PicklingTools Documentation

Release 1.7.0

Richard T. Saunders

December 14, 2016

CONTENTS

Contents:

CHAPTER

ONE

THE PICKLINGTOOLS 1.7.0 USER'S GUIDE

1.1 Introduction

The PicklingTools distribution is an Open Source package of tools to allow systems written in the Python Language and/or the C++ Programming Language to exchange information easily. (Recently, some support for Java has been added as well) It allows:

- Socket communications
  - UDP (for speed)
  - TCP/IP (for reliability)
- Multiple file formats
  - text (for readability)
  - binary (for speed)
  - older Python formats (for compatibility)
  - older Midas 2K formats: OpalTable (for legacy)
  - JSON (for ubiquity)
  - XML (for modern systems)
- Support for Shared Memory
  - Processes related by inheritance
  - Processes unrelated

Historically, the PicklingTools was a collection of tools to allow non-Midas 2k clients to talk to a popular software package written in Midas 2k (Midas 2k is a legacy framework). Since then, it has evolved into a full-fledged set of client/server and file tools to allow any C++ and Python applications to talk easily (whether they be raw C++, raw Python, Midas 2k, X-Midas C++ Primitives, or XMPY). *It is worth noting that the PicklingTools is not tied to any particular framework, the PicklingTools libraries are plain C++ and Python code that any system can use.* Java support is new as of PicklingTools 1.5.1.

For example:

- Users who don't care about Midas: A client written in raw C++ and a client written as a raw Python script can both talk to server written in raw C++.

- Users in the Midas world: An client written in XMPY (Python) and a client written in C++ as an X-Midas primitive could both talk to an Midas 2k server (OpalDaemon or better yet, OpalPythonDaemon).

The PicklingTools facilitates Python and C++ and Java applications communicating.

All communication (via sockets or files) is done via *Python dictionaries*. An example Python dictionary looks like:

```
{'Request': { 'PING': {'TimeOut': 5.5, 'PORTS':[88, 89]} } } # Text format
```

```
Figure 1:  An example of a Python dictionary in text format
```

A dictionary is essentially a collection of key-value pairs where the values can be integers, real numbers, arrays or other dictionaries. Another phrase that describes Python dictionaries: dynamic, recursive (because it can contain other dictionaries), heterogeneous (because it can contain many types of objects) collections. Python dictionaries provide very similar functionality to XML formats, but Python dictionaries tend to be easier to read and manipulate. In fact, the JSON (JavaScript Object Notation) is fairly backwards compatible with Python Dictionaries and is a competing standard to XML (see http://json.org).

Python dictionaries are the *currency* of the PicklingTools.

1.2 Formats

The PicklingTools allow Python dictionaries to be exchanged in two major formats: text or binary (In fact, the reason this distribution is named "PicklingTools" is because the major binary format in Python are "pickled" dictionaries). Text (like Figure 1) is a good human readable and editable format. It's easy to read, but tends to be slower to exchange in a file or over a socket. The binary format tends to be much faster to exchange in a file or over a socket, but it's harder to read/write without special editors. The choice of serialization really depends on your constraints.

- Text: There are four "human readable" formats:
  1. **Python Dictionaries:** the standard Python dictionary
  2. **Midas 2k OpalTables:** A stovepipe construction from Midas 2K that mirrors Python Dictionaries.
  3. **JSON:** A world-wide standard for recursive, heterogeneous containers
  4. **XML:** A world-wide standard for recursive, heterogeneous containers

  There are tools in both Python and C++ to read/write both formats.
- **Binary: There are a myriad of choices for Binary Serialization, in order**

  from fastest to slowest:

  1. **OpenContainers Serialization:** This format exists only inside the PicklingTools and has been specialized for the C++ components. If you want absolute speed, this is probably the fastest format. New! As of PicklingTools 1.6.0, there is now a Python C Extension module for this.
  2. **Python Pickling Protocol 2:** This is a binary serialization that Python has built-in. If most of your communications are with Python systems, this is probably the best format as Python understands it natively and it is just about as fast as OpenContainers serialization.
  3. **Midas 2k Serialization:** This is the binary serialization taken directly from Midas 2K. If you need to talk to legacy Midas 2k applications, this is a good, fast choice.
  4. **Python Pickling Protocol 2 for Python 2.2:** This is Protocol 2 as implemented by Python 2.2. Python interpreters above version 2.2 implement this differently. ONLY USE this if you have to talk to a Python interpreter that MUST BE 2.2.

5. **Python Pickling Protocol 0:** Strictly speaking, this is an ASCII format (7-bit clean), but not nearly as readable as the textual formats. This is by far the most backwards compatible format, as all Pythons of interest seem to understand this protocol well. This is the default of most components in PicklingTools for compatibility reasons, but it definitely not the fastest.

Your choice of protocol will frequently be dictated by the components involved in your system. If you must talk to a legacy M2k application that can't be changed, M2k serialization is your only choice. If you are using Python in your system, you are probably constrained to use Pickling Protocol 0 or 2 (as Python currently doesn't understand OpenContainers or Midas 2k serialization).

Serializations can also mix-and-match as needed: if the majority of a system is in C++, OpenContainers serialization is probably the best choice when possible, with selectively chosen Python or M2k serialization when needed. Many people choose the text format (eschewing binary serialization altogether) for transparency.

One other note: The C++ versions of the socket clients and servers (this includes the Midas 2k and X-Midas Primitives) use *adaptive* serializations: This means on a per-client basis, the servers and clients can recognize different serializations. In other words, a C++ server can simultaneously talk to a Python client using Protocol 2 and a M2k client using M2k serialization. The Python servers and clients don't currently support adaptive serialization because they are very much constrained to their native serialization (for example: Python 2.2 supports a different version of Protocol 2 than Python 2.3, 2.4, etc.).

1.3 Media

There are two major ways to communicate with the PicklingTools: via files or over sockets from Python and C++ (and Java to a lesser extent). *NOTE: Whenever we say "Python and C++", we mean the whole range of systems supported directly: XMPY, X-Midas C++ primitives, Midas 2K C++ components, raw C++, raw Python.*

A file can be read/written using any of the the formats (within a few constraints) described from the previous section: This can be done from Python and C++ and Java. See later sections for example of this within C++, Python, etc.

We can also build socket servers and clients from Python and C++ and Java:

The MidasTalker/MidasServer pair are TCP/IP (respectively) clients and servers. The same classes exist in both C++ and Python and they have very similar interfaces.

The MidasListener/MidasYeller pair are UDP (respectively) clients and servers. The same classes exist in both C++ and Python and they have very similar interfaces.

The *OpalPythonDaemon* is the Midas 2K equivalent of the MidasServer. This replaces the original OpalDaemon, but is still backwards compatible with OpalDaemon.

The *OpalPythonSocketMsg* is the Midas 2K equivalent of the MidasTalker. This replaces the OpalSocketMsg component, but is still backwards compatible with OpalSocketMsg.

There will be more discussion and full examples later in this document.

1.4 Conclusion...But Keep Reading!

The primary major goal of the PicklingTools is to allow users to talk to legacy systems (X-Midas, Midas 2k) from systems written in Python or C++ or Java. This has been accomplished: a number of users have been able to interface to legacy systems. [This is also the reason we didn't consider Twisted as a network communication system: it didn't support the legacy protocol, and it seems to be Python only].

A later major goal was to allow C++ and Python (and later Java) systems to interact easily. In other words, even if you don't have a legacy system that you must talk to, the tools provided here still allow you to build new systems out of both C++ and Python pieces and have them communicate easily (over files or sockets).

A minor goal of the PicklingTools was to allow the C++ experience to be similar to the Python experience, yet still allow threads in a C++ system. Anyone who wishes to use threads from Python knows that Python (at least CPython, the most prevalent Python implementation) doesn't support truly concurrent threads. To this end, the OpenContainers library has been provided within the distribution: It provides abstractions for both threads and dictionaries within C++. [This is also the reason we don't just embed a Python interpreter directly within C++ to get a "Python-like" experience: The interpreter doesn't support concurrent threads.]

This minor goal now extends to Java: the Python dictionary is fairly easy to manipulate in Java.

One other minor goal when using C++ is the ability to use *valgrind* and other such tools. Most tests and code in the framework should be valgrind-clean, and you should always be able to work with valgrind to help you debug.

The rest of this document gives examples for many of the tools provided herein. The full API for the PicklingTools distribution is also included in the distribution, in the "Pickling API" document.

1.5 The Python Experience

Python is the easiest place to get started and get familiar with the tools. Just about everything we need is built-in to Python: dictionaries, socket code (*import socket*) and serialization code (*import cPickle*).

One minor goal of the Python experience for the PicklingTools is that *any* Python interpreter could just use the PicklingTools:

```
>>> import midastalker  # Make sure PicklingToolsXXX/Python on PYTHONPATH
```

Done. All the PythonPickling tools are written in pure Python so they be imported directly without worrying about any unnatural dependencies. In other words, we didn't want to write any C extension modules which would cause issues with linking: Who builds the extension module? Which Python is it linked against? Can you even build on your machine? Luckily everything we need was built-in and fast enough to support the Python PicklingTools. The NumPy or Numeric support may an issue: see the XMPY experience below.

Whether you are using Python that just comes on the machine or XMPY (a version of Python built explicitly for X-Midas), you should just be able to *import* and it'll work. [Sidebar with PYTHONPATH?]

1.5.1 Files

Reading and Writing Python Dictionaries from Text Files:

This can be done with all built-in constructs. If we want "rfile" to contain a textual Python dictionary:

```
>>> # Write out a text dict
>>> o = { 'a':1, 'b':2.2, 'c':'three' }   # dict to write
>>> f = file('rfile', 'w')
>>> f.write(repr(o))            # get the string representation to write
>>> f.close()

# The file 'rfile' contains text:  {'a':1, 'c':three', 'b':2.2 }

>>> # Read in a text dict
>>> d = eval(file('rfile', 'r').read())    # d has the dictionary
```

The pretty module is very useful for writing dictionaries to files in a way that exposes the hierarchical structure better than a plain repr. There is more discussion in a section below:

```
>>> o = {'a':1, 'b':{'nest': None} }
>>> import pretty
>>> pretty.pretty(o)        # Exposes nesting and structure: easier to read
{
    'a':1,
    'b':{
        'nest':None
    }
}
>>> f = file('prettyout.txt', 'w')
>>> pretty.pretty(o, f)                  # Write out file pretty
>>> exit()
% cat prettyout.txt
{
    'a':1,
    'b':{
        'nest':None
    }
}
```

Note that the eval method still works for pretty printed dictionaries in files:

```
>>> d = eval(file('prettyout.txt', 'r').read())
>>> print d
{'a': 1, 'b': {'nest': None}}
```

Reading OpalTables from Text Files as Dictionaries:

```
>>> import opalfile    # Part of the PicklingTools distro, in Python subdir
>>> d = opalfile.readtable('opaltextfilein.tbl')
*** Unfortunately, this currently only works if you are using XMPY
```

Writing Dictionaries to Text Files as OpalTables:

```
>>> import opalfile  # Part of the PicklingTools distro, in Python subdir
>>> opalfile.writetable('opaltextfileout.tbl')
*** Unfortunately, this currently only works if you are using XMPY
```

The "opalfile.py" also contains code for reading and writing OpalFiles (a large data binary file format from M2k). For more information, use the built-in help facility:

```
>>> import opalfile
>>> help(opalfile)
```

The *opalfile* module only works if you have Numeric (which XMPY, a version of Python specifically compiled with Numeric and a few X-Midas libraries): there is also a module for pretty printing OpalTables *that does NOT require Numeric*: *prettyopal*:

```
>>> from prettyopal import prettyOpal
>>> prettyOpal( [ 1, 'two', 3.5 ] )       # Plain Python: no external depends
{
    "0" = L:1 ,
    "1" = "two" ,
    "2" = D:3.5
}
```

1.5.2 Sockets

MidasTalker: TCP/IP client

Always start with the built-in documentation:

```
>>> import midastalker
>>> help(midastalker)
```

If you wish to create client to talk to a MidasServer or OpalPythonDaemon, it's very easy. You need to know:

1. what machine the server is running on
2. The port the server is using
3. (probably) the type of serialization the server is using

Once you know that, using a Midastalker is easy. Let's say the server is running on "bradmach" on port 8888 using Python Pickling 2 serialization protocol. Then, to create a client:

```
>>> from midastalker import *
>>> mt = MidasTalker("bradmach", 8888, SERIALIZE_P2)
```

Once you have created the client, you need to open the connection:

```
>>> mt.open()  # tries to open, throws exception if fails
```

Sending and receiving is easy once the connection is open: Remember, the currency of PicklingTools is Python Dictionaries, so that's what flows over the socket!:

```
>>> request = { 'a': 1, 'b': 2 }
>>> mt.send(request)           # send a simple dictionary to server
>>> result = mt.recv()         # get a response dictionary back from server
>>> print result
{ 'a': 1, 'b': 2 }    # ... echoed back the same response
```

This is a very simple example, but shows all the major pieces of communicating with a server. Of course, there are a lot of other issues to worry about:

1. What if the server isn't available? (More likely, used the wrong server and port)
2. What is the server goes away after it opens up?
3. What if I want a timeout?

A full example called *midastalker_ex2.py* in included in distribution: It's a full Python program that shows how to deal with real issues: typically, if the server goes away or you can't connect, an exception will be thrown and you have to deal with by trying to reconnect. This is probably the best example to copy for creating a robust client.

See the full documentation of MidasTalker in API documentation. Note you can see the same documentation from Python using *help(midastalker)*.

MidasServer: TCP/IP server

Always start with the built-in documentation:

```
>>> import midasserver
>>> help(midasserver)
```

Creating a server is a little more complicated. It requires you to create a class that inherits directly from the MidasServer. When you create an instance, you will have three methods that get called for you (callbacks):

acceptNewClient_: Called for you whenever a new client connects to the server: It is a callback that gives the file descriptors for read/write access to the socket

readClientData_: Called for you whenever a client sends you data. The client can be uniquely identified by his file descriptor (the same one that was passed in with the `acceptNewClient_`)

disconnectClient_: Called for you whenever a client disconnects from socket

You will write a MidasServer that encapsulates the server behavior you want: what kind of messages you send back to clients, what to do when they disconnect or disconnect. Once you have one written, you tend to use them like the MidasTalker:

```
from midasserver import *

# New kind of Server written in Python
class MyNewMidasServer(MidasServer) {
    def __init__ (self, host, port, serialization) :
        MidasServer.__init__(self, host, port, ser, 1)
        ...
    def acceptNewClient_(self, read_fd, read_addr, write_fd, write_addr):
        ...
    def readClientData_(self, read_fd, write_fd):
        ...
    def disconnectClient_(self, read_fd, write_fd):
        ...

# Usage
MyNewMidasServer server("host", port, SERIALIZE_P2)
server.open();   # To start it going, accepting clients and responding
```

There is an example of a MidasServer in the Python area called *midasserver_ex.py* which shows a simple echo server (it echoes back what you send it). The servers are typically written to *respond to client requests*, but they could very easily be active servers. Whenever a new client connects, you could spawn a thread that immediately begins talking to the client (using the file descriptor given). It's important to keep track of the file descriptor handed to you: it's how you communicate back to the client: see *midasserver_ex.py*.

MidasYeller and MidasListener: UDP

The MidasListener (client) and MidasYeller (server) have very similar interfaces to that of MidasTalker (client) and MidasServer (server). The major difference is the the Yeller/Listener use UDP (User Datagram Protocol, frequently called Unrealiable Datagram Protocol because there are no guarantees on whether packets will be delivered, nor are there guarantees on the order). The listener and yeller both require the user specify a message length limit in the constructors: *both the Yeller and Listener need to match!!* See the *midaslistener_ex.py* and *midasyeller_ex.py* examples in the Python area.

1.5.3 Pretty Printing

One output format that Midas 2k users became very comfortable with is the prettyPrint routine for OpalTables: it exposes nesting of dictionaries and lists in a very human readable way. Below is the prettyPrint of an M2k OpalTable:

```
a = {
    ATTRIBUTE_PACKET={ },
    FILE_VERSION=UL:3,
```

```
    GRANULARITY=UL:4096,
    KEYWORDS={ },
    MACHINE_REP="EEEI",
    NAME="group(,)",
    TIME={
        DELTA=DUR:1,
        KEYWORDS={ },
        LENGTH=UX:4096,
        NAME="Time",
        START=DUR:0,
        UNITS="s"
    },
    TIME_INTERPRETATION={
        AXIS_TYPE="CONTINUOUS"
    },
    TRACKS={
        "0"={
            AXES={
                "0"=@"TIME"
            },
            FORMAT="D",
            KEYWORDS={ },
            NAME="Track 0",
            UNITS=""
        }
    }
}
```

Because this output format was so successful and useful with Midas 2k, there is a module called *pretty* that gives the Python user similar types of output for Python Dictionaries:

```
>>> import pretty
>>> help(pretty)
```

Note that this is different from the built-in Python module *pprint*:

```
>>> from pretty import pretty
>>> from pprint import pprint
>>> a = {'a':1, 'b':{'nest':None} }
>>> print a
{'a': 1, 'b': {'nest': None}}
>>> pprint(a)       # Built-in pretty print: Tends to keep on fewer lines
{'a': 1, 'b': {'nest': None}}
>>> pretty(a)       # PicklingTools: Exposes list and dictionary structure
{
     'a': 1 ,
     'b': {
         'nest': None
     }
}
```

The PicklingTools pretty print exposes structure better, and tends to be easier to read for larger tables, although it can be more verbose.

The pretty print function can also write to files:

```
>>> pretty(a, file('outfile', 'w'))     # Write pretty repr to outfile

% cat outfile
{
    'a':1,
    'b':{
        'nest':1
    }
}
```

1.5.4 Binary Formats

If you wish to read and write files in a binary format, use the cPickle format:

```
>>> import cPickle
>>> a = {'a': 1, 'b': 2 }
>>>    #### SAVING
>>> string1 = cPickle.dumps(a, 0)# binary dump a to a string using Protocol 0
>>> string2 = cPickle.dumps(a, 2)# binary dump a to a string using Protocol 2
>>> file('p0format", 'w').write(string1)

>>> file('p2format", 'w').write(string2)

>>>    #### LOADING
>>> f = file('p0format', 'r').read()
>>> l1 = cPickle.loads(f)     # loads both protocol 0 and protocol 2
>>> print l1
{'a':1, 'b': 2}
```

See the *cPickle* documentation that comes with Python for more examples. We recommend using Python Pickling Protocol 0 for compatibility and Python Pickling Protocol 2 for speed. **DO NOT USE Python Pickling Protocol 1!!!** The C++/Midas 2k/X-Midas PicklingTools DO NOT SUPPORT Protocol 1: Only 0 and 2. (3 will be in a future release).

The files produced by above technique can then be read/written by the C++/Midas 2k/X-Midas PicklingTools or Python.

Note that when we specify SERIALIZE_P0 or SERIALIZE_P2 from the MidasTalker/Server/Yeller/Listener, we are simply taking the data passed to "send" or "recv" and calling cPickle.dumps()/loads() on it.

NEW: As of PicklingTools 1.6.0, Python now has a C Extension module that works with OC Serialization.

```
>>> import pyocser  ### Make sure build/../pyocsermodule.so on your PYTHONPATH
>>> a = {'a':1, 'b':2 }
>>>   ##### SAVING
>>> string1 = pyocser.ocdumps(a)    # Binary dump using OC Serialization
>>> file('something.oc','w').write(string1)


>>>   ##### LOADING
>>> f = file('something.oc', 'r').read()
>>> l1 = pyocser.ocloads(f)
>>> print l1
{'a':1, 'b':2}
```

The OC Serialization tends to be faster than pickling. OC Serialization also works with very large strings and arrays (over 4G), whereas pickling probably doesn't (maybe will be fixed by the time you read this?).

1.5.5 Conclusion to the Python Experience

When in doubt, check the help page for the module of interest: there should be enough documentation to get you going.

```
>>> import midassocket  # Base class for ALL Midas socket thingees
>>> help(midassocket)
```

We strongly suggest learning how the Python PicklingTools work first because they are easy to use, well documented, and easy to try. There are many examples in the baseline: start by trying to copy/modify one of the examples to get going.

Once you feel comfortable with the Python PicklingTools, many of the same interfaces exist for the C++ PicklingTools: the C++ experience will hopefully feel very similar to the Python Experience (except there will be more *{ }* and *;* in the C++ experience).

1.6 The XMPY Experience

The only real difference between XMPY and Python is that XMPY has access to a few more C extension modules that are not available from a "standard Python" distribution: most of these are X-Midas specific and not of concern to this document. The important exceptions are the *Numeric* or *NumPy* modules.

1.6.1 Numeric or NumPy

The Numeric module (or the NumPy) allows the Python programmer to deal with large arrays of complex/real numbers and operate on them AT THE SPEED OF COMPILED C. The Numeric module is written in C, and implements a lot of common numeric operations in C. In other words, if there is a lot of numeric processing (multiplying matrices, manipulating large arrays), the Numeric module makes that functionality fast and available from Python.

NumPy is the current de-facto standard, but Numeric is the older, deprecated standard. NumPy is in current maintenance, whereas Numeric has fallen out of maintenance. Unless you have major backwards compatibility concerns, we strongly recommend using NumPy over Numeric.

NumPy comes with most Linux distributions, or is easy to install. RedHat and Fedora both have numpy RPMs that are easy to install. Few machines will actually have Numeric installed by default unless you are XMPY (there is a Red-Hat RPM that will allow Numeric to be installed inside of a standard RedHat Python in /usr/bin), but most of the time you'll have to install it yourself. Again, if you use XMPY, Numeric comes built-in.

The reason this is an issue: if you are serializing large amounts of POD type (Plain Old Data—this is the kind of data Numeric operates on: ints, float, complexes), then choosing Numeric/NumPy as your array serialization can make a world of difference in speed.

Consider (in Python):

```
>>> # Construct some data, and send it over a socket
>>> import numpy
>>> from midastalker import *

>>> a = numpy.array([1.0,2.0,3.0])    # Contiguous array of real_8s
```

```
>>> mt = MidasTalker("host", port, SERIALIZE_P2)
>>> mt.send(a)
```

By default, the MidasTalker will convert the *Array<real_8>* to a Python List (aka. C++ Arr) and then send that converted data over the socket. In code, it essentially does:

```
>>> # When we don't use Numeric for serialization, all Arrays of POD
>>> # data are converted to Python Lists and THEN sent over
>>> python_list = []
>>> for ii in xrange(0, a.length) :
...   python_list.append(a[ii])
>>> mt.send(python_list)
```

For two reasons, this is a lot slower:

1. An extra conversion to an Python List has to happen
2. copying elements one-by-one is a lot slower than copying the contiguous memory of an array by "BIT-BLIT" (which is an optimization with POD types).

Why is "without Numeric/NumPy" the default? That is the most backwards compatible way to send Array<POD> data because not all Pythons support Numeric or NumPy. If you try to send Numeric/NumPy data to a version of Python that doesn't have it built-in, the Python side will probably fail will an esoteric error message.

If you are convinced that all your clients understand NumPy (all C++ components do, all X-Midas components do, all XMPY interpreters do), then using NumPy can drastically decrease your serialization time of large amount of scientific data:

```
>>> mt = MidasTalker("host", port, SERIALIZE_P2, DUAL_SOCKET, AS_NUMPY)
>>> mt.open()     # Have to open before sending ...
>>> mt.send(a)    # Send NumPy arrays very fast
```

This extra argument on the Midastalker (similar for MidasServer, MidasListener, MidasYeller) to force the Midastalker to use NumPy is called the *ArrayDisposition*. This option ONLY APPLIES if you are using SERIALIZE_P0 or SERIALIZE_P2—it is ignored if you use any other serializations.

1.6.2 ArrayDisposition

ArrayDisposition is how to handle Arrays of POD (Plain Old Data). ArrayDisposition is only relevant if you are using SERIALIZE_P0 or SERIALIZE_P2 as your serialization (which you will be if you are talking to a Python client).

You can ignore the discussion below on ArrayDisposition if you use SERIALIZE_OC or SERIALIZE_M2K.

There are actually four choices for the ArrayDisposition argument on the MidasTalker/MidasServer/MidasListener/MidasYeller:

```
AS_NUMERIC      = 0
AS_LIST         = 1  # the default
AS_PYTHON_ARRAY = 2
AS_NUMPY        = 4
```

We currently recommend AS_NUMPY, as NumPy is in active maintenance. Unless you have backwards compatibility issues, we recommend moving from Numeric (which is out of maintenance) to NumPy (actively developed). Even AS_PYTHON_ARRAY is deprecated, as Python changed how it pickles array.array from Python 2.6 to Python 2.7 (so a Python 2.6 and 2.7 client would be incompatibile, and it is significantly slower as well).

The default is usually AS_LIST. To use NumPy:

```
>>> import numpy
>>> from midastalker import *
>>>
>>> a = numpy.array([1,2,3])
>>> mt = midastalker("host", port, SERIALIZE_P2, DUAL_SOCKET, AS_NUMPY)
>>> mt.open()     # Must open before sending
>>> mt.send(a)
```

Versions of Python prior to 2.5 do not support the serialization of arrays even though the Python array module has been built-in to the Python interpreter for ages. This simply gives the non-XMPY users or users who don't have access to Numeric another way to send Arrays of POD data.

If you try to use AS_NUMPY, AS_NUMERIC or AS_PYTHON_ARRAY and your version of Python does not support it, a large error will be issued to let you know you can't do this: THIS IS ON PURPOSE. It is better to get a big, graphic error up front saying "you can't use Numeric" rather than crashing later with an esoteric Python exception.

Again, we suggest using NumPy as it is fairly ubiquitous and actively in maintenance:

```
>>> import numpy
```

Contact your system administrator to install NumPy on your machine if the numpy import above doesn't work.

1.7 The C++ Experience

The work to build the PicklingTools from Python was very simple: After all, Python has just about everything built-in to the language, so making tools to handle files and sockets was straight-forward.

1. Python Dictionaries: Built-in the language
2. File and Socket Support: Built-in library (*import socket*)
3. Serialization Support: Built-in library (*import cPickle*)

The work to build the PicklingTools from C++ was harder: much less was built-in.

1. Python Dictionaries: Use *OpenContainers* library, included in distro
2. File and Socket Support: UNIX libraries, included on machine (hopefully)
3. Serialization Support: Written from scratch, included in distro

1.7.1 Files

The main tools for dealing with files (as well as Arrays) are the routines from *chooseser.h*. These routines allow you to read/write files (as well as Arrays) with serialized data. The main routines are:

```
// C++
DumpValToFile (const Val& thing_to_serialize,
               const string& output_filename,
               Serialization_e how_to_dump_the_data);

LoadValFromFile (const string& input_filename,
                 Val& result,
                 Serialization_e how_data_was_dumped);
```

With these routines, you can read and write data back and forth between Python systems (and other C++ systems of course). The *DumpValToFile* routine writes our data ("serializes" or "pickles") to a binary file. The *LoadValFromFile* routine reads our data ("deserializes" or "unpickles") from a binary file. Note that *LoadValFromFile* and *DumpValToFile* are inverses of each other so that a load gets back exactly what a dump did.

The choices for serialization are numerous:

1. **SERIALIZE_P0: Serialize as Python Pickling Protocol 0 would.** This is 7-bit clean and printable, so you can always look at this file with an editor and get an idea of what's in it. This protocol tends to be slower, but very backwards compatible.
2. **SERIALIZE_P2: Serialize as Python Pickling Protocol 2 would.** This is a binary protocol, so much more difficult to understand without a binary editor. This protocol tends to be very fast.
3. **SERIALIZE_M2K: Serialize as Midas 2k would.** This uses the binary serialization of Midas 2k and is most useful for talking to legacy systems.
4. **SERIALIZE_OC: Serialize using the OpenContainers serialization.** In general, this is the fastest binary protocol, but currently only other C++ systems using the OpenContainers (like PicklingTools, PTOOLS) understand this. OpenContainers comes built-in with PicklingTools. NEW! This now works with Python!
5. **SERIALIZE_TEXT: Simply stringize the given data.** This outputs 7-bit data you can then easily edit. Not very fast, but very human-readable.
6. **SERIALIZE_PRETTY: Like SERIALIZE_TEXT, but it uses the prettyPrint option.** The prettyPrint makes the tables much more human readable, at the cost of some extra white space. A slightly more efficient way to do this is to use the *WriteValToFile* and *ReadValFromFile* routines from the *ocvalreader.h* file (but these routines are limited to ONLY prettyPrinting: no other serialization).

With these routines, you can very easily exchange file data between C++ and Python: Below are a number of examples. The first example shows how to write some data from C++ so that Python can read it:

```
// C++ side: Write a Value
Val v = Tab("{ 'a':1, 'b':2.2, 'c':'three' }"); // .. something ..
DumpValToFile(v, "state.p0", SERIALIZE_P0);

# Python side: read the same value
>>> import cPickle
>>> result = cPickle.load( file('state.p0') ) # load figures out the protocol
>>> print result
```

Another example: have C++ read a file that Python created:

```
# Python side: write a file
>>> v = {'a':1, 'b':2.2, 'c':'three' } # ... something ...
>>> import cPickle
>>> cPickle.dump( v, file('state.p2'), 2 ) # Use Pickling Protocol 2

// C++ side: read the same file
Val result;
LoadValFromFile("state.p2", result, SERIALIZE_P2);
cout << result << endl;
/// .. and we have the same value from Python!
```

This is the best way to get started using the PicklingTools from C++: See if you can write a file from C++ and have Python read it.

There are some lower-level routines for serialization you may also find useful: You can also take your Val and load/dump it to an Array of char (for shoving over your own socket protocol, etc.):

```
// C++
DumpValToArray (const Val& thing_to_serialize,
                Array<char>& array_to_dump_to,
                Serialization_e how_to_dump_the_data);

LoadValFromArray (const Array<char>& array_to_load_from,
                  Val& result,
                  Serialization_e how_data_was_dumped);
```

In fact, the *DumpValToFile* is implemented using *DumpValToArray* and *LoadValFromFile* is implemented using *LoadValFromArray*.

With these routines, you should be able to get started. Of course, it might useful to know what a *Val* is: the next section talks about the *Val*: the main currency of the C++ PicklingTools.

1.7.2 OpenContainers

A minor goal of the PicklingTools was to try to make the C++ experience when dealing with Python Dictionaries very similar to the Python experience. Python Dictionaries are the currency of PicklingTools, so we needed to make sure dictionaries are easy to manipulate from C++.

Part of the problem is that Python is a dynamically-typed language (the type of an object is known only at runtime) and C++ is a statically-typed language (the type of an object is known at compile time). This makes supporting the dynamic, recursive, heterogeneous typing of Dictionaries difficult in C++. Consider the Python code:

```
>>> a = 10
>>> b = "hello"
>>> c = { 'key1': 17 }
>>> a = "a string, not an int!"   # okay in Python
```

In the above Python code, each variable's type is dynamic: *a* starts life as an int, then becomes a string. C++ has the opposite philosophy: All types need to be known at compile time:

```
int a = 10;
string b = "hello";
Tab c("{ 'key1: 17 }");
a = "a string, not an int!"  /// a is an int, not a string!! ERROR in C++
```

To work with dynamic typing in C++, we introduce a new type called the *Val*: (so called because it is always passed by value or deep-copy). The Val represents a dynamic container that can contain a variety of different types:

```
Val a = 10;
Val b = "hello";
Val c = Tab("{ 'key1': 17 }");
a = "a string, not an int!";    // Okay now in C++:  Val is a dynamic type
```

In fact, there is only a set number of types that a Val can contain.

- **INTEGER types: int_1, int_u1, int_2, int_u2, int_4, int_u4, int_8, int_u8** (note that int is always typedefed to one of these)
- REAL types: real_4, real_8
- COMPLEX types: complex_8, complex_16, and cx_t<T> for all INTEGER types
- MISC types: None (empty type), string, Proxy, int_n

- **CONTAINER types: Tab (like Python Dictionary), Arr (like Python List)** OTab (like OrderedDict), Tup (like Tuple)
- ARRAY type: Array<POD> where POD is any INTEGER, REAL, or COMPLEX type

(Sidebar: Note that the typedefs for integer/real/complex values use the *number of bytes* the specify the size of the numbers, as opposed to the number of bits in the standard C types (such as uint32_t which has 32 bits). This again belies the FORTRAN history of Midas/PTOOLS where integers and doubles and complex values are specified in terms of bytes: real*4, real*8, integer*4, complex*8, complex*16 and so on.)

Note that cx_t<INTEGER> is new as of PicklingTools 1.6.0. For certain DSP operations, it's useful to be able to represent what comes directly off an antenna as complex integers. There is not really a direct correspondance in Python or NumPy, but it's useful enough for C++ where hardcore DSP happens.

Vals cannot contain *any type* because we want compatibility with Python: with the limitations above, we can always serialize the data and give it to Python. In practice, this restriction hasn't been problematic: most data can be formulated in terms of Tabs, Arrs and elementary data types (In essence, this is the same argument of XML: all data can be formulated in the basic XML formats).

Since Val is a dynamic type, it has a *tag* and *subtype* to tell you what is inside it:

```
's' 'S' : int_1, int_u1
'i' 'I' : int_2, int_u2
'l' 'L' : int_4, int_u4
'x' 'X' : int_8, int_u8
'f' 'd' : real_4, real_8
'c' 'C' : cx_t<int_1>, cx_t<int_u1>:    # New as of PicklingTools 1.6.0
'e' 'E' : cx_t<int_2>, cx_t<int_u2>:    # New as of PicklingTools 1.6.0
'g' 'G' : cx_t<int_4>, cx_t<int_u4>:    # New as of PicklingTools 1.6.0
'h' 'H' : cx_t<int_8>, cx_t<int_u8>:    # New as of PicklingTools 1.6.0
'F' 'D' : complex_8, complex_16
'a'     : string (like 'a' in ASCII)
'n'     : array  (like n elements in array) [Like Python list]
't'     : Tab                               [Like Python dict]
'o'     : OTab                              [Like Python OrderedDict]
'u'     : Tup                               [Like Python tuple]
'q' 'Q' : int_n, int_un                     [like Python arbitrary-sized ints]
'Z'     : None
```

The tag is just a public data member on the class. This tag belies the Midas history of the product: they feel very similar to the tags on X-Midas and Midas 2k data:

```
Val v = 10.0;   // real_8
cout << v.tag;  // letter 'D'

a = int_1(10);
cout << v.tag;  // letter 's'

a = None;
cout << v.tag;  // letter 'Z'
```

There is another field called subtype which indicates what type an array is: this field is only valid if the Val is some kind of array:

```
Val a = Array<int_1>(10);
cout << v.tag << v.subtype;  // letters 'n' 's'

a = None;
cout << v.tag << v.subtype;  // letters 'Z',  subtype UNDEFINED if not array
```

```
a = Arr("[1,2,3]");
cout << v.tag << v.subtype;  // letter 'n', 'Z'
```

Note that an Arr is essentially an Array<Val>, but augmented with the ability to parse string literals.

1.7.3 Val and Conversions

One feature of C++ that makes Vals so easy to use are the (implicit and explicit) conversions. The Val has a constructor for every single type Vals can contain, so creating a Val from something else is simple and easy to read:

```
Val a = "hey";
Val b = 3.1415;  // constructors for ALL TYPES the Val supports.
Val c = 1;
Val d = Arr("[1,2,3]");
Val e = complex_8(1,2);
```

Notice that we overload the constructor on ALL integer types and ALL real types. Experience has shown that the compiler gets confused (read: compile-time errors) if you don't have an explicit constructor for every single type you expect. More important seems to be if you have a routine that takes a Val, the compiler won't get confused:

```
void PrintF (Val v); // ... prototype for some function ...

PrintF(1);       // no confusion
PrintF("hello"); // no confusion
```

(Incidentally, this is a type-safe way to support a better *printf*)

The outconversion process is equally important: Once you have placed a value inside a Val, how do you get it out? Very simple: ask for it!:

```
Val a = 123.456;

real_8 in = a;    // Ask for the value out!
```

The Val class also contains an outconverter (read: *operator T* for some type *T*) for every type the Val can contain. That's very specific C++ nomenclature, but the upshot is, you can ask for any type out and it will convert it for you, if the conversion makes sense. The general rule is that the conversion will happen just like C would do it:

```
Val v = 1;

real_8 f     = v;     // Sure, convert to 1.0
real_4 d     = v;     // Sure, convert to real_4(1.0)
complex_16 F = v;     // Sure, convert to 1.0+0.0i
string s     = v;     // Sure, turn it into the string "1"
Tab t        = v;     // DOESN'T MAKE SENSE!  runtime_error thrown

Val vv = 3.3;
int i = vv;           // Sure, truncates to 3 just like C would
```

Incidentally, we see here a very simple way to stringize a Val: just ask for its string out:

```
Val v = ReturnSomeVal();
string repr1 = v;              // Method 1:
string repr2 = Stringize(v);   // Method 2:
```

Method 1 and Method 2 of stringizing above do exactly the same thing, except if the Val v in question is a string: in that case Method 2 puts quotes around the string, Method 1 does not:

```
Val v = string("123");
string repr1 = v;               // repr1 is  123     (no quotes!)
string repr2 = Stringize(v);    // repr2 is '123'
```

1.7.4 C++ Arrs and Python Lists

The Val supports two main kinds of containers: The *Tab* (which is just like the Python Dictionaries, see below) and the *Arr*. Arrs are just like Python Lists: they are dynamically resizing arrays of Vals.

Python:

```
>>> a = [1, 2.2, 'three']         # Python List
>>> a.append("hello")
>>> print a.length()
```

C++:

```
Arr a = "[1, 2.2, 'three']";    // C++ Arr (like Python List)
a.append("hello");
cout << a.length() << endl;
```

The Array class comes from the OpenContainers collection: it is NOT the STL array class (there is further discussion of why we choose not to use in the FAQ).

The OpenContainers Array class is templatized on the type it supports. For using Arrays with Val, the type needs to be either POD (Plain Old Data which is ints, reals, or complexes) or Val. For example:

```
Array<real_8> demod_data(10); // Initial empty: Reserve space for 10 elements
demod_data.fill(0.0);         // Fill to capacity (10) with 0.0
for (int ii=0; ii<demod_data.length(); ii++) {
   demod_data[ii] = demod_data[ii] + ii;
}
```

One potential gotcha with Arrays is that they are ALWAYS constructed empty, with an initial capacity. If you wish to put elements in the array, you need to either fill the Array (as above), or append/prepend to the Array. For example:

```
Array<complex_8> ac(100);  // Initially empty: Reserve Space for 100 elements
for (int ii=0; ii<20; ii++) {
   ac.append(complex_8(1,0));
}
cout << ac.length() << endl;  // Only 20 items in array, space for 80 more
```

or:

```
Array<int_2> ai(100);
ai.fill(777);   // Fill array to capacity with 100 777s
```

If you exceed the capacity of the Array when you append/prepend, then the class automatically doubles the capacity and copies all the old data into the resized memory. This can bite you if you hold onto an element for too long:

```
Array<int_1> a(1);  // Capacity of 1, length of 0
a.fill(127);        // Capacity of 1, length of 1
int_1& hold_too_long = a[0];  // Currently valid reference to first data
a.append(100);      // Array resizes, hold_too_long is now INVALID
cout << hold_too_long << endl; ///   ??? Seg fault ???
```

The Array is implemented as a contiguous piece of memory so that array accesses are constant time. This is also important if you need to interface with legacy C routines:

```
Array<char> a(5);
a.fill('\0');
char* data = a.data();  // Returns &a[0]
strcpy(data, "hi");     // expect contiguous piece of memory
```

See C++ API document for documentation on the Array class. It is a basic OpenContainers inline class.

The *Arr* is essentially an *Array*<*Val*>, with one exception: it has a few extra methods to make them easier to use with Vals and Tabs. The most important is the constructor: If you give an Arr a string, it will attempt to parse it as Python would:

```
Arr a = "[1,2.2,'three']";   // Parses the string literal
```

This is the same as:

```
Arr a;
a.append(1);
a.append(2.2);
a.append("three");
```

The string literal can be as complex as you want, with recursive Arrs and Tabs inside it:

```
Arr a = "[1, 2.2, ['sub',2], {'a':1, 0: None}]";
```

Basically, you should be able to construct literals just as you would in Python.

1.7.5 C++ Tabs and Python Dictionaries

A *Tab* is the C++ equivalent of the Python Dictionary. You may notice that the Val/Tab/Arr all have three letters: This is on purpose. Since we are trying to emulate a dynamic language where you don't need to put an explicit type on, we are trying to save typing by having Val/Tab/Arr all be three letters. A *Tab* is a dynamic, recursive, heterogeneous container with key-value pairs.

Python:

```
>>> t = { }                            # Empty Table
>>> t = {'a': 1, 0: 'something' }      # table with 2 key-value pairs
>>> for (key, value) in t.iteritems(): # Iterate through table
...   print key, value
```

C++:

```
Tab t;                                 // Empty table
Tab t = "{ 'a': 1, 0:'something'}";    // Table with 2 key-value pairs
for (It ii(t); ii(); ) {               // Iterate through table
```

```
        cout << ii.key() << ii.value();
    }
```

A *Tab* is initially constructed empty, unless you provide a string literal (much like *Arr* above). Items are usually inserted into the *Tab* one of two ways:

```
Tab t;
t.insertKeyAndValue("key", 1.23);         //
t.insertKeyAndValue(0, Arr("[1,2,3]"));   // Direct insert
```

or:

```
t["key"] = 1.23;
t[0]     = Arr("[1,2,3]");                // []
```

Notice that the keys and values of the *Tab* are both of type *Val*: They can be any type Val supports.

Values are looked up a number of ways:

```
Tab t = "{'a': 1, 'b': 2 }";
cout << t["a"];                // using [], value 1
cout << t.contains("c");       // is the key "in" there?  No in this case
cout << t.lookup("a");
cout << t("a");
```

A *Tab* is implemented as an *AVLHashT*, which is an extensible HashTable that handles growth well, but still very fast lookups and removals. See the OpenContainers documentation for more discussion of the the different HashTables in OpenContainers.

Note that *[]* and *()* for lookup have slightly different semantics: if the key is in the table, they do the same thing:

```
Tab t = "{ 'a':1, 'b': 2 }";
cout << t["a"];   // Okay, in there, return Val 1
cout << t("a");   // Okay, in there, return Val 1
```

If, however, the key is not in there, the two do different things. The *[]* operator will insert the key into the table and give it a default value of *None*:

```
cout << t["NOT THERE"];  // Inserts key "NOT THERE" into table with None
```

Using operator *()*, if the key is not there, an exception will be thrown:

```
try {
   cout << t("NOT THERE");
} catch (exception& e) {
   cerr << e.what();    // Error message describing which key was NOT THERE
}
```

The *[]* notation is useful for assignment, because it allows us to change the table using the *[]*. The *()* notation is useful for lookup, because you don't want to change anything when you are just looking up something:

```
Tab t;
t["a"] = 3.3;     // Inserts "a":None into table, then overwrites with 3.3
```

```
Val& nv = t("a"); // Lookup, gives us a reference to the Val in the table
                     // (and throws an exception if not there)
```

See the FAQ for more discussion of *()* vs. *[]*. The basic rule: use *[]* for assignment, *()* for lookup.

1.7.6 Nested Lookup and Assignment

Tabs and Arrs can also handle cascading lookups and assignments. This makes it easy to get Vals in and out of nested structures: Again, this should feel very much like Python:

```
Tab t = "{'nest': { 'a':[0,1,2] }, 'b': 2 }";
Val& inside = t("nest")("a")(0);   // reference to the Val containing 0
inside = 777;
cout << t;      // {'nest': { 'a':[777,1,2] }, 'b': 2 }";

t["nest"]["a"][0] = 999;  // Cascading assignment
```

All lookups with Tabs and Arrs typically return *Val&* so they can be used for lookups and assignment like above.

1.7.7 Tricks and Tips for Efficient Tab and Arr Usage

When you start getting into more complicated *Arr* and *Tab* usage, there are tricks and tips that are helpful for more efficient usage.

First of all, as a debug tool or simply readability tool, there is a prettyPrint method on *Val*, *Tab*, and *Arr*. If there is complex, nested structure in a *Val/Tab/Arr*, *prettyPrint* is a nice human readable way to print the data:

```
Tab t = "{'nest': {'a',1 } }";

cout << t << endl;
// OUTPUT:  {'nest':{'a':1}}

t.prettyPrint(cout);
// OUTPUT:
{
    'nest': {
        'a': 1
    }
}
```

Notice that both ways are still backwards-compatible with Python Dictionaries (in fact, you could cut-and-paste the dictionaries directly into Python and they would still work), but the *prettyPrint* shows nested structure a lot better, even if it is a little wordy.

If you need OpalTable output, there is a *prettyPrintOpal* routine as well.

Copying

Whenever you copy a *Tab* or an *Arr*, they are copied by deep-copy. This means the entire recursive structure is copied:

```
Tab t = "{'nest': {'a':1 } }";
Tab deep_copy = t;   // Full Deep Copy

t["new"] = 1;  // No effect on deep_copy table
```

```
cout << t << endl;                // {'nest':{'a':1},'new':1}
cout << deep_copy << endl;        // {'nest':{'a':1}}
```

Once a copy is made under OpenContainers, it is a separate copy. (There is way to share *Tabs* and *Arrs* using Proxy: see the FAQ and more examples below). Note that this is a departure from how Python copies lists and dictionaries around:

```
# Copying in Python: by reference by default
>>> t = {'nest': {'a':1 } }
>>> copy = t;
>>> t["new"] = 1   // copy and t 'share' the dictionary
>>> print t        // {'nest':{'a':1},'new':1}
>>> print copy     // {'nest':{'a':1},'new':1}
```

The reason for this two fold: threads and understandability. From a threads perspective, it almost always makes sense to pass a separate copy of data so each thread can work independently on its own data. If you are going to share with threads, sharing should be EXPLICIT for the sake of human readability. Thus, if you choose to pass a pointer to data, that's fine, but experience has taught us (from the Midas 2k days, with systems built from 100s of threads) that threads almost always work better with EXPLICITLY shared data, otherwise every thread should have its own copy of data (to avoid problems like false sharing, race conditions, over-synchronization, and linearizing). Implicitly shared data always seems to cause collateral damage that's hard to trace.

We make this threads distinction for another reason: Python can be slow. We encourage people to write in Python when possible: Python code tends to be simpler, easier to read, and easier to maintain. *But,* if you need all the speed of a compiled language like C++, you probably also need all the tools you can use in C++ to make code faster: one such tool is truly concurrent threads (which Python does not have). So, if you are already in C++, there is a good chance you need to use real threads for speedup, so you should using data structures that are efficient and well-behaved with threads. [The OpenContainers collection was extracted from Midas 2k: one of the main design goals of the Midas 2k collections was the ability to work with 100s of threads in an application].

Why Deep Copy by Default?

If you are in Python, everything is fast enough. If you are in C++, there is a good chance you need the extra speed of compiled language. Let's allow the C++ programmer all the tools and abstractions needed to get extra speed, including threads.

References:

(Feel free to skip this section on first reading)

We mentioned earlier that the name of basic class is *Val* to remind ourselves that all things are copied by value (deep-copy, see the previous section). However, this sometimes means you make extra copies that you don't mean to:

```
SomeFunction(Tab t);          // prototype

Tab t("{'a':1 }");
SomeFunction(t);      // Will deep copy t, so SomeFunction has own copy
```

You may just need a read-reference to the given Tab, which means you could just as easily have SomeFunction take a *Tab&* (a Tab reference):

```
SomeFunction(const Tab& t);   // Passes EXPLICIT reference to SomeFunction

Tab t("{'a':1 }");
SomeFunction(t);
```

When you pass the reference in, we are essentially just passing the pointer to the Tab of interest: this a very fast copy, and we won't accidentally change the Tab because it is const Tab&. Note that when we do this, we are EXPLICITLY sharing the Tab.

A similar problem is when you copy a Tab in or out of a Val: you may be doing extra copies you don't mean to:

```
Tab t = "{'a':1}";
Val v = t;          // Two copies of the table: one in t, one in v

Tab t_out = v;     // Three copies of the table, t, v, and t_out
```

Let's tackle the second problem first (where *t_out* is an extra copy), as it gives us a hint how to deal with the first problem. When we ask for t_out, we can ask EXPLICITLY to share the implementation of the Tab inside of v. In other words:

```
Tab& t_ref = v;    // EXPLICITLY share the copy of the table
                   // contained inside of v

t_ref["insert"] = 1;  // Inserts into table inside of v
```

Strictly speaking, you can't ask for a Tab from a Val, you can only ask for a Tab&. So when you ask for a Tab, what really happens is:

```
Tab& t_ref = v;    // Can only get a Tab& from a v
Tab t_copy(t_ref); // Invoke copy constructor from reference

// EQUIVALENT TO:

Tab t_copy = v;
```

Let's go back to the original problem, avoiding the copy in:

```
Tab t = "{'a':1}";
Val v = t;          // Two copies of the table: one in t, one in v
```

If we just wanted one copy of a table copied into v, then we probably wanted something like this:

```
Val v = Tab();  // empty table
Tab& t = v;
t["a"] = 1;     // Put things into table inside of v
```

Note that when using Arrs, the same principles work: you ask for an Arr& when you want EXPLICITLY to share. Now of course, Val supports [], so sometimes it's easier to:

```
Val v = Tab();
t["a"] = 1;     // Put things inside table inside of v
```

1.7.8 C++ OTab and the Python OrderedDict

As of PicklingTools release 1.2.0, this is a new type of container: the OTab (Ordered Tab), which behaves just like Python OrderedDict. Unfortunately, the Python OrderedDict is only supported well in Python 2.7 and up, so your Python may not understand it yet:

```
>>> import collections  # Does My Python support the OrderedDict?
>>> a = collections.OrderedDict(([('a', 1), ('b', 2)])
                                  # Only in Python 2.7 and above
```

For all intensive purposes, OrderedDict is like the Python built-in dict, except that it preserves the order of insertion. This is most visible when iterating through the table:

```
>>> for (key, values) in a.iteritems() :
...    print key, value
a 1
b 2
```

The order that the key-value pairs are listed in the constructor is preserved. For more information on the Python OrderedDict, take a look at PEP 327: Adding an ordered dictionary to collections at *http://www.python.org/dev/peps/pep-0372/*.

The OTab, which in the C++ equivalent of the Python OrderedDictionary behaves just like its brethren. The OTab is very much like the Tab, except that it preserves the order of insertion:

```
// OTab is just like Tab
OTab o("OrderedDict(([('a', 1), ('b', 2)])");  // C++ respect Python syntax
o["goes at end"] = 500;
It ii(o);
while (ii()) {
   cout << ii.key() << " " << ii.value();  // preserves order of insertion
}
```

The OTab has exactly the same interface as the Tab. Adding the idea of order to the Tab is very simple, and doesn't cost much in terms of implementation (it's essentially just an extra doubly linked list): all the speed of the OTab is preserved for key-lookup, insertion, etc. In other words, the only real difference is that the order of insertion is preserved so what when you print the table, you can see the order:

```
OTab o;
o["a"] = 10;
o["b"] = 20;
o["c"] = 30;
cout << o << endl;  // o{ 'a':10, 'b':20, 'c':30 }
```

Contrast this to a plain dictionary:

```
Tab t;
t["a"] = 10;
t["b"] = 20;
t["c"] = 30;
cout << t << endl;  // { 'c':30, 'a':10, 'b':20 }  // Looks "random" order
```

Intuitively, the OTab is easier for beginners to understand because the input matches the output better (the beginner may ask "Why is my dict in some weird order?"). The real utility is for bridging data structures in other languages.

1. XML is an ordered data structure: the next version of PicklingTools will have tools to read/write XML using OTab
2. The C struct is inherently ordered: if you needed to go back and forth between some C struct and PicklingTools, an OTab would be essential (in fact: this was the driving need: we needed to be able to read/write BlueFiles which is C-struct based).
3. Windows .ini files are ordered

The next version of the PicklingTools will have XML parsing tools and will rely heavily upon the OTab.

The C++ OTab currently supports a shorter syntax for OrderedDict:

```
OTab o("o{'a': 1}"); // short syntax, an o just before first {
OTab longer( "OrderedDict([('a',1)])");  // Python syntax
```

We are hoping to perhaps introduce the shorter syntax to Python: the longer Python syntax seems harder to read. The nice thing about the shorter syntax is that it looks JUST LIKE a normal dictionary, with braces and such but only ONE EXTRA CHARACTER: the little o to indicate its ordered. Which is easier to read?:

```
Val v1 = Eval("o{ 'a':1, 'b':2, 'c':3 }");  // short syntax: ordered dict

Val v2 = Eval("OrderedDict([('a', 1), ('b',2), ('c',3)])"); // longer
```

1.7.9 C++ int_n and the Python arbitrary size ints (long)

Python has an arbitrary sized integer so that when you exceed the precision of an int_u4 or int_u8, it creates an arbitrary sized int (essentially implemented as an array of bytes).:

```
>>> google = 10**100  # Obviously bigger than an int_u8 can hold
>>> print google
10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

The PicklingTools, as of release 1.2.0 has the *int_n* and the *int_un* types. The *int_n* corresponds to the Python arbitrary type. The *int_un* is an unsigned version. In general, they have just like other ints and can combine with other ints:

```
// Use of int_n just like normal ints
int_n google = 1;
for (int ii=0; ii<100; ii++) {
   google *= 10;
}

int_n top = 1+ 2;
int_n bottom = 1000;
int_n dd = top / bottom;
```

The int_n is implemented as an array of unsigned ints. They work with Vals and other ints and reals just like all the other ints:

```
Val g = int_n(100);
int_n out = g;

Val pi = 3.14159265;
int_n three = pi;  // truncates just like any int would
```

Some integers constants are too large, but that's easy to get around with some strings:

```
// literal too big, compiler complains and worse, truncates the number!
int_n t = 123456789123456789123456789;

int_n a0 = StringToBigInt("123456789123456789123456789"); // Ah! Use string!
int_n a1 = "123456789123456789123456789"; // In PicklingTools 1.4.1 & above
```

In general, the int_n/int_un seem to have roughly the same performance as the Python arbitrary ints (gleaned from an informal test computing combinatorics).

1.7.10 C++ Tup and Python Tuples

As of version 1.2.0, the PicklingTools supports the Tup, which behaves very much like the Python tuple:

```
>>> a = (1,2.2, 'three')
>>> print a[1]   # 2.2
>>> print len(a) # 3
>>> a.append('NOT ALLOWED')  # ERROR!

// C++
Tup a(1, 2.2, "three");
cout << a[1];        // 2.2
cout << a.length();  // 3
a.append("NOT ALLOWED");   // Syntax Error
```

The Tup, like the Python tuple, is just for building a "const" array that you can't change. Once gotcha: the Tup does NOT EVAL the string arguments like the Tab, OTab and Arr:

```
// OTab, Tab, Arr all evaluate the first argument when constructing
OTab o("o{ 'a':1 }");  // Eval
Tab  t("{'a': 2'} ");  // Eval
Arr  a("[1,2,3]");     // Eval

Tup  u("[1,2,3]");          // DOES NOT EVAL
                            // So, this is a tuple of 1 argument: a string
                            // ("[1,2,3]")

Tup  uu(Eval("[1,2,3"]));  // Force Eval
                            // So, this is a tuple of 1 argument: a list
                            // ([1,2,3])
```

Tups are most useful when just constructing long argument trains of very different types in C++:

```
Tup lotsa(1, 2.2, "three", Tab(), Arr("[1,2,3]"), OTab(), None);
```

The Tup is implemented as an array of Vals, where the number of Vals is fixed at construction time.

1.7.11 C++ and cx_t<INT>

New in PicklingTools 1.6.0, a Val can carry a complex type for integers as well as floats and doubles. This is in recognition that certain DSP operations make good use of complex integers. For example: D2A sampling from an antenna: an array of data can be held as `cx_t<int_2>` rather than `complex8` for half the cost of memory. For very large arrays of data, that can be a signficant savings.

The cx_t<> support fewer operations, (mag2 still works but may roll over if the values get large) as many operations (such as magnitude) require floating values. Note that these values can easily convert between different precisions.

1.7.12 Sockets

The MidasTalker in C++ in very similar to the Python MidasTalker, and this is on purpose. Similarly for the MidasServer, MidasYeller and MidasListener.

Creating and using the basic socket classes from C++ should feel almost exactly the same as the Python experience:

```
#include "midastalker.h"
MidasTalker mt("host", port, SERIALIZE_M2K);
mt.open();
mt.send(Tab(" {'a':1} ");
Val v = mt.recv();
```

The major difference (besides the syntax) is that the C++ components support all the major serialization protocols: M2k, OpenContainers, Python P0, P2, P-2. The Python components generally only support the Python 0 (and maybe P2) protocols that are built-in.

There are plenty of examples in the C++ area of the PicklingTools distribution. The best place to start is copy one of the examples (following the example in the *Makefile.Linux*) and looking at some of the sample source code.

1.7.13 JSON

JSON stands for JavaScript Object Notation and comes from the dictionaries (or objects) in the Javascript programming Language. Some people prefer JSON over XML as a language independent exchange format because JSON tends to be smaller, easier to read, and have less overhead.

JSON is a text format for storing recursive, heterogeneous dictionaries and lists. JSON is very much like text-Python dictionaries: the differences are minor. In JSON, 'true', 'false', 'null' replace the 'True', 'False', 'None' of Python dictionaries. JSON strings can only use double quotes (whereas Python Dictionaries can use either single quotes or double quotes). Other than that, they are very similar.

The PicklingTools C++ library has the capability to read JSON and write JSON. (By default, Python has a JSON library builtin: the *json* library: 'import json') The JSON text is converted to Tab/Arr/Vals, so that we can use the dictionaries in the standard Tab manipulations. For example, to read in file which contains a JSON dictionary:

```
#include "jsonreader.h"

Val result;
ReadValFromJSONFile("json.txt",
                    result);
// 'result' is now a plain Tab we can manipulate
```

Writing JSON is even easier: anything that is a Tab/Arr can be written into a stream:

```
#include "jsonprint.h"

Val t=Tab("{'a':1, 'b':2.2, 'c':'three'}");
JSONPrint(std::cout, t);   // Python dictionary written as JSON dict
```

These tools can also be useful when building HTTPClients and Servers.

(Caveat: PicklingTools JSON doesn't support anything other than ASCII chars)

1.7.14 HTTPClient and HTTPServer

There are currently some tools in the PicklingTools to manipulate HTTP. The interfaces for the C++ are in flux to a certain extent, so they aren't quite ready for full exposure yet, but they are included in release 1.3.2 so we can get some feedback. The code does function.

In general, the HTTPClient looks/feels very much like the HTTPClient of the Python library: Take a look at the "httpclient_ex.cc" for a sample of how to write a simple client.

The HTTPServer framework is much more complicated: the server framework has been overhauled and refactored so that we can support one connection per thread easily. A sample server is written: see "httpserver_ex.cc".

Currently, there is support for HTTP 1.0 and some support for HTTP 1.1 (including chunked encoding). Some more work needs to be done here.

1.7.15 Conformance Checking

Is there a mechanism like XML schema for dictionaries? Yes, the *Conforms* routine. This area describes the C++ version, but there is a Python version which behaves almost identically.

One feature that has been asked for by a number of users is the ability to check a dictionary for conformance. In other words:

1. Does a dict have the right number of keys?
2. Does a dict have the right structure?
3. Does a dict have the keys with the right names?
4. Are the types of the values the right types?

For example, if a user wants to send a message, where the host and port are required keys in a message, the user wants to be able to "validate" a dictionary has the proper keys and/or types:

```
// C++
Val message = Tab("{'port':8888, 'host':'ai1', 'data':'ping'}");
```

A simple way to validate this message would be to manually check for all the keys:

```
if (message.contains("port") &&
    message["port"].tag == 'i' && // check for int port
    message.contains("host") &&
    message["host"].tag == 'a') { // check for string host
    //// Valid message
} else {
    throw runtime_error("Invalid Message to send");
}
```

While this will work, it scales poorly for larger and larger structures. A simple idea is to have a "prototype" message that we try to match against. If the message we want to send matches the prototype structure (has the same keys and types), then it is a valid message. As of PicklingTools 1.3.3, there is a new function called *Conforms* which checks an instance against a prototype message. If the message to send matches, then this is a valid message:

```
#include "occonforms.h"

Val message  = Tab("{'port':8888, 'host':'ai1', 'data':'ping'}");
Val prototype= Tab("{'port':0,    'host':'',    'data':''    }");

if (Conforms(message, prototype)) {
   //// Valid message
} else {
   throw runtime_error("Invalid Message to send");
}
```

Matching (by default) means that all keys of the prototype are present in the message and all the types (int, string, etc) of the message match the types of the prototype. Note that in the prototype, the actual value doesn't matter as much as the type of the field.

If the message forgets the port, host, or data fields, then the *Conforms* check would fail.:

```
Val message  = Tab("{'port':8888}");
Val prototype= Tab("{'port':0,    'host':'',    'data':''    }");

// Conforms will return false!!
if (Conforms(message, prototype)) {

}
// NO! this message does not conform
```

There are many kinds of options in performing matches. For instance, sometimes the type of the fields doesn't matter at all: all that matters is the presence of the keys and that the key names match. For instance, the data to send to a client may a table, a string, an int, etc. You can use *None* in the prototype to specify a key can take any value:

```
Val message  = Tab("{'port':8888, 'host':'ai1', 'data':'ping'}");
Val prototype= Tab("{'port':0,    'host':'',    'data': None  }");

if (Conforms(message, prototype)) {
   //// Valid message because None in prototype matches ANY TYPE!
} else {
   throw runtime_error("Invalid Message to send");
}
```

Since the data field in the prototype is *None*, any type is valid! Thus the conform check above will succeed. If we change the data to a table, or anything else, the conform check will still succeed:

```
Val message  = Tab("{'port':8888, 'host':'ai1', 'data':{'stuff':'123'}");
```

As will:

```
Val message  = Tab("{'port':8888, 'host':'ai1', 'data':4 }");
```

Of course, we can send None data as well:

```
Val message  = Tab("{'port':8888, 'host':'ai1', 'data':None }");
```

There are many types of conformance matching that make sense, depending on what a user may be doing. For instance, the user may only looking to make sure that *some* keys are there: other keys can be crucial, but others may be optional. For example, a key specifying the sender can be useful for our message (for debugging), but not required:

```
// The sender key is optional, but both of these tables are still valid
Val message1 = Tab("{'port':8888, 'host':'ai1', 'sender':'ai2' }");
Val message2 = Tab("{'port':8888, 'host':'ai1'  }");
```

The third argument to *Conforms* (called *exact_structure* in the code) controls how pedantic the check is when looking at the structure:

- If exact_structure is true, then Conforms is checking that all keys in the prototype MUST be present AND that only those keys are present. In other words, the prototype and message must have the same number of keys and all keys must match.
- If exact_structure is false, then Conforms is simply checking that all the keys of the prototype are present in the message: if there are more keys in the message, that is not a problem.

In summary:

```
Conforms(message, prototype, false) -> all keys of prototype must be in
                                       message and match

Conforms(message, prototype, true)  -> all keys of prototype must be in
                                       message and match AND
                                       number of keys must match prototype
```

This means that:

```
Val message1 = Tab("{'port':8888, 'host':'ai1', 'data':4, 'sender':'ai2' }");
Val prototype= Tab("{'port':0,    'host':'',    'data': None  }");


Conforms(message1, prototype, true) -> false
// FAILS because message1 has too many keys,
// so doesn't have EXACTLY same structure


Conforms(messag21, prototype, false) would SUCCEED (return true)
// SUCCEEDS because message1 has all necessary keys,
// but message1 can have a few more (because doesn't have to
// have EXACTLY the same structure)
```

In the *exact_structure* parameter specifies how structure and keys match, the next parameter *type_match* specifies how *values* match. In particular, the *type_match* parameter control how matching works when comparing the types of two entries in a table or array. By default, types must match exactly:

```
Val mesg     = Tab("{'port':8888, 'host':'ai1' }");
Val proto    = Tab("{'port':0,    'host':''    }");

if (Conforms(mesg, proto, true, EXACT_MATCH)) { ... }
// same as Conforms(mesg, proto, true)
```

The above *Conforms* return true because (a) the structure matches but more importantly (b) the type of *port* is an *int* and the type of *host* is a *string* and that matches what's in the prototype. The fourth argument to *Conforms* is a enumeration and defaults to *EXACT_MATCH*. All values of the enumeration are:

```
EXACT_MATCH:  The types of compared values must match exactly

LOOSE_MATCH:  All ints match each other
              All reals match each other
              All complexes match each other
              Tab/OTab match other Tab/OTab
              Arr/Tup match other Tab/OTab
              Array<POD1> will match Array<POD2>
                  if POD1 loose matches POD2

LOOSE_STRING_MATCH:
              Like LOOSE_MATCH, but strings in the
              given message will match anything.
```

An example showing different types of integers can match or not:

```
Val mesg1 = int_8(1);
Val proto1 = int_4(1);
```

```
Conforms(mesg1, proto1, true, EXACT_MATCH) -> return false
// FAILS because different types int_8 and int_4


Conforms(mesg1, proto1, true, LOOSE_MATCH) -> return true
// SUCCEEDS because similar types (2 types of int)
```

Another example showing how reals and ints don't match:

```
Val mesg2 = real_8(1.0);
Val proto2 = int_4(1);


Conforms(mesg2, proto1, true, EXACT_MATCH) -> return false
// FAILS because different types real_8 and int_4


Conforms(mesg2, proto1, true, LOOSE_MATCH) -> return false
// FAILS because reals and ints NOT really same types
```

When *LOOSE_STRING_MATCH* is turned on, strings in the message will match anything:

```
Val mesg3 = "1234";
Val proto2 = int_4(0);


Conforms(mesg3, proto3, true, EXACT_MATCH) -> return false
// FAILS because different types string and int_4


Conforms(mesg3, proto3, true, LOOSE_MATCH) -> return false
// FAILS because reals and ints NOT really same types


Conforms(mesg3, proto3, true, LOOSE_STRING_MATCH) -> return true
// SUCCEEDS because strings match anything under LOOSE_STRING_MATCH
```

The purpose of *EXACT_MATCH* is to strictly enforce matches. The purpose of *LOOSE_MATCH* is to allow some leeway so that related types will match. The purpose of *LOOSE_STRING_MATCH* is to recognize that many times strings are filled into fields of a message, but are really supposed to be something else (like an int, a time, a list of people); this makes it easier to move from XML to dictionaries.

If it's not clear from the above examples, all matching is performed *recursively* throughout the table: if the prototype has a nested dictionary inside of an array inside of a dictionary, the message must have the same structure recursively:

```
Val instance =Tab("{'a':1, 'b':[1,2.2, {'nested': {} }] }");
Val prototype=Tab("{'a':0, 'b':[0,0.0, {'nested': {} }] }");

Conforms(instance, prototype) -> true

// Recursively checks that 'b' of instance matches 'b' of prototype,
// and that 'nested' of instance matches 'nested' of prototype
```

One final note: sometimes when validation fails, it's unclear why the validation failed. By default, a validation check simply returns *false* with no other information why the check failed:

```
 Val instance = 1.0;
 Val prototype= "";

 // FAILS by returning false to show that 1.0 and "" mismatch types
 if (Conforms(instance, prototype, 1, EXACT_MATCH)) {
   ///  -> return false
 }
// Why exactly did this fail?
```

There is one final parameter on the *Conforms* function call called *throw_exception_with_message* which can be set to true to give more information. By default, this parameter is set to false, which means failed *Conforms* calls return false to show failure. If, on the other hand, it is set the true, a *runtime_error* is thrown instead of a false return, and embedded in the exception text is information about why the fail checked:

```
// FAILS by returning throwing runtime_error
bool result = false;
try {
   result = (Conforms(instance, prototype, 1, EXACT_MATCH, true));
} catch (const runtime_error& re) {
   cerr << re.what() << endl;
}

// Prints out
*********FAILURE TO MATCH instance against prototype:
 instance=1.0 with type:d
 prototype='' with type:a
 exact_structure=1
 type_match=EXACT_MATCH
Requested Exact Match of two primitive types that didn't match
```

Inside the error message is information on what the types were and why they didn't match: in this case, type 'd' doesn't match type 'a' in an *EXACT_MATCH*. In a very large table, this can be very useful, as it will tell you exactly what keys don't match. This avoids hunting around the table to find the problem.

This debugging mechanism *should not be turned on by default*. It's very expensive to build error strings, and it's also expensive to throw exceptions to get information from a conformance check. We envision this mechanism as being most useful when debugging: For production code, the last parameter should probably always be false so that any conformance checks are fast. Probably only when debugging should you set the last parameter to true.

The Python version of the conforms module is very similar.

Test to see if an instance of a Python object conforms to the specification prototype. This is similar to checking if an XML document conforms to an XML DTD or schema. For example:

```
>>> from conforms import conforms
>>> from conforms import EXACT_MATCH
>>> from conforms import LOOSE_MATCH
>>> from conforms import LOOSE_STRING_MATCH

>>> # EXAMPLE 1
>>> ##### At a simple level, we want to exactly match a prototype:
>>> instance  = {'a':1, 'b':2.2, 'c':'three'}   # value to check
>>> prototype = {'a':0, 'b':0.0, 'c':""}    # prototype to check against
>>> if conforms(instance, prototype, True, EXACT_MATCH) :
...       # should be true: all keys match, and value TYPES match

(1) Note that the instance has all the same keys as the prototype, so it
```

```
    matches
(2) Note that on the prototype table, that the VALUES aren't important,
    it's only matching the the TYPE of the val

>>> # EXAMPLE 2
>>> #### We may not necessarily need all keys in the prototype
>>> instance1 = {'a':1, 'b':2.2, 'c':'three', 'd':777 }
>>> prototype1= {'a':0, 'b':0.0 }
>>> if conforms(instance1, prototype1, False, EXACT_MATCH) :
...     # should be true: instance has all keys of prototype

(1) Note that the instance has more keys than the prototype, but that's
    okay because we specified exact_structure to be false.
(2) by setting EXACT_MATCH, all the types of the values that are
    compared MUST match (not the value just the types of the values)

>>> # EXAMPLE 3
>>> #### If you just want the structure, but don't care about the
>>> #### types of the keys, use None in the prototype.
>>> instance2 = {'a':1, 'b':2.2, 'c':'three'}
>>> prototype2= {'a':None, 'b':None, 'c':None }
>>> if conforms(instance2, prototype2, True, EXACT_MATCH) :
...     # should be true, only comparing keys


>>> # EXAMPLE 4
>>> #### If you want to match value types, but want to be a little
>>> #### looser: sometimes your int is a long, sometimes an int_u4, etc.
>>> instance3 = {'a':1, 'b':2L, 'c':'three'}
>>> prototype3 ={'a':0, 'b':0, 'c':'three'}
>>> if conforms(instance3, prototype3, true, LOOSE_MATCH) :
...     # should be true because long(2) is a LOOSE_MATCH of int(2)
```

The Python version of the *conforms* module is a standalone module that can easily be dropped into other Python baselines. It is more general than the C++ version (because Python has a plethora of types) and so it returns slightly different results on things that are iterable: If a type is iterable, things are more easily comparable, and thus more likely to conform in a LOOSE_MATCH. In general, though, the Python and C++ versions of conforms will give the same results.

In summary, the purpose of the *Conforms* routines to all some simple kind of schema validation like XML has. Enough customers have asked for a feature like this that we felt it was time to embrace some XML schema-like mechanism for Python dictionaries. In the end, *Conforms* is simply a tool you can use to help when you make start constructing bigger dictionaries to make sure structure is preserved.

1.8 Java

The Java experience is discussed in a self-contained document in the "Docs" area of the PicklingTools distribution. It's also on the http://picklingtools.com website.

In general, Java support is still relatively new as of PicklingTools 1.5.1, but it has been used in some big systems already.

1.9 The X-Midas PTOOLS Experience

The PicklingTools is an entire distribution with C++, Python, M2k, and X-Midas support. PTOOLS is an option tree for using with X-Midas that comes with the PicklingTools distribution.

Everything we have learned in the previous sections applies to the PTOOLS option tree.

All of the code in the python subdirectory of PTOOLS is just a copy of the Python code from the distribution. The nice thing is that you don't have to set the PYTHONPATH to pick up that code from XMPY: X-Midas handles that for you.

Most of the rest of PTOOLS is just a copy of the code from the C++ area. The C++ primitives in the *host* area of PTOOLS are essentially the same examples from the C++ area coded to fit inside of an X-Midas C++ primitive.

The real reason to use PTOOLS is because you want to use the PicklingTools with X-Midas. PTOOLS has been packaged to work with X-Midas as a standard option tree:

```
X-Midas> xmopt ptools /path/to/ptools/only/lowercase/characters
X-Midas> xmp +ptools
X-Midas> xmbopt ptools
```

There are plenty of examples in the host area demonstrating how to write an X-Midas C++ primitive with the PTOOLS libraries. The only major gotcha is that you have to be sure your own X-Midas C++ primitives use the same compiler and linking flags that the example host primitives do. Take a look at the *library.cfg* and *primitives.cfg files in the 'cfg/* area for examples before you build your own primitives.

1.10 The Midas 2k Experience

Recall that the original goal of the PicklingTools was to work with legacy M2k applications. If you use the SERIALIZE_M2K option on your MidasTalker/MidasServer/MidasYeller/MidasListener, you should be able to talk to the legacy M2k components (OpalDaemon, OpalSocketMsg, UDPSocketMsg) without problems.

The M2k area of the PicklingTools contains replacement components for some of the standard M2k components. In particular, the OpalPythonDaemon is a replacement for the OpalDaemon and the OpalPythonSocketMsg is a replacement for OpalSocketMsg. The original M2k OpalDaemon/OpalSocketMsg components works fine, but are limited:

Original M2k components only supports Midas 2k serialization : if you want to talk to them from Python, OpenContainers, out of luck

Original M2k components don't support adaptive serialization: the newer components can adopt to the message being sent to them so they can understand multiple clients with multiple protocols AT THE SAME TIME. Original components ONLY understand one protocol.

Later versions of legacy applications can use the OpalPythonDaemon and OpalPythonSocketMsg to open their options: adaptive serialization and multiple serialization choices.

Essentially, the OpalPythonSocketMsg and OpalPythonDaemon close the loop: with them, Midas 2k can talk every serialization protocol—this means ANY system (XMPY, X-Midas, raw C++, raw Python, Midas 2k) can talk to your legacy M2k app.

To use the new Components, copy everything you need out of the *unit.cfg* in the M2k area, and copy all the listed files into your own project.

1.11 The Final Experience

All the components from the PicklingTools system should be compatible. For a final test, the examples in each of the different areas should work together: For example, the *xmserver.cc* X-Midas primitive from the PTOOLS option tree should work with the Python *midastalker_ex2.py* and the raw C++ *midastalker_ex2.cc*.

In the end, this is just a set of open source tools to help you get your job done. You are welcome to change them and modify them as needed.

1.12 Appendix A: C++ and User-Defined Types

A frequent criticism of the *Val/Tab/Arr* is "Why can't Vals contain user-defined types?" The simple answer is that there are a lot of issues to be concerned with:

1. Is a new type a POD type? (And thus bit-blittable?)
2. How does a new type compare with other types of Vals?
3. How does a new type serialize?
4. How does a new type Stringize itself?
5. How does a new type convert to other types?
6. What happens if we have virtual functions in a type?
7. What letter do I use for a tag? Has it already been used?

Certainly there are ways to handle these issues, but the philosophy of OpenContainers (for the moment) is that most data, if not all, can be represented with the core types: all ints, all floats, complexes, Arrs, Tabs, Array<POD>, strings, and None. This is the essential argument of JSON and XML anyway, so we aren't asserting anything controversial.

One thing that has proved useful is the ability is to have classes that can convert to and from Vals easily so that you can take advantage of all the infrastructure. For example:

```
// A class for computing simulations of a biological nature
class BiologicalSimulation {
  public:

   Start ();  // Start the simulation
   Stop  ();  // Stop
   Resume (); // Very expensive, so have to stop and resume every so often

  private:
    // All sorts of data
    real_8 start_time, real_8 stop_time;
    float* paramecium_state;
    int    number_of_paramecium;
};
```

It would be nice if we could convert BiologicalSimulation to and from Vals so we could easily serialize it and send over sockets or save it to files. It's actually pretty straight forward to do. Here's a sketch of what we want:

```
// Take current BiologicalSimulation and save to file
BiologicalSimulation b;
Val repr = b;
WriteValToFile(repr, "simulation_backup");
```

```
...

// Read Biological Simulation from file and reconstruct
Val old_sim;
ReadValFromFile("simulation_backup", old_sim);
BiologicalSimulation update = old_sim;
```

The idea is simple: The class of interest needs to know how to convert between itself and Vals. Notice that to do this easily, the class has two new methods:

1. A New Constructor to create BiologicalSimulations from Vals: *BiologicalSimulation (const Val& v)*
2. An outconverter to create Vals from BiologicalSimulations: *operator Val () const*

Here's an example how we might do this: The *BiologicalSimulations* becomes a *Tab* with some keys representing state:

```
// A class for computing simulations of a biological nature
class BiologicalSimulation {
  public:

   // Construct myself from a Val
   BiologicalSimulation (const Val& v) {
     start_time = v("start_time");
     stop_time  = v("stop_time");
     Array<real_4>& p = v("paramecium_data");
     number_of_paramecium = p.length();
     paramecium_state = new float[number_of_paramcium];
     memcpy(paramecium_state, p.data(), sizeof(float)*p.length());
   }

   Start ();  // Start the simulation
   Stop  ();  // Stop
   Resume (); // Very expensive, so have to stop and resume every so often

   // Create a Val which represents my current state
   operator Val () const
   {
      Val ret_val = Tab();  // enable return value optimization

      Tab& table = ret_val;
      table["start_time"] = start_time;
      table["stop_time"]  = stop_time;
      table["paramecium_data"] = Array<real_4>(number_of_paramecium);
      Array<real_4>& a = table("paramecium_data");
      a.expandTo(number_of_paramecium);
      memcpy(a.data(), paramecium_state,sizeof(real_4)*number_of_paramecium);

      return ret_val;
   }

  private:
    // All sorts of data
    real_8 start_time, real_8 stop_time;
    float* paramecium_state;
    int    number_of_paramecium;
};
```

By creating a new constructor that takes a Val, you allow a Biological Simulation to be created from Vals. This assumes you you have been able to "map" your data structure onto Val/Tab/Arr and create a "representation" of the BiologicalSimulation as Tab:

```
// Read Biological Simulation from file and reconstruct
Val old_sim;
ReadValFromFile("simulation_backup", old_sim);
BiologicalSimulation update = old_sim;
```

By creating an outconverter for Val, you allow the BiologicalSimulation to create a representation of itself as Vals:

```
// Take current BiologicalSimulation and save to file
BiologicalSimulation b;
b.Start();  b.Stop();
Val repr = b;           // use outconverter
WriteValToFile(repr, "simulation_backup");
```

Certainly, it's nice to be able to convert between Vals and your user classes, but of course, you can't always change your classes (perhaps someone else owns them and doesn't want them modified in any way). You can always write global routines (in another file so they don't mess up the original class) that do the same thing:

```
Val CreateValFromBiologicalSimulation (const BiologicalSimulation& b);
BiologicalSimulation CreateBiologicalSimulationFromVal (const Val& v);
```

This isn't quite as "compact", but still works and you can still take advantage of all the PicklingTools infrastructure.

There are several nice things about being able to convert between Vals and User types: besides the obvious (Vals can be pickled, saved to files, sent over sockets), this is also useful for debugging: PrettyPrinting a Val is a human-readable way to see the state of a class without attaching a complex debugger.

The two most common ways to write user-defined types like this it are:

1. Represent the state as a Tab (like the *BiologicalSimulation* above). Simply keep all the fields of your class as entries to a table.
2. Represent the state as a string. This is more for situations when you have binary data and need to "bit-blit" the data back. In fact, the *SockAddr* representation uses this because the socket data needs to be plain binary data. See the *SockAddr* for an example.

CHAPTER

TWO

FREQUENTLY ASKED QUESTIONS FOR PICKLINGTOOLS 1.7.0

2.1 General Questions

0. **What are the PicklingTools?**

   The PicklingTools are an open-source collection of tools for communicating between C++ and Python components (and Java as of PicklingTools 1.5.1): they allow a developer to write pieces of an application in C++ and other pieces in Python, yet still have those pieces talk *easily*. **All code is plain C++ or plain Python so any framework can incorporate the PicklingTools code easily**

   The PicklingTools supports multiple legacy frameworks (M2k, X-Midas, XMPY), *but it is not bound to those frameworks*: it's all raw Python and raw C++ (and raw Java) that is all compatible. You can write a server in M2k and a client in XMPY and another client in raw C++ and another client in an X-Midas primitive and they all work together.

   A minor goal of the PicklingTools is to make the C++ experience as friendly as the Python experience. (And Java as well).

   In the end, this is just a library. Use it or not.

1. **Why are they called the PicklingTools?**

   The main currency of PicklingTools is essentially Python dictionaries. For example:

   ```
   { 'key1': 323.1, 'key2': 'available' }
   ```

   is a Python Dictionary.

   When you serialize Python dictionaries, Python says that they are *pickled*: that's where the term PicklingTools comes from. All clients and servers talk and send Python Dictionaries around in requests.

2. **Why do you use Python Dictionaries as the "common currency" of PicklingTools?**

   Short Answer: Python dictionaries are a very good "recursive, heterogeneous" data structure.

   For encoding data in requests and responses from clients, Python dictionaries are a very simple, easy-to-read, easy-to-write data structure based on a widely accepted standard. A Python Dictionary can contain any number of 'key-value' pairs where the values themselves can be other Python Dictionaries. For example:

   ```
   { 'REQUEST': { 'PingRequest':1000.1 }, 'Host': "dl380" } }
   ```

is a simple Python Dictionary for encoding what might be a ping request every 1000.1 seconds to host machine named "dl380". Note that the structure can be dynamic, which is one of the advantages of Python Dictionaries.

The original currency of Midas 2k (the precursor of the PicklingTools) was the OpalTable, but it was a bit of a stovepipe construction at the time.

There is a standard called "JSON" (JavaScript Object Notation) which specifies a standard for tables which is "kind of" backwards compatible with Python Dictionaries: See http://json.org for more discussion and examples.

Those of you familiar with XML should think of Python Dictionaries as a smaller, cleaner, and more user-friendly version of XML.

2. **Where did the PicklingTools come from?**

   Short Answer: They were created to allow easy access/use of a a popular software product.

   A popular software product used the Midas2k OpalDaemon as a "server" and historically you had to use a Midas2k OpalSocketMsg as a "client" to talk to it. The product was popular enough that non-Midas2k users wanted to be able to talk to the product without having to use Midas 2k.

   The first versions of the PicklingTools included only code to write clients. Currently, a user can write a server or client easily in one of many different systems/languages: raw C++, raw Python, XMPY, X-Midas C++ Primitives, Midas 2k. Raw Java is newly supported.

3. **Why do I want to use the PicklingTools?**

   Answer 1: If you wish to use the "popular software product" mentioned above, PicklingTools makes it VERY easy.

   Answer 2: If you have a legacy product written in Midas 2k that you wish to transition to X-Midas, PicklingTools makes it very easy. There is an X-Midas option tree called PTOOLS (which is included in the PTOOLS distribution) that contains several tools to make it easy to transition legacy M2k products to X-Midas:

   (a) OpalTable tools: read and write OpalTables from/to disk

   (b) M2k Binary Serialization tools: understand M2k Serialization

   (c) Interface with OpalDaemon and OpalSocketMsg

   Answer 3: If you wish to talk to X-Midas, but prefer to write code in raw C++ or raw Python, PicklingTools enables that paradigm.

   Answer 4: You want tools with cross language support for C++ and Python and Java.

   In the end, PicklingTools is just a LIBRARY you can use or not. It is not tightly bound to any particular product.

4. **What languages does PicklingTools support?**

   Short Answer: Python and C++ and Java.

   The Java interface is a bit more immature than the C++ and Python, but it is supported. There are currently some features missing from the Java codebase (see the Java docs).

5. **What is a MidasTalker/MidasServer/MidasYeller/MidasListener?**

   Short Answer: These are the PicklingTools clients and servers.

   A MidasTalker is a client that talks to MidasServers over TCP/IP. Multiple clients can talk concurrently to the same server.

   A MidasListener is a client that listens to a MidasYeller over UDP. Multiple clients can talk concurrently to the same server.

   The "Midas" forename is to represent that these are compatible with the Midas 2k system: For example, the PicklingTools MidasServer gives the same functionality as the Midas 2k OpalDaemon.

At this point, though, you do not need to have Midas 2k anymore for these tools to be useful.

6. **Can I use PicklingTools to enable easy back and forth between C++ and Python and Java?**

   Short Answer: Absolutely, that's one of the design goals

   If you do a lot of cross language development in C++ and Python, you know that C++ is good at certain things and Python is good at certain things. The PicklingTools makes it easy to talk back and forth between C++ and Python. In particular:

   (a) Interactions via files: It is very easy to read and write Python dictionaries from Python (as it is built into Python), and the PicklingTools allows the user to do that just as easily from C++.

      i. If readability is a concern, files can contain textual (ASCII) Python Dictionaries

      ii. If speed and size are concerns, files can contain binary serialized (pickled) Python Dictionaries

   (b) Interactions via sockets: As stated many times, it is easy to talk back and forth via sockets using the PicklingTools Midastalker, MidasSocket, MidasYeller and MidasYeller. For example, A Python MidasTalker can talk interoperably with a C++ MidasServer or a Python MidasServer.

   An alternative is to consider embedding Python within a C++ program or writing a C/C++ module for Python. For most interactions, consider using the PicklingTools as they should be simpler than either of the above options.

7. **What is the easiest way to get started with the PicklingTools?**

   The cPickle module in Python allows you to manipulate files and dump/load dictionaries very easily. The chooserser.h file from C++ allows to manipulate files and dump/load dictionaries very easily.

   Example 1 (Python -> C++): you want to dump a dictionary from Python into a file and have C++ read that file. Use the cPickle module in Python (note we are using Pickling Protocol 0):

```
>>> my_dict = {'a':1, 'b':2}      # My dictionary to dump
>>> import cPickle
>>> cPickle.dump( my_dict, file('myfile.p0'), 0)
```

   Then, from C++, use the chooseser.h include:

```
// C++: Want to load a file from python
#include "chooseser.h"

int main()
{
  Val result;  // put it here!
  LoadValFromFile("myfile.p0", result, SERIALIZE_P0);

  cout << result << endl;  // print it out! {'a':1, 'b':2}
}
```

   Example 2 (C++ -> Python): you want to dump a dictionary from C++ into a file and have Python read that file (note we are using Pickling Protocol 2):

```
// C++: Want to pickle a dictionary into a file from C++
#include "chooseser.h"

int main()
{
  Val my_dict = Tab("{ 'key1': 1, 'key2': 2.2 }");
  DumpValToFile(my_dict, "my_file.p2", SERIALIZE_P2);
}
```

Now, load the file from Python:

```
>>> import cPickle
>>> result = cPickle.load( file('my_file2.p2') )
           # Python load figures out whether P0 or P2
>>> print result
           # {'key1': 1, 'key2': 2.2}
```

If you want to do stuff over sockets, read on about the MidasTalker, the MidasServer, the MidasListener and the MidasYeller.

See the Java documentation for how to work with Java.

2.2 Python

0. **What versions of Python does PicklingTools support?**

   Historically, versions 2.1.x to 2.6.x have been tested extensively. More recently, 2.7 has been tested and should work, but it has not been tested as much as the other versions.

   3.x has not been tested: We are waiting for our main paying customer to adopt the 3.x series.

1. **What do I need to use the PicklingTools in Python?**

   Short Answer: A handful of Python files and an `import`

   All the Python files live in the "Python" area of the PicklingTools distribution. If you are using the X-Midas option tree PTOOLS, the Python area under that option tree contains the exact same files.

   There are no C or C++ modules you "have to build". All the PicklingTools modules are written in straight Python, so all you have to do is `import` the proper module and move forward. For example, to create a client to talk to a server on machine dl380, port 8888:

```
>>> # Simple example in Python
>>> import midastalker   # Make sure midastalker.py on PYTHONPATH
>>> a = midastalker.MidasTalker("dl380", 8888)
>>> a.open()
>>> data = a.recv()  # Read a Python Dictionary from server

UPDATE: There are now two Python C Extension modules.  The first
allows very fast XML/dict conversions.  The second allows using OC
serialization (for example, if you want pickles over 4G, OC may be
your only option).  These modules have to be built specially.
See the later section on the Python C Extension Modules.
```

2. **Where can I find examples?**

   Short Answer: In the same area where all the Python files are.

   There several examples in the Python area demonstrating several uses of all the different clients and servers.

   midastalker_ex.py: Simple usage of a Python MidasTalker (TCP Client)

   midastalker_ex2.py: More complex example, showing how to reconnect

   midasserver_ex.py: Simple MidasServer, works with midasTalker examples

   permutation_client.py: More realistic MidasTalker example

   permutation_server.py: More realistic MidasServer example, with threads

midasyeller_ex.py: Simple MidasYeller (UDP Server) example

***midaslistener_ex.py*: Simple MidasListener (UDP Client) example, works** with the midasyeller_ex.py

There are also some examples in PythonCExt area.

3. **What are Numeric and NumPy and why do I need them from Python?**

   Short Answer: No, you don't need them, but they can be useful if you do lots of scientific computation.

   Numeric and *NumPy* are Python modules written in C that allows you to do very fast numeric vector operations. Because they are written in C, it does those operations very fast. For some DSP, having these operations available from Python is very useful.

   The reason you care about Numeric/NumPy: If you want to send large amounts of data (resampled data, demod data, etc.) over sockets, it is generally MUCH faster if you hold your data in Numeric/NumPy arrays. For example:

   ```
   { 'DATA' : array(1,2,3,....) }   # 1e6 data in Numeric array
   ```

   vs.:

   ```
   { 'DATA' : [1,2,3,....]  }# 1e6 data in standard Python List
   ```

   Everything will still work with holding data in standard Python Lists, but you may see some dramatic speed increases (and dramatic memory reductions) if you use Numeric arrays for holding/shipping around numeric data.

   The problem is that Numeric does NOT come with most machine-installed versions of Python. You have to install it on your machine either by compiling it yourself with your version of Python, or you have to find the RPM for it.

   NumPy seems to be installed on most machines as a standard package these days. If it's not, the RPM/aptget/whatever is usually very easy to install on your machine.

   XMPY (The X-Midas Python) comes with Numeric installed already. XMPY as of 4.0.0 will *only* come with NumPy. If you need Numeric, you will have to use earlier versions of XMPY.

4. **Can I have both Numeric and NumPy in Python?**

   Short Answer: No (Well ... Mostly No)

   We can only support *ONE* of *Numeric* or *NumPy* inside of Python (C++ and M2k don't have this restriction). *Numeric* was the original de-facto package for handling arrays, but maintenance for it has faded. *NumPy* is now the de-facto standard. One of the *NumPy* design goals was to be backwards compatible with *Numeric*, so they both have about the same API, which is why it's very difficult to support them both inside of Python as the same time. For more information on NumPy, take a look at the web page.

   (We have seen systems with both installed, but it's messy. It can work, but it's better not to open this can of worms unless you have to.)

5. **Do you recommend Numeric or NumPy?**

   For backwards compatibility, Numeric may be your only choice. However, we strongly urge users to move forward to NumPy, as Numeric seems to be out of maintenance, and bugs are no longer getting fixed. NumPy is very active.

2.3 C++

0. **What versions of C++ does PicklingTools works with?**

   Earlier versions of PicklingTools worked with ARM C++ and ANSI C++. Current versions take advantage of complicated templates, so you need ANSI C++.

   Almost all testing has been done on Linux (Red Hat, Fedora) under g++. A fair amount of testing has been done under Tru64 C++ compiler CC. Some testing has been done with the Intel compiler (icc).

1. **If Python Dictionaries are the "currency" of PicklingTools, how does C++ deal with them? After all, C++ does not have Python Dictionaries built into the language.**

   Short Answer: A library provides "Python-like" Dictionaries

   The OpenContainers library provides the "Val/Tab/Arr" abstraction. The Tab gives us a "Python-like" Python Dictionary:

```
Python:  t = {'a': 1}
C++:     Tab t("{ 'a': 1 }");
```

   The Arr gives us a "Python-like" Python List:

```
Python:  a = [1,2,3]
C++:     Arr a("[1,2,3]");
```

   The Val gives us a "Python-like" dynamic type (a variable that can hold all the basic types, as well as Arr and Tab):

```
Python:  a = 1
         a = "string"
         a = { }

C++:     Val a = 1;
         a = "string";
         a = Tab("{ }");
```

2. **How do Tab, Val and Arr in OpenContainers work in C++?**

   Short Answer: As close to Python as C++ will allow

   Python obviously comes with Python Dictionaries built into the language and integrated well. In fact, because Python is a dynamically typed language, tables are very easy to build and manipulate in Python, as a variable can contain a table, an integer, a real number or any type:

```
>>> # Python
>>>  n = 10
>>>  a = { 'sample': None }
>>>  a['sample'] = n
>>>  t = { }
>>>  t['new entry'] = a
```

   C++ is a statically typed language, so every variable has to have a type at compile time. To represent that a variable can contain a table or array or number or integer, we use the type *Val*:

```
// C++
Val n = 10;
Val a = Tab("{ 'sample' : None }");
```

```
a["sample"] = n;
Val t = Tab();
t["new entry"] = a;
```

One major difference between Python and the C++ OpenContainers Val/Tab/Arr abstraction is that Python does reference counting and OpenContainers does deep copies by default, but can do reference counting (see below).

3. **How does OpenContainers Val/Tab/Arr differ from Python?**

   Short Answer: They tend to look similar, but by default they copy data differently

   Python uses reference counting for copying, and OpenContainers uses both deep copies and reference counting. By default, if you are NOT using Proxies (see below), everything is copied by deep copies, meaning EVERYTHING is recursively copied. If you are using a Proxy, then copying happens just like Python. Here's some examples of what happens when we copy by reference counting vs. deep copy.

   ## In Python:

```
>>> a = [1,2]
>>> b = a       # Uses reference counting, a and b SHARE the list
>>> b[0] = 17
>>> print a,b  # BOTH a and b have changed!!
*****[17, 2] [17,2]
```

   // In C++:

```
Arr a("[1,2]");
Arr b = a;    // Makes a deep copy
b[0] = 17;
cout << a << " " << b << endl  // ONLY b has changed!!
*****[1,2] [17.2]
```

   If you wish to 'completely' emulate Python semantics, you can use Proxies in C++ (see below), but there are issues to be aware of (see the later FAQ below).

   // In C++:

```
Val a = new Arr("[1,2]");   // This creates a PROXY
Val b = a;    // Use reference counting, a and b SHARE the list
b[0] = 17;
cout << a << " " << b << endl  # BOTH a and b have changed!!
*****[17,2] [17.2]
```

   Thus, Python users are used to "sharing" data when they copy (unless they explicitly deep copy). The OpenContainers does a full deep copy (unless they explicitly uses Proxies) so that a and b have their own explicit, unshared copy of the data.

4. **Why does the Val/Tab/Arr abstractions do deep copies by default but Python uses reference counting by default?**

   Short Answer: For more intuitive copy semantics and to allow better threads performance in C++

   Most people, when they copy a table (until they are used to Python's mutable/immutable semantics) are surprised when the table they "copied" changes underneath them: this is because the table was shared. PicklingTools does the more "functional" copy (full deep-copy) by default because it tends to be more intuitive for new users. Experience has shown that sharing is usually better understood when EXPLICIT (which is why we have Proxies, see below).

The other issues is that Python does not have TRUE concurrent threads (i.e., multiple threads in the same interpreter running concurrently within the same process on different CPUS). Python uses something called the "Global Interpreter Lock" (GIL) and a thread must hold the lock in order to make progress inside the interpreter. Since only only one thread can hold the lock at a time, all threads are inherently serialized. Although Python supports the notion of threads, they cannot truly run in parallel due to the GIL.

The OpenContainers collection, however, was written with truly concurrent threads in mind. Experience with Midas 2k (the birthplace of the OpenContainers collection) demonstrated many issues of threads with container classes: See the wikipedia: OpenContainers for more discussion.

So, once a thread has its own deep copy of a data structure, it shouldn't have to worry about synchronization with other threads (not all libraries, including the STL (until 2009) give this guarantee). OpenContainers was written so that a thread with its own Dictionary does NOT have to worry about synchronization with other threads: One lesson from Midas 2k is that synchronization is VERY expensive, and that any any excessive synchronization is very limiting.

Deep copies of tables allow threads to independently process tables without excessive synchronization or false sharing or extra serialization.

5. **So how does Val work?**

   Short Answer: Val is a simple container for any kind of type

   A Val is a dynamic container that can hold any of the following types:

   - Ints: int_1, int_u1, int_2, int_u2, int_4, int_u4, int_8, int_u8
   - Reals and Complexes: real_4, real_8, complex_8, complex_16
   - Specials: None and bools (True and False)
   - **Complex Containers: Tab (like Python Dictionary)** Arr (like Python Lists)

   For example:

```
Val a = 4;        // Put an int into variable a
cout << a.tag;    // See what type is in there 's' means int_4
a = 3.3;          // ... now a holds a real_8
cout << a.tag;    // 'd' means real_8
a = Tab();
a = real_4(3.3); // Force it to be a real_4 instead of real_8
a = None;         // Empty value
```

   Getting the values out is just as easy: the natural conversion will occur for you:

```
int_4 actual = a; // a is currently real_4(3.3), will be truncated and
                  // turned into int_4 as expected
cout << actual;   // .. value in actual is 3

Tab t = a;        // Throws an exception!  No natural conversion
                  // from 3.3 to a Tab
```

6. **So how does Arr work?**

   Short Answer: Like a Python List (or an "array")

   The only "gotcha" is that Arrs have to be filled either with (a) string literals or (b) using append or (c) using fill

   Creation Examples:

```
Arr a;  // Empty list
cout << a[0];  // Throws an exception!  Nothing in the list!

Arr a("[1,2,3]");  // Initialize with string literal, using []
cout << a[2];  // Okay, value 3
cout << a[3];  // Exception!  Beyond the ends
```

Append Examples:

```
Arr a(10);   // Creates EMPTY Array with space for 10 elements
cout << a[0]; // Throws an exception!  Nothing in list!

a[0] = 10;    // WON'T WORK!!! Throws an exception because list empty

// In order to initialize, you have to APPEND elements into the list
for (int ii=0; ii<10; ii++) {
  a.append(ii);
}

a.append(100);  // Capacity was only 10, so this causes a resize
                // of the array capacity to 20, and now a[10]==100.
```

Fill Examples:

```
Arr a(10);    // Empty, with capacity of 10
a.fill(None); // ... fills with 10 Nones, a now has 10 elements
```

7. **So how does Tab work?**

   Short Answer: Like Python Dictionaries

   A Tab is basically a container for a bunch of Key-Value pairs. Both the Key and the Value are of type Val.

   Creation Examples:

   ```
   Tab a;   // Empty
   Tab b("{ 'a': 1 }");  // String literal looks just like Python Dictionary
   ```

   Insertion Examples:

   ```
   a["hello"] = "there";  // Insert key "hello" with value "there"
   a[100] = 3.3;          // Insert int key 100 with real_8 value 3.3
   cout << a(100);        // Show value(3.3) associated with key 100
   ```

   Lookups:

   ```
   if (a.contains("Hello")) {  // Case matters
     cout << "there" << endl;
   }
   ```

8. **Why do I use [] sometimes and () sometimes with a Tab?**

   Short Answer: When inserting into the table, use [] on the left hand side. When looking up a key, use ().

   Canonical Usage:

   ```
   a[101] = "hi";    // [] on left hand side of =
   cout << a(100);   // () on right hand side
   ```

Why? Both [] and () do the right thing if the key is already in the table, but they behave differently if the key is NOT in the table. With [], if the key is NOT in the table, it inserts it into the table with a default Value of *None*. For example:

```
// DO NOT DO THIS!!
Val xx = a["Nope"]; // Nope is NOT a key in table, so it forces
                    // changes to the table.  This behaves as if:
                    // a["Nope"] = None;
                    // Val xx = a("Nope");
```

With (), if they key is NOT in the table, an exception is thrown:

```
// DO NOT DO THIS!!
a("NotThere") = 10; // Will throw exception because NotThere NOT in
                    // table and will NOT insert into table
```

Experience has shown that you WANT an exception thrown if you lookup a key that is not there, so use () on the right hand side, [] on left.

Here's how an example of how you want to use the () and []:

```
int start=0, end=0;
try {
  start = a("start");
  end   = a("end");
} catch (const exception& e) {
  // Has failed key in exception message so can see which lookup failed
  cerr << "Oops!  Forgot to populate table with:" << e.what();
}
a["interval"] = end-start;  // Insert diff into table
```

9. **Why do I sometimes see a Tab& returned and sometimes a Tab? (Similarly, sometimes I see a Arr& returned and sometimes a Arr)**

   Short Answer: **Tab& is for changing the Tab in place.** Tab is for getting a whole new copy out.

   For example:

```
Val v = Tab();
Tab& t = v;     // Get a reference to the Tab
t[0] = 17;
cout << v << t; // Changing t changes the Tab inside v

Val z = Tab();
Tab copy = z;   // Makes a full COPY of the table in z
copy[0] = 17;
cout << z << copy;  // z and copy are DIFFERENT COPIES
```

10. **How can I get a stringized version of the my variable?**

    You can either call "Stringize" directly, or ask for a string version of your Val.

    For example:

```
Val a = 1.3;
string a_string = string(a);  // Stringizes 1.3 for you

string s = Stringize(1.3);
```

11. **Can I cascade lookups and inserts into Tabs and Arrs?**

    Short Answer: Yes

    For example:

```
Tab t("{ 'a': 1, 'nested': { 'start': 1.1, 'end': 2.2 } }");
t["nested"]["start"] = 1.11;  // cascading inserts
cout << t("nested")("end");   // cascading lookups

Arr a("[0,[1,2,3],555]");
a[1][0] = "hello";
cout << a;   //  [0, ["hello",2,3], 555]
```

12. **I like that I can do cascading lookups/insertions with Arrs and Tabs. Why can't I do that with strings and Array<T>?**

    Short Answer: Limitations of C++

    It would be nice if you could do this:

```
Val a = "abc";
a[0] = "A";   // WILL THROW AN EXCEPTION, a is a STRING
```

    or similarly:

```
Val b = Array<real_8>(10);
cout << b[0];  // WILL THROW AN EXCEPTION, b is an Array<real_8>
```

    The problem is that the [] operation always returns a `Val&`: It's the static typing of C++.

    Consider:

```
Val a = "abc"; a[0] = "A";   // THIS DOESN'T WORK!!!!
```

    In order for this to work, [] would have to return a `char&`. Or for the `Array<real_8>` to work, the [] operation would have to return a `real_8&`. (Some kind of proxy may fix this, but makes the code even more complex and slow).

    The workaround: Get the value out, and mess with it. See the examples:

```
// With Array<T>, you can get a reference out and mess with it in place
Val vv = Array<real_8)(10);
Array<real_8>& a = vv;
a.expandTo(100);   // changing a changes the value inside

// With strings, all you can do is copy it back in,
// you CAN NOT get a string&
Val vvv = "abc";
string s = vvv;
s[0] = "A";
vvv = s;
```

13. **What is the difference between OpenContainers and PicklingTools?**

    Short Answer: PicklingTools is a library that ships with the OpenContainers library: PicklingTools USES OpenContainers.

OpenContainers is "essentially" the C++ containers collection from Midas2k that used the RogueWave interfaces. The main philosophy is that these classes are open-source, inlineable container classes with an eye towards speed for the core classes and an eye towards usability for the Val classes.

PicklingTools is a library for writing socket servers and clients C++ and Python. The central philosophy is that it should be easy to write interoperable socket servers or clients in raw C++, X-Midas, raw Python, XMPY, or Midas 2k. (By interoperable, we mean that clients written in one system (such as raw C++) should be able to talk directly to servers written in a different system (such as Python).

The C++ PicklingTools uses the OpenContainers libraries to allow a more "Python like" experience when using dictionaries.

14. **Why do the Tab/Val/Arr classes not use the C++ Standard Template Library (STL)?**

    Short Answer: Usability issues, threads issues, historic issues, preference issues

    Historically, the OpenContainers were a re-implementation of the RogueWave containers classes. They used the RogueWave interfaces.

    From a threads perspective, the STL (until TR2, supposedly available in February 2009) was silent on the issues of threads. You had to "hope" that the STL was built with thread issues in mind. Being a thread-friendly class has much more to do than just simple locking and unlocking issues. See the wikipedia: OpenContainers for more discussion. The OpenContainers cares strongly about threads and couldn't depend on the STL to get it right.

    From a usability issue, the STL tends to be very long-winded and clumsy. The Val/Tab/Arr classes were written to be simple, friendly, easy-to-use and look like Python. Providing a "Python-like" experience with C++ tends to be at odds with using the STL.

15. **Now that C++ TR2 is out, are you going to rewrite pieces of the PicklingTools to be more STL like?**

    Short Answer: Grumble, maybe.

    See above for discussion.

2.4 C++ and Proxys: New in PicklingTools 1.0.0

1. **What is a Proxy?**

    As of PicklingTools 1.00, Val can support Proxies. A Proxy is a way of adopting a Tab, OTab, Arr or Array<T> so that it can be shared.

    In its simplest form:

```
Proxy p = new Tab("{'a': 1}");  // Share this table

Val v1 = p;   // shared copy
Val v2 = p;   // shared copy
v1["a"] = 17;   // v1 and v2 BOTH see the change!
```

    The Proxy exists as a value for a type of "link" or "pointer": a way to share data without having to do a full copy. There are three current ways to really use this:

    (a) In a system with no threads (EASY)

    (b) In a system with threads where you have to worry about coordinating data sharing between threads (MEDIUM)

    (c) In a system with multiple processes sharing a piece of memory, where you have to worry about coordinating data sharing between processes (HARD)

2. **Why do I want a Proxy?**

   Recall that everything in OpenContainers is passed by DEEP COPY. When you copy a Proxy, however, you only copy a handle (not the entire piece of data). In order to avoid excessive copying, you may want larger tables and/or arrays inside of Proxies. Or you may simply wish to share some data structure among multiple tables.

   For example:

```
Tab t;

Array<real_8> a(1000);
fillWithZeroes(a);

t["full copy"] = a;  // FULL, DEEP COPY of a: 1000 elements copied

Proxy p = new Array<real_8>(1000);
Array<real_8>& ar = p;
fillWithZeroes(ar);

t["proxy"] = p;     // Just a copy of the handle, much cheaper! 4 bytes
```

3. **Does a Proxy handle the memory management for you?**

   Short Answer: Yes, if you want it to.

   By default, the Proxy "adopts" pointer to data given to it. (You can change this, but it's rare to want to do this).

   The Proxy uses a reference-counting scheme, so everytime you copy or destruct a Proxy, it updates the reference count. When the reference count goes to 0, the item is destructed and the memory handed back to the allocator.

   For example:

```
{
  Proxy p = new Tab();  // ref count at 1
  {
     Proxy p2 = p;      // ref count up to 2
  }          // back to 1 when p2 goes away
}  // ref count at 0 when p goes away, so Tab destructed, memory returned
```

4. **What is the easiest way to use a Proxy?**

   Short Answer: If I don't worry about threads or processes and I just want to share some data and avoid excessive copying

   For example:

```
// Example: CREATION
Proxy p = new Tab("{'a':1}");
   // or
Proxy p(new Tab("{'a':1}"), true, false);

// Example: USAGE
Tab& t = p;      // Get table pointed to by proxy
t["b"] = "add";
```

   This proxy adopts the memory so when the last proxy goes away, the table will be destructed and the memory will go away:

```
// Common idiom
Proxy p = new Tab("{'a':1}");
Val v = p;

// ... so below is supported to "automatically" create a Proxy in v
Val v = new Tab("{'a':1}");

// Once the proxy for a Tab is in a Val, you can use [] notation
v["changing the table in proxy"] = "yup";
```

5. **If I am worried about Threads, how does that change how I use a Proxy?**

   Short Answer: Creation is slightly different, and you have to use a *TransactionLock* to enforce mutual exclusion when you you write or read from a table that is shared among threads.

   When using threads, you are probably aware that you have to be careful when sharing data among threads so it doesn't get corrupted. In general for Vals, you don't worry about this too much because Vals are copied by DEEP COPY for this very reason: once a thread has its own DEEP COPY, then it can usually use the table without worrying.

   If you use Proxies, however, you have to make sure you co-ordinate sharing. The Proxy gives some "controlled sharing" capabilities.

   Creation is slightly different. Note that you will create a Proxy that contains a Lock so you can enforce single thread access:

   ```
   Proxy p = Locked(new Tab("{'a':1}");
   ```

   By saying "Locked", you are are creating a Proxy that is managed atomically and you expect multiple threads to copy proxies around. Note that you only have to use a *Locked* Proxy if you know multiple threads will be copying this table:

   ```
   // Usage:
   Tab& t = p;   // Rare: I know NO OTHER THREAD is looking at
      // or
   {
     TransactionLock tl(p);   // yes, this is the same Proxy p

     Tab& t = p;
     t["b"] = "add";
   }
   ```

   Inside the { }, only one thread at a time may hold the TransationLock: This guarantees that only one thread may read or write the table. In other words, you can "lock" your table for atomic transactions.

   Note that memory is still managed for you by the Proxy and the data destroyed when the last reference goes away.

   TransactionLocks now support timeouts, so a runway thread can't inhinit progress. If an exception is thrown, then that means the TransactionLock timeout expired, and the user may want to see if he needs to clean up.

6. **Can I use Proxies with custom allocators? I need to use a special region of memory.**

   Short Answer: Yes.

   You may wish to put all your tables in one piece of shared memory:

   (a) to facilitate sharing

   (b) to cross process boundaries (HARD! see below)

By default, OpenContainers 1.6.0 and beyond contain some default allocators you can use. Currently, the only allocators supported are the custom StreamingAllocator and the default "new/delete". But, the entire Val/Str/Tab/Arr suite contained support for the allocators:

```
// Example: CREATION
Proxy  p = Shared(SHM, Tab("{'a': 1}"));

// Example: USE
Tab& t = p;  // I __KNOW__ no one else has the lock .. rare ...
         // or
{ // Lock is held so no one else in any process (or this one) may use
  // or modify this
  TransactionLock tl(p);

  Tab& t = p;
  t["b"]  = "add";    // all components of t (key, value) IN SHARED MEM!
}
```

The "SHM" is an allocator that allocates memory from some shared memory pool. The pool may be a simple allocated with new/delete or may be a pool mapped and shared among multiple processes. When the last proxy goes away, the Tab is deallocated from OUT of the pool.

7. **Can I use Proxies with Shared Memory across multiple processes?**

   Short Answer: You can, but there are limitations and it's difficult.

   The PicklingTools contains wrapper code to use the UNIX shared memory facilities of UNIX. They are in "sharedmem.h,.cc"

   The easiest way to use Tabs/Vals, etc. with memory shared among processes is to follow the example in "sharedmem_test.cc". In that example, you create a shared memory region in the parent, fork a child, then that child (through inheritance without extra work) maps the same region. This works and works well.

   If you wish to use processes which don't share a common lineage, it's a lot harder and may not work. It basically requires you to use SHMCreate to create shared memory region in one process, record where the region gets mapped into memory, then force a SHMAttach to attach to EXACTLY that region. This is difficult and may or may nor work (depending on how well your implementation of mmap supports the MAP_FIXED option). One thing we have found is that certain versions of RedHat have the "Address Randomization" optimization (to keep hackers from exploiting address space regularity). This optimization prevents the SHMCreate/SHMAttach method from working and you will have to turn it off to get this to work.

   Later versions of PicklingTools will support this better, but right now only the parent/child idiom is recommended.

8. **When I use shared memory (between processes) with Tabs, etc., why does it matter that we map the shared memory to the same address space?**

   Short Answer: Tabs contain pointers, and those pointers MUST have the same value in both address spaces.

   When we we put a Tab in shared memory, its data contains pointers to nodes, pointers to memory, etc. To make sure that all pointers point to the same thing, (1) all the data must be in shared memory and (2) All the data must be at the same address.

   If (1) is not met, then the data is not in shared memory and it cannot be seen by the other process.

   If (2) is not met, then if we try to "look" at the data via the pointer, we'll get different answers! (because they point to different locations)

   In other words, if you are using shared memory between processes, you need to be very careful. This is why there is a special interface for initially creating a table in shared memory:

```
Proxy p = Shared(shm, Tab("{ }"));
```

This ensures the table and all its data are stored in the shared memory. Once the table is in shared memory, PicklingTools makes sure that inserts into Tabs and Arrs and Array<T> stay in Shared memory:

```
Val v=p;
v["allin"] = "shared memory";  // All inserted data in shared memory
```

Let's repeat that: The PicklingTools, if you use standard operations like [] *guarantee* that the keys and values of the Tab will be in shared memory.

9. **What is the RedHat "Address Randomization" and why does it affect SHMAttach?**

   Short Answer: It randomizes where memory maps in address spaces, and causes SHMAttach to fail if you have to "force" a memory address.

   Details: Basically, to stop hackers from exploiting address space regularities, the RedHat Address Randomization (also referred to as ExecShield) makes mmap picks "random" addresses throughout memory (as well as other process data structures). The end result is that if memory chosen 'randomly' conflicts with the address space you need, mmap will segfault. The problem, of course, is that it is random and sometimes will work, and sometimes won't!

   By turning "Address Randomization" off, you can have a MUCH better chance of not having address space conflicts between processes (because the address spaces will be grown the same way). Thus, if two processes do not share the same lineage, they can still mmap the same addresses confidently and share memory, tables, etc.

   To turn off the Address Randomization feature on a per process basis, use `setarch` on the executable you are going to run. For example, when you run the *sharedmem_test* from this baseline:

   ```
   % setarch i386 sharedmem_test 0
   ```

   Turning off the "Address Randomization" feature for the whole machine is more difficult and requires root privies and a reboot:

   - Add the following to /etc/sysctl.conf file:

     ```
     kernel.exec-shield = 0
     ```

   (It can be made effective for the current session by using):

   ```
   # sysctl -w kernel.exec-shield=0
   ```

10. **Where can I find more information shared memory in PicklingTools?**

    As of PicklingTools 1.4.1, there is better documentation (as well as better abstractions) discussing how to use Vals and shared memory. Check out the "Shared Memory" document in the Documentation area on the http://www.picklingtools.com web site or the PicklingTools141Release/Docs/shm.txt file.

    As of PicklingTools 1.6.0, the Shared Memory abstractions have been augmented to be more robust and handle errors better: you can specify timeouts, and break handlers when things fail. See code for more details.

2.5 C++ and OTab/Tup/int_un/int_n: New in PicklingTools 1.2.0

0. **What is an OTab?**

Short Answer: An OTab is like the Python OrderedDictionary: It's just like a dictionary, but it preserves the insertion order.

Starting with Python 2.7, the Python collections module supports the OrderedDict data structure. In terms of implementation, speed, and interface, it really is just a Dictionary. What distinguishes it is that the iteration order preserves the order of insertion:

```
>>> from collections import OrderedDict
>>> a = OrderedDict([('a',1, 'b',2)])
                    # like {'a':1, 'b':2}, but order preserved
>>> for key in a :
...    print key     # iterates in order of insertion 'a', then 'b'
```

Note that the notion of order has nothing to with the sort order: it's the order things are *inserted*. From C++:

```
OTab o = "OrderedDict([('a',1)])";  // like Python syntax
n["key12"] = 17;
n["again"] = 18;
for (It ii(n); ii(); ) {
   const Val& key = ii.key();
   Val& value = ii.value();
   cout << key << endl;        // Insertion order: a key12 again
}
```

In general, the collections.OrderedDict behaves like the dict except for preserving insertion order. Similarly, the OTab behaves like the Tab except for preserving insertion order.

1. **The syntax for OTab and collections.OrderedDict is clumsy. Is there a better way to enter an OrderedDict literal?**

   Short Answer: Yes and No.

   Since we always want the PicklingTools C++ side to feel as much like Python as possible, the PicklingTools must support the clumsy literal syntax:

   ```
   // C++ OTab
   OTab ot("OrderedDict([('a',1), ('b',2)])");
   ```

   By default, this is how they print as well: again, this is to stay compatible with Python as much as possible:

   ```
   // C++
   cout << ot << endl;    // output: OrderedDict(['a',1, 'b',2)])
   ```

   However, from the C++ side, we currently support a simpler syntax: simply add the single letter *o* to a dictionary literal, and that makes it an OrderedDict:

   ```
   OTab o(" o{ 'a':1, 'b':2 }" );  // like dict literal, but the
                                   // o distinguishes it
   ```

   We are hopeful something like this makes it into Python. Right now, you can recompile and set the OTabRepr to 2 in ocval.cc to use the short behavior.

   Although we like the short form better, we have to stay true to out Python roots until Python decides to create a better literal.

2. **Why would I want an OTab or an OrderedDict?**

   Short Answer: XML, C/C++ structs

   If you deal with XML or C/C++ structs, then the order of the keys/elements matters. If you want to process those data structures, an OTab/OrderedDict can make dealing with those structures much easier.

3. **What is a Tup?**

   Short Answer: A Tup is like a Python tuple

   A Python tuple is useful for creating an immutable heterogeneous list:

```
>>> t = (1, 2.2, 'three', None, {}) #  Python tuple
>>> print t[1]  # 2.2
```

   Similarly, a Tup is very much like a Python tuple, but used within C++:

```
Tup t(1, 2.2, "three", None, Tab()); // C++ tuple
cout << t[1];   # 2.2
```

   Like python tuples, you can't resize or grow tuples once you grown them. They really just are an "easy" way to pass around a bunch of heterogeneous data, just like Python.

4. **Why is the first argument of the Tup constructor not working?**

   In other words, why isn't *t* below a tuple of three arguments?:

```
Tup t("(1,2.2,'three')"); /// NOT a TUPLE of three arguments???
```

   Tup is slightly different than Tab/Arr/OTab: In those data structures, the first argument is a string argument that's MEANT to be Eval'ed:

```
Tab t = "{'a':1}";    // Like:  Val tv = Eval("{'a':1}");
Arr a = "['a', 'b']"; // Like:  Val av = Eval("['a', 'b']");
OTab o=  "o{'a':1}";  // Like:  Val ov = Eval("o{'a':1}");
```

   The first (and all) arguments of a Tup are simply Vals that are taken as is:

```
Tup a("17");  // Tup is a 1-element tuple with a single string: '17'
Tup b("{}");  // Tup is a 1-element tuple with a single string: '{}'
Tup c(1,2);   // Tup is a 2-element tuple: int, int
```

   Understanding this, we revisit the example:

```
Tup t("(1,2.2,'three')"); // 1-element tuple of single string:
                          //   '(1,2.2,"three")'
```

   This means t is a 1-element string. Probably what the user wanted was a 3-element tuple like:

```
Tup tt(1, 2.2, 'three');
```

   The point is that the Tup does NOT Eval any of its arguments like the Tab or OTab: it simply takes them as-is.

5. **What are the int_n and int_un?**

   Short Answer: The int_n is equivalent to the Python arbitrary-sized integer (sometime called the long). The int_un is just the unsigned version.

If you need to compute with larger integers than int_u8, then PicklingTools 1.2.0 now supports arbitrary-sized integers. From Python:

```
>>> a = 700;                          // small enough to be normal int
>>> b = 700L;                         // force to Python long
>>> c = 10000000000000000000000000L;  // only fits in Python long
>>> google = 10**100
```

From C++:

```
// C++ int_un, int_n
int_4 a = 700;                        // small enough for normal int
int_n b = 700;                        // force to C++ arbitrary-sized int
int_n c = StringToBigInt("10000000000000000000000000L");
          // We can't represent literals too large in C++, so we have
          // to turn larger ints into strings.
int_n d = "10000000000000000000000";   // NEW syntax in PicklingTools 1.4.1
int_n google = IntExp(10, 100);
```

6. **What's the performance of the int_n?**

   The int_n seems to be about as fast as the Python long. The main test of this assertion was computing large combinations (n choose k) in both C++ and Python: incidentally, there is tremendous support for combinations in PicklingTools 1.2.0.

7. **Why do I have to use StringToBigInt? It seems clumsy.**

   Short Answer: You don't if you update to PicklingTools 1.4.1.

   Yes, using *StringToBigInt* to make very integers is clumsy:

   ```
   int_n c = StringToBigInt("10000000000000000000000"); // Pre 1.4.1
   ```

   This interface still works, but as of PicklingTools 1.4.1, you can use strings directly:

   ```
   int_n c1 = "10000000000000000000";  // In 1.4.1 and beyond
   string s("247365789243052435234757892340");
   int_n c2 = s;
   ```

   This should make using int_un and int_n much easier to use.

8. **Some of the interactions with plain ints and int_n/int_un don't work?**

   In PicklingTools *before* 1.4.1, this code compiles ...

   ```
   int_un ii = StringToBigUInt("10000000000000000000000000") + 1;
   cout << ii << endl;
   ```

   ...but gives the wrong answer!

   ```
   2003764205206896641   // Should be 100000000000000000000000001
   ```

   Why? Because when overloading *int_un* + *int*, the compiler chooses to downconvert (using the operator *int_8* of the *int_un* class) rather than upgrade the *int* to a *int_un*. In other words, the downcast forces the addition to be *int* + *int* which exceeds the size of a normal int.

   The interactions with overloadings and outcastings are complex: especially when native types (like *int*) are involved.

   The way around this in early PicklingTools? Force the *int(1)* to a *int_un(1)*:

```
int_un ii = int_un("10000000000000000000000000") + int_un(1);
cout << ii << endl;
```

BUT as of PicklingTools 1.4.1, **this is all fixed!**

```
int_un ii = int_un("10000000000000000000000000") + 1;
cout << ii << endl;
```

This gives what's expected:

```
100000000000000000000000001
```

A few minor changes were made to make int_un/int_n work better with plain ints. One result is that we got rid of the need to use StringToBigInt (see point above), another is that we can use ints and big ints as expected. The only interface that changed (yes, this should be a major release change, but considering that it was fundamentally flawed, this is excusable) as that you cannot ask for ints out of an int_n:

```
int_n in = "10000";
int_8 out = in;     // WORKS in < PicklingTools 1.4.0
                    // FAILS in PicklingTools 1.4.1 and above

int_8 out = in.as(); // How to get out in PicklingTools 1.4.1. and above
```

Allowing the int_8 outcast simply allows too many ways for the overload engine to run into ambiguities: we prefer to make int_n work well with plain ints. In our tests, there were only a few places where we did, so we justify that this shouldn't be too big a code change. But it is a code change. There is a macro you can put in your code:

```
#if defined(OC_BIGINT_OUTCONVERT_AS)
#  define AS(x) ((x).as())
#else
#  define AS(x) ((x).operator int_8())
#endif
```

With this macro, you can use the following code, and it will work with any version of PicklingTools:

```
int_n ii = "10000";
int_8 n = AS(ii);    // Works with all versions of PicklingTools
```

In C++11, there is support for *explicit* outcasts which would fix the problem, but currently, as most of our users are back in C++0x or earlier, we can't take this upgrade right now. This is very frustrating and we are looking into cleaning this up.

2.6 C++ and the new PickleLoader: New in PicklingTools 1.2.0

0. **Why is there a new implementation for loading Pickling Protocol 0 and 2?**

   Short Answer: Speed, simplicity, maintainability, features.

   As of PicklingTools 1.2.0, there is a new implementation of the the loader for Pickling Protocol 0 and 2. It's significantly faster than the previous versions, usually 2-10x faster. From the speed_test.cc metric: this shows version 1.2.0, where the new implementation appeared and version 1.3.1, with minor speed improvements:

```
# C++ is on par with Python with Unpickling 0
OLD IMPL: Unpickling 0: 36.66 seconds
NEW IMPL: Unpickling 0:  7.48 seconds  # 5 times faster! (1.2.0)
NEW IMPL: Unpickling 0:  7.20 seconds  #                  1.3.1

# C++ implementation is 2x faster than previous implementation
OLD IMPL: Unpickling 2:  9.24 seconds
NEW IMPL: Unpickling 2:  6.24 seconds  # 1.5x faster   (1.2.0)
NEW IMPL: Unpickling 2:  4.34 seconds  # 2x   faster   (1.3.1)
```

The core of the new loader supports BOTH protocols (Pickling 0 and Pickling 2) easily so there aren't two separate implementations: this makes maintenance significantly easier. Also, the older loaders DO NOT support all the new features of PicklingTools 1.2.0 (OTab/Tup/int_un/int_n) very well, if at all. ONLY the new loader supports the new features.

In general, the new loader code is significantly simpler, and reflects much more closely what Python does. It really is better on all axes. By default, all the MidasThingees use the new loader.

1. **What if I really want to use the old loaders?**

   You can. When you specify your serialization protocol (to LoadValFromArray for example), use either SERIALIZE_P0_OLDIMPL to get the old Protocol 0 depickler or SERIALIZE_P2_OLDIMPL to the the old Protocol 2 depickler:

   ```
   // From C++:  Load using new loader for Pickling Protocol 2
   LoadValFromArray(result, buffer, SERIALIZE_P2); // uses new better loader

   // Use older loader for Pickling Protocol 2
   LoadValFromArray(result, buffer, SERIALIZE_P2_OLDIMPL); // old loader
   ```

   Caveat Emptor. The older loaders do not support the newer features found in PicklingTools 1.2.0.

   The real reason we support the old loaders is just in case you really need to go back (because the new loader doesn't work), you can.

2. **Why is there not a newer *saver* for Pickling Protocol 0 and 2?**

   Short Answer: There is no major reason currently.

   Currently, we can dump data close to Python speeds (these numbers are from the speed_test.py and speed_test.cc where we serialize "about" the same dictionary in both):

   ```
   # Speedup so that C++ is on par with Python Pickling 0
   Python Pickling 0:  Python  12.70 seconds  # pre 1.2.0
   Python Pickling 0:  C++     14.56 seconds  # 1.2.0
   Python Pickling 0:  C++     12.23 seconds  # 1.3.1

   # C++ 6x faster than Python Pickling!
   Python Pickling 2: Python:   8.05 seconds  # pre 1.2.0
   Python Pickling 2:  C++      1.36 seconds  # version 1.2.0
   Python Pickling 2:  C++      1.30 seconds  # version 1.3.1
   ```

   We are about as fast as Python for Pickling, so there's no real need to rewrite it. The picklers also support all features of PicklingTools 1.2.0.

2. **How can I verify these speeds or other speeds of the PicklingTools?**

   Short Answer: Take a look at the speed_test.cc in the C++ directory of the PicklingTools or the speed_test.py in the Python directory of the PicklingTools.

Here are some recent results from PicklingTools 1.2.0:

```
 Current speeds in seconds: (-O4 on 64-bit Fedora 13 machine)
                          C++        Python
                          g++ 4.4.4   2.6   # Comments
                          ----------------------------
Pickle Text                5.64       5.12  # About equiv
Pickle Protocol 0         14.56      12.70  # Python slightly faster
Pickle Protocol 2          1.36       8.05  # C++ significantly faster
Pickle M2k                 3.03        N/A
Pickle OpenContainers      1.31        N/A  # OC is fastest overall

UnPickle Text             32.55      38.23  # About equiv
UnPickle Protocol OLD 0   36.66       7.20  # Why OLD P0 is deprecated!
UnPickle Protocol NEW 0    7.48       7.20  # About equiv
UnPickle Protocol OLD 2    9.24       4.46  # Why OLD P2 is deprecated!
UnPickle Protocol NEW 2    6.24       4.46  # Python still faster
UnPickle M2k               9.00        N/A
UnPickle OpenContainers    4.12        N/A  # OC is fastest overall
```

More recent results from PicklingTools 1.3.1 (Notice the speedups!):

```
 Current speeds in seconds: (-O4 on 64-bit Fedora 14 machine)
                          C++        Python
                          g++ 4.5.1   2.7   # Comments
                          ----------------------------
Pickle Text                5.90       4.82  # Python slightly faster
Pickle Protocol 0         12.23      12.65  #
Pickle Protocol 2          1.30       3.41  # C++ significantly faster
Pickle M2k                 2.98        N/A  #
Pickle OpenContainers      1.25        N/A  # OC is fastest overall

UnPickle Text             23.40      38.19  # C++ faster
UnPickle Protocol OLD 0   29.53       7.13  # Why OLD P0 is deprecated!
UnPickle Protocol NEW 0    7.24       7.13  #
UnPickle Protocol OLD 2    8.23       3.66  # Why OLD P2 is deprecated!
UnPickle Protocol NEW 2    4.34       3.66  # Python still faster
UnPickle M2k               9.72        N/A  #
UnPickle OpenContainers    2.41        N/A  # OC is fastest overall
```

The speed of pickling2/unpickling2 in Python 2.7 = 3.41+3.66 = 7.07 secs

The speed of pickling2/unpickling2 in Ptools 1.3.1 = 1.30+4.34 = 5.64 secs

The round trip time of the C++ impl is about 15-20% faster than Python 2.7.

3. **Why aren't the PicklingTools C++/Val pickling/unpickling routines always faster than the Python version?**

   Short Answer: The Python routines use the cPickle module (used by speed_test.py) which is written in C already. Thus the Python is comparable to the PicklingTools C++/Val PickleLoader implementation.

   Discussion: The speed_test.py uses cPickle, which is a Python module written in C. The speed_test.cc uses LoadValFromArray, which is written purely in C++. The tight-loops in the speed_test are all in the pickling modules are written in C/C++, so they are substantially the same speed.

4. **Do you want to talk about why the Unpickling with C++/Val seems slightly slower than cPickle/Python?**

   Short Answer: Yes.

   Pickling/Unpickling is very much tied into the dynamic objects of the implementation.

In Python, all PyObject objects are passed around by pointer, and moving them around is trivial. This fact is important for unpickling because the "values" stack (for temporary storage) can be small (an array of pointers): it's trivial to move things on and off of that stack (by moving a pointer). That simple notion contributes quite a bit to the speed of the Python unpickling. Also, the creation and allocation of Python objects has been very heavily optimized for a single threaded engine: For example: the PyList_New (for both dict and list) has a cached freelist that makes many allocations trivial. Internally, allocating lists and dicts and lists don't have to necessarily hit a generic "malloc", and that can be a major speedup: especially since those dict/lists tend to get reused.

In C++, all Val objects are passed around by value. Using the move-semantics and/or swaps, we can still move objects around in constant time, just not as fast a pointer move. This "by value" characteristic unfortunately means the "values" stack (for unpickling) has a bigger footprint in memory than a plain array of pointers: that extra memory makes the speed_test slower. Also, the creation and allocation of objects (Val, Tab, etc.) *on purpose* relies on the generic "new/malloc" machinery for allocating memory: this makes *some* object creation significantly more expensive. Thus the creation of some objects (strings, dicts) for unpickling is not as fast a Python. Note that this decision to use the generic memory allocation pool is on purpose for two major reasons. One, we can use valgrind directly to help us find memory leaks (although Python can as well, it's not quite as easy) Two, the data structures can be used generically by threads. We have considered adding "per data structure" allocators (specialize allocation for Tab, Arr, etc.), but this takes away from the thread-neutrality of the OpenContainers data structures.

The Python choices make some objects easier to move, create, allocate, but at the cost of disallowing concurrent threads. The C++ choices make some objects some objects more expensive to move, create, allocate, but allow threads (and valgrind).

Simply speaking, pickling and unpickling is tied directly to the object model: the limits/strengths of that object model affect the speed. Creation dynamic objects in C (PyObject) or C++ (Val) has an inherent cost.

2.7 XML Support: New in PicklingTools 1.3.0

0. **Do the PicklingTools support XML?**

   Short Answer: If the XML is strictly a recursive key-value structure, yes. If the XML represents a generic document, no.

   If the XML just represents recursive key-value entries, then there is a equivalent and obvious mapping between Python dictionaries and XML. Consider:

```
<top>
  <Futurama>
   <name>Phillip</name>
   <age>1036</age>
  </Futurama>
  <Simpsons>
   <name>Homer</name>
   <age>36></age>
  </Simpsons>
<top>
```

   There is nested structure, but all the tags are either simple keys or just containers for other keys, so there is an obvious XML correspondence:

```
>>> top = {
...     'Futurama': {
...         'name': 'Phillip',
...         'age': 1036
```

```
...     },
...     'Simpsons': {
...         'name': 'Homer',
...         'age': 36
...     }
... }
```

A *document* (which the PicklingTools have real trouble supporting) is something with content and keys interspersed. For example:

```
<top>
  <text>It was the <it>best</it> of times,
        it was the </it>worst<it>of times</text>
</top>
```

In the example above, the text has content and tags interspersed together: what would be a good equivalent Python dictionary?:

```
>>> # SOME HACKS?  DOES NOT WORK THIS WAY!!!!!!!!!!
>>> top = { 'text': ['It was the', { 'it':'best' }, 'of times,'] }
...          #  or
>>> top = { 'text': o{0:'It was the', 'it':'best', 1:' of times,' } }
```

There's not really a good correspondence, so we don't support it; In that case, the tools will throw an exception or ignore the content, depending on what the context and/or user specification.

1. **Do the PicklingTools XML tools handle lists?**

   Yes, there is support for lists (but take a look at the PicklingTools XML Documentation, included in this distro, for more details for corner cases):

   ```
   <top>
     <friend>Zoidberg</friend>
     <friend>Lrrr</friend>
   </top>
   ```

   The equivalent Python dictionary would be:

   ```
   >>> top = {
   ...     'friend': [ 'Zoidberg', 'Lrrr' ]
   ... }
   ```

   Lists can be arbitrarily complex (just like Python dictionaries) containing primitive types, other lists, or other dictionaries.

2. **What about attributes in XML?**

   Short Answer: As long as the XML is strictly recursive key-value, yes.

   There are a few conventions (depending on user options), but attributes are supported. The default is make a special dictionary called __attrs__:

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <top>
        <book title="A Tale of Two Cities" date="1859">
           <chapter>text</chapter>
        </book>
   </top>
   ```

The attributes get put in a special table:

```
>>> {
...   'book':{
...      '__attrs__':{
...         'date':1859,
...         'title':'A Tale of Two Cities'
...   },
...   'chapter':'text'
>>> }
```

If you use the *unfolding* feature (XML_LOAD_UNFOLD_ATTRS), then the attributes are unfolded into the book table as keys that start with '_':

```
>>> {
...   'book':{
...      '_date':1859,
...      '_title':'A Tale of Two Cities',
...      'chapter':'text'
...    }
... }
```

There is also an option to drop attributes all together.

3. **Can I go back and forth between XML and Python Dictionaries?**

   Short Answer: Yes.

   A lot of effort has gone into making the tools be able convert back and forth between XML and Python dictionaries, preserving all structure and information. There are a number of options for the tools which seem silly, but are there for allowing the user to fine-tune the transformations so as to not lose information (if possible). Again, this assumes a recursive key-value structure only.

   The transformations are completely invertible (depending on how the options are tweaked):

   ```
   XML_to_dict(dict_to_XML(something))-> something
   dict_to_XML(XML_to_dict(something))-> something
   ```

3. **Where can I find further information on the PicklingTools XML tools?**

   There is relatively comprehensive document (about 20 pages) full of examples and descriptions of the tools: this document is included in the docs area of the PicklingTools distribution. The docs are in 3 formats: text document, PDF document, and HTML version describing the how the XML tools work.

4. **What advanced features of XML does the PicklingTools support?**

   There is no current support or planned support for DTDs (the world seems to have turned to XML schemas anyway, which are written in XML). As of 1.3.1, we read but completely ignore DTD.

   There is currently no namespace support, although we intend to support it in a future release. As of 1.3.1, we recognize namespaces (and the :) but don't do much with them.

   There is currently only support for UTF-8.

5. **Why are there two different versions of the XML tools for Python?**

   Short Answer: speed.

   The original version of the XML tools (in xmltools.py) was written in pure Python: it used *pure Python* to parse and do I/O. It's easy to get going and try out the raw Python (as simple as `import xmltools`),

but those routines tended to be slow. For smaller XML tables, this was fine, but larger XML tables really felt the slowness.

As of PicklingTools 1.4.1, there is a C Extension module (in cxmltools.py) which does all the XML to/from dict conversion from C++ which increases the speed of the dict to XML by 6x-10x and the speed of XML to dict by 60x-100x.

The Python C Extension module is more difficult to build and use, but significantly faster. See the README in the PythonCExt directory or the XMLtools document (mentioned in item 3 above).

2.8 C++ and JSON: New in PicklingTools 1.3.2

1. **What is JSON?**

   JSON stands for "JavaScript Object Notation": it comes from the JavaScript Programming Language. See http://www.json.org

   JSON is, with a very few differences, just plain textual dictionaries:

   ```
   { "a": True, "b":11, "c":3.1 }
   ```

2. **What are the differences between JSON and Python Dictionaries?**

   JSON uses 'true', 'false' and 'null' for 'True', 'False', and 'None' (respectively). Strings are unicode, and quotes around strings are ONLY double-quotes (no single quotes). Other than that, they are just like Python Dictionaries:

   ```
   { 'a':1, 'b':True, "c":None}    # Python Dictionary

   { "a":1, "b":true, "c":null}    # Equivalent JSON
   ```

3. **What kind of support does PicklingTools offer for JSON?**

   From Python, there are already many tools available (already built-in), so there is no reason for PicklingTools to extra work to support JSON. From Python, the 'json' module ('import json') should have all you need.

   From raw C++, there is a new reader that turns JSON files/text into Tab/Arr/Vals:

   ```
   #include "jsonreader.h"
   Val json;
   ReadValFromJSONFile("filename.txt", json)
   // json now contains a Tab which represents
   //   the JSON structure
   ```

   You can take any Tab/Arr and turn it into a JSON text file or textual representation with the *JSONPrint* routine:

   ```
   #include "jsonprint.h"
   Val v = Tab("{ 'a':1, 'b':2.2, 'c':'three' }"); // Manipulate like plain Tabs

   JSONPrint(std::cout, v);  // ... but print out as JSON
   ```

4. **Do the MidasTalkers, etc. support JSON?**

   Not right now. We are evaluating whether it makes sense. Most people just use the tools orthogonally to the MidasTalker/MidasServers.

2.9 Conformance or Validation Support: New in PicklingTools 1.3.3

0. **Do the PicklingTools support something like an XML schema for Python dicts?**

   Yes. There is a new routine in the C++ opencontainers library called *Conforms* which implements something like XML schema checking for Python dicts. There is also a standalone Python module (called *conforms.py*) in the Python area which behaves almost exactly like the C++ version.

   A key customer has asked for something like an XML schema for Python dictionaries. An XML schema allows a user to "validate" an XML document against a template to see if the structure of the document matches the schema. *Conforms* allows a similar type of operation.

1. **How does Conforms work?**

   The user provides a message to be "validated" and a prototype which demonstrates what a valid message looks like.

```
// C++
if (Conforms(message, prototype)) {
    // message is valid
} else {
    // message is malformed
}
```

   The Python is similar.

2. **What structure do messages to Conforms look like?**

   Typically Python dicts or lists, but any valid type will work. For example, in C++:

```
#include "occonforms.h"

Val instance = Tab("{ 'host': 'ai1', 'port': 8888 }");
Val prototype= Tab("{ 'host': '',    'port': 0    }");
if (Conforms(instance, prototype)) { ... }
```

   The *Conforms* routine sees if the structure and keys of the instance match the keys and types of the prototype. If they do (as they do in this case), the instance is considered conformant.

   The equivalent Python would be:

```
>>> from conforms import *
>>>
>>> instance = { 'host': 'ai1', 'port': 8888 }
>>> prototype= { 'host': '',    'port': 0    }
>>> if conforms(instance, prototype) :   # conformance check
...    pass
```

   The Python version is more lenient of types under a LOOSE_MATCH than the C++ version because Python has a plethora of types.

3. **Where can I find more information about conforms?**

   The PicklingTools 1.3.3 User's Guide has a dedicated section discussing all the gory options in detail. It mostly discusses the C++ version, but Python usage is almost identical (modulo language differences between C++ and Python).

   The help page for the Python *conforms* is also quite informative:

```
>>> import conforms
>>> help(conforms)
```

2.10 Java Support: New as of PicklingTools 1.5.1

0. **Is there Java support in PicklingTools?**

   Yes. As of PicklingTools 1.5.1. The goals are two-fold:

   (a) Allow Java to talk to C++ and Python easily

   (b) Make Python dictionaries easy to manipulate in Java

1. **What documentation and examples are there?**

   The Java subdirectory contains the needed code: anything that ends with "_ex*.java" is an example.

   There is a full-fledged Java document in the Docs area (and on the Web site) for Java.

2.11 Python C Extension Modules: New as of PicklingTools 1.6.0 (and 1.3.3)

0. **What are Python C Extension Modules?**

   Python C Extension are just new Python libraries. They call be called from Python, but they are written in C and "linked in" to Python.

   Python C Extension Modules are written in C usually because they need the extra speed of C.

1. **What are the Python C Extension Modules?**

   There are currently two as of PicklingTools 1.6.0.

   The first is the pyocconvert module. It's purpose is to be able to convert between Python objects and C++ Val objects so that we can use the very fast XML/Dict conversion modules written in C++. Most people won't use this module directly, but use the "cxmltools" module (which imports the "pyocconvert" module and wraps it nicely). There is currently a full document describing this: See the XML documentation for more information about the XML/dict conversion tools.

   The second is the pyocser module. It is a very simple module with only two functions, and they are inverse of each other:

   ```
   ser = ocdumps(pyobject)    : Serialize pyobject into string ser
   pyobject = ocloads(ser)    : Deserialize from string ser into pyobject
   ```

   The pyocser module allows the use of OC serialization (instead of pickling from the pickle/cPickle modules). Currently, OC Serialization tends to be 1.5-2x faster than Python pickling, and is the *only* way to serialize very large (over 4G) strings and numpy arrays (as Python pickling (until Python 3.5?) doesn't work for strings or numpy arrays over 4Gig).

2. **How do you build the Python C Extension modules?**

   It depends.

   If you are using the PTOOLS X-Midas option tree, it gets built for you automatically when you `xmbopt ptools`.

If you are just building straight out of the PicklingTools basline, change to the `PythonCExt` dir and type 'python setup.py build'. E.g.:

```
% cd PicklingTools1.6.2Release/PythonCExt
% python setup.py build
```

Make sure the python you want to use is the python you build with. The C extension module is only compatible with the very of Python it was built with. When you build, you should see something like this (and it might be a little slow):

```
running build
running build_ext
building 'pyobjconvertmodule' extension
creating build
creating build/temp.linux-x86_64-2.6
gcc -pthread -fno-strict-aliasing -O2 -g -pipe -Wall -Wp,-D_FORTIFY_SOURCE=2 -fexceptions -fstac
gcc -pthread -fno-strict-aliasing -O2 -g -pipe -Wall -Wp,-D_FORTIFY_SOURCE=2 -fexceptions -fstac
creating build/lib.linux-x86_64-2.6
g++ -pthread -shared build/temp.linux-x86_64-2.6/pyobjconvertmodule.o build/temp.linux-x86_64-2.
running build
running build_ext
building 'pyocsermodule' extension
gcc -pthread -fno-strict-aliasing -O2 -g -pipe -Wall -Wp,-D_FORTIFY_SOURCE=2 -fexceptions -fstac
gcc -pthread -fno-strict-aliasing -O2 -g -pipe -Wall -Wp,-D_FORTIFY_SOURCE=2 -fexceptions -fstac
g++ -pthread -shared build/temp.linux-x86_64-2.6/pyocsermodule.o build/temp.linux-x86_64-2.6/pyc
```

There should be a `build` directory. Under there should be something like `lib.linuxx86_64-2.6` with two files: `pyosermodule.so` and `pyobjconverrmodule.so`.

2. **How I can't import pyocser or pyocconvert from Python?**

   Short Answer: You need to set-up your PYTHONPATH

   If you try to `import pyocser` and it fails, that probably means you to add the build directory to your PYTHONPATH. In order for the import of this foreign C module to work, you have to tell python where to look. That's what the PYTHONPATH is for:

   ```
   # If your PYTHONPATH is empty
   % setenv PYTHONPATH "/path/to/PicklingTools160Release/PythonCExt/build/lib.linux-x86_64-2.6"

   # If you already have a PYTHONPATH
   % setenv PYTHONPATH "${PYTHONPATH}:/path/to/PicklingTools160Release/PythonCExt/build/lib.linux-x
   ```

   Once your PYTHONPATH is set, you should be able to import correctly. You need to make sure the files `pyocsermodule.so` and `pyocconvertmodule.so` are both files in one the directories specified by the PYTHONPATH.

   Note that the name of the path will be different depending on the system you are on: In the example above, we were on a 64-bit Intel linux box using Python 2.6, this the final path component was `lib.linux-x86_64-2.6`. On a 32-bit Intel linux box using Python 2.4, it would be `lib.linux-i686-2.4`. Adjust the PYTHONPATH setting for your system.

3. **How come it imports but fails?**

   The Python C Extension modules have to built for *each* different platform and different platform it uses. So, if you have multiple Pythons on the same system, or multiple systems, make sure your PYTHONPATH points to the proper version of linux/Python.

4. **Why is the purpose of the pyoceser module? Doesn't pickle do that?**

The pyocser module does indeed offer similar functionality of the pickle module: it allows you to turn arbitrary Python objects into strings for putting to a file or socket, which you can them recover and turn back into Python objects. The pyocser module implements something called "OC Serialization" and the pickle module implements "pickling" (which is the default Python serialization).

There are two main reasons for the pyocser module:

1. Speed.

   In some tests, the serialization/deserialization can be 1.5-2.0x faster. There is a speed_test.py in the Python directory where you can compare the speed of different serializations.

2. Works with very large data.

   As of this writing, Python 2.4-2.7 cannot handle strings or numpy arrays greater than 4 Gig. Python 3.5 is about to add/already has support for very large data, but this hasn't been back-ported, and it's unclear if it will. (Hopefully?)

The OC Serialization originally came from C++ (OC stands for OpenContainers) and tends to be faster than Pickling. See the speed numbers from the "new unpickler" previously.

Currently, OC serialization has some limitations:

1. It only can serialize base Python types and a few others:

   ```
   int, long, float, complex, bool, dict, list, string, None
   OrderedDict, Numpy Arrays, Numeric Arrays
   ```

2. It uses the endian of the machine it's on.

Because OC serialization has to work with both C++ and Python, the first limitation allows C++ and Python to still work together; Passing class instances is very difficult between two languages, and many times, a dict will suffice.

2.12 cx_t<INT>: New as of PicklingTools 1.6.0

0. **What are the complex ints types? What is cx_t<INT>?**

   Short Answer: complex types for all integers.

   As of PicklingTools 1.6.0, the Val supports all different types of complex integers (besides `complex_8` and `complex_16`). These are:

   ```
   cx_t<int_1>, cx_t<int_u1> :  Val tags 'c' 'C'
   cx_t<int_2>, cx_t<int_u2> :  Val tags 'e' 'E'
   cx_t<int_4>, cx_t<int_u4> :  Val tags 'g' 'G'
   cx_t<int_8>, cx_t<int_u8> :  Val tags 'h' 'H'
   ```

   All support for `complex_8` and `complex_16` remains the same.

1. **Why do I need complex integers? A lot of operations don't make sense**

   Short answer: As a container.

   If you have to ask, you probably don't need it. But, a lot of Digital Signal Processing (DSP) happens after data has been sampled in integer form and mixed as complex integers (or sampled as complex integers). So data comes in over the wire as complex ints.

For a lot of complex operations, integer math is much cheaper than floating point math. Adding, multiplying two complex ints or by a int constant can be significantly faster than the equivalent complex floating point math.

For large data sets. `cx_t<int_2>` is much smaller than the `complex_8` (half as large) or `complex_16` (quarter as large). If you are saving data to disk, or moving data through a system, the extra savings in memory can be important for speed.

Many operations are exactly the same: getting the real or imaginary components, copying, adding, subtracting, multiplying complex ints. Division will work, but it usually doesn't make sense. Some operations still make sense (like `mag2`), but they can roll-over quickly, caveat emptor.

In general, complex integers are very useful for a limited set of operations.

2. **Why do you not use C++ STL complex numbers?**

   Short answer: speed and interface

   One main reason is that the C++ STL complex template doesn't allow setting just a real component or an imaginary component: you have to create a brand new complex with the new component, or use methods for everything:

```
// C++ STL complex
complex<float> a(1.2, 3.4);
a.real = 77.7;     // DOESN'T COMPILE
a.real() = 77.7;   // May compile if using gcc?
a.real(77.7);      // May compile?

complex<float> newa(77.7, a.imag());  // Have to create brand new
```

   For people with experience with FORTRAN complex, this is very non-intuitive. The complex values are much to manipulate when can set the components individually:

```
cx_t<int_1> a(1,2);
a.re = 7;   // change the real component
a.im = 15;  // change the imaginary component
```

   Another major reason is speed: The X-Midas baseline (a DSP framework) did an experiment sometime ago comparing the speed of C's complex, C++ STL complex, and a similar `cx_t` class, and saw some major impacts of having to create new complex temporaries all the time: it was slower.

   Because we care about speed and the interface, Picklingtools has it's own `cx_t` type.

3. **Are C++ STL complex and PicklingTools complex types compatible?**

   Short answer: They should be layout compatible.

   If you have an array of C++ STL complexes, you should be able to go `cx_t` types and vice-verse because they should be layout compatible (i.e., a struct with a real and imaginary component). For example, to zero the imaginary component of every element of a vector:

```
vector<complex<float> > a = ... some big array ...;

complex_8 *ap = reinterpret_cast<complex_8*>(&a[0]);
for (int ii=0; ii<a.size(); ii++) {
    ap[ii].im = 0;  // zero the imaginary component
}
```

   If you like the complex class of the STL C++, great, use it. However, if you need to put something in a Val, it needs to be a `cx_t` or `complex_8` (same as `cx_t<real_4>`) or `complex_16` (same as `cx_t<real_8>`):

```
complex<int_1> a;
Val v = cx_t<int_1>(a.real(), a.imag()); // Copy to a val
```

The only way to carry an array of complex ints in a Val is with the `Array` class from PicklingTools:

```
// Create an Array of 10 complex ints
Array<cx_t<int_1> > a(10);
a.fill(cx_t<int_1>(0,0));

// Copy into a Val
Val v = a;

// Get a reference from the Val
Array<cx_t<int_1> >& aref = v;
for (int ii=0; ii<aref.length(); ii++) {
    aref[ii].re = 1;
}
```

In general, you should be able to reinterpret cast from a `cx_t<T>` to a `complex<T>` and vice-versa.

4. **Why do I care about complex ints?**

   If you need them for DSP, complex ints are very important. Otherwise, they probably won't matter to you too much.

   Having said that, it may be useful to use the complex ints if you need a pair of integers. For example, if you have a graphing package with x and y co-ordinates, the real and imaginary components of a complex int can be used as the the x and y components for a point on the screen.

2.13 M2k

0. **Why is there an M2k area provided in the PicklingTools distribution?**

   Short Answer: To provide the better OpalDaemon and OpalSocketMsg

   The "baseline" OpalDaemon and OpalSocketMsg "work", but are limited to exactly one type of serialization when running. The OpalPythonDaemon (a newer version of the OpalDaemon meant to replace the OpalDaemon) gives the user adaptive serialization: the ability dynamically to decide on a per connection basis the type of serialization. Another feature is that components can utilize the Python Pickling Protocol 0 and 2: this gives more options for serialization and allows Python-only clients to talk to the OpalPythonDaemon easily.

   Strictly speaking, these components (OpalPythonDaemon and OpalSocketMsg) probably should have gone into the Midas 2k baseline at some point, but since Midas 2k development and support was suspended, the OpalPythonDaemon and OpalPythonSocket languish in the M2k area of the PicklingTools.

1. **What is the current status of the M2k area?**

   As of PicklingTools 1.3.2, the M2k area has been cleaned up. The OpalPythonDaemon and OpalPythonSocketMsg both support NumPy. OrderedDictionaries and Tuples can be processed, even though M2k has no support for them. The loader for Python Pickling 0 and 2 in m2k is significantly better (faster, cleaner, robust).

   After many long debates, we have currently decided NOT to use the XML tools to support XML natively in M2k (although if someone really cares, we can).

2. **Does M2k work with OCSerialization?**

It has worked with OC Serialization for some time, but has recently been updated to behave better and work with large arrays. The OpalPythonDaemon and PythonOpalPythonTableWriter have both been updated so they can handle very large (over 4Gig) data and strings.

3. **Why doesn't very large data work with M2k?**

   Short Answer: No one ever thought we'd have Vectors or strings over 4Gig.

   In order for very large files to work, a few changes have to be made to MITE/M2k: newer versions should have these changes, but essentially, the `int_u4s` of `Vector` need to changed to ``size_t` and the `unsigned len` of `oceasystring.h` needs to be an `size_t` as well.

2.14 X-Midas

0. **What is the difference between PicklingTools and PTOOLS?**

   Short Answer: PTOOLS is an X-Midas option tree packaged with the PicklingTools distribution.

   PTOOLS is just a repackaging of all the C++ and Python code in the C++ and Python areas of the PicklingTools distribution. This packaging makes it easy for X-Midas users to use all the PicklingTools features in X-Midas.

   In particular

   (a) XMPY scripts can use the Python MidasTalker, MidasServer, MidasYeller and MidasListener easily (found in python subdir of PTOOLS)

   (b) X-Midas C++ primitives can easily write Midastalker, MidasServer, Midasyeller, MidasListener. They also support the Tab/Arr/Val abstractions so C++ primitives can feel like they are using Python.

   (c) The Python C Extension modules get built for you

1. **How do I use the PTOOLS option tree?**

   Copy the ptools100 (or whatever) to your X-Midas option tree area. Then add and build it like standard X-Midas option trees:

```
xm> xmopt ptools /full/path/to/copy/ptools100  # must lower-case path
xm> xmp +ptools
xm> xmbopt ptools
```

   Watch out for CAPITAL LETTERS in the Pathname: X-Midas doesn't like!

2. **How do I write a C++ primitive or an XMPY script?**

   Short Answer: Look for an example in the option tree

   There are plenty of example primitives in the "host" area (with corresponding explain pages in the "exp" area). The only potential gotcha for a primitive: make sure you look at the "cfg" area and imitate how "primitives.cfg" sets it options.

   There are plenty of Python examples in the "python" area that should look suspiciously like all example from the Python piece of the PicklingTools distribution.

2.15 SerialLib

0. **What is seriallib?**

   Seriallib is a Python library with utilities for converting back and forth between Python dictionaries and one of three other formats: Windows .ini files, Key-Value strings, or (in some limited cases) Python objects.

1. **How do I convert between .ini files and Python dicts?**

   For converting back and forth between Windows .ini files and Python dictionaries, the important routines are:

```
dict_to_ini(d)
    Given a dictionary, return a ConfigParser instance reflecting the dict's
    structure.

    Currently only handles dicts that look like .ini files; that is:

    { 'section name 1': { k1: v1, ..., kN: vN},
      'section name 2': ...
      ...
    }

    N.B.: The returned ConfigParser will store the values as they were given,
    NOT convert them to strings. This means that you will need to do 'raw'
    gets to bypass the value interpolation logic that assumes all values are
    strings.  getint() and getfloat() will also fail on non-string
    values. (This caution applies to Python 2.4; it is unknown whether it
    still applies in newer Pythons.)


ini_to_dict(ini_file)
    Given a ConfigParser instance or pathname to an INI-formatted file,
    return a dict mapping of the ConfigParser. Each section becomes a
    top-level key.
```

   Simply import seriallib and do a help to see these help pages.

2. **How do I convert between Key-Value Strings and Python dicts?**

   For converting back and forth between Key-Value Strings and Python dictionaries, the important routines are:

```
kvstring_to_dict(s, sep=':')
    Given a multi-line string containing key-value pairs, return a flat dict
    representing the same relationships. Whitespace around the separator is
    ignored.

    Expects one entry per line in the input.

    Note that all keys and values are interpreted as strings upon return.

dict_to_kvstring(d, sep=':')
    Given a flat dictionary, return a simple key-value string. Each entry
    in the dict gets one line in the output.

    Examples:

      { 'key1': 'the first value',
        2: 'the second value' }

        ->
      'key1:the first value\n2:the second value'


    If optional 'sep' is given, it is used to separate keys and values.  In any
    case, 'sep' should probably not be in the string representation of any key!
```

```
    If the input dict is not flat, the behavior is undefined.
```

Simply import seriallib and do a help to see these help pages.

3. **What else can seriallib do?**

   Using "dict_to_instance", you can use a dictionary to assign into the attributes of an object. Using "instance_to_dict", you can snapshot the given object as a dictionary; The resulting dictionary can be used as a memento for the important state.

   Take a look at the help pages for those two routines. There are also a number of helper routines related to those two routines.

2.16 Serialization

0. **What is serialization?**

   The process of turning a complex data structure that spans memory (such as strings, lists, tables) into a self-encapsulated compact, storable structure. The entire structure is captured in a small piece of memory. This capture can be sent across a socket, saved on disk, saved in shared memory.

   The whole point of serialization is to "save" a complex structure in a form that can be turned back (deserialized) into the original complex data structure at a future date.

   Remember, pickling is another word for serialization in the "Python world".

1. **Why are there so many different kinds of serialization options?**

   Short Answer: PicklingTools needs to be able to support multiple systems.

   You may never use M2k serialization (you may not even not even know what M2k is), but those that do use it find it critical. PicklingTools gives you many different options for flexibility.

   The defaults usually work perfectly well without much work (usually Pickling Protocol 0) but it's good to know your different options (see below).

2. **What are all the different serialization options?**

   From Python (or XMPY): In order, from fastest to slowest

   (a) OCSerialization (need Python C Extension module from this release)

   (b) Python Pickling Protocol 2

   (c) Python Pickling Protocol 0

   (d) Python Dictionaries in textual form (eg., "{'a':1}")

   (e) OpalTables in textual form (eg., "{ a=1 }')

   (f) JSON in textual form

   (g) XML in textual form

   Format (0) is only available as of PicklingTools 1.6.0, and needs to be compiled specially so it can be imported (*import pyocser*). Formats (1)-(3) are native to Python. (4) is supported by doing a simple *import opalfile.py*. Note that Python doesn't support all protocols because it tends to be limited by whatever the "native" Python supports. (5) is built-in (*import json*). (6) is supported by the new tools available in PicklingTools 1.3.1: see the XML tools document for more discussion.

   From C++: In order, from fastest to slowest

(a) OpenContainers serialization (binary)

(b) Python Pickling Protocol 2 (binary)

(c) M2k Serialization (binary)

(d) Python Pickling Protocol 0 (some consider it binary)

(e) Python Dictionaries in textual form (eg., "{'a':1}")

(f) OpalTables in textual form (eg., "{ a=1 }')

(g) JSON in textual form (eg., *{ "a":1 }*)

(h) XML in textual form

From M2k: The M2K OpalPythonDaemon understands all binary serializations above plus the OpalTable ASCII serialization. NumPy is supported as of PicklingTools 1.3.2.

From X-Midas: The MidasServer/Talker/Yeller/Listener support all binary serializations, plus "raw" data (any string). From the raw data, one can easily construct tables using the ValReader and OpalReader classes.

0. **Why would I choose one serialization protocol over another?**

   Short Answer: Depends on what you need. Frequently, the choice of clients and servers seems to dictate what protocol you use.

   If you care about speed, use OpenContainers serialization. It tends to beat most serializations speed by at least 15%. That assumes that all your clients and servers are in C++, or that you can use the Python C Extension module 'pyocser'.

   Any (older) Python clients or servers in the mix tend to dictate using Python Pickling Protocol 2. If you are stuck with an older version of Python, Python Pickling Protocol 0 may be your only choice.

   If you save a lot of small things to disk, text versions of dictionaries are much easier to look at a later date.

   There's no reason you can't mix and match protocols as well: all C++ components can communicate with OpenContainers serialization, and any Python components in the mix can use Python Protocol 2. The MidasTalker/Listener/etc default to using adaptive serialization, where each client sends a small header indicating what type of serialization THAT particular client is using: that makes it easy to combine serializations in one system.

1. **What's all this crazy "ArrayDisposition" stuff when I serialize or unserialized data?**

   Short Answer: ArrayDisposition indicates how you serialize POD array data. The capabilities of your Python typically dictate this.

   POD means "Plain Old Data" and usually refers to simple numeric types like int, float, real, complex (anything that can be bit-blitted). POD Array data is contiguous, homogeneous data. In the different systems:

   (a) OpenContainers Arrays (eg., `#include "ocarray.h" Array a<real_8>;`)

   (b) M2k Vector (eg., `#include "m2vector.h" Vector d(DOUBLE, 10);`)

   (c) Python Numeric Arrays (eg., `import Numeric; a = Numeric.array()`)

   (d) Python Array (eg., `import array; a = array.array()`)

   (e) X-Midas (uses OpenContainers `Array<real_8>`)

   (f) NumPy Arrays (eg., `import numpy; a = numpy.array([1,2,3])`)

   ArrayDisposition answers the question of HOW the POD array data is serialized. There are many different options because not all Pythons support all options.

2. **How do I choose between the ArrayDispositions?**

   All C++ components support all the different ArrayDispositions Array, Numeric, List or NumPy. It's really your Python that decides. If your entire system is in C++, AS_NUMPY is probably your best choice (as it's the most compatible and the fastest).

   The different options for ArrayDisposition are (from fastest to slowest):

   AS_NUMPY or 4: The new de-facto standard for handling arrays within the Python scientific communities is NumPy. It is still an external package which you *may* have to install manually, but it is very common and easy to install.

   AS_PYTHON_ARRAY or 2: Python 2.2, 2.3 and certain versions of 2.4 *DO NOT* support the Python array modules serialization of arrays, so you can't even use this option. Python 2.5 does.

   AS_NUMERIC or 0: XMPY has Numeric built-in, but most versions of Python *DO NOT* come with this built in. You can always install the Numeric module, but it's complicated. From a speed perspective, this is about as fast as AS_PYTHON_ARRAY.

   AS_LIST or 1: ALL versions of Python can serialize POD Data as Python lists. This is the default, but it can be significantly slower than the other two.

3. **What's the other option on serialization called PicklingIssues?**

   If you use Python 2.3 and above, DO NOT WORRY ABOUT THIS! Set this option to ABOVE_PYTHON_2_2 (the default just about everywhere) and don't waste any more brain cells.

   If you use Python version 2.2 _AND_ you use Pickling Protocol 2, you will have to set this to AS_PYTHON_2_2. Unfortunately, the Python 2.2 cPickle module serializes Python Pickling Protocol 2 DIFFERENTLY than later Pythons. Stay away from AS_PYTHON_2_2 unless you absolutely have to.

   Some very important users still use Python 2.2 and that's the only reason we support this.

4. **When I serialized from X to Y, I lost information. Why?**

   Short Answer: Life is complex and not all structures (M2k OpalValue OpenContainers Val, Python values) are 100% compatible.

   If you don't want to want to lose information, talk from like to like. For example:

   - have Python talk to Python using Pickling Protocol 0 or 2.
   - have C++ talk to C++ using OpenContainers Serialization
   - have M2k talk to M2k

   The problem is that M2k, Python, C++, X-Midas all have slightly different philosophies for their structures and serializations. And in some cases, information may be lost. Most information lost is minor (for example: int_4 becomes int_8). Here are a few that might bite you:

   Python Pickling 0 or 2 to C++: Proxies only deserialize as proxies if there are multiple copies

   M2k->Anything else:

   Opalheaders: m2k serializes OpalHeaders EXACTLY like OpalTables, so there is no way to distinguish them

   OpalLink: Links in M2k are poorly done, and rarely used. Links become strings when serialized (TODO: Maybe become Proxies?) If you serialize Proxys to M2k, that information is lost, and it just becomes another copy.

2.17 Misc

0. **Where can I find some examples of X?**

   Short Answer: Look around the baseline.

   The PicklingTools baseline is littered with examples all over the place. The examples usually end in either _test or _ex.

   The C++/Examples directory contains a fairly complex example demonstrating how to build a threaded framework using PicklingTools.

   The C++ directory contains a number of files ..._ex.cc with examples of how to use the Socket clients and servers.

   The C++/opencontainersXXX/tests directory contains code examples (as well as expected outputs) for using the different OpenContainers classes.

   The C++/opencontainersXXX/examples directory contains code examples for using OpenContainers classes.

   The Python directory contains a number of file ..._ex.py with examples of how to use socket clients and servers.

   The Xm/ptoolsXXX/host directory contains X-Midas primitives demonstrating how to write X-Midas primitives using ptools.

CHAPTER

THREE

XML SUPPORT: PICKLING TOOLS 1.7.0

3.1 XML Support: New in PicklingTools 1.3.0

The XML format and Python dictionary are fairly equivalent formats: they both allow recursive, heterogeneous structures for storing data. In many ways, XML is yet another serialization format and the PicklingTools embraces XML as yet another serialization (with some limitations: DTD is not supported (1.3.0 has no support at all, 1.3.1 reads but ignores DTD), although support for namespaces is coming).

> New in PicklingTools 1.4.1 is support for a Python C-Extension module that speeds up conversion from dict to XML by 6-10x and from XML to dict by 60-100x.

If you are using XML as a key-value format, then PicklingTools and XML are essentially equivalent. Consider the tags and content of following simple XML document:

```
<doc>
  <chapter>1</chapter>
  <chapter>2</chapter>
  <appendix>
    <A>3.0</A>
    <B>4.0</B>
  <appendix>
<doc>
```

These are equivalent to the keys and values of the following Python dictionary:

```
>>> d = {
...   'chapter': [1,2],
...   'appendix': {
...       'A': 3.0,
...       'B': 4.0
...     }
... }
```

Python dictionaries tend to be easier to manipulate in Python and C++ (which is why they are the currency of the PicklingTools). but XML does have some advantages over Python Dictionaries:

1. XML is intrinsically ordered, whereas Python dictionaries aren't (but can be with the OrderedDict, see below)
2. XML can represent true documents: this is XML's intrinsic advantage

3.2 XML, Dictionaries and Ordering

Consider the ordering issue: in XML, the order of tags and content is preserved in an XML document as the tags and content always are processed in the order they appear. A Python dictionary, however, doesn't necessarily preserve the order of keys-values. Consider:

```
>>> d = { 'chapter':[1,2], 'appendix':{'A':3.0, 'B':4.0} }
>>> for key,value in d.iteritems() :
...   print key,
# Output:  chapter appendix        OR      appendix chapter
```

In the example above, "chapter appendix" is JUST AS LIKELY as "appendix chapter" as output because the Python dictionary is a *hash table only* and doesn't preserve order.

If insertion order is really a desired feature, The OrderedDict in a new data structure Python 2.7 that captures this (The equivalent in C++ is the OTab and was introduced in PicklingTools 1.2.0). The OrderedDict is just like a Python dictionary (in fact, it inherits from it), but it preserves insertion order just like XML:

```
>>> from collections import OrderedDict # Available as of Python 2.7
>>> od = OrderedDict([
...       ('chapter': [1,2]),            # Long form of the OrderedDict
...       ('appendix', OrderedDict([
...             ('A', 3.0),
...             ('B', 4.0),
...       ])),
... ])
>>> print od['chapter']  # Just like Python dict otherwise
```

Unfortunately, the default output form of OrderedDict is not as clean as the equivalent dictionary (as the OrderedDict is currently represented as a list of tuples), but it is still just as easy to manipulate in Python or C++.

To be clear: *ordered dictionary* means that keys are ordered by insertion order *NOT by sorting order*. You can still look up values via key (i.e., a['key']), but if you ITERATE through items, you iterate through them in the order they were inserted (or in the case of literals, the order they were listed).

Python uses the above long form to represent OrderedDicts, but we are hopeful to upgrade Python to make OrderedDicts a more first-class object. The notation that C++ uses is to use 'o{ }' to represent Ordered Dictionaries. Consider, C++ can use a little letter o to indicate it's an ordered dictionary:

```
o{
  'chapter': [1,2],
  'appendix': o{        // C++ output can choose between the short
      'A': 4.0          // form (this example) or the long form
      'B': 3.0,         // (above) of OrderedDict.  Unfortunately,
   }                    // Python DOES NOT understand this short form.
}
```

From Python, the OrderedDict behaves JUST like the Python dictionary, except for the fact that the insertion order of the keys is preserved. Thus, when you iterate (or print), you see the original insertion order.:

```
# Python:  2.7 and above
>>> from collections import OrderedDict
>>> od = OrderedDict([('chapter',[1,2]),\
...                   ('appendix',OrderedDict([('A',3.0),('B',4.0)]))])
>>> for (key, value) in od.iteritems() :
...    print key                    # For OrderedDict, preserves
...                                 # insertion order
```

```
chapter
appendix

// C++:  PicklingTools 1.2.0 and above
OTab oo = OTab("o{'chapter':[1,2], 'appendix':o{'A':4.0,'B':3.0} }");
for (It ii(od); ii(); ) {
    cout << ii.key() << endl;
}
// chapter
// appendix
```

By using the OrderedDict, the insertion order can be preserved. Many times, however, the insertion order is not relevant: a user may simply care for the absence/presence of keys in the table, in which case, a Python dictionary is fine to use.

In the case ordering is an issue, XML and Python dictionaries can be equivalent: you just have to use the Python OrderedDict instead of the dict: Simply choose the XML_LOAD_USE_OTABS options in the XMLLoader when translating from XML to Python dictionaries.

3.3 XML, Dictionaries, and Documents

When it comes to representing documents, XML is the medium to use; This is XML's raison d'etre: tags interspersed with content and data and attributes, Consider the very simple XML document below:

```
<text>It was the <it color='green'>best</it> of times,
      it was the <it color='red'>worst</it> of times.
</text>
```

There is no real "easy" equivalent of the above in Python dictionaries: you can make up a format which captures all the information, but in the end, it is just a hack over XML:

```
>>> { 'text': o{ 0:'It was the ', 'it':{ 'color='green, '_':'best',
...              2:'of times\n',
...              3:'     it was the ', 'it':'worst', 4:' of times.\n'  } }
```

The above *kind of* works as a translation between XML and Python dictionaries, but breaks down quickly with more complex documents with attributes, nested tags or content interspersed.

If you are manipulating documents like above, don't use Python Dictionaries! Use some other format that is made for documents: XML, LaTeX, and REStructed Text are some alternatives for expressing documents. (In fact, REStructured Text permeates the PicklingTools documentation because it's a simple way to produce documentation in text, PDF and HTML)

For key-value pairs, we can translate directly between XML and Python dictionaries. For documents, Python dictionaries are the wrong choice.

> Sidebar: It can make sense to have a document embedded with the text of a Python dictionary, if you want to keep meta-information around it:
>
> ```
> >>> book = {
> ...   'book': 'A Tale of Two Cities',
> ...   'REStructuedText': 'It was the *best* of times, it was the *worst* of times',
> ...   'XML': '<top>It was the <it>best</it> of time, it was the <it>worst</it> of times</top>',
> ... }
> ```

3.4 Translating between XML and Python Dictionaries

The PicklingTools offers tools to translate between XML and Python dictionaries (both directions) from both C++ and Python. The tools assume one major maxim:

```
Assumption: We are using XML to represent recursive, heterogeneous
key-values data structures.  In this case, we can translate back and forth
between XML and Python dictionaries and not lose information.
```

The interfaces are essentially the same in both Python and C++: there is an XMLDumper which converts from plain dictionaries to XML, and an XMLLoader which converts from XML to plain dictionaries. The Python tools are easier to use than the C++ tools, so we'll discuss those first, but all the interfaces for both are basically the same.

> There was quite a bit of work when doing the XML tools in Python and C++ to make sure the interfaces were the same and the outputs were the same as well. There are two tests called xmldump_test.[py/cc] and xmlload_test.[py/cc] that use exactly the same output to compare against. Although there are tools in Python and C++ to deal with XML separately, the C++ and Python XML tools here have been written completely from scratch so both the Python and C++ can be maintained in parallel. The Python and C++ code in the XMLDumper/XMLLoader is remarkably similar for maintenance purposes: any changes in the Python can easily be propagated to the C++ and vice-versa.

3.5 Python Tools: XMLDumper

To convert from Python dictionaries to XML, use the XMLdumper. The online documentation is quite good:

```
>>> import xmldumper
>>> help(xmldumper)
```

Let's start with a simple example and convert a simple dict to XML:

```
>>> example = { 'a':1, 'b':2.2, 'c':'three' }
>>> from xmldumper import *
>>> import sys     # for sys.stdout
>>> xd = XMLDumper(sys.stdout)       # dump XML to stdout
>>> xd.XMLDumpKeyValue('top', example)
```

The output:

```
<top><a>1</a><b>2.2</b><c>three</c></top>
```

This is a tad unreadable, but sometimes you may want to compress your XML output all together. Most of the time, though, you will probably want to use the *pretty print* version, which indents to show nesting:

```
>>> xd = XMLDumper(sys.stdout, XML_DUMP_PRETTY)
>>> xd.XMLDumpKeyValue('top', example)
```

The output:

```
<top>
  <a>1</a>
  <b>2.2</b>
  <c>three</c>
</top>
```

Notice the top-level container: this is actually an XML requirement that there be exactly one outer tag (in this case, it is called 'top') containing the content. The 'top' tags surround the input table. If we want, we can just output the value:

```
>>> xd.XMLDumpValue(example)
```

The output:

```
<a>1</a>
<b>2.2</b>
<c>three</c>
```

This isn't legal XML by itself, but it can be part of a larger XML document composed piecewise.

There are actually a number of options for the XMLDumper: each option is ored in. For example, if we want pretty-printed XML and strict XML (with the header):

```
>>> xd = XMLDumper(sys.stdout, XML_DUMP_PRETTY | XML_STRICT_HDR )
>>> xd.XMLDumpKeyValue('top', example)
```

The output:

```
<?xml version="1.0" encoding="UTF-8"?>
<top>
    <a>1</a>
    <b>2.2</b>
    <c>three</c>
</top>
```

In this case, we output the XML header which, strictly speaking, is needed to be a standard conforming XML document. Currently, we only support version 1.0 and UTF-8 (namespaces are coming in a future release).

3.6 Attributes and Folding

Attributes are a critical part of any XML document: the XML tools here use a default convention that all keys that start with '_' are to be placed as attributes:

```
>>> a = { "chapter": { '_length': 100, '_pages':200, 'text': 'hello' } }
>>> xd.XMLDumpKeyValue("top", a)
```

The output:

```
<?xml version="1.0" encoding="UTF-8"?>
<top>
    <chapter length="100" pages="200">
        <text>hello</text>
    </chapter>
</top>
```

Notice that the keys '_length' and '_pages' got turned into attributes in the output XML because they started with '_'. This process is called *folding* and allows attributes to be represented simply in key-value structures. If you aren't comfortable with this, consider the following analogy: In UNIX, all files that start with a '.' are treated specially in an 'ls'. In the PicklingTools, all keys that start with '_' are treated specially in XML processing.

You can turn this folding feature off easily enough:

```
>>> xd=XMLDumper(sys.stdout, XML_DUMP_PRETTY | XML_DUMP_PREPEND_KEYS_AS_TAGS)
>>> xd.XMLDumpKeyValue("top", a)
```

The output:

```
<top>
    <chapter>
        <_length>100</_length>
        <_pages>200</_pages>
        <text>hello</text>
    </chapter>
</top>
```

You can also change the prepend_char to be anything you want in the constructor to XMLDumper (see help(xmldumper)).

Many people using XML support the convention that simple data should be in attributes and only structure (lists, dictionaries) should be in tags. The XML_DUMP_SIMPLE_TAGS_AS_ATTRIBUTES option allows you to do just that:

```
>>> xd=XMLDumper(sys.stdout,XML_DUMP_PRETTY|XML_DUMP_SIMPLE_TAGS_AS_ATTRIBUTES)
>>> xd.XMLDumpKeyValue("top", a)
```

The output:

```
<top>
    <chapter length="100" pages="200" text="hello">
    </chapter>
</top>
```

This option allows all simple data to sit in attributes.

All the options for XMLDumper are below. Some of them make more sense when coupled with the XMLLoader (see next section):

```
# Options for dictionaries -> XML
#  If XML attributes are being folded up, then you may
#  want to prepend a special character to distinguish attributes
#  from nested tags: an underscore is the usual default.  If
#  you don't want a prepend char, use XML_DUMP_NO_PREPEND option
XML_PREPEND_CHAR = '_'


# When dumping, by DEFAULT the keys that start with _ become
# attributes (this is called "unfolding").  You may want to keep
# those keys as tags.  Consider:
#
#   { 'top': { '_a':'1', '_b': 2 }}
#
# DEFAULT behavior, this becomes:
#   <top a="1" b="2"></top>       This moves the _names to attributes
#
# But, you may want all _ keys to stay as tags: that's the purpose of this opt
#   <top> <_a>1</_a> <_b>2</b> </top>
XML_DUMP_PREPEND_KEYS_AS_TAGS = 0x100

# Any value that is simple (i.e., contains no nested
```

```
# content) will be placed in the attributes bin:
#  For examples:
#    { 'top': { 'x':'1', 'y': 2 }} ->  <top x="1" y="2"></top>
XML_DUMP_SIMPLE_TAGS_AS_ATTRIBUTES = 0x200

# By default, everything dumps as strings (without quotes), but those things
# that are strings lose their "stringedness", which means
# they can't be "evaled" on the way back in.  This option makes
# Vals that are strings dump with quotes.
XML_DUMP_STRINGS_AS_STRINGS = 0x400

# Like XML_DUMP_STRINGS_AS_STRINGS, but this one ONLY
# dumps strings with quotes if it thinks Eval will return
# something else.  For example in { 's': '123' } : '123' is
# a STRING, not a number.  When evaled with an XMLLoader
# with XML_LOAD_EVAL_CONTENT flag, that will become a number.
XML_DUMP_STRINGS_BEST_GUESS = 0x800

# Show nesting when you dump: like "prettyPrint": basically, it shows
# nesting
XML_DUMP_PRETTY = 0x1000

# Arrays of POD (plain old data: ints, real, complex, etc) can
# dump as huge lists:  By default they just dump with one tag
# and then a list of numbers.  If you set this option, they dump
# as a true XML list (<data>1.0/<data><data>2.0</data> ...)
# which is very expensive, but is easier to use with other
# tools (spreadsheets that support lists, etc.).
XML_DUMP_POD_LIST_AS_XML_LIST = 0x2000

# When dumping an empty tag, what do you want it to be?
# I.e., what is <empty></empty>
# Normally (DEFAULT) this is an empty dictionary 'empty': {}
# If you want that to be empty content, as in an empty string,
# set this option: 'empty': ""
# NOTE: You don't need this option if you are using
# XML_DUMP_STRINGS_AS_STRINGS or XML_DUMP_STRINGS_BEST_GUESS
XML_DUMP_PREFER_EMPTY_STRINGS = 0x4000

# When dumping dictionaries in order, a dict BY DEFAULT prints
# out the keys in sorted/alphabetic order and BY DEFAULT an OrderedDict
# prints out in the OrderedDict order.  The "unnatural" order
# for a dict is to print out in "random" order (but probably slightly
# faster).  The "unnatural" order for an OrderedDict is sorted
# (because normally we use an OrderedDict because we WANTS its
# notion of order)
XML_DUMP_UNNATURAL_ORDER = 0x8000

# Even though illegal XML, allow element names starting with Digits:
# when it does see a starting digit, it turns it into an _digit
# so that it is still legal XML
XML_TAGS_ACCEPTS_DIGITS  = 0x80

# Allows digits as starting XML tags, even though illegal XML.
# This preserves the number as a tag.
XML_DIGITS_AS_TAGS = 0x80000
```

```
# When dumping XML, the default is to NOT have the XML header
# <?xml version="1.0">:  Specifying this option will always make that
# the header always precedes all content
XML_STRICT_HDR = 0x10000
```

3.7 Python and the XMLLoader

The XMLLoader reads XML and converts it to a Python dictionary: this is the inverse operation of the XMLDumper. (Note this assumes the type of XML we are processing is key-value kind of XML, not document XML).

The online docs are always helpful:

```
>>> import xmlloader
>>> help(xmlloader)
```

Let's start with a simple example. In a file named 'example.xml', we will put the following XML:

```
<top>
  <a>1</a>
  <b>2.2</b>
  <c>three</c>
</top>
```

To process this file and turn it into a dictionary:

```
>>> from xmlloader import *
>>> example = file('example.xml', 'r')
>>> xl = StreamXMLLoader(example, 0)  # 0 = All defaults on options
>>> result = xl.expectXML()
>>> print result
{'top': {'a': '1', 'c': 'three', 'b': '2.2'}}
```

We can match the *pretty print* nature of the original XML using the pretty module (which comes with the PicklingTools):

```
>>> from pretty import pretty
>>> pretty(result)
{
    'top':{
        'a':'1',
        'b':'2.2',
        'c':'three'
    }
}
```

From the previous section, we know that all XML has to have exactly one outermost container: in this case, the 'top' key. Many times, when translating from XML to a dictionary, the outer most container is superfluous. There is a simple option to 'drop' the outer most container:

```
>>> example = file('example.xml', 'r')
>>> xl = StreamXMLLoader(example, XML_LOAD_DROP_TOP_LEVEL)
>>> result = xl.expectXML()
>>> pretty(result)
{
```

```
    'a':'1',
    'b':'2.2',
    'c':'three'
}
```

You might notice that the values above are strings and not integers or floats: the default when turning XML into a dict is to just keep whatever string of content was in the XML as, well, a string. Using the eval function built-in to Python, we can turn these strings into their appropriate values. Or, we can use the XML_LOAD_EVAL_CONTENT option (which uses eval, but is a little bit smarter):

```
>>> example = file('example.xml', 'r')
>>> xl = StreamXMLLoader(example, XML_LOAD_DROP_TOP_LEVEL | XML_LOAD_EVAL_CONTENT)
>>> result = xl.expectXML()
>>> pretty(result)
{
    'a':1,
    'b':2.2,
    'c':'three'
}
```

This brings the keys to real values. Internally, the XMLLoader uses eval (which can be a security problem if you XML from untrusted sources), but is a little bit smarter: it only keeps the result of the eval if the entire output would be consumed in a tag. For example <tag>123 #12</tag> should will stay a string using XML_LOAD_EVAL_CONTENT, even though plain eval would return 123. And this is good! We don't want to lose any content!:

```
>>> xml_text = '<tag>123 #12</tag>'
>>> xl = XMLLoader(xml_text, XML_LOAD_EVAL_CONTENT)
>>> xl.expectXML()
{'tag':'123 #12'}
```

Notice the example above shows the difference between XMLLoader and StreamXMLLoader: the former takes input from a string, the latter takes input from a stream. You might also note that in every example we have created a new XMLLoader: if we didn't, the loader would just read from where we left off in the previous input. Rule of thumb: create a new XMLLoader for each XML document to process.

3.8 Attributes and the XMLLoader

There are about 4 ways of dealing with attributes in XML when converting to Python dictionaries.

1. Put them in a special '__attrs__' sub-table: the default
2. Unfold them: use XML_LOAD_UNFOLD_ATTRS
3. Unfold them, but drop the _: use XML_LOAD_NO_PREPEND_CHAR
4. Ignore them: use XML_LOAD_DROP_ALL_ATTRS option

Consider the following XML file (book.xml):

```
<?xml version="1.0" encoding="UTF-8"?>
<top>
    <chapter length="100" pages="200">
       <text>hello</text>
   </chapter>
</top>
```

We will convert this XML to a dict using the default way of handling attributes: stick the attributes in a special table called '__attrs__':

```
>>> book = file('book.xml','r')
>>> xl = StreamXMLLoader(book, XML_LOAD_DROP_TOP_LEVEL)
>>> result = xl.expectXML()
>>> pretty(result)
{
    'chapter':{
        '__attrs__':{
            'length':'100',
            'pages':'200'
        },
        'text':'hello'
    }
}
```

Just like normal XML, we can turn the strings into real values using XML_LOAD_EVAL_CONTENT:

```
>>> book = file('book.xml','r')
>>> xl = StreamXMLLoader(book, XML_LOAD_DROP_TOP_LEVEL | XML_LOAD_EVAL_CONTENT)
>>> result = xl.expectXML()
>>> pretty(result)
{
    'chapter':{
        '__attrs__':{
            'length':100,
            'pages':200
        },
        'text':'hello'
    }
}
```

This method makes it clear which values are attributes and which values are tags:

```
>>> print result['chapter']['__attrs__']['length']
>>>    # attributes of chapter are under chapter/__attrs__ table
```

Another different way to handle attributes (if you don't like the above) is to use the special character '_' in front of tags to indicate those came from the attributes section. With the XML_LOAD_UNFOLD_ATTRS option, the attributes get *unfolded* into the table of interest, as special keys starting with an '_':

```
>>> book = file('book.xml','r')
>>> xl = StreamXMLLoader(book, XML_LOAD_DROP_TOP_LEVEL |
...                            XML_LOAD_EVAL_CONTENT | XML_LOAD_UNFOLD_ATTRS)
>>> result = xl.expectXML()
>>> pretty(result)
{
    'chapter':{
        '_length':100,
        '_pages':200,
        'text':'hello'
    }
}
```

It's still pretty obvious what keys are attributes:

```
>>> print result['chapter']['_length']
>>>     # The attributes of chapter all start with an _
```

Of course, you can change the prepend character in the constructor of XMLLoader (see help(XMLLoader)), or you can get rid of it altogether with XML_LOAD_NO_PREPEND_CHAR:

```
>>> book = file('book.xml','r')
>>> xl = StreamXMLLoader(book, XML_LOAD_DROP_TOP_LEVEL |
...                            XML_LOAD_EVAL_CONTENT | XML_LOAD_UNFOLD_ATTRS | XML_LOAD_NO_PREPEND_CH
>>> result = xl.expectXML()
>>> pretty(result)
{
    'chapter':{
        'length':100,
        'pages':200,
        'text':'hello'
    }
}
```

Finally, you can just drop all your attributes using the XML_LOAD_DROP_ALL_ATTRS:

```
>>> book = file('book.xml','r')
>>> xl = StreamXMLLoader(book, XML_LOAD_DROP_TOP_LEVEL |
...                            XML_LOAD_EVAL_CONTENT | XML_LOAD_DROP_ALL_ATTRS )
>>> result = xl.expectXML()
>>> pretty(result)
{
    'chapter':{
        'text':'hello'
    }
}
```

A list of options is available below (or look in xmlloader.py):

```
###################### OPTIONS for XML -> dictionaries

# ATTRS (attributes on XML nodes) by default becomes
# separate dictionaries in the table with a
# "__attrs__" key.  If you choose to unfold, the attributes
# become keys at the same level, with an underscore.
# (thus "unfolding" the attributes to an outer level).
#
# For example:
#   <book attr1="1" attr2="2>contents</book>
# WITHOUT unfolding  (This is the DEFAULT)
#   { 'book' : "contents",
#     '__attrs__' : {'attr1'="1", "attr2"="2"}
#   }
# WITH unfolding:  (Turning XML_LOAD_UNFOLD_ATTRS on)
#   { 'book' : "contents",
#     '_attr1':"1",
#     '_attr2':"2",
#   }
XML_LOAD_UNFOLD_ATTRS = 0x01
```

```
# When unfolding, choose to either use the XML_PREPEND character '_'
# or no prepend at all.  This only applies if XML_LOAD_UNFOLD_ATTRS is on.
#   <book attr1="1" attr2="2">contents</book>
# becomes
#  { 'book': "content",
#    'attr1':'1',
#    'attr2':'2'
#  }
# Of course, the problem is you can't differentiate TAGS and ATTRIBUTES
# with this option
XML_LOAD_NO_PREPEND_CHAR = 0x02

# If XML attributes are being folded up, then you may
# want to prepend a special character to distinguish attributes
# from nested tags: an underscore is the usual default.  If
# you don't want a prepend char, use XML_LOAD_NO_PREPEND_CHAR option
XML_PREPEND_CHAR = '_'

# Or, you may choose to simply drop all attributes:
# <book a="1">text<book>
#   becomes
# { 'book':'1' }   # Drop ALL attributes
XML_LOAD_DROP_ALL_ATTRS = 0x04

# By default, we use Dictionaries (as we trying to model
# key-value dictionaries).  Can also use ordered dictionaries
# if you really truly care about the order of the keys from
# the XML
XML_LOAD_USE_OTABS = 0x08

# Sometimes, for key-value translation, somethings don't make sense.
# Normally:
#   <top a="1" b="2">content</top>
# .. this will issue a warning that attributes a and b will be dropped
# because this doesn't translate "well" into a key-value substructure.
#   { 'top':'content' }
#
# If you really want the attributes, you can try to keep the content by setting
# the value below (and this will suppress the warning)
#
#  { 'top': { '__attrs__':{'a':1, 'b':2}, '__content__':'content' } }
#
# It's probably better to rethink your key-value structure, but this
# will allow you to move forward and not lose the attributes
XML_LOAD_TRY_TO_KEEP_ATTRIBUTES_WHEN_NOT_TABLES = 0x10

# Drop the top-level key: the XML spec requires a "containing"
# top-level key.  For example: <top><l>1</l><l>2</l></top>
# becomes { 'top':[1,2] }  (and you need the top-level key to get a
# list) when all you really want is the list:  [1,2].  This simply
# drops the "envelope" that contains the real data.
XML_LOAD_DROP_TOP_LEVEL = 0x20

# Converting from XML to Tables results in almost everything
# being strings:  this option allows us to "try" to guess
# what the real type is by doing an Eval on each member:
# Consider: <top> <a>1</a> <b>1.1</b> <c>'string' </top>
# WITHOUT this option (the default) -> {'top': { 'a':'1','b':'1.1','c':'str'}}
```

```
# WITH this option                  -> {'top': { 'a':1, 'b':1.1, 'c':'str' } }
# If the content cannot be evaluated, then content simply says 'as-is'.
# Consider combining this with the XML_DUMP_STRINGS_BEST_GUESS
# if you go back and forth between Tables and XML a lot.
XML_LOAD_EVAL_CONTENT = 0x40

# Even though illegal XML, allow element names starting with Digits:
# when it does see a starting digit, it turns it into an _digit
# so that it is still legal XML
XML_TAGS_ACCEPTS_DIGITS = 0x80

# Allows digits as starting XML tags, even though illegal XML.
# This preserves the number as a tag.
XML_DIGITS_AS_TAGS = 0x80000

# When loading XML, do we require the strict XML header?
# I.e., <?xml version="1.0"?>
# By default, we do not.  If we set this option, we get an error
# thrown if we see XML without a header
XML_STRICT_HDR = 0x10000
```

3.9 Lists and the XMLLoader/XMLDumper

Lists present some interesting challenges when converting back and forth. XML supports lists, but Python lists and XML lists are very different beasts in a few areas.

Consider: by default, multiple entries of the same tag in XML form a list:

```
<top>
   <ch>text1</ch>   # list item 1
   <ch>text2</ch>   # list item 2
</top>
```

There is an easy, obvious way to convert this into Python lists, and it works well:

```
>>> xl = XMLLoader("<top><ch>text1</ch><ch>text2</ch></top>")
>>> xl.expectXML()
{ 'top': { 'ch': ['text1', 'text2'] } }
```

What happens, though, if there is only one tag in the XML list?:

```
<top>
  <ch>text1</ch>   # list item?  plain data?
</top>
```

In the absence of extra information, 'ch' becomes a plain string: in XML, the only way to signal a list is with multiple entries, which don't exist here:

```
>>> xl = XMLLoader("<top><ch>text1</ch></top>", 0)
>>> xl.expectXML()
{'top': {'ch': 'text1' }}
```

In a full XML world, where schemas abound, the solution might be to have a schema to enforce this. Unfortunately, the XML tools here assume we *just* have the information of the table itself: there is no extra information. So, we

have to make due with what XML gives us. In this case, we use the convention of having a special attribute **type__** to indicate that 'ch' is a list:

```
<top>
  <ch type__="list">text1</ch>
</top>
```

Adding this attribute forces the XML translation to keep ch as a list:

```
>>> xl = XMLLoader("<top><ch type__='list'>text1</ch></top>", 0)
>>> xl.expectXML()
{'top': {'ch': ['text1']}}
```

For consistency, you can *always* put it on the first entry to tag an entity as a list:

```
<top>
 <ch type__="list">text1</ch>
 <ch>text2</ch>
</top>
```

And when you translate:

```
>>> xl = XMLLoader("<top><ch type__='list'>text1</ch><ch>text2</ch></top>", 0)
>>> xl.expectXML()
{'top': {'ch': ['text1', 'text2']}}
```

But you don't have to put the **type__** tag on if you have multiple entries: in that case the tools can easily figure out if it's a list or not. It's really only if you have a *single lonesome tag* that you need the special **type__** to force a list. (but ONLY put in on the first entry: if you put it on all the entries, you won't get what you expect).

How do you represent an empty list? Use an empty tag with the **type__** attribute:

```
<top>
  <ch type__="list"/>
</top>
```

Translating, you'll see, yes, 'ch' becomes an empty list:

```
>>> xl = XMLLoader("<top><ch type__='list'/></top>", 0)
>>> xl.expectXML()
{'top': {'ch': []}}
```

There's another way in which XML lists and Python lists differ: all XML lists have to be *named*, whereas Python has the notion of *anonymous* lists and dictionaries. Consider in Python:

```
>>> a = { 'top': [ [1,2,3],['a','b'], {'a':1} ] }
```

The 'top' list contains two anonymous lists and one anonymous dictionary. Basically, because lists can hold anything (including other lists), the content inside the list exists without a name. Python programmers normally think of that as just a['top'][0], a['top'][1], etc. using indices, so they don't care that the inner lists don't have names.

In XML, all tags *HAVE* to have a name. To preserve the XML notion of lists, all entries should have the same name as well (we saw above that XML enforces lists by having repeated tags). This means we can't use 0,1,2, etc. like a Python programmer would. The XML needs a name for the anonymous list: the tools use '**list__**':

```
>>> xd = XMLDumper(sys.stdout, XML_DUMP_PRETTY)
>>> a =  { 'top': [ [1,2,3],['a','b'], {'a':1} ] }
>>> xd.XMLDumpKeyValue('a', a)
<a>
    <top>
        <list__>1</list__>
        <list__>2</list__>
        <list__>3</list__>
    </top>
    <top>
        <list__>a</list__>
        <list__>b</list__>
    </top>
    <top>
        <a>1</a>
    </top>
</a>
```

The convention that the toolset uses is that "**list__**" will be the name XML uses to correspond to the anonymous Python lists. In the case of the dictionary, being inside a list like that makes it "obvious" it's a dictionary, so we don't need any special mechanism for that. To be sure this converts back faithfully:

```
>>> x = """
...   <a>
...       <top>
...           <list__>1</list__>
...           <list__>2</list__>
...           <list__>3</list__>
...       </top>
...       <top>
...           <list__>a</list__>
...           <list__>b</list__>
...       </top>
...       <top>
...           <a>1</a>
...       </top>
...   </a>
... """
>>> xl = XMLLoader(x, 0)
>>> xl.expectXML()
{'a': {'top': [['1', '2', '3'], ['a', 'b'], {'a': '1'}]}}
```

There is one final corner case:

```
>>> a= { 'top': [ {} ] }
```

An empty dictionary inside a list (so the dictionary is anonymous). Any keys in a dict usually offer enough information for the XML tools to figure out that it's a dictionary. In this case, since there are no keys, we need a special key to indicate an anonymous dictionary:

```
>>> xd.XMLDumpKeyValue('a', {'top': [ {} ] } )
<a>
    <top type__="list">
        <dict__>
        </dict__>
    </top>
</a>
```

In fact, you can use '**dict__**' to name an anonymous dictionary, and the tools will do the right thing:

```
>>> x = """
... <a>
...       <top type__="list">
...           <dict__>
...           </dict__>
...       </top>
...   </a>
... """
>>> xl = XMLLoader(x, 0)
>>> xl.expectXML()
{'a': {'top': [{}]}}
```

In fact, adding the '**dict__**' in the anonymous list works just fine, it's just clumsier:

```
>>> x = """
... <a>
...   <top>
...     <dict__><a>1</a></dict__>
...   </top>
...   <top>
...     <dict__/>
...   </top>
... </a>
... """
>>> xl = XMLLoader(x, 0)
>>> xl.expectXML()
{'a': {'top': [{'a': '1'}, {}]}}
```

3.10 Array Disposition

The ArrayDisposition parameter of the XMLLoader and XMLDumper is often misunderstood. Basically, the array disposition tells the XMLLoader/XMLDumper how to deal with arrays of POD: POD stands for Plain Old Data, meaning data like ints, floats, complexes (In hard core C, POD is quick and easy to manipulate). POD arrays are very important for efficient processing of lots of scientific data as they are are stored efficiently as contiguous data in memory.

Python lists *are not* POD arrays: lists have to deal with heterogeneous data (i.e., [1, 2.2, 'three']) and thus aren't as efficient for storing large amounts of data. In a crunch, however, POD arrays can be stored as Python lists.

There are five different array dispositions (number 4 is new to PicklingTools 1.3.0):

0. ARRAYDISPOSITION_AS_NUMERIC : Assume all array data is using the Python Numeric module which keeps arrays of POD:

   ```
   >>> import Numeric
   >>> a = Numeric.array([1,2,3], 'f') # POD array of floats
   ```

   The Python Numeric module may or may not be installed on your platform: many versions of RedHat Linux allow an RPM to be installed. If you use older XMPY (pre 4.0), Numeric is installed by default. Newer XMPY should use NumPy (see next bullet).

   As of 1.5.x series, Numeric is out of maintenance and we tend to prefer NumPy.

1. ARRAYDISPOSITION_AS_NUMPY : Assume all array data is using the Python NumPy module which keeps arrays of POD:

```
>>> import numpy
>>> a = numpy.array([1,2,3], 'f') # POD array of floats
```

The Python numpy module may or may not be installed on your platform: many versions of RedHat Linux allow an RPM to be installed. NumPy really only works with XMPY greater than version 4.0.

2. ARRAYDISPOSITION_AS_LIST: Turn all array POD data into a list. This is the most inefficient way to store POD arrays, but it is the most compatible, as all versions of Python support the Python list:

```
>>> l = [1,2,3]  # Not stored as anything special: uses
>>>              # overhead of lists which is not the most
>>>              # efficient way to store lots of POD
```

3. ARRAYDISPOSITION_AS_ARRAY: Most versions of Python have the array module:

```
>>> import array
>>> a = array.array('f', [1,2,3]) # POD array of floats
>>>                               # NOTE! Python arrays
>>>                               # have different interfaces
>>>                               # than Numeric arrays
```

The Python array module doesn't support complex data, but is available on almost Pythons as a default module.

4. ARRAYDISPOSITION_AS_NUMERIC_WRAPPER: This is new to simple array class which wraps the Python array, but retains the interface of the Numeric array and also supports complex data.

```
>>> from simplearray import SimpleArray as array
>>> a = array([1,2,3], 'D')    # POD array of complex doubles:
>>>                            # works like Numeric arrays but
>>>                            # just a simple Python class
>>>                            # that wraps Python array module
```

In general, the array disposition indicate what the XMLLoader/XMLDumper will try to do with arrays of POD. The XMLLoader and XMLDumper do slightly different things based on the array disposition.

1. In the XMLDumper case, if the Python dictionary contains any POD data (Numeric array, NumPy array, Python array, or Numeric array wrapper), it will dump it as Numeric data UNLESS the array disposition is AS_LIST, in which case it will dump it as a list. Here's an example using the Numeric wrapper array, and how it dumps:

```
>>> from simplearray import SimpleArray as array
>>> e = { 'data': array([1,2,3], 'D'), 'time':'12:00' }
>>>
>>> import sys
>>> from xmldumper import *
>>> xd = XMLDumper(sys.stdout, e, XML_DUMP_PRETTY,
...                ARRAYDISPOSITION_AS_NUMERIC_WRAPPER)
>>> xd.XMLDumpKeyValue('top', e)
```

The output:

```
<top>
    <data arraytype__="D">(1+0j),(2+0j),(3+0j)</data>
    <time>12:00</time>
</top>
```

The POD array dumps a long list of comma-separated values (CSV), with an attribute indicating what the original type of the data was. But, if we use the default array disposition (which is ARRAYDISPOSITION_AS_LIST), that array will be turned into a plain list, and all POD array information will be lost (including the type tag):

```
>>> xd = XMLDumper(sys.stdout, XML_DUMP_PRETTY)
>>>                           # Default is ARRAYDISPOSITION_AS_LIST)
>>> xd.XMLDumpKeyValue('top', e)
```

The output is:

```
<top>
    <data>(1+0j)</data>
    <data>(2+0j)</data>
    <data>(3+0j)</data>
    <time>12:00</time>
</top>
```

Although we lose some information with this array disposition, this format is very compatible with many XML tools (as the notion of XML lists is well understood by those tools).

2. From the XMLLoader's point of view: it will only try to convert POD arrays if it actually encounters POD arrays! If the XML has only plain XML lists, the array disposition doesn't matter:

```
<!-- XMLLoader won't care what the array disposition is for this
     data, because all key values are standard lists: there is
     no special tags or anything indicating otherwise --!>
<top>
    <data>(1+0j)</data>
    <data>(2+0j)</data>
    <data>(3+0j)</data>
    <time>12:00</time>
</top>

<!---- XMLLoader *will care* because it sees the special
       arraytype__ tag below, so it knows that array POD --!>
<top>
    <data arraytype__="D">(1+0j),(2+0j),(3+0j)</data>
    <time>12:00</time>
</top>
```

When loading, the XMLLoader will follow the array disposition: all array POD data will be converted to either a Numeric array, Python array, Numeric wrapper array or plain Python list. As an example for Numeric Wrapper array:

```
>>> x = """
... <top>
...     <data arraytype__="D">(1+0j),(2+0j),(3+0j)</data>
...     <time>12:00</time>
... </top>
... """
>>> from xmlloader import *
>>> xl = XMLLoader(x, 0, ARRAYDISPOSITION_AS_NUMERIC_WRAPPER)
>>> xl.expectXML()
{'top': {'data': array([(1+0j),(2+0j),(3+0j)], 'D'), 'time': '12:00'}}
```

In the above example, the POD array was preserved, because we set the array disposition to load all POD arrays using the Numeric wrapper (simplearray.py). If we don't specify an array disposition, it uses the ARRAYDISPOSITION_AS_LIST as the default:

```
>>> x = """
... <top>
...    <data arraytype__="D">(1+0j),(2+0j),(3+0j)</data>
...    <time>12:00</time>
... </top>
... """
>>> from xmlloader import *
>>> xl = XMLLoader(x, 0) # Default is ARRAYDISPOSITION_AS_LIST
>>> xl.expectXML()
{'top': {'data': [(1+0j), (2+0j), (3+0j)], 'time': '12:00'}}
```

In the above case, the array was converted to a Python list: it preserves the data, but not the original type of the POD data (was the data complex float or complex double?) or the fact that is was a POD array.

The real reason for ArrayDisposition is because Python doesn't have a "good" standard POD array:

1. Numeric arrays are a standard at many places of work, but they aren't installed as standard
2. Python array don't handle complex data, which is a non-starter for some kinds of processing. They also don't do array operations (vector add, multiply, etc.) AND the serialization for Python arrays has changed between 2.6 and 2.7, so they are incompatibile between Pythons.
3. The Numeric wrapper handles complex, but still doesn't have the vector operations
4. Python lists are standard, but are a poor way to store large amounts of data
5. The NumPy package is fairly standard on most modern distributions, but you may still have to go out of your way to get it. In general NumPy is the best choice, as NumPy is in maintenance and fairly standard across multiple platforms.

C++ doesn't have this problem: the Array<T> is the standard way to deal with POD arrays. AS_NUMERIC and AS_PYTHON_ARRAY are handled the same (using the C++ Array<T> class). The AS_LIST is offered as a compatibility option and will convert POD arrays to the equivalent Python List (the C++ Arr()).

In XML, the 'arraytype' tags are the following:

```
s: 1 byte signed char
S: 1 byte unsigned char
i: 2 byte signed char
I: 2 byte unsigned char
l: 4 byte signed char
L: 4 byte unsigned char
x: 4 byte signed char
X: 4 byte unsigned char
f: 4 byte float
d: 8 byte double
F: 8 byte complex (2 4-byte floats)
D: 16 byte complex (2 8-byte doubles)
```

Thus:

```
<s arraytype__='S'>1,2,3</s>
```

Is an array of unsigned 1 byte integers. These typetags correspond to the C++ Val type tags.

When building arrays from Python, look at the Numeric typecodes from Numeric (print Numeric.typecodes) or array (help array).

Note that from a Python dictionary perspective, the arrays are usually printed as Numeric arrays. Their typecodes:

```
>>> import Numeric
>>> print Numeric.typecodes
{'Integer': '1sil', 'UnsignedInteger': 'bwu', 'Float': 'fd', 'Character': 'c', 'Complex': 'FD'}
```

Why didn't we use Numeric typecodes as the standard for the XML type tags? Unfortunately, the Numeric typecodes don't have a 8-byte unsigned integer, and depending on the type of machine (32-bit or 64-bit), the 'l' typecode may be (resp.) a 4-byte integer or an 8-byte integer. The Numeric typecodes are unfortunately inconsistent. The Python array typecodes have similar problems (the tags aren't guaranteed to be a x-bytes, and there are no complex typecodes). The Val typecodes (listed above) are always guaranteed to be exact number of bytes and they support complex data.

3.11 Back and Forth Between XML and Python Dictionaries

The XMLLoader and XMLDumper have been written in such a way to allow you to convert back and forth between XML and Python dictionaries and not lose information (if your XML is key-value XML). The previous sections made it look easy, but there are a lot of places where it be tricky.

1. lists: Lists can be problematic (see above discussion), but the solutions to those problems are outlined in the previous section.
2. floating point numbers: With any floating point data, the number of places you print can be important. The 'default' printing of Python is usually not good enough, so the *pretty* module has gone to great lengths (and the XMLDumper and XMLLoader both import the *pretty* module) to make sure that floating point numbers are handled responsibly: floats are printed with 7 places, doubles are printed with 16 places. Both the C++ and Python versions of pretty should behave exactly the same way.
3. array disposition: When going back and forth between dictionaries and XML, be judicious with the array disposition, or you may lose information. See the previous section for more details on POD data.
4. dropping the top-level: In XML, the top-level document is frequently irrelevant. Converting back and forth between XML and dicts you will probably want to drop the top-level.

There are two tools for converting between XML and Python dictionaries from the command line: they are xml2dict.py and the dict2xml.py (or in the C++ area, xml2dict and dict2xml: the C++ version is significantly faster (60x), but will have to be compiled. As usual, the C++ and Python interfaces are exactly the same). Sample usage:

```
# From a UNIX prompt
% cd /fullpath/to/PicklingTools130/Python
% cat INPUT.XML

<?xml version="1.0" encoding="UTF-8"?>
<root>
  <list__>
    <data arraytype__="d">100.0,200.0</data>
    <data arraytype__="D">(100-100j),(600+1j)</data>
    <here>1</here>
  </list__>
  <list__>7</list__>
  <list__>(1-2j)</list__>
</root>

% python xml2dict.py INPUT.XML
```

```
[
  {
      'data':[
          array([100.0,200.0], 'd'),
          array([(100-100j),(600+1j)], 'D')
      ],
      'here':1
  },
  7,
  (1-2j)
]
```

The options on there were chosen to try to make it easy to go back and forth between the two representations without losing any info.

3.12 C++ and the XMLLoader and XMLDumper

The C++ version is remarkable similar to the Python version: in almost all respects, their behaviors, their interfaces, their options, and even their names should be exactly alike. Even though we said this earlier in this document, it is worth saying again: there has been considerable effort to make the C++ and Python versions of the XMLLoader and XMLDumper to be as close to the same as possible (for ease of maintenance). Thus, the Python and C++ should be almost interchangeable.

The C++ version is significantly faster (60x), but the Python will be easier to use.

Some notable differences: the C++ version deals with Vals instead of Python objects, C++ uses OTabs instead of OrderedDict.

Consider the following simple C++ example for XMLDumper:

```
// Includes needed
#include <iostream>
#include "xmldumper.h"

// C++ code
int main()
{
  Val v = Tab("{'a':1, 'b':2.2, 'c':'three'}");
  XMLDumper xd(std::cout, XML_DUMP_PRETTY | XML_STRICT_HDR);
  xd.XMLDumpKeyValue("top", v);
}

/* Output:
<top>
   <a>1</a>
   <b>2.2</b>
   <c>three</c>
</top>
*/
```

Like the XMLDumper of Python, the options are specified the same and the behavior and interfaces are essentially the same. Instead of using "sys.stdout", we use C++ stream std::cout.

There are multiple examples in the baseline of using this the XMLLoader and XMLDumper: take a look at xml2dict.cc and dict2xml.cc and the Makefiles to see examples of how to compile and use C++.

3.13 Different Types of Keys Of Dictionaries

Most of the discussions above assume the keys of dictionaries are strings. They can be other types:

1. **numbers:** If a key is an int, it will be converted to a tag with "0" around it. Unfortunately, an XML tag of <0> is not legal. By default, the tools will error out saying that keys with only numbers would be illegal XML. There are two options which help mitigate this: XML_TAGS_ACCEPTS_DIGITS and XML_DIGITS_AS_TAGS (yes, they are closely named). The first option (XML_TAGS_ACCEPTS_DIGITS) turns a dict key that starts with digits into an _digit. In other words, `0` would become `_0`. This gives legal XML, but doesn't translate back from XML to dict well. The second option (XML_DIGITS_AS_TAGS) allows "illegal" XML where `0` becomes the tag <0>: on other words, the numberness is preserved, even though the underlying XML is strictly speaking illegal.

   For conversions between Midas 2k OpalTables and XML, we suggest using the XML_DIGITS_AS_TAGS conversion.

2. **tuples and other types:** Right now, having a tuple as a key in a dict will cause "undefined" translations. As The Python will likely error out, and the C++ will try to convert the tuple to a string with varying degrees of success.

In general, we suggest keeping keys as strings to keep the XML translations clean and well-defined.

3.14 Python C-Extension Module: New In PicklingTools 1.4.1

The XMLtools for Python (`xmldumper.py` and `xmlloader.py`) was originally written all in raw Python: this was on purpose it would be easy to include the Python "as-is" without any special build process: *import xmldumper* and you are ready to go. The C++ routines, however, are significantly faster than the Python routines (character based I/O is usually much faster in C/C++ than Python). Take a look at the output of `xmltimingstest.py`:

```
% python xmltimingschecks.py
Time to create big table 0.00489687919617
Time to deepcopy big table 0.0030460357666
...time to convert PyObject to Val ... 0.00101280212402
...time to convert PyObject to Val and back... 0.00146102905273
Time for Python XMLDumper to dump an XML file 0.0433909893036
Time for C XMLDumper to dump an XML file: 0.00465703010559
 -----------
 *Warning: This version of Python doesn't support ast.literal_eval, so XML_LOAD_EVAL_CONTENT can be a
Time for Python Loader to load an XML file 0.61283493042
Time for C Ext Loader to load an XML file 0.00649094581604
```

From these numbers, it's easy to see that dict to XML C++ conversion routines are about an order of magnitude faster than their Python equivalents, and the XML to dict C++ conversion routines are about two orders of magnitude faster than their Python equivalents. Although it is nice to have raw Python solution to convert between dicts and XML (for smaller tables), for any larger tables, the extra speed of C++ may be essential.

The guts of the C++ converters are available in the Python C-Extension module `pyobjconvert`, but the real interface most Python users will use is `cxmltools`.

3.14.1 Building the pyobjconvert Python C-Extension Module

The README describes how to create the pyobjconvert module, which has the XML conversion wrappers:

```
CReadXMLFromStream, CReadXMLFromString, CReadXMLFromFile
CWriteXMLToStream, CWriteXMLToString, CWriteXMLToFile
```

This C extension module for Python increases the speed of the XML to dict conversion by 60x-100x and the dict to XML conversion by 6-10x.

The user probably doesn't want to use the pyobjconvert module directly (as it has a different API than the previous xmltools.py): instead, the user will *import cxmltools* which brings all the appropriate definitions in and the interfaces/names are converted to interfaces that are consistent with the xmldumper.py and xmlloader.py.

HOW TO BUILD

1. Check 'setup.py':

   Make sure it includes the paths to the code in *PicklingTools141/C++* and *PicklingTools141/C++/opencontainers1_7_5/include* (of course, the version numbers may change in later releases).

   By default, this should work, but these are relative paths from the PicklingTools main directory. You may want to move those directories.

2. Once you are sure those are correctly set-up, type:

   ```
   % python setup.py build   # % is the UNIX prompt
   ```

   This starts the build process and builds the C extension module for you. You should see something like this (and notice that it creates three sub-directories):

   ```
   creating build
   creating build/temp.linux-x86_64-2.4
   gcc -pthread -fno-strict-aliasing -DNDEBUG -O2 -g -pipe -Wall -Wp,-D_FORTIFY_SOURCE=2 -fexce
   gcc -pthread -fno-strict-aliasing -DNDEBUG -O2 -g -pipe -Wall -Wp,-D_FORTIFY_SOURCE=2 -fexce
   creating build/lib.linux-x86_64-2.4
   c++ -pthread -shared build/temp.linux-x86_64-2.4/pyobjconvertmodule.o build/temp.linux-x86_6
   ```

3. Underneath PythonCExt should be three subdirs with names "something" like below:

   ```
   build
   build/temp.linux-x86_64-2.4
   build/lib.linux-x86_64-2.4
   ```

   Take a look at the *build/lib.linux-x86_64-2.4* dir: under there should be a `pyobjconvert.so` file. This is the file that contains your library.

   Note these names aren't likely to be the same on your installation. The 'x86' means the machine is a 64-bit installation: yours may be a 32-bit installation and would be 'i686'. The '2-4' means this is for Python 2.4; you are probably using a newer version of Python like 2.6 or 2.7.

   Use the appropriate names for your system.

4. Set your PYTHONPATH so it picks up the .so when you import:

   ```
   % setenv PYTHONPATH "/full/path/to/PicklingTools141/PythonCExt/build/lib.linux-x86_64-2.4"

   % python
   >>> import pyobjconvert  # without PYTHONPATH, it probably won't find your .so
   >>> dir(pyobjconvert)
   ['CReadFromXMLFile', 'CReadFromXMLStream', 'CReadFromXMLString', 'CWriteToXMLFile', 'CWriteT
   ```

5. Try it out! An easy way to see if it works is to run the xmltimingtools.py script which shows the relative times of xmldumper vs. C XMLDumper, etc:

```
% python xmltimingschecks.py
Time to create big table 0.00489687919617
Time to deepcopy big table 0.0030460357666
...time to convert PyObject to Val ... 0.00101280212402
...time to convert PyObject to Val and back... 0.00146102905273
Time for Python XMLDumper to dump an XML file 0.0433909893036
Time for C XMLDumper to dump an XML file: 0.00465703010559
 -----------
 *Warning: This version of Python doesn't support ast.literal_eval, so XML_LOAD_EVAL_CONTENT
Time for Python Loader to load an XML file 0.61283493042
Time for C Ext Loader to load an XML file 0.00649094581604
```

   Note the C XMLDumper is about 10x faster than the Python version and the C XMLLoader is about 100x faster than the Python version

   Surprisingly, converting from PyObject->Val->PyObject (which accomplishes a deep copy) is faster than the Python deepcopy (!)

6. To use the C version of the XML tools, try using `cxmltools`

   The `cxmltools` requires the 'PicklingTools141/Python' to be on the Python path along with the extension module:

```
% setenv PYTHONPATH "${PYTHONPATH}:/full/path/PicklingTools141/Python"
% python
>>> from cxmltools import *   # make sure cxmltools.py is on PYTHONPATH
...                           # as is

>>> d = {'a':1, 'b':2 }
>>>
>>> a = WriteToXMLFile(d, 'top')   # Don't forget the "top"
>>> print a
<?xml version="1.0" encoding="UTF-8"?>
<top>
  <a>1</a>
  <b>2</b>
</top>

>>> res = ReadFromXMLString(a)
>>> print res
{'a': 1, 'b': 2}
```

   In case the `cxmltools` aren't built, you can *import xmltools* and get the same behavior as above, just not as fast!

Note that the `cxmltools` does NOT have everything the `xmltools` has: it only has the simplified wrappers (listed below). These are the same simplified wrappers that the xmldumper/xmlloader have as well. Really, the only thing you *don't* have are the classes (XMLDumper/XMLLoader) that implements the conversions: all the functionality is still available, but through the easier-to-use wrappers.

For converting from XML to dict, the simplified wrappers are:

```
ReadFromXMLFile(filename, options, array_disp, prepend_char);
ReadFromXMLStream(stream, options, array_disp, prepend_char);
ReadXMLString(xml_string, options, array_disp, prepend_char);
```

```
defaults:
     options=XML_STRICT_HDR | XML_LOAD_DROP_TOP_LEVEL | XML_LOAD_EVAL_CONTENT
     array_disp=AS_NUMERIC
     prepend_char=XML_PREPEND_CHAR
```

For converting from dict to XML, the simplified wrappers are:

```
WriteToXMLFile(dict_to_convert, filename, top_level_key, options,
               array_disp, prepend_char)
WriteToXMLStream(dict_to_convert, stream, top_level_key, options,
                 array_disp, prepend_char)
WriteToXMLString(dict_to_convert, top_level_key, options,
                 array_disp, prepend_char)

defaults:
     top_level_key=None   (this should probably always be "top" instead)
     options=XML_DUMP_PRETTY | XML_STRICT_HDR | XML_DUMP_STRINGS_BEST_GUESS
     array_disp=AS_NUMERIC
     prepend_char=XML_PREPEND_CHAR
```

Note that the `WriteToXMLString` returns the string of interest, whereas the WriteToXMLFile/Stream instead both write to the given entity (and return None).

3.15 Conclusion

There are two sets of Python conversion routines: *xmltools.py* and *cxmltools.py*. The former allows a raw Python solution so you can get going immediately. The latter requires more work to build and use correctly, but gives routines that are significantly faster.

The XML Tools described herein have a particular job: allow dictionaries and XML to be used relatively interchangeably. When there is not a clear translation, it's probably worth stepping back and re-evaluating if XML or dict is the better solution. Some of the questionable conversions:

1. Does the XML have content and tags freely interspersed? These don't map well to dicts and maybe the user is using the "document" part of XML heavily.
2. Do dicts have complex keys (like tuples)? These don't map well to XML, and the user is maybe relying on the complex nature of dicts within Python.

If, on the other hand, your XML or dicts are used strictly for key-value type relationships (with a straight-forward use of attributes in XML), then these conversions make sense and may solve the problem for you. These tools have been written with a particular usage in mind and hopefully fit your bill.

3.16 Appendix A:

What features of XML we support:

1. hex escape sequences: Sequences such as * are supported as of 1.3.1
2. DTDs: Usually, a DTD has a <!SOMETHING ... > format: we don't enforce or use the DTD at all, but it can be read—it is simply ignored. 1.3.0 couldn't recognize DTDs and 1.3.1 recognizes but ignores them.
3. namespaces: Neither 1.3.0 nor 1.3.1 recognize namespaces. 1.3.1 can at least parse XML with namespaces (by recognizing the : in names), but there is no support beyond that. A later release will embrace namespaces fully.

4. comments: XML comments can be interspersed in more places: they reduce to nothing. A future release may try to preserve the comment in a Python #
5. version: only 1.0
6. encoding: BUG: We specify UTF-8, but currently only works with ASCII. This won't be a problem unless you use any non-ASCII chars.

CHAPTER

FOUR

C++ CROSS-PROCESS SHARED MEMORY TOOLS

The C++ Cross-Process Shared Memory Tools is new In PicklingTools 1.4.0. There have been some extensive updates as of PicklingTools 1.6.2.

Although the PicklingTools library has had tools to handle cross process shared memory for sometime (since 1.0.0), this release introduces some simple abstractions to help make using shared memory a little bit easier.

4.1 Reminder

First, a quick reminder: C++ Vals *can* be used with shared memory: this is the reason the `Allocator` became an inherent part of the PicklingTools back in 1.0.0. For example, to create a Tab in some shared memory region:

```
char* mem = ... create/attach shared memory across processes ...
StreamingPool *shm=StreamingPool::CreateStreamingPool(mem, bytes, 8);
Val v = Shared(shm, Tab());
v["a"] = "hello";   // Table and keys and values in shared memory
```

The shared memory can be created with `SHMCreate` or connected to with already created memory with `SHMAttach`, but it's tricky to get this right with the basic tools provided by the simple abstractions in "sharedmemory.h":

```
size_t bytes = 1024*1024;
char* mem = SHMCreate("shm_region", bytes);
// Created, but is it available yet?  Do you have to check
// if the entire shared memory has been mapped into the process
// with SHMInitialized, etc.
```

The process for using shared memory is a little clumsy: Who creates and who attaches? The creator is responsible for calling `SHMInitialize` and the user is responsible for called `SHMInitialized` to see if the region is ready, even it already mapped it. But these have to done in the right order, and it's not well documented.

To address these concerns, there are three new abstractions that handle Vals in shared memory a little better.

4.2 SHMMain, ServerSide and ClientSide

There are three new classes that make using shared memory a lot easier: SHMMain, ServerSide and ClientSide. These all come from:

```
#include "shmboot.h"   // Gets defn of SHMMain, ClientSide and ServerSide
```

4.2.1 SHMMain

SHMMain is responsible for creating the shared memory region of the proper size that all ServerSide and ClientSides can use. And that's it. It needs to be called exactly once.

Ideally, the SHMMain gets called exactly once in a startup process that gets called before everyone else: frequently at the start of some main process that has to be started before anything else in the app:

```
#include "shmboot.h"

int main (int argc, char**argv)
{
   int bytes = atoi(argv[1]);

   // Initial set-up code
   bool debug = true;
   SHMMain mem("shm_region", bytes, debug);
   mem.start();  // Actually calls and creates region

   ... start rest of application, fork processes, etc. ...

}
```

Sometimes, if an application is built piecewise, the model described above won't quite work: imagine something similar to a UNIX pipeline where each process may be communicating piecewise to server and a client. In a case like that, the "head" of the pipeline would be where the SHMMain should have to be created.

Once the SHMMain is created (and started), this establishes a shared memory zone or "pool" where queues and Vals can be used. Note that the SHMMain can be created from any process, even if there are no servers or clients in it: its sole purpose is to create and establish the shared memory region that later ServerSides and ClientSides use.

Expert Notes: If you are creating a system where the clients and servers all are all created from one main process, then that makes things easier: you don't *have* to specify a region, as all clients/servers that inherit from the main process can use the region already mapped in the main process: assuming all clients/servers `fork` from the main above and don't `exec`, the shared memory set-up from SHMMain will be inherited (in a process sense, not in a OOP sense) and be in the same address space in all the clients. However, if (like all the examples in the baseline), the clients and servers are completely separate processes, they HAVE to be mapped to the same area: in this case, you want to specify where in memory to map:

```
// On a 32-bit i386 Linux machine, force into a pretty unused area
SHMMain mem("shm_region_32bit", bytes, debug, (void*)0xB0000000);

// On a 64-bit x86_64 Linux machine, force into pretty unused area
SHMMain mem("shm_region_64bit", bytes, debug, (void*)0x700000000000ULL);
```

Although the small standalone programs tend to work without forcing to use a particular region, larger "real" applications tend to need to be forced. Although the ServerSide and ClientSide *can* explicitly set the memory needed, usually you only want to do this in the SHMMain (which sets it up in the right region) and that forces all clients and servers to attach to the right region.

4.2.2 ServerSide

The ServerSide presents the abstraction of a server, although strictly speaking it can also be either side of a communication. The most important thing a ServerSide does is to create the pipe (in shared memory) which will be used for queuing and enqueueing.

To create a pipe, you have to give it a string name (in the example below, `pipename`) and start the ServerSide:

```
// Create a pipe in the shared memory region (string name used in SHMMain)
// The pipe has the given capacity (capacity is in packets).
bool debug = true;
int packet_capacity = 4;
ServerSide server("shm_region",
                  "pipename", packet_capacity. true);

server.start();   // pipe only created when started
CQ& pipe = server.pipe();
```

Once the pipe has been created, it can be enqueued or dequeued from: *Note that queueing and dequeueing is threadsafe*: I.e., you can have multiple processes operating on the queue simultaneously and its state is never inconsistent. Typically, the ServerSide enqueues and the ClientSide dequeues from the given CQ:

```
// ServerSide enqueues
for (int seq=0; ; seq++) {

   // Create the value to enqueue in shared memory
   Val data = Shared(server.pool(), Tab());  // Create a Tab in shared memory
   data["sequence_number"] = seq;

   // Now enqueue
   bool enqueued = false;
   real_8 timeout_in_seconds = 3.2;
   while (!enqueued) {
       enqueued = pipe.enqueue(data, timeout_in_seconds);
       if (!enqueued) {
           cerr << "Failed to enqueue after " << timeout_in_seconds
                << " in seconds ... trying again ..." << endl;
       } else {
           cout << "Enqueued! Going to next packet" << endl;
           break;
       }
   }

}
```

Note that the enqueue has a timeout: in case the queue is full and the Val can't be placed in the queue, it will wait up to `timeout_in_seconds` for some space to open up. If the space opens up in the given time, the data is enqueued and enqueue returns true: the data has now been enqueued. If space doesn't open up in the given time, enqueue returns false. In the example code above, the sender just keeps retrying, but gives a warning every 3.2 seconds warning indicating the queue is full.

Although typically the ServerSide is used as the *send* side only, there's no reason the ServerSide can't dequeue as well (see below for examples of how to dequeue). This ability might be useul if that queue needed to be cleaned as part of a shutdown or restart.

There is another call to enqueue data `pipe.enq(data)` which will block forever; this is simpler and will work but makes it harder to detect error conditions and/or gracefully exit.

4.2.3 ClientSide

The ClientSide is the other end of the pipe: if the ServerSide creates the pipe and writes to it, then the ClientSide waits for the pipe to be created so it can attach and read from the pipe:

```
ClientSide client("shm_region", "pipename", true);
client.start(); // blocks waiting for pipe to be created

CQ& pipe = client.pipe();

// Once pipe is available, we can read from it
real_8 timeout_in_seconds = 5.5;
while (1) {

  // Try to dequeue a single packet
  Val packet;
  bool valid=false;
  while (!valid) {

      valid = pipe.dequeue(timeout_in_seconds, packet);
      if (!valid) {
            cerr << "Couldn't dequeue after " << timeout_in_seconds
                 << " ... trying to dequeue again ... " << endl;
            continue;
      } else {
            cout << "Got packet!" << endl;
            break;
      }

  }
}
```

Note that the client (in order to read from the proper pipe) has to match *both* the name of the shared memory region *and* the pipename the server writes to.

In the above example, if the client can't get something from the pipe immediately, thee queue blocks for up to a few seconds. If something appears on the pipe before the timeout, then that value is dequeued and taken off the pipe and valid becomes true. If after those few seconds the pipe is still empty, valid becomes false and the dequeue fails.

This is basic paradigm: Use SHMMain to create the shared memory region before anything else starts, start and create a sender with a ServerSide and client with ClientSide.

Note that the ClientSide and ServerSide will block until SHMMain is called and creates the memory pool. The ClientSide will block and wait until the the ServerSide creates the pipe. The only way this can become problematic if the shared memory pool has a "unclean" shutdown: if you can't guarantee SHMMain is called before all the clients and servers are created, it's important to make sure the shared memory region has been destroyed, otherwise the client and server may pick up the shared pool from the LAST invocation. To make sure the shared memory is clean, make sure /proc/shm does has been cleaned up and does NOT contain the memory pools; this is discussed a bit more in the *Five Major Headaches* section below later.

When the Val is dequeued from the pipe, it is copied out using the copy constructor of the Val. So, Vals with Proxys simply increase/ decrease reference counts. Prior to PicklingTools 1.6.0, the value was left in the pipe until it was overwritten by later puts to the pipe. As of PicklingTools 1.6.0, when a value is dequeued, the value is copied out and then the Val in the pipe is immediately destructed. Why is this important? If your Val uses a lot of shared memory, letting a copy live in the pipe after a dequeue wastes resources. By immediately destructing after the dequeue copies it out, its resources are immediately released.

4.2.4 Middleside

Many times a server is also a client: it reads from one queue and posts to another queue (like a UNIX pipeline): this role is frequently called a *transformer* in X-Midas or M2k speak. The example below show how to *both* read from a client and post to a server in the same process:

```
ClientSide client("shm_region", "pipe_0", true);
client.start(); // read from this pipe
CQ& input = client.pipe();

ServerSide server("shm_region", "pipe_1", 4, true);
server.start(); // write to this pipe
CQ& output = server.pipe();

while (1) {

    // Read from input pipe
    Val in_packet;
    if (!input.dequeue(.1, in_packet))
        continue; // retry to get input

    // Write to output pipe
    bool enqueued = false;
    while (!enqueued) {
      enqueued = out.enqueue(in_packet, .1);
    }
}
```

In this way, a client and server can be used together. In fact, there is no apriori limit on the number of clients and servers that can be in a single process (limited only by the amount of memory). Multiple clients and/or servers allows programming any type of semantics you want: see the upcoming section.

4.3 Timeouts and External Shutdown Conditions

The SHMMain (and ServerSide and ClientSide) all take the same last two (normally defaulted) arguments: an external break checker and microsecond sleep period:

```
SHMMain (const string& memory_pool_name, size_t bytes,
         bool debug=false, void* forced_addr=0,
         BreakChecker external_break=0, int_8 micro_sleep=int_8(1e5))
```

Some of the internal routines to the SHMMain (and ServerSide and ClientSide) have loops where they are waiting for things to happen (shared memory to come up, first time it's zero filled, wait for that, etc). In case there are problems, there is a systematic way to monitor a global shutdown conition and shutdown an application cleanly. The *external_break* is simply a typedef for a pointer to a function:

```
typedef bool (*BreakChecker)();
```

If the user has somehow set an external condition that signifies "time to shutdown", the user can wrap that check in a routine and pass it to the SHMMain. How often that external condition is checked is the next arguments *micro_sleep*: How many microseconds we sleep before we check (a) coming up conditions and (b) external break. You have to be a little careful with this: if you set it too big, the SHMMain will come up quickly, but there will be a lot of polling (potentially taking too much CPU). If you set it too small, it will take longer to come up and you will might miss

external shutdown conditions. The default, 1e5 in microseconds (thus, .1 seconds) is a fairly reasonable compromise. By default, there is no break checker: it's a hook the user can fill in.

Example: In X-Midas, there is a global flag called *Mc->break*. When an application is coming down (because contrl-C was hit, or an error happened), the *Mc->break* is set for about 5 seconds to tell all primitives to come down. The external break checker for X-Midas would look something like:

```
// Wrap the break checker into a pointer to function we can call
inline bool Mc_break_checker ()
{
  // memory barrier before fetch: may not need ... may be able to comment
  // out if we don't have gcc atomics ...
  __sync_synchronize();

  volatile bool_4* volatile mcb = &Mc->break_;
  volatile bool_4 br = *mcb;
  if (br) {
    return true;
  } else {
    return false;
  }
  // return Mc->break_; // this is volatile, but only this routine needs that!
}
```

It's a simply hook that allows an application to get out if there are problems; without this check, it's possible for an app to "hang" waiting for something to happen. Another possible break checker would be: if a total of n seconds have elapsed and the app still hasn't come up, kill the app.

4.4 Complex Interactions

By default, the CQ class (so-called because it is a Queue that preserves thread-safety using a Condition variable: CQ) has a simple interface: Vals (which exist in shared memory, this is critical!) can either be enqueued or dequeued. Period. There can be any number of processes enqueueing or dequeueing simultaneously, but (this is important) *there is no notion of multicast*: once a Val is dequeued by any reader, it's gone, even if there are twelve readers. If we wanted to support multicast, or any complicated semantics for multiple readers, we have to enforce those ourselves.

For example, if we wanted to have the multicast semantics to only dequeue when all the readers have read the packet (or anything more complicated), we could put all the logic in a simple component with one input and multiple outputs:

```
// Implement a simple multicast with n clients where
// all clients see all data that is dequeued.

// Start up the single input
ClientSide single_input("shm_region", "pipe_input", true);
single_input.start();
CQ& input = single_input.pipe();

// Array of outputs: each output queue will see the input exactly once
Array<ServerSide*> outputs;
for (int ii=0; ii<number_of_outputs; ii++) {
   ServerSide* ssp=new ServerSide("shm_region", "out"+Stringize(ii), 4, true);
   ssp->start();
   outputs.append(ssp);
}

while (1) {
```

```
   // Pull input off the input queue
   Val in_packet;
   if (!input.dequeue(.1, in_packet)) {
      continue;  // nothing available, try again
   }

   // Got input: implement multicast semantics so
   // all outputs see the same input
   for (int ii=0; ii<outputs.length(); ii++) {
      ServerSide* ssp = outputs[ii];
      CQ& out = ssp->pipe();

      // Try to enqueue; blocks until delivered
      out.enq(in_packet);
   }
}
```

With the example above, all outputs see a copy of the input. Since *hopefully* the input packet is just a proxy (where the underlying Tab is shared), this should be a quick and easy dispersal.

Although this abstraction makes it a little harder to implement multiple readers and writers, it gives you full mechanism to implement any policy for multiple readers. For example, in the example above, the multicast can get stuck if one the readers never reads its pipe: the enqueue blocks forever. What if we wanted data to drop if the reader hadn't read it after 5 seconds?:

```
bool enqueued = out.enqueue(in_packet, 5.0);
if (!enqueued) {
   cerr << "Reader " << ii " << is too slow, dropping packet" << endl;
   continue;
}
```

Thus with a simple change, we have implemented a custom multi-cast semantics. For systems where dropped data is okay, we can implement whatever semantics we need with whatever time constraints are relevant. If dropped data is not okay, we can keep retrying to send data, with some messages. We could also implement a nice GUI to show the status of a pipe. Whatever is needed can be built on ServerSide, ClientSide and timeouts.

4.4.1 Address Randomization

Many Linuxes today implement the Address Randomization "feature" to stop hackers from exploiting address space regularity. In other words, the code and data part of your program can be randomly place "anywhere" in main memory. This can be problematic for systems where shared memory needs to be mapped in: What if regions conflict? What part of memory is used? To that end, you may have to force your processes to turn off this feature. This feature can be turned off on a per-process basis easily:

```
% setarch i386 -L -R serverside_ex ...    # 32-bit machine
       or
% setarch x86_64 -L -R serverside_ex ...  # 64-bit machine
```

Frequently, the simple examples work without the above, but the more complex programs may need to do the above. Note that the examples remind you to use setarch when you run them.

If you forget to turn off the feature, then each of the SHMMain, ClientSide and ServerSide will *warn* you with a large message to standard error:

```
% serverside_ex ...   # forgot to run with setarch!

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! WARNING !!!!!!!!!!!!!!!!!!!!!!!!
! It appears that the address randomization feature is still on.
! Your SHMMain/ServerSide/ClientSide is unlikely to work correctly.
! Program will continue running ... but may not run correctly ...
!
! Make sure the process that's gets started up has this feature
! turned off using setarch.  For example:
!  % setarch i386 -L -R startup_program     # 32-bit machine
!           or
!  % setarch x86_64 -L -R startup_program   # 64-bit machine
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

Detecting the absence of the Address Randomization feature is a bit tricky, so you may be using setarch correctly but it still outputs the warning. Lots of false positive warning messages are clumsy and messy, so there is a mechanism to turn off the potential warning. For example, to turn off this message for SHMMain:

```
SHMMain mem(...);
mem.warning(false);  // Turn off warning message above
mem.start();         // Warning message above suppressed
```

Both ClientSide and ServerSide have the same method. By default, the warning is turned on: the idea is that it is better to get a warning message when you are first starting so you can figure out how things work; Once you are comfortable and always turning on the setarch feature, you don't need the warning anymore and can turn it off if needed.

4.4.2 Examples

There are three standalone examples and three X-Midas examples in the PicklingTools baseline demonstrating the shared mem client/server tools.

In the C++ area are serverside_ex.cc, middleside_ex.cc and clientside_ex.cc examples. These three examples should be run together *on the same machine*:

```
# In one shell prompt
% setarch i386 -L -R serverside_ex mempool 1000000 pipe1 4
```

The `serverside_ex` creates the shared memory region (called `mempool`) of one million bytes. The pipe the server will write to is `pipe1` and it will have a capacity of four packets:

```
# In another shell prompt
% setarch i386 -L -R middleside_ex mempool pipe1 pipe2 4
```

The `middleside_ex` waits for the shared memory region (called `mempool`) to be available. Once it is, the middleside reads from `pipe1` and writes to `pipe2`; `pipe2` also has a capacity of four packets:

```
# In yet another shell prompt
% setarch i386 -L -R clientside_ex mempool pipe2
```

The `clientside_ex` waits for the memory pool, waits from `pipe2` to be created, then reads from `pipe2`.

This will cause the server to talk the middle and the middle will talk to the final client. Note that SHMMain is created *exactly once* by serverside_ex.

Also note that the X-Midas primitives can talk to the standalone executables.

4.4.3 Five Biggest Headaches

The five biggest problems getting shared memory across platforms to work are:

1. Memory size: Most Linuxes are constrained by how much shared memory they can allocate. If the allocation is too big, then Linux will simply fail. Carefully try larger and larger sizes of shared memory from SHMMain to make sure that your box can legally create and use that much shared memory.

2. Using Vals in shared memory. Once a Val is created in shared memory, most updates on the table will cause the new keys/values to be created in the same shared memory. If you wish to enqueue a table or Array, make *sure* the entire table is in shared memory:

   ```
   // Okay
   Val data = Shared(shm, Tab());  // created in shared memory
   pipe.enqueue(data);             // Okay, because data in shared memory

   // !!!! NOT OKAY!!!
   Val data2 = Tab();  // data2 NOT in shared memory!!
   pipe.enqueue(data2);  // Will seg fault
   ```

   There is a routine called `IsSHM` from #include "checkshm.h" which allows the user to check and see if a Val is completely contained in shared memory. In debug mode, this is a very useful tool; before data is enqueued on a shared memory pipe, the table can be checked to make sure all of its parts are in shared memory.

3. Left-Over files. By default, the shared memory regions have an file in the /dev/shm area. There is good news and bad news about this. If you do not destruct SHMMain (in the C++ sense), then the region persists. This can be good because your queues can persist across time: assuming a client connect and disconnects frequently, this can just work as the new connection will simply pick up where the last one left off.

   This can be horrible if the queues get into an inconsistent state: then every client will simply break.

   It can be useful to completely clean /dev/shm of your shared memory. By default, when you create shared memory region like the "shm_region" in all the examples above, two files are created:

   ```
   /dev/shm/shm_region_boot
   /dev/shm/shm_region
   ```

   The boot region is very small and used only to pass global information to each client (where memory should be mapped, the size of the memory, where to find the pipes, etc). The boot is first mapped in anywhere in memory. The data in the boot informs where to map the main section: the main section contains the giant memory pool.

   To make sure your application starts in a fresh start, it's probably worth removing `/dev/shm/shm_region_boot` and `/dev/shm/shm_region` before starting, or restarting your application. Note that *by default*, SHMMain will completely clean-up for you when you create and start the SHMMain component.

4. Forgetting About Address Randomization

   Frustratingly, sometimes things will work with the address randomization on, then one small change will cause everything to stop working. It's best put this as part of a start-up script where all processes will "inherit" this attribute:

   ```
   % setarch i386 -L -R startup_process
   ```

   See the *Address Randomization* section above.

5. Holding On Too Long or Letting Go Too Soon

   What does `Shared(client.pool(), Tab())` return? A proxy to a Tab in shared memory protected by a process-safe reference count. Remember, that every copy of the proxy increments the reference count:

```
// Serverside
{
  Val v = Shared(shm, Tab());  // ref count at 1
  pipe.enq(v);                 // ref count at 2
} // Val destructed, ref count at 1


// Clientside
{
  Val off = pipe.deq();  // Ref count at 1
}  // Val destructed, ref count at 0, memory reclaimed
```

Thus, once you have enqueued your shared Tab, you can let go of the packet and let the client dequeue it. Once the client dequeues it and is done, it will be reclaimed by the pool.

Be careful not to keep your Tab alive after you have enqueued it, or that could become a memory growth.

4.5 Tips and Tricks

Here's a collection of tricks and tips that can make your life easier.

1. You can always ask any of the SHMMain, ClientSide or ServerSide for it's allocator:

   ```
   ServerSide server_side(...);
   Allocator *a = server_side->pool(); // Get the allocator
   ```

   That way you can be assured to get the right allocator.

2. You can ask a Val for it's allocator:

   ```
   Val v = ...;
   Allocator *a = v.allocator();
   ```

   If the value was from a SharedPool, it will give you access to that. If this Val was using the "standard" heap, a will be NULL.

3. If you have an allocator, you can call allocate to get some memory from it, and deallocate to free it. If you are just using Vals/Tabs/etc. with it, you should be able to use the Shared or Protected stubs:

   ```
   Val v = Shared(a, Tab());  // Give me a Tab from allocator a
   ```

Just a few tricks and tips. The SHMQ X-Midas option tree shows one way to use these routines to implement a shared memory system.

4.6 Conclusion

With three simple abstractions, tables across shared memory can be much easier to manipulate.

4.6.1 Known Bugs:

There are still some known bugs: we prefer to release early so as to get feedback, even if there are some known issues.

1. The `int_un` and `int_n` DO NOT work with shared memory. A future version will fix this. **FIXED: in PicklingTools 1.4.1**
2. Should a Val initialized from a Proxy copy the allocator? This makes the `IsSHM` check without an explicit work-around for Proxies.

CHAPTER

FIVE

JAVA SUPPORT FOR PICKLINGTOOLS: NEW AS OF PICKLINGTOOLS 1.5.0

As of PicklingTools 1.5.0, there is support for Java: this means that Java can handle Python dictionaries and pickle them into files or sockets.

1. What does it means that PicklingTools 1.5.0 supports Java?

   Short Answer: In general, Java is supported like C++, but since the Java baseline isn't quite as fleshed out as the C++ baseline, there are some missing features.

   Long Answer.

   (a) Java can talk to MidasServers using a Java MidasTalker (i.e., we can have a Java client talking to any C++/Python MidasServer or M2k OpalPythonDaemon). There is currently no support for Java MidasServers, Midasyellers or MidasListeners, but that will be available in a future release. MidasTalker simple example:

   ```
   MidasTalker mt = new MidasTalker("localhost", 8888);
   mt.open();
   mt.send(new Tab("{'a':1, 'b':2.2, 'c':'three'}");
   Object o = mt.recv(5.0);  // block upto 5 seconds waiting
   Tab result = (Tab)o;
   result.prettyPrint(System.out);  // print as Python dict
   ```

   (b) Java can read Python dictionaries from files or sockets. See below for examples with textual and binary (pickled) data.

   (c) Java can read/write pickled data.

   (d) Java can read/write textual Python dictionaries. Example:

   ```
   import pythonesque.*;

   // Create text dictionary from a Python compatible string
   Tab t = new Tab("{'a':1, 'b':2.2, 'c':'three', 'arr':[1,2.2,'t']}");

   // Write a textual Python dict to a file
   Ptools.WritePythonTextFile("python_dict.txt", t);

   // Read a text Python dictionary from a file
   Object o = Ptools.ReadPythonTextFile("python_dict.txt");
   Tab res = (Tab)o;

   // Write a pretty printed textual Python dict to output
   ```

```
res.prettyPrint(System.out);

// Write python dict to output (no extra spaces)
System.out.println(res);
```

(e) Java can manipulate Python-esque dict and lists. Example:

```
Tab t = new Tab("{'a':1, 'nest':{'b':2.2, 'c':'three'}");
t.put("newkey", 16);          // Insert new key at top level
t.put("nest", "newer", 2.2);  // Cascading insert into nested dict

Arr a = new Arr("[10000, 2.2, 'three', [1,2,3]]");
int ii = (Integer)a.get(1);     // get int
int ii = (Integer)a.get(3, 0);  // get nested int
```

2. Where does the Pickling support come from?

   The Pyro project released an OpenSource pickling package which PicklingTools 1.5.0 has embraced and is working with. The Pyro project has a similar license as the PicklingTools.

3. How do you handle Python types in Java?

   Directly from the Pyro documentation:

```
Pyrolite does the following type mappings:

PYTHON    ---->     JAVA
------              ----
None                null
bool                boolean
int                 int
long                long or BigInteger  (depending on size)
string              String
unicode             String
complex             net.razorvine.pickle.objects.ComplexNumber
datetime.date       java.util.Calendar
datetime.datetime   java.util.Calendar
datetime.time       java.util.Calendar
datetime.timedelta  net.razorvine.pickle.objects.TimeDelta
float               double   (float isn't used)
array.array         array of appropriate primitive type (char, int, short, long, float, double)
list                java.util.List<Object>
tuple               Object[]
set                 java.util.Set
dict                java.util.Map
bytes               byte[]
bytearray           byte[]
```

   The unpickler simply returns an Object. Because Java is a statically typed language you will have to cast that to the appropriate type. Refer to this table to see what you can expect to receive.:

```
JAVA     ---->      PYTHON
-----               ------
null                None
boolean             bool
byte                int
char                str/unicode (length 1)
String              str/unicode
```

```
double              float
float               float
int                 int
short               int
BigDecimal          decimal
BigInteger          long
any array           array if elements are primitive type (else tuple)
Object[]            tuple
byte[]              bytearray
java.util.Date      datetime.datetime
java.util.Calendar  datetime.datetime
Enum                the enum value as string
java.util.Set       set
Map, Hashtable      dict
Vector, Collection  list
Serializable        treated as a JavaBean, see below.
JavaBean            dict of the bean's public properties + __class__ for the bean's type.
```

4. Does the Java support Python Object, dict, and list?

   Short Answer: Yes and No.

   PicklingTools wants to make Java programmers as comfortable with native Java types and tools as much as possible, so choices have been made that tend to fit the Java model. See below.

5. Is there an equivalent for C++ Val?

   Short Answer: No, we use the Java Object instead.

   There is no equivalent Val in Java: the Val in C++ was required because C++ doesn't have a heterogeneous, dynamic container type (as C++ is very statically typed, and the library predates the C++ `any` class). Java *already* has a heterogeneous, dynamic container: the `Object`. Most types inherit from `Object`, and can be cast down from `Object` to the appropriate type easily.

6. Is there a Java equivalent for a Python dict?

   Yes.

   By default, a Java Tab "is-a" `HashMap<Object, Object>`. So, Tab inherits all the interface from HashMap. BUT: Tab extends the interface significantly to make the Java Tab feel much more like the Python dict.

7. What does Tab add to `HashMap<String, Object>`?

   Five things.

   (a) Less typing. Seriously, which would you rather type?:

   ```
   HashMap<String, Object> m = new HashMap<String, Object>();
   Tab t = new Tab();
   ```

   (b) The constructor supports creating a literal from a string, i.e.,:

   ```
   Tab t = new Tab("{'a':1, 'somefloat':2.2, 'nest':{'oo':'str'} }");
   // Constructs same table as above
   ```

   This is exactly the syntax of dictionary literals in Python, so these tables can be cut-and-pasted between Python and Java.

   (c) Supports pretty print that looks EXACTLY like Python dictionaries so you can cut and paste Python dicts between Python and Java:

```
t.prettyPrint(System.out);
// overrides toString to System.out.println(t) also works
```

(d) Support for cascading lookup:

```
Tab t = new Tab("{'a':1, 'somefloat':2.2, 'nest':{'oo':'str'} }");
String s = (String)t.get("nest", "oo");
```

(e) Support for cascading insertion:

```
Tab t = new Tab("{'nest':{ 'nest2':{} }");
t.put("nest", "nest2", "value");
```

8. How do Java Tab interactions compare with Python dicts?

   They are similar in many ways. Newer features of Java (such as boxing and unboxing) make it a little easier to get stuff in and out of Tabs. For example, In Python:

```
>>> a = { 'a': 1 }
>>> a['b'] = 17.7
```

   The equivalent in Java:

```
Tab a = new Tab("{'a':1}");
a.put("b", 17.7);              // Because of boxing, don't have to type
                               // a.put("b", new Double(17.7));
```

   Overloading and variable number of arguments make it easier to represent cascading lookups and inserts. In Python:

```
>>> t = {'nest':{'a':1} }
>>> lookup =  t['nest']['a']     # Cascading Lookup
>>> print lookup                 # output: 1

>>> t['nest']['a'] = 777         # Cascading insert
>>> print t                      # output: { 'nest': {'a': 777} }
```

   In Java:

```
Tab t = new Tab("{'nest': {'a':1}}");
Object lookup = t.get("nest", "a");   // Cascading lookup
System.out.println(lookup);           // output: 1

t.put("nest", "a", 777);              // Cascading insertin
System.out.println(t);                // output: { 'nest': {'a': 777 } }
```

   Note that Java uses double quotes ("") for strings and Python can use both single and double quotes ("", '') for strings. In general, it's easier to type single quotes inside the literal string of the Tab.

9. How do we get types out of Java?

   Use the type-casting of Java. For example, to get a nested Tab from another Tab:

```
Tab t = new Tab("{'a':1, 'b':2.2, 'c':'three', 'nest':{} }");
Object o = t.get("nest");
Tab nest = (Tab)o;
```

Or, a little less typing:

```
Tab t = new Tab("{'a':1, 'b':2.2, 'c':'three', 'nest':{} }");
Tab nest = (Tab)t.get("nest");
```

With unboxing, getting POD types like int and floats out isn't quite perfect: they have to go through the Object version. For example:

```
int i = (int)t.get("a");      // SYNTAX ERROR: too much unboxing to do

int i = (Integer)t.get("a");  // Okay, unboxing helps
```

So, primitive types can work, but do require a cast. (C++ gets around this extra cast because C++ supports a language feature called user-defined conversion). This isn't ideal, but it is a reasonably small amount of typing.

10. Is there an equivalent for a Python list?

    Yes.

    By default, an Arr "is-a" `ArrayList<Object>`. So, Arr inherits all the interface from ArrayList. BUT: Arr extends the interface significantly to make the Java Arr feel much more like the Python list.

11. What does Arr add to `ArrayList<Object>`?

    Five Things. The Arr adds some features to the ArrayList that make it easier to manipulate. The Arr "is-a" ArrayList<Object> so it still supports all the same methods, as well as:

    (a) Less typing:

    ```
    ArrayList<Object> a = new ArrayList<Object>();
    Arr a = new Arr();
    ```

    (b) Support for string literals, where string literals can be cut-and-paste directly between Java and Python:

    ```
    Arr a = new Arr("[1, 2.2, 'three', [44]]");
    ```

    (c) Supports pretty print that looks EXACTLY like Python dictionaries so you can cut and paste Python dicts between Python and Java:

    ```
    Arr a = new Arr("[1, 2.2, 'three', [44]]");
    a.prettyPrint(System.out);
    // overrides toString to System.out.println(a) also works
    ```

    (d) Supports cascading lookups in nested Arr/Tab:

    ```
    a.get(3, 0);
    //  --> gets 44 from nested array
    ```

    (e) Supports cascading inserts into nested Arr/Tab:

    ```
    a.put(3, 0, 777);
    // --> results in [1, 2.2, 'three', [777]]
    ```

12. How do Java Arr interactions compare with Python lists?

    Like Tabs, boxing and unboxing make it easier to deal with heterogeneous types in Java. Consider a Python list:

    ```
    >>> a = [1, 2.2, 'three']
    >>> a.append(6)
    ```

Similarly in Java:

```
Arr a = new Arr("[1, 2.2, 'three']");
a.add(6);         // Because of boxing, don't have to do
                  //    a.add(new Integer(6));
```

Overloading get and put, along with Java supporting a variable number of arguments (as well as boxing/unboxing) make cascading gets and puts (that Python deals with so easily) easy to deal with. Consider Python:

```
>>> aa = [ 5, 6, [10, 11] ]
>>> lookup =  aa[2][0]      # Cascading lookup: 10
>>> print lookup

>>> aa[2][0] = 100      # Cascading inserts: [5,6, [100,11]
>>> print aa
```

In Java, the equivalent is:

```
Arr aa = new Arr("[5,6, [10,11]]");
Object lookup = aa.get(2,0);    // Cascading lookup
System.out.println(lookup);

aa.put(2, 0, 100);              // Cascading insert: [5,6, [100,11]
System.out.println(aa);
```

13. Is there support for Tuples and OTabs?

    Tuples: yes, limited. This will be expanded in a later version. OTab. No. May be added later.

14. What's missing?

    There's still some development to do.

    (a) No MidasServer. This is currently no MidasServer like in C++/Python there is only the client.

    (b) No Numeric or NumPy support. The original Pyrolite dist. only supported Python Arrays.

    (c) No "shared" data structures. The pytolite dist. did not implement the get/put feature of pickling which allows dictionaries to be shared (i.e., only deep copies are made).

    (d) No Unicode support. The C++/Python tools were developed to work with Python 2.x series, where strings where ASCII. The unicode support for PTOOLS is non-existant currently: this is a PTOOLS problem, not a Java problem.

    (e) No byte array support. It becomnes unicode, which (4) can't handle.

    (f) Can't read textual arrays. Cannot currently read arrays from strings or files.

    (g) No MidasYeller or MidasListener.

    These are the major missing pieces, which we will flesh out as people need them. This is a first release to get the basic functionality out the door so we can get feedback.

CHAPTER

SIX

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*