# The PicklingTools 1.4.1 User's Guide

## 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. 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++ Primtives, 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.*

**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++ 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.

# 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, heterogenous 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.
    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 interepreters 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.).

# Media

There are two major ways to communicate with the PicklingTools: via files or over sockets from Python and C++. *NOTE: Whenever we say "Python and C++", we mean the whole range of systems supported directly: XMPY, X-Midas C++ primtives, 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++. See later sections for example of this within C++, Python, etc.

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

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 componet, but is still backwards compatible with OpalSocketMsg.

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

# 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++. 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 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. ]

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 distrubution, in the "Pickling API" document.

# The Python Experience

Python is the easist 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 unnatrual 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?]

## 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
}
```

# 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 whenver 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 behaviour 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 communuicate 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 (Unreliable Data Packet) which requires you to 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.

## 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 priny: 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
    }
}
```

## 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 technque 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.

## 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).

# 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.

## 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 functionaility 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 actualy 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 serialing 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 you 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.send(a)    # Send Numeric.array 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.

## ArrayDisposition

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 currenty 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.

By default, we use 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.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 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 above doesn't work.

# 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

## 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 undertand 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.

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 printPrint 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.

# 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 typedef-ed to one of these)
- REAL types: real_4, real_8
- COMPLEX types: complex_8, complex_16
- 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

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
'F' 'D' : complex_8, complex_16
'a'     : string (like 'a' in ASCII)
'n'     : array  (like n elements in array)
't'     : Tab
'o'     : OTab
'u'     : Tup
'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 with the ability to parse string literals.

## 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 typesafe 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;
```

The Val class also contains an out-converter (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'
```

## 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);     // Inital 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 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.

# 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 intitally 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, they 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.

## 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.

## 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:*

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
```

## 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
```

## 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
10000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

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 the same performance as the Python arbitrary ints.

## 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 arguments like the Tab, OTab and Arr:

```
// OTab, Tab, Arr all evaulate 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.

## 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.

## 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.

## 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.

## 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  }");

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

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

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 enumration 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);

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

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

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

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

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

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

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

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

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

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

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

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!:

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

// Recursively checks that 'b' of instance matches 'b' of prototype,
// and that 'nested' of instance matches 'nested' of prototype
Conforms(instance, prototype) -> true
```

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 always be false so that any conformance checks are fast. Only in 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 necesssarily 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 sumary, 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 stucture is preserved.

# 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.

# 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.

# 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.

# 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:

a. Is a new type a POD type? (And thus bit-blittable?)

b. How does a new type compare with other types of Vals?

c. How does a new type serialize?

d. How does a new type Strinigize itself?

e. How does a new type convert to other types?

f. What happens if we have virtual functions in a type?

g. What letter do I use for a tag? Has it already been used?

Certainally 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:

a. A New Constructor to create BiologicalSimulations from Vals: *BiologicalSimulation (const Val& v)*

b. 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");
```

Certainally, 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.