv3',...]
6  waitTime=1.0 #seconds
7  try:
8    cafe.openPrepare()
9    hList = cafe.open(pvList] #ret. handles
10   cafe.openNowAndWait(waitTime)
11 except Exception as e:
12   ...
13
14 #<handlePV> in {hList[0], pvList[0]}
15 #<hPVList>  in {hList, pvList}
16 #dt =  {'native', 'int', 'float', 'str'}
17 #art = {'memoryview', 'numpy', 'array', ...}
18 try:
19   #get value in native (default) type
20   value   = cafe.get(handlePV)
21   #returns structured data, value as float
22   pvData  = cafe.getPV(handlePV, [dt='float'])
23   #waveform, returns list in native type
24   valList = cafe.getList (handlePV, [dt='native'])
25   #waveform, returns memoryview in native type
26   mv = cafe.getArray(handlePV, [art='mv'])
27   #control parameter data from cache
28   pvCtrl = cafe.getCtrlCache(handlePV)
29   #write operation for scalars and waveforms
30   cafe.set(handlePV, value) #value, any data type
31 except Exception as e:
32   ...
33
34 #synchronous multiple channel operations
35 #s gives overall status
36 #vList,sList, individual scalar values/status
37 vList,s,sList = cafe.getScalarList(<hPVList>)
38 s,sList = cafe.setScalarList(<hPVList>, vList)
39 #valList may contain lists within a list
40 valList,s,sList = cafe.getCompoundList(<hPVList>)
41 s,sList = cafe.setCompoundList(<hPVList>, valList)
42
43 #asynchronous multiple channel operations
44 s,sList = cafe.getAsyn(<hPVList>)
45 #apply any future cache method
46 pvData  = getPVCache(<hPVList[0]>, [dt='float'])
47
48 #synchronous group operations
49 s = cafe.defineGroup('groupName', pvList)
50 groupHandle = cafe.openGroup('groupName')
51 #<gHandleName> in {groupHandle,'groupName'}
52 valList,s,sList = cafe.getGroup(<gHandleName>)
53 s,sList = cafe.setGroup (<gHandleName>, valList)
54 #returns a pvgroup object, with native data types
55 pvgroup = cafe.getPVGroup(<gHandleName>)
56
57 #returns pvgroup from a multiple channel operation
58 pvgroup = cafe.getCompoundPVGroup(<gHandleName>,
59                                   [dt='native'])
60 cafe.terminate() #tidy up
```

ISBN 978-3-95450-189-2

Listing 2: PyCafe Monitor Example

```
...
#Callback function
def py_callback(handle):
  #Any method that retrieves data from cache
  pvData=cafe.getPVCache(handle, [dt='str'])
  #user supplied code
  ...
  return
...
#start monitor with user supplied callback
mon=cafe.monitorStart(<handlePV>, cb=py_callback,
      [dbr=cyca.CY_DBR_TIME],
      [mask=cyca.CY_DBE_VALUE|cyca.CY_DBR_ALARM])
...
```

monitor. This is highlighted in Listing 2 where a monitor on a PV is activated. Optional keyword arguments govern the behaviour of the monitor. A notable aspect of the PyCafe interface, here, is that only the handle (i.e., object reference) is, and need be, reported back to the callback function. Since the CAFE API takes the provision to cache the data in its internal storage, the user may call upon any one of a number of CAFE methods that retrieve data directly from the cache. The precedence of sifting through Python dictionaries is dispensed with. A typical use case of the callback would be to trigger the passage of data to a graphical widget.

The CAFE connection event handler has also been configured to trigger the given callback in the event of a channel disconnection or reconnection.

PERFORMANCE IMPROVEMENTS

Cython, in certain use cases, has the potential to enhance performance for CPU-bound operations by several orders of magnitude. Performance differences between Python and Cython are significantly decreased, however, for memory-, input/output-, and network-bound operations. As channel access transactions involve movement of data across the network, the resulting latency is dominated by the data transfer. This, consequently, allows for some leeway in the design of the interface. For instance, in certain cases, static type declaration for arguments can be omitted, preserving flexibility without detriment to performance. This is evident in methods that accept either a process variable name or object handle as a positional argument. Nonetheless, PyCafe, due to Cython's generated optimized C++ code, still manages a factor of four improvement in performance for a single scalar retrieval when compared with the Python/C API, and a 40% performance enhancement compared with `ctypes`, which are subject to their Python call overhead. For large waveforms, e.g., $\sim 10^6$ elements, the performance difference between Python and Cython essentially vanishes, although interfaces exploiting the `memoryview` array type result in a gratifying factor of two performance gain when compared with `numpy.ndarray`.

SUMMARY

The Cython programming language has been used to interface the external C++ CAFE library to Python, thereby providing a well-tested, high-performance, and extensive channel access interface to high-level beam dynamics application developers at SwissFEL. The project software may be downloaded from the CAFE website [9], where further examples of various usages are given.

ACKNOWLEDGMENTS

The author is grateful to Masamitsu Aiba, whose extensive use of PyCafe in the SwissFEL virtual accelerator, deploying a few thousand EPICS soft channels, led to improvements to CAFE's functionality and verified its Cython interface.

REFERENCES

[1] NumPy, `http://numpy.org`

[2] SciPy, `http://scipy.org`

[3] J. D. Hunter, "Matplotlib: A 2D graphics environment", in *IEEE Computing in Science and Engineering*, vol. 9, no. 3, pp. 90–95, May/Jun. 2007, `doi:10.1109/MCSE.2007.55`; matplotlib, `http://matplotlib.org`

[4] *SwissFEL Conceptual Design Report*, R. Ganter, Ed. PSI, Villigen, Switzerland, Rep. 10-04, Version Apr. 2012.

[5] T. Zhang, J. H. Chen, B. Liu, and D. Wang, "Python-based high-level applications development for Shanghai soft X-ray free-electron laser", in *12th Int. Computational Accelerator Physics Conf. (ICAP'15)*, Shanghai, China, Oct. 2015, pp. 23–25, `doi:10.18429/JACoW-ICAP2015-MODWC4`

[6] Cython C-Extensions for Python, `http://cython.org`

[7] EPICS, `http://www.aps.anl.gov/epics/`.

[8] J. O. Hill and R. Lange, "EPICS R3.14 Channel Access Reference Manual",
`http://www.aps.anl.gov/epics/docs/ca.php`

[9] CAFE, `http://ados.web.psi.ch/cafe/`.

[10] J. Chrin and M. C. Sloan, "CAFE, A modern C++ interface to the EPICS channel access library", in *Proc. 13th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'11)*, Grenoble, France, Oct. 2011, paper WEPKS024, pp. 840–843.

[11] J. Chrin, "MATLAB objects for EPICS channel Access", in *Proc. 14th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'13)*, San Francisco, CA, USA, Oct. 2013, paper MOPPC146, pp. 453–456.

[12] J. Chrin, "An update on CAFE, a C++ channel access client library and its scripting language extensions", in *Proc. 15th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'15)*, Melbourne, Australia, Oct. 2015, pp. 1013–1016,
`doi:10.18429/JACoW-ICALEPCS2015-WEPGF132`

[13] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The best of both worlds", *IEEE Computing in Science and Engineering*, vol. 13, no. 2, pp. 31–39, Mar./Apr. 2011, `doi:10.1109/MCSE.2010.118`

[14] K. W. Smith, in *Cython*, Sebastopol, CA, USA: O'Reilly Media, Inc., Jan. 2015, pp. 128–134.