Python wrappers for d0me

[Python Module : d0me]

1 Introduction

This is a collection of python wrappers for the d0me messaging library. Familiarity with that library is assumed.
For basic usage, see the classes Processor, Client_Connection, and Server.
New message types can now be added, but only in C++ (not Python), and you must supply the appropriate wrappers.

2 Message classes

2.1 IPC_Message

2.1.1 class IPC_Message

```
[ created from class Py_IPC_Message ]

The IPC message, abstract base class for all IPC Messages.
Note that you cannot at present override any
of these methods from python such that they can be called from C++.
```

self.MSG_ID

```
[ Constant: int  ]
The message ID for this type.
```

IPC_Message()

```
[ Constructor: returns Py_IPC_Message * ]
Constructor.
Since this is an abstract class, it isn't useful
to call this from Python.
```

del this

```
[ Destructor: returns void  ]
Destructor.
```

msg_ID()

```
[ Member : returns int  ]
Return the message ID of this instance.
```

length()

```
[ Member : returns size_t  ]
Return the length of the data.
```

copy_in(data,data_len)

```
[ Member : returns size_t  ]
Copy the input buffer DATA to this object, decoding the message
(after transmission).
nb. Renamed this from ‘in’, since that’s a reserved word in python.
The SWIG pointer library can be used to construct buffers to be
passed to this routine.
```

copy_out(data,data_len)

```
[ Member : returns size_t  ]
Will copy the data for this object to the output buffer,
encoding the message (before transmission).
nb. Renamed this from ‘out’ to correspond with ‘copy_in’ above.
The SWIG pointer library can be used to construct buffers to be
passed to this routine.
```

dump()

```
[ Member : returns void  ]
Dump to stdout.
```

base_msg_ID()

```
[ Member : returns int  ]
Will return a MSG_ID for a base class, to which the message can be
upcasted when the message is been dispatched, 0 is no upcast
(upcast to MSG_ID 0 (an IPC_Message) is already supported).
An example of use is in Status_Message.h.
Note: currently only upcast to one base is supported.
```

connection()

```
[ Member : returns Connection * ]
Return the Connection which generated this message, or None
if there wasn't one.
Note: this will return a new python object which is borrowing
a reference to the underlying C++ object.  This has the following
consequences:
 --- This object should not be kept around after the original
     connection object is deleted.
 --- These objects won't compare equal to other connection
     objects referencing the same C++ object.  For now, you'll
     have to compare the SWIG pointers directly.
```

2.2 String_Message

2.2.1 class String_Message

`[ created from class Py_String_Message ]`

A message which holds a string. Derives from IPC_Message.

self.MSG_ID

`[ Constant: int ]`

The message ID for this type.

String_Message(DEFAULT_EMPTY_STRING)

`[ Constructor: returns Py_String_Message * ]`

Constructor. The argument is a string with which to initialize the new message.

del this

`[ Destructor: returns void ]`

Destructor.

str()

    [Member : returns std::string]
    Get back the message contents as a string.

2.3 Opaque_Message

2.3.1 class Opaque_Message

```
[ created from class Py_Opaque_Message ]

A message which holds some opaque blob of data.
Derives from IPC_Message.
```

self.MSG_ID

```
[ Constant: int  ]
The message ID for this type.
```

del this

```
[ Destructor: returns void  ]
Destructor.
```

rawdata()

```
[ Member : returns void * ]
Return the data as a swigged void*.  The SWIG pointer library
may be useful in manipulating this buffer.
```

data()

```
[ Member : returns PyObject * ]
Return the data as a python string.
```

Opaque_Message(data,data_size)

```
[ Constructor: returns Py_Opaque_Message * ]
Constructor.
This may be called with the following arguments:
  ---  Opaque_Message ()   ->  null message
  ---  Opaque_Message (None, N) -> message N bytes long.
  ---  Opaque_Message (string) -> Message initialized from string.
  ---  Opaque_Message (void*, N) -> Message N bytes long from the void*.

The SWIG pointer library may be useful in forming the buffer
for the last of these forms.
```

2.4 Status_Message

2.4.1 class Status_Message

`[ created from class Py_Status_Message ]`

Abstract base class for all status messages Derives from IPC_Message.

self.SM_UNKNOWN

`[ Constant: int ]`
Unknown status.

self.SM_START

`[ Constant: int ]`
Reporting a new connection.

self.SM_STOP

`[ Constant: int ]`
Reporting a broken connection.

self.SM_ERROR

`[ Constant: int ]`
Reporting a network error.

self.SM_END

`[ Constant: int ]`
Sent when the connection object can be deleted.

self.SM_READ

`[ Constant: int ]`

This message was sent from a reader thread.

self.SM_WRITE

`[ Constant: int ]`

This message was sent from the writer thread.

self.MSG_ID

`[ Constant: int ]`

The message ID for this type.

Status_Message()

`[ Constructor: returns Py_Status_Message * ]`

Constructor. Since this is an abstract class, it isn't useful to call this from Python.

del this

`[ Destructor: returns void ]`

Destructor.

status()

`[ Member : returns int ]`

What sort of status is this reporting? SM_UNKNOWN, SM_START, SM_STOP, S-M_ERROR, SM_END.

thread()

`[ Member : returns int ]`

From which thread did this originate? SM_READ or SM_WRITE.

error()

`[ Member : returns int ]`

An associated errno code.

str()

`[ Member : returns string ]`

An associated error message.

queueEmpty()

`[ Member : returns bool ]`

The queue empty flag.

2.5 Connection_Start_Msg

2.5.1 class Connection_Start_Msg

`[ created from class Py_Connection_Start_Msg ]`

Status message sent when a connection has been established. Derives from Status_Message.

self.MSG_ID

`[ Constant: int ]`

The message ID for this type.

Connection_Start_Msg(thread)

`[ Constructor: returns Py_Connection_Start_Msg * ]`

Constructor. THREAD is either SM_READ or SM_WRITE; it defaults to SM_READ.

2.6 Connection_Stop_Msg

2.6.1 class Connection_Stop_Msg

`[ created from class Py_Connection_Stop_Msg ]`

Status message sent when a connection has gone away. Derives from Status_Message.

self.MSG_ID

`[ Constant: int ]`

The message ID for this type.

Connection_Stop_Msg(thread,qE,err,DEFAULT_EMPTY_STRING)

`[ Constructor: returns Py_Connection_Stop_Msg * ]`

Constructor. THREAD is either SM_READ or SM_WRITE; it defaults to SM_READ. qE is the queue empty flag; it defaults to true. ERR is an additional error code; it defaults to 0. The final argument is a string giving an error message; it defaults to an empty string.

queueEmpty()

`[ Member : returns bool ]`

Return the queue empty flag.

2.7 Connection_End_Msg

2.7.1 class Connection_End_Msg

`[ created from class Py_Connection_End_Msg ]`

Status message sent when a connection is going away and can be deleted. You normally should use Connection_Start_Message instead of this one. Derives from Status_Message.

self.MSG_ID

`[ Constant: int ]`

The message ID for this type.

Connection_End_Msg(thread)

`[ Constructor: returns Py_Connection_End_Msg * ]`

Constructor. THREAD is either SM_READ or SM_WRITE; it defaults to SM_READ.

3 Low-level communication classes

3.1 IPC_Handle

3.1.1 class IPC_Handle

`[ created from class IPC_Handle ]`

Low-level class for communication. In most cases, you probably want to use one of the Connection classes instead of this.

IPC_Handle(str)

`[ Constructor: returns IPC_Handle * ]`

Constructor. The argument is a string giving the remote 'host:port' to which to connect.

del this

`[ Destructor: returns void ]`

Destructor. Note: deleting an IPC_Handle, does not close the stream.

send_message(msg,timeout)

`[ Member : returns ssize_t ]`

Send a message over the channel. TIMEOUT is a timeout in seconds; -1 means to wait forever. The return value is the number of data bytes sent, or one of the error codes defined in Mstream.

connect()

`[ Member : returns bool ]`

Establish the connection. Returns true if successful.

close()

`[ Member : returns void ]`

Close the connection.

is_open()

`[ Member : returns bool ]`

Test if the connection is open.

remote_addr()

`[ Member : returns bool ]`

Get the address of the other end of the connection. Returns a string in the form 'host:port'. (nb. From python, this really is a no-argument function returning a string.)

3.2 Acceptor

3.2.1 class Acceptor

`[ created from class Acceptor ]`

Low-level class to accept a network connection. In most cases, you probably want to use the Server class instead.

Acceptor(port)

`[ Constructor: returns Acceptor * ]`

Constructor. PORT is the port on which to listen.

del this

`[ Destructor: returns void ]`

Destructor.

open()

`[ Member : returns bool ]`

Start listening on the port. Returns true if successful, false if there's an error.

accept(timout)

[Member : returns int]
Accept a new connection, waiting for up to TIMEOUT seconds. -1 means to wait forever. Automatically calls open if that has not yet been done. Returns 1 on success, ETIME on timeout, and -1 for other errors.

is_open()
[Member : returns bool]
Test if the listen has been started.

get_handle()
[Member : returns IPC_Handle]
Return an IPC_Handle for the last successful accept.

get_port()
[Member : returns unsigned int]
Return the port number with which we were initialized.

4 High-level communication classes

4.1 Processor

4.1.1 class raw_Processor

[created from class Py_Processor]

The raw_Processor class. Instances of this class run a thread to dispatch messages. There is a single callback to which all messages are sent, set by the raw_callback function. In most cases, you'll want to use the Processor class instead of this one; it has better facilities for dispatching messages.

raw_Processor(qsize)
[Constructor: returns Py_Processor *]
Constructor. Create a new instance, and start the processor thread running. QSIZE is the maximum queue size; it defaults to 20.

del this
[Destructor: returns void]
Destructor. Stop the processor thread, and delete the instance.

dispatch(msg)

`[ Member : returns void ]`
Dispatch an arbitrary message; i.e., enter it in the processor queue. The processor thread will then call the callback function for this message at some later time. MSG should be either an instance of a class deriving from IPC_Message or a string.

dump()
`[ Member : returns void ]`
Debugging dump to stdout.

raw_callback(CALLABLE_OR_NONE)
`[ Member : returns void ]`
Set the callback function. It should be a function of a single argument, and will be called for each message received. It can also be None, in which case no callbacks will be made. The argument to the callback will be a swigged pointer to the message instance. Note: this will be blocked by the callback function itself. Also note: the callback function is responsible for seeing that this object gets deleted.

4.1.2 class Processor

```
The Processor class is what most user code should use for setting
up callbacks.  It derives from raw_Processor, so all of its methods
are available.  In addition, one can also set up callbacks keyed
on the message type.  The set_callback method will set up a callback
to be called on all messages which derive from a given class.  In
addition, Processor supports setting a few ``convenience callbacks,''
to be called when the connection is established or goes away, and
when a string message is received.

Note that the Processor class is implemented in python.

Processor (qsize = 20)
        [ Constructor ]

        Constructor.
        Create a new processor instance, and start the processor thread
        running.   QSIZE is the maximum size of the processor queue.


del this
        [ Destructor]
```

```
        Destructor.
        Stop the processor thread, and delete this instance.


set_callback (cls, cb)
        [ Member :  returns void ]

        Add a callback.
        CLS should be a class which derives from IPC_Message
        (or IPC_Message itself).
        CB should be some callable object.
        For every method with a class which derives from or
        is equal to CLS, CB will be called with the message
        as an argument.

        Multiple callbacks may be registered for the same CLS;
        they will be called in the order in which they were registered.


del_callback (cls, cb)
        [ Member :  returns bool ]

        Delete a callback.
        Both CLS and CB must match exactly the parameters passed
        to an earlier set_callback call.  If a match is found,
        that callback is deleted, and true is returned.  Otherwise,
        false is returned.


string_msg_callback (cb)
        [ Member :  returns void ]

        Set a convenience callback for string messages.
        CB should be some callable object.
        When a String_Message is received, CB will be called with
        two arguments: the Connection instance on which the message
        was received, and the message contents as a string.  To cancel
        the callback, pass in None.

        Note that unlike the callbacks set with set_callback, this
        callback will be called only for messages which are exactly
        of class String_Message -- not for any derived classes.
```

```
connect_callback (cb)
        [ Member :  returns void ]

        Set a convenience callback for connection events.
        CB should be some callable object.
        When a connection which uses this processor is established,
        CB will be called with the Connection instance as an argument.
        To cancel the callback, pass in None.


disconnect_callback (cb)
        [ Member :  returns void ]

        Set a convenience callback for disconnection events.
        CB should be some callable object.
        When a connection which uses this processor is broken,
        CB will be called with the Connection instance as an argument.
        To cancel the callback, pass in None.
```

4.2 Connection

4.2.1 class Connection

`[ created from class Connection ]`

Base class for connections. This has methods for sending messages. To establish the connection, see the methods in the derived classes.

Connection(service_name,proc,max_qsize)

`[ Constructor:  returns Connection * ]`

Constructor. This class is abstract, so there's not much point in calling this from python.

del this

`[ Destructor:  returns void ]`

Destructor.

close()

`[ Member :  returns void ]`

Close the connection.

send_message(msg)

`[ Member : returns bool ]`
Queue a message to be sent over the connection. MSG should be either an instance of a class deriving from IPC_Message or a string. Returns true if the message was successfully queued (but but necessarily sent), false if there was a problem (for example, the connection is not open).

dispatch_message(msg)
`[ Member : returns bool ]`
Dispatch an arbitrary message; i.e., enter it in the processor queue. The processor thread will then call the callback function for this message at some later time. MSG should be either an instance of a class deriving from IPC_Message or a string. Returns false if there's no processor associated with the connection, true otherwise.

is_open()
`[ Member : returns bool ]`
Test if the connection is open.

remote_addr()
`[ Member : returns int ]`
Get the address of the other end of the connection. Returns a string in the form 'host:port'. (nb. From python, this really is a no-argument function returning a string.)

4.2.2 class raw_Client_Connection

`[ created from class Client_Connection ]`

The raw_Client_Connection class. This class derives from Connection, and includes methods to make a connection. Note the memory management restriction of the constructor; in most cases, you'll want to create instances of Client_Connection (which handles that for you) instead of this class.
raw_Client_Connection(service_name,proc,max_qsize)
`[ Constructor: returns Client_Connection * ]`
Constructor. SERVICE_NAME describes the service to which the connection will attempt to connect. It should be a string of the form "host::port". PROC is an instance of Processor. MAX_QSIZE is the maximum queue size for the reader and writer threads; it defaults to 20. WARNING: This object will refer to the processor instance you pass in, but will not make a python reference to it. Therefore, you must ensure that the processor instance is not deleted before this object is. Using the Client_Connection class instead of this one will take care of that automatically.

del this

`[ Destructor: returns void ]`

Destructor.

connect()

`[ Member : returns bool ]`

Attempt to connect to the service described in the string passed to the constructor. Returns true on success, false otherwise.

4.2.3 class Client_Connection

```
The Client_Connection class is what most user code should use
for establishing connections.  It derives from raw_Client_Connection,
and provides the following functionality:

  - Establish connections.
  - Send messages.
  - Set receive callbacks.

A Client_Connection has a Processor instance associated with it.
Multiple connections may share a single processor.
The processor is given to the connection via the constructor.
If no processor is provided, a new processor instance
is created automatically.

For convenience, the Processor methods for establishing callbacks
are also provided here.  They simply forward to the associated
Processor.

Note that the Client_Connection class is implemented in python.

Here is a simple example:

    import time
    import d0me

    def doprint (conn, s):
        print s

    def test (host, port):
        cn = d0me.Client_Connection ('%s:%s' %  (host, port))
```

```
        cn.string_msg_callback (doprint)
        cn.connect ()
        cn.send_message ('boo')
        time.sleep (1)
        cn.close ()
```

```
Client_Connection (service_name, proc = None, max_qsize = 20)
        [ Constructor ]

        Constructor.
        SERVICE_NAME describes the service to which the connection
        will attempt to connect.  It should be a string of the form
        ``host::port''.
        PROC is an instance of Processor.  If it is defaulted to None,
        a new instance will be created automatically.
        MAX_QSIZE is the maximum queue size for the reader and writer
        threads; it defaults to 20.


del this
        [ Destructor]

        Destructor.


set_callback (cls, cb)
        [ Member :  returns void ]

        Add a callback.
        CLS should be a class which derives from IPC_Message
        (or IPC_Message itself).
        CB should be some callable object.
        For every method with a class which derives from or
        is equal to CLS, CB will be called with the message
        as an argument.

        Multiple callbacks may be registered for the same CLS;
        they will be called in the order in which they were registered.


del_callback (cls, cb)
        [ Member :  returns bool ]
```

```
        Delete a callback.
        Both CLS and CB must match exactly the parameters passed
        to an earlier set_callback call.  If a match is found,
        that callback is deleted, and true is returned.  Otherwise,
        false is returned.


string_msg_callback (cb)
        [ Member :  returns void ]

        Set a convenience callback for string messages.
        CB should be some callable object.
        When a String_Message is received, CB will be called with
        two arguments: the Connection instance on which the message
        was received, and the message contents as a string.  To cancel
        the callback, pass in None.

        Note that unlike the callbacks set with set_callback, this
        callback will be called only for messages which are exactly
        of class String_Message -- not for any derived classes.


connect_callback (cb)
        [ Member :  returns void ]

        Set a convenience callback for connection events.
        CB should be some callable object.
        When a connection which uses this processor is established,
        CB will be called with the Connection instance as an argument.
        To cancel the callback, pass in None.


disconnect_callback (cb)
        [ Member :  returns void ]

        Set a convenience callback for disconnection events.
        CB should be some callable object.
        When a connection which uses this processor is broken,
        CB will be called with the Connection instance as an argument.
        To cancel the callback, pass in None.
```

4.2.4 class raw_Server_Connection

[created from class Server_Connection]

The raw_Server_Connection class. This class derives from Connection. Objects of this type are intended to be created from the information returned from Acceptor.accept. Note the memory management restriction of the constructor; in most cases, you'll want to create instances of Client_Connection (which handles that for you) instead of this class.

raw_Server_Connection(handle,proc,server,max_qsize)

`[ Constructor: returns Server_Connection * ]`

Constructor. HANDLE is an IPC_Handle instance, as returned from Acceptor.accept. PROC is an instance of Processor. The SERVER argument should be defaulted to null for calls from python. MAX_QSIZE is the maximum queue size for the reader and writer threads; it defaults to 20.

WARNING: This object will refer to the processor instance you pass in, but will not make a python reference to it. Therefore, you must ensure that the processor instance is not deleted before this object is. Using the Server_Connection class instead of this one will take care of that automatically.

del this

`[ Destructor: returns void ]`

Destructor.

4.2.5 class Server_Connection

```
The Server_Connection class is for connections established via
Acceptor.accept.  It derives from raw_Server_Connection.
In most cases, user code should probably use the Server.accept
or Server.run methods, rather than creating Server_Connection
objects directly.

A Server_Connection has a Processor instance associated with it.
Multiple connections may share a single processor.
The processor is given to the connection via the constructor.
If no processor is provided, a new processor instance
is created automatically.

For convenience, the Processor methods for establishing callbacks
are also provided here.  They simply forward to the associated
Processor.

Note that the Server_Connection class is implemented in python.
```

```
Server_Connection (handle, proc = None, max_qsize = 20)
        [ Constructor ]

        Constructor.
        HANDLE is an IPC_Handle instance, as returned from Acceptor.accept.
        PROC is an instance of Processor.
        MAX_QSIZE is the maximum queue size for the reader and writer
        threads; it defaults to 20.


del this
        [ Destructor]

        Destructor.


set_callback (cls, cb)
        [ Member :  returns void ]

        Add a callback.
        CLS should be a class which derives from IPC_Message
        (or IPC_Message itself).
        CB should be some callable object.
        For every method with a class which derives from or
        is equal to CLS, CB will be called with the message
        as an argument.

        Multiple callbacks may be registered for the same CLS;
        they will be called in the order in which they were registered.


del_callback (cls, cb)
        [ Member :  returns bool ]

        Delete a callback.
        Both CLS and CB must match exactly the parameters passed
        to an earlier set_callback call.  If a match is found,
        that callback is deleted, and true is returned.  Otherwise,
        false is returned.


string_msg_callback (cb)
        [ Member :  returns void ]
```

```
        Set a convenience callback for string messages.
        CB should be some callable object.
        When a String_Message is received, CB will be called with
        two arguments: the Connection instance on which the message
        was received, and the message contents as a string.  To cancel
        the callback, pass in None.

        Note that unlike the callbacks set with set_callback, this
        callback will be called only for messages which are exactly
        of class String_Message -- not for any derived classes.


connect_callback (cb)
        [ Member :  returns void ]

        Set a convenience callback for connection events.
        CB should be some callable object.
        When a connection which uses this processor is established,
        CB will be called with the Connection instance as an argument.
        To cancel the callback, pass in None.


disconnect_callback (cb)
        [ Member :  returns void ]

        Set a convenience callback for disconnection events.
        CB should be some callable object.
        When a connection which uses this processor is broken,
        CB will be called with the Connection instance as an argument.
        To cancel the callback, pass in None.
```

4.3 Server

4.3.1 class Server

```
The Server class is used to accept connections from remote
processes.  This is the class which most user code should probably
use for this.

The main point of this class is the accept method, which accepts
a connection from a remote process, and creates a new connection
instance using a specified Processor instance.  There is also
a simple run method, which sits in a loop repeatedly calling accept.
```

```
Note that the Server class is implemented in python.

Here's a very simple example of using it:

    import d0me

    def doecho (conn, s):
        print s
        conn.send_message (s)

    def echoserve (port):
        s = d0me.Server (port)
        s.open ()
        p = d0me.Processor ()
        p.string_msg_callback (doecho)
        while 1:
            s.accept (p)


Server (port)
        [ Constructor ]

        Constructor.
        PORT is the port number on which to listen.
        Note that this does not actually start listening
        on the port --- call open() to do that.


del this
        [ Destructor]

        Destructor.


open ()
        [ Member :  returns bool ]

        Start listening on the port specified in the constructor.
        Returns true on success, false on error.


is_open ()
        [ Member :  returns bool ]
```

```
        Test if we're presently listening on the port.


accept (proc, timeout = -1, cls = d0me.Server_Connection)
        [ Member :  returns int ]

        Accept a connection on the port.
        If open has not yet been called, it will be called
        automatically.

        Wait for at most TIMEOUT seconds.  If TIMEOUT is negative,
        then wait forever.  It defaults to -1.

        If a connection is accepted, a new instance of the class
        CLS will be created.  If it is unspecified, it defaults
        to Server_Connection.  The constructor for CLS should
        match that of Server_Connection.

        PROC is the processor to be passed to the newly created
        connection.  Note that a callback will automatically be added
        to notice when the connection breaks, so that we can
        clean up the instance.

        Returns 1 if a connection was successfully accepted.
        Returns ETIME on a timeout.


reap ()
        [ Member :  returns void ]

        When a connection breaks, the connection instance created
        in accept() is queued for deletion.  This method actually
        deletes any such connection instances which have been
        so queued.  [Actually, it simply deletes all the references
        that the Server class had to that instance.  If you have
        other references to the connection instance, then it will
        of course not be deleted.]

        Note this this method is called by accept(), so it should
        usually not be necessary to call this method explicitly.


run (proc)
```

```
[ Member : returns int ]

Sit in a loop calling accept() with processor PROC
until accept() returns an error.
```

5 Handling external message classes

5.1 class Register_Class

`[ created from struct Register_Class ]`

register_message_class(reg,msg_id)

`[ returns void ]`

Register a new message class with d0me. REG is the registration object, as returned from the class's make_registration_class method. MSG_ID is the id number for the message.

6 Miscellaneous d0me features

6.1 Setup

6.1.1 Global variables

cvar._deb

`[ Global : int _deb ]`

Debug level.

cvar._Timeout

`[ Global : float _Timeout ]`

Thread timeout.

cvar._bigTimeout

`[ Global : float _bigTimeout ]`

Socket timeout.

cvar._queueSize

`[ Global : int _queueSize ]`

Default queue size.

6.1.2 Error codes

Return values from low level send/recv. A return > 0 is the number of bytes written/sent.

SE_UNKNOWN = SE_UNKNOWN
`[ Constant: int ]`
Unknown message type.

SE_BADHEADER = SE_BADHEADER
`[ Constant: int ]`
Bad header (bad magic number).

SE_BROKENCONN = SE_BROKENCONN
`[ Constant: int ]`
Client broke the connection.

SE_ERROR = SE_ERROR
`[ Constant: int ]`
Any other error.

SE_TIMEOUT = SE_TIMEOUT
`[ Constant: int ]`
Timeout.

7 Pointer Handling Library

```
%include pointer.i

The pointer.i library provides run-time support for managing and
manipulating a variety of C/C++ pointer values.  In particular,
you can create various kinds of objects and dereference common
pointer types.  This is done through a common set of functions:

    ptrcast      - Casts a pointer to a new type
    ptrvalue     - Dereferences a pointer
    ptrset       - Set the value of an object referenced by
                   a pointer.
    ptrcreate    - Create a new object and return a pointer.
    ptrfree      - Free the memory allocated by ptrcreate.
    ptradd       - Increment/decrement a pointer value.
```

```
    ptrmap       - Make two datatypes equivalent to each other.
                   (Is a runtime equivalent of typedef).

When creating, dereferencing, or setting the value of pointer
variable, only the common C datatypes of int, short, long, float,
double, char, and char * are currently supported.   Other
datatypes may generate an error.

One of the more interesting aspects of this library is that
it operates with a wide range of datatypes.  For example,
the "ptrvalue" function can dereference "double *", "int *",
"long *", "char *", and other datatypes. Since SWIG encodes
pointers with type information, this can be done transparently
and in most cases, you can dereference a pointer without
ever knowing what type it actually is.

This library is primarily designed for utility, not high
performance (the dynamic determination of pointer types takes
more work than most normal wrapper functions).  As a result,
you may achieve better performance by writing customized
"helper" functions if you're making lots of calls to these
functions in inner loops or other intensive operations.
```

ptrcast(ptr,type)

```
Casts a pointer ptr to a new datatype given by the string type.
type may be either the SWIG generated representation of a datatype
or the C representation.  For example :

   ptrcast(ptr,"double_p");   # Python representation
   ptrcast(ptr,"double *");    # C representation

A new pointer value is returned.   ptr may also be an integer
value in which case the value will be used to set the pointer
value.  For example :

   a = ptrcast(0,"Vector_p");

Will create a NULL pointer of type "Vector_p"

The casting operation is sensitive to formatting.  As a result,
"double *" is different than "double*".  As a result of thumb,
there should always be exactly one space between the C datatype
and any pointer specifiers (*).
```

ptrvalue(ptr,index,type)

```
Returns the value that a pointer is pointing to (ie. dereferencing).
The type is automatically inferred by the pointer type--thus, an
integer pointer will return an integer, a double will return a double,
and so on.   The index and type fields are optional parameters.  When
an index is specified, this function returns the value of ptr[index].
This allows array access.   When a type is specified, it overrides
the given pointer type.   Examples :

   ptrvalue(a)             #  Returns the value *a
   ptrvalue(a,10)          #  Returns the value a[10]
   ptrvalue(a,10,"double") #  Returns a[10] assuming a is a double *
```

ptrset(ptr,value,index,type)

```
Sets the value pointed to by a pointer.  The type is automatically
inferred from the pointer type so this function will work for
integers, floats, doubles, etc...  The index and type fields are
optional.  When an index is given, it provides array access.  When
type is specified, it overrides the given pointer type.  Examples :

  ptrset(a,3)            # Sets the value *a = 3
  ptrset(a,3,10)         # Sets a[10] = 3
  ptrset(a,3,10,"int")   # Sets a[10] = 3 assuming a is a int *
```

ptrcreate(type,value,nitems)

```
Creates a new object and returns a pointer to it.  This function
can be used to create various kinds of objects for use in C functions.
type specifies the basic C datatype to create and value is an
optional parameter that can be used to set the initial value of the
object.  nitems is an optional parameter that can be used to create
an array.  This function results in a memory allocation using
malloc().  Examples :

  a = ptrcreate("double")     # Create a new double, return pointer
  a = ptrcreate("int",7)      # Create an integer, set value to 7
  a = ptrcreate("int",0,1000) # Create an integer array with initial
                              # values all set to zero

This function only recognizes a few common C datatypes as listed below :

       int, short, long, float, double, char, char *, void

All other datatypes will result in an error.  However, other
datatypes can be created by using the ptrcast function.  For
example:

 a = ptrcast(ptrcreate("int",0,100),"unsigned int *")
```

ptrfree(ptr)

Destroys the memory pointed to by ptr. This function calls free() and should only be used with objects created by ptrcreate(). Since this function calls free, it may work with other objects, but this is generally discouraged unless you absolutely know what you're doing.

ptradd(ptr,offset)

Adds a value to the current pointer value. For the C datatypes of int, short, long, float, double, and char, the offset value is the number of objects and works in exactly the same manner as in C. For example, the following code steps through the elements of an array

```
a = ptrcreate("double",0,100);    # Create an array double a[100]
b = a;
for i in range(0,100):
    ptrset(b,0.0025*i);           # set *b = 0.0025*i
    b = ptradd(b,1);              # b++ (go to next double)
```

In this case, adding one to b goes to the next double.

For all other datatypes (including all complex datatypes), the offset corresponds to bytes. This function does not perform any bounds checking and negative offsets are perfectly legal.

ptrmap(type1,type2)

```
This is a rarely used function that performs essentially the same
operation as a C typedef.  To manage datatypes at run-time, SWIG
modules manage an internal symbol table of type mappings.  This
table keeps track of which types are equivalent to each other.  The
ptrmap() function provides a mechanism for scripts to add symbols
to this table.  For example :

   ptrmap("double_p","Real_p");

would make the types "doublePtr" and "RealPtr" equivalent to each
other.  Pointers of either type could now be used interchangably.

Normally this function is not needed, but it can be used to
circumvent SWIG's normal type-checking behavior or to work around
weird type-handling problems.
```

ptrsize(type,numelements)

```
Returns the size, in bytes, of the memory which ptrcreate would
allocate when called with TYPE and NUMELEMENTS.
```