Frequently Asked Questions for PicklingTools 1.5.3

General Questions

1. **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.

2. **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.

3. **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 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 (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?**

   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:

1. 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++.
    a. If readability is a concern, files can contain textual (ASCII) Python Dictionaries
    b. If speed and size are concerns, files can contain binary serialized (pickled) Python Dictionaries
2. 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.

Python

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

   Historically, versions 2.1.x to 2.4.x have been tested extensively. More recently, 2.5 - 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.

2. **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
   ```

3. **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

4. **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.

5. **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.)

6. **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.

C++

1. **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).

2. **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("{ }");
```

3. **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).

4. **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.

5. **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.

6. **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
```

7. **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
```

8. **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;
   }
   ```

9. **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
```

10. **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
```

11. **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);
```

12. **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]
```

13. **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;
```

14. **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.

15. **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.

16. **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.

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, 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:

1. In a system with no threads (EASY)
2. In a system with threads where you have to worry about coordinating data sharing between threads (MEDIUM)
3. 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.

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.

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

1. **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 = "100000000000000000000000";   // 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("100000000000000000000000"); // Pre 1.4.1
```

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

```
int_n c1 = "100000000000000000000";  // In 1.4.1 and beyond
string s("24736578924305243523475789234");
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("100000000000000000000000") + 1;
cout << ii << endl;
```

   ...but gives the wrong answer!

```
2003764205206896641   // Should be 100000000000000000000001
```

   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("100000000000000000000000") + int_un(1);
cout << ii << endl;
```

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

```
int_un ii = int_un("100000000000000000000000") + 1;
cout << ii << endl;
```

   This gives what's expected:

```
100000000000000000000001
```

   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.

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

1. **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.

XML Support: New in PicklingTools 1.3.0

1. **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).

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.

Conformance or Validation Support: New in PicklingTools 1.3.3

1. **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)
```

Java Support: New as of PicklingTools 1.5.1

1. **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

2. **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.

M2k

1. **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.

2. **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).

X-Midas

1. **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

   1. XMPY scripts can use the Python MidasTalker, MidasServer, MidasYeller and MidasListener easily (found in python subdir of PTOOLS)
   2. 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.

2. **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.

SerialLib

1. **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.

2. **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.

3. **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.

4. **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.

Serialization

1. **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".

2. **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).

3. **What are all the different serialization options?**

   From Python (or XMPY): In order, from fastest to slowest

      1. Python Pickling Protocol 2
      2. Python Pickling Protocol 0

3. Python Dictionaries in textual form (eg., "{'a':1}")
4. OpalTables in textual form (eg., "{ a=1 }')
5. JSON in textual form
6. XML in textual form

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

1. OpenContainers serialization (binary)
2. Python Pickling Protocol 2 (binary)
3. M2k Serialization (binary)
4. Python Pickling Protocol 0 (some consider it binary)
5. Python Dictionaries in textual form (eg., "{'a':1}")
6. OpalTables in textual form (eg., "{ a=1 }')
7. JSON in textual form (eg., *{ "a":1 }*)
8. 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.

4. **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++.

Any 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.

5. **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.

6. **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.

7. **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.

8. **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.

Misc

1. **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.