Hurricane Python/C++ API Tutorial

Contents

1. Introduction 2
1.1 First, A Disclaimer 2
1.2 About Technical Choices 2
1.3 Botched Design 3
2. Basic File Structure and CMake configuration 4
3. Case 1 - DBo Derived, Standalone 5
3.1 Class Associated Header File 5
3.2 Class Associated File 6
3.2.1 Head of the file. 6
3.2.2 The Python Module Part 7
3.2.3 Python Type Linking 9
3.2.4 The Shared Library Part 10
3.3 Python Module (C++ namespace) 10
4. Case 2 - Hierarchy of DBo Derived Classes 11
4.1 Base Class Header 11
4.2 Base Class File 12
4.3 Intermediate Class Header. 14
4.4 Intermediate Class File 14
4.5 Terminal Class Header 16
4.6 Terminal Class File 16
4.8 Python Module 17
5. Case 3 - Non-DBo Standalone Classe 18
5.1 Class Header 18
5.2 Class File 19
5.2 Class File 20
6. Encapsulating DbU 20
7. No C++ Hurricane::Name encapsulation 21

1. Introduction

- This document is written for people already familiar with the Python/C API Reference Manual.
- The macros provided by the Hurricane Python/C API are written using the standard Python C/API. That is, you may not use them and write directly your functions with the original API or any mix between. You only have to respect some naming convention.
- Coriolis is build against Python 2.7.

1.1 First, A Disclaimer

The Hurricane Python/C++ API has been written about ten years ago, at a time my mastering of template programming was less than complete. This is why this interface is build with old fashioned C macro instead of C++ template.

It is my hope that at some point in the future I will have time to completly rewrite it, borrowing the interface from `boost::python`.

1.2 About Technical Choices

Some would say, why not use *off the shelf* wrappers like `swig` or `boost::python`, here are some clues.

1. **Partial exposure of the C++ class tree.** We expose at Python level C++ base classes, only if they provides common methods that we want to see. Otherwise, we just show them as base classes under Python. For instance `Library` is derived from `DBo`, but we won't see it under Python.
2. **Bi-directional communication.** When a Python object is deleted, the wrapper obviously has a pointer toward the underlying C++ object and is able to delete it. But, the reverse case can occurs, meaning that you have a C++ object wrapped in Python and the database delete the underlying object. The wrapped Python object *must* be informed that it no longer refer a valid C++ one. Moreover, as we do not control when Python objects gets deleteds (that is, when their reference count reaches zero), we can have valid Python object with a dangling C++ pointer. So our Python objects can be warned by the C++ objects that they are no longer valid and any other operation than the deletion should result in a severe non-blocking error.

   To be precise, this apply to persistent object in the C++ database, like `Cell`, `Net`, `Instance` or `Component`. Short lived objects like `Box` or `Point` retains the classic Python behavior.

   Another aspect is that, for all derived `DBo` objects, one and only one Python object is associated. For one given `Instance` object we will always return the *same* `PyInstance` object, thanks to the bi-directional link. Obviously, the *reference count* of the `PyInstance` is managed accordingly. This mechanism is implemented by the `PyInstance_Link()` function.
3. **Linking accross modules.** As far as I understand, the wrappers are for monolithic libraries. That is, you wrap the entire library in one go. But Hurricane has a modular design, the core database then various tools. We do not, and cannot, have one gigantic wrapper that would encompass all the libraries in one go. We do one Python module for one C++ library.

   This brings another issue, at Python level this time. The Python modules for the libraries have to share some functions. Python provides a mechanism to pass C function pointers accross modules, but I did found it cumbersome. Instead, all our modules are split in two:

- The first part contains the classic Python module code.
- The second part is to be put in a separate dynamic library that will hold the shared functions. The Python module is dynamically linked against that library like any other. And any other Python module requiring the functions will link against the associated shared library.

Each module file will be compiled *twice*, once to build the Python module (`__PYTHON_MODULE` is defined) and once to build the supporting shared library (`__PYTHON_MODULE__` **not** defined). This tricky double compilation is taken care of though the `add_python_module cmake` macro.

For the core Hurricane library we will have:

- `Hurricane.so` the Python module (use with: `import Hurricane`).
- `libisobar.so.1.0` the supporting shared library.

The `PyLibrary.cpp` file will have the following structure:

```
#include "hurricane/isobar/PyLibrary.h"

namespace  Isobar {

  extern "C" {

#if defined(__PYTHON_MODULE__)

  // +=================================================================+
  // |                "PyLibrary" Python Module Code Part              |
  // +=================================================================+
  //
  // The classic part of a Python module. Goes into Hurricane.so.


#else  // End of Python Module Code Part.

  // x=================================================================x
  // |               "PyLibrary" Shared Library Code Part              |
  // x=================================================================x
  //
  // Functions here will be part of the associated shared library and
  // made available to all other Python modules. Goes into libisobar.so.1.0


# endif  // Shared Library Code Part.

  }  // extern "C".

}  // Isobar namespace.
```

This way, we do not rely upon a pointer transmission through Python modules, but directly uses linker capabilities.

1.3 Botched Design

The mechanism to compute the signature of a call to a Python function, the `__cs` object, is much too complex and, in fact, not needed. At some point I may root it out, but it is used in so many places...

What I should have used the `"O!"` capablity of `PyArg_ParseTuple()`, like in the code below:

```
static PyObject* PyContact_create ( PyObject*, PyObject *args )
{
  Contact* contact = NULL;
  HTRY
    PyNet*        pyNet        = NULL;
    PyLayer*      pyLayer      = NULL;
    PyComponent*  pyComponent  = NULL;
    DbU::Unit     x            = 0;
    DbU::Unit     y            = 0;
    DbU::Unit     width        = 0;
    DbU::Unit     height       = 0;

    if (PyArg_ParseTuple( args, "O!O!ll|ll:Contact.create"
                        , &PyTypeNet  , &pyNet
                        , &PyTypeLayer, &pyLayer
                        , &x, &y, &width, &height)) {
      contact = Contact::create( PYNET_O(pyNet), PYLAYER_O(pyLayer)
                               , x, y, width, height );
    } else {
      PyErr_Clear();
      if (PyArg_ParseTuple( args, "O!O!ll|ll:Contact.create"
                          , &PyTypeComponent, &pyComponent
                          , &PyTypeLayer    , &pyLayer
                          , &x, &y, &width, &height)) {
        contact = Contact::create( PYCOMPONENT_O(pyComponent), PYLAYER_O(pyLayer)
                                 , x, y, width, height );
      } else {
        PyErr_SetString( ConstructorError
                       , "invalid number of parameters for Contact constructor." );
        return NULL;
      }
    }
  HCATCH
  return PyContact_Link( contact );
}
```

2. Basic File Structure and CMake configuration

As a first example we will consider the `Hurrican::Library` class. To export a class into Python, we must create three files:

1. `PyLibrary.h`, defines the `PyLibrary` C-Struct and the functions needed outside the module istself (mostly for `PyHurricane.cpp`).
2. `PyLibrary.cpp`, contains the complete wrapping of the class and the Python type definition (`PyTypeLibrary`).
3. `PyHurricane.cpp`, the definition of the Python module into which the classes are registered. The module act as a `namespace` in Python so it is good practice to give it the same name as it's associated C++ namespace.

To build a Python module in **cmake** , use the following macro:

```
                     set( pyCpps      PyLibrary.cpp
                                      PyHurricane.cpp )
                     set( pyIncludes  hurricane/isobar/PyLibrary.h

   add_python_module( "${pyCpps}"
                      "${pyIncludes}"
                      "isobar;1.0;1"       # Name & version of the supporting
                                           # shared library.
                      Hurricane            # Name of the Python module will give:
                                           #   Hurricane.so
                      "${depLibs}"         # List of dependency libraries.
                      include/coriolis2/hurricane/isobar
                                           # Where to install the include files.
                    )
```

3. Case 1 - DBo Derived, Standalone

As example, we take `Library`. This a `DBo` derived class, but we choose not to export the parent classes. From Python, it will appear as a base class.

3.1 Class Associated Header File

Here is the typical content of a header file (for `PyLibrary`):

```
#ifndef  PY_LIBRARY_H
#define  PY_LIBRARY_H

#include "hurricane/isobar/PyHurricane.h"
#include "hurricane/Library.h"

namespace  Isobar {
  using namespace Hurricane;

  extern "C" {

    typedef struct {
        PyObject_HEAD
        Library* _object;
    } PyLibrary;

    extern  PyTypeObject  PyTypeLibrary;
    extern  PyMethodDef   PyLibrary_Methods[];
    extern  PyObject*     PyLibrary_Link        ( Hurricane::Library* lib );
    extern  void          PyLibrary_LinkPyType ();


#define IsPyLibrary(v)  ( (v)->ob_type == &PyTypeLibrary )
#define PYLIBRARY(v)    ( (PyLibrary*)(v) )
#define PYLIBRARY_O(v)  ( PYLIBRARY(v)->_object )

  }  // extern "C".
}  // Isobar namespace.

#endif  // PY_LIBRARY_H
```

The code is organized as follow:

1. It must have, *as the first include* `PyHurricane.h`, which provides the complete bunch of macros needed to build the module. Then the include of the C++ class we want to wrap (`Library.h`).
2. As Python is written in C, all the wrapper code has to be but inside an `extern "C"` namespace.
3. Definition of the wrapped **`struct`** , `PyLibrary`. It is standard Python here.

Note

For our set of macros to work, the name of the pointer to the C++ class must always be **_object**, and the various functions and macros defined here must take the name of the class (either in lowercase, camel case or capitals).

4. Declaration of the Python type `PyTypeLibrary` (standard).
5. Declaration of the Python type table of methods `PyLibrary_Methods` (standard).
6. Declaration of `PyLibrary_Link()`, helper to convert a C++ `Lybrary` into a `PyLibrary` (put in the support shared library).
7. Declaration of `PyLibrary_LinkPyType()`, this function setup the class-level function of the new Python type (here, `PyTypeLibrary`).
8. And, lastly, three macros to:
   - `IsPylibrary()`, know if a Python object is a `PyLibrary`
   - `PYLIBRARY()`, force cast (C style) of a `PyObject` into a `PyLibrary`.
   - `PYLIBRARY_O()`, extract the C++ object (`Library*`) from the Python object (`PyLibrary`).

3.2 Class Associated File

3.2.1 Head of the file

```
#include "hurricane/isobar/PyLibrary.h"
#include "hurricane/isobar/PyDataBase.h"
#include "hurricane/isobar/PyCell.h"

namespace Isobar {
  using namespace Hurricane;

  extern "C" {

#define  METHOD_HEAD(function)   GENERIC_METHOD_HEAD(Library,lib,function)
```

As for the header, all the code must be put inside a `extern "C"` namespace.

A convenience macro `METHOD_HEAD()` must be defined, by refining `GENERIC_METHOD_HEAD()`. This macro will be used in the method wrappers below to cast the `_object` field of the Python object into the appropriate C++ class, this is done using a C-style cast. The parameters of that macro are:

1. The C++ encapsulated class (`Library`).
2. The name of the *variable* that will be used to store a pointer to the C++ working object.
3. The name of the C++ method which is to be wrapped.

3.2.2 The Python Module Part

First, we have to build all the wrappers to the C++ methods of the class. For common predicates, accessors, and mutators macros are supplied.

Wrapping of the `Library::getCell()` method:

```
static PyObject* PyLibrary_getCell ( PyLibrary* self, PyObject* args )
{
  Cell* cell = NULL;

  HTRY
    METHOD_HEAD( "Library.getCell()" )
    char* name = NULL;
    if (PyArg_ParseTuple(args,"s:Library.getCell", &name)) {
      cell = lib->getCell( Name(name) );
    } else {
      PyErr_SetString( ConstructorError
                     , "invalid number of parameters for Library::getCell." );
      return NULL;
    }
  HCATCH

  return PyCell_Link(cell);
}
```

Key points about this method wrapper:

1. The `HTRY` / `HCATCH` macros provides an insulation from the C++ exceptions. If one is emitted, it will be catched and transformed in a Python one. This way, the Python program will be cleanly interrupted and the usual stack trace displayed.
2. The returned value of this method is of type `Cell*`, we have to transform it into a Python one. This is done with `PyCell_Link()`. This macro is supplied by the `PyCell.h` header and this is why it must be included.

Wrapping of the `Library::create()` method:

```
static PyObject* PyLibrary_create( PyObject*, PyObject* args )
{
  PyObject* arg0;
  PyObject* arg1;
  Library* library = NULL;

  HTRY
    __cs.init( "Library.create" );                          // Step (1).
    if (not PyArg_ParseTuple( args, "O&O&:Library.create"
                            , Converter, &arg0
                            , Converter, &arg1 )) {         // Step (2).
      PyErr_SetString( ConstructorError
                     , "invalid number of parameters for Library constructor." );
      return NULL;
    }
    if (__cs.getObjectIds() == ":db:string") {              // Step (3.a)
      DataBase* db = PYDATABASE_O(arg0);
      library = Library::create( db, Name(PyString_AsString(arg1)) );
    } else if (__cs.getObjectIds() == ":library:string") {  // Step (3.b)
      Library* masterLibrary = PYLIBRARY_O(arg0);
      library = Library::create( masterLibrary, Name(PyString_AsString(arg1)) );
    } else {
      PyErr_SetString( ConstructorError
                     , "invalid number of parameters for Library constructor." );
      return NULL;
    }
  HCATCH

  return PyLibrary_Link( library );
}
```

Key point about this constructor:

1. We want the Python interface to mimic as closely as possible the C++ API. As such, Python object will be created using a static `.create()` method. So we do not use the usual Python allocation mechanism.
2. As it is a *static* method, there is no first argument.
3. Python do not allow function overload like C++. To emulate that behavior we use the `__cs` object (which is a global variable).
    1. Init/reset the `__cs` object: see *step (1)*.
    2. Call `PyArg_ParseTuple()`, read every mandatory or optional argument as a Python object (`"O&"`) and use `Converter` on each one. `Converter` will determine the real type of the Python object given as argument by looking at the encapsulated C++ class. It then update the `__cs` object. Done in *step (2)*
    3. After the call to `PyArg_ParseTuple()`, the function `__cs.getObjectIds()` will return the *signature* of the various arguments. In our case, the valid signatures will be `":db:string"` *(step (3.a)*a) and "":library:string"" (*step (3.b)*).
    4. Call the C++ method after extracting the C++ objects from the Python arguments. Note the use of the `PYLIBRARY_O()` and `PYDATABSE_O()` macros to perform the conversion.
4. Return the result, encapsulated through a call to `PyLibrary_Link()`.

Wrapping of the `Library::destroy()` method:

```
DBoDestroyAttribute(PyLibrary_destroy, PyLibrary)
```

For C++ classes **that are derived** from `DBo`, the destroy method wrapper must be defined using the macro `DBoDestroyAttribute()`. This macro implements the bi-directional communication mechanism using `Hurricane::Property`. It **must not** be used for non `DBo` derived classes.

Defining the method table of the PyLibrary type:

```
PyMethodDef PyLibrary_Methods[] =
  { { "create"     , (PyCFunction)PyLibrary_create , METH_VARARGS|METH_STATIC
                   , "Creates a new library." }
  , { "getCell"    , (PyCFunction)PyLibrary_getCell, METH_VARARGS
                   , "Get the cell of name <name>" }
  , { "destroy"    , (PyCFunction)PyLibrary_destroy, METH_NOARGS
                   , "Destroy associated hurricane object The python object remains." }
  , {NULL, NULL, 0, NULL}          /* sentinel */
  };
```

This is standard Python/C API. The name of the `PyMethodDef` table must be named from the class: `PyLibrary_Methods`.

3.2.3 Python Type Linking

Defining the `PyTypeLibrary` class methods and the type linking function.

Those are the functions for the Python object itself to work, not the wrapped method from the C++ class.

> **Note**
> At this point we **do not** define the `PyTypeLibrary` itself. Only it's functions and a function to set them up *once* the type will be defined.

```
DBoDeleteMethod(Library)
PyTypeObjectLinkPyType(Library)
```

The macro `DBoDeleteMethod()` define the function to delete a `PyLibrary` *Python* object. Again, do not mistake it for the deletion of the C++ class (implemented by `DBoDestroyAttribute()`). Here again, `DBoDeleteMethod()` is specially tailored for `DBo` derived classes.

To define `PyLibrary_LinkPyType()`, use the `PyTypeObjectLinkPyType()` macro. This macro is specific for `DBo` derived classes that are seen as base classes under Python (i.e. we don't bother exposing the base class under Python). `PyLibrary_LinkPyType()` setup the class functions in the `PyTypeLibrary` type object, it **must** be called in the Python module this class is part of (in this case: `PyHurricane.cpp`). This particular flavor of the macro *will define* and setup the following class functions:

- `PyTypeLibrary.tp_compare` (defined by the macro).
- `PyTypeLibrary.tp_repr` (defined by the macro).
- `PyTypeLibrary.tp_str` (defined by the macro).
- `PyTypeLibrary.tp_hash` (defined by the macro).
- `PyTypeLibrary.tp_methods` sets to the previously defined `PyLibrary_Methods` table.
- `PyTypeLibrary.tp_dealloc` is set to a function that *must* be named `PyLibrary_DeAlloc`, this is what `DBoDeleteMethod` does. It is *not* done by `PyTypeObjectLinkPyType`.

Defining the `PyTypeLibrary` type:

3.2.4 The Shared Library Part

This part will be put in a separate supporting shared library, allowing other Python module to link against it (and make use of its symbols).

```
DBoLinkCreateMethod(Library)
PyTypeObjectDefinitions(Library)
```

To define `PyTypeLibrary`, use the `PyTypeObjectDefinitions()` macro. This macro is specific for classes that, as exposed by Python, are neither *derived* classes nor *base* classes for others. That is, they are standalone from the inheritance point of view.

The `DBoLinkCreateMethod()` macro will define the `PyLibrary_Link()` function which is responsible for encapsulating a C++ `Library` object into a Python `PyLibrary` one.

3.3 Python Module (C++ namespace)

We use the Python module to replicate the C++ *namespace*. Thus, for the `Hurricane` namespace we create a Python `Hurricane` module which is defined in the `PyHurricane.cpp` file, then we add into that module dictionary all the Python types encapsulating the C++ classes of that namespace.

```
DL_EXPORT(void) initHurricane ()
{
  PyLibrary_LinkPyType();  // step 1.

  PYTYPE_READY( Library )  // step 2.

  __cs.addType( "library", &PyTypeLibrary, "<Library>", false ); // step 3.

  PyObject* module = Py_InitModule( "Hurricane", PyHurricane_Methods );
  if (module == NULL) {
    cerr << "[ERROR]\n"
         << "  Failed to initialize Hurricane module." << endl;
    return;
  }

  Py_INCREF( &PyTypeLibrary );                                            // step 4.
  PyModule_AddObject( module, "Library", (PyObject*)&PyTypeLibrary ); // step 4.
}
```

The `initHurricane()` initialisation function shown above has been scrubbed of everything not relevant to the `PyLibrary` class. The integration of the `PyLibrary` class into the module needs four steps:

1. A call to PyLibrary_LinkPyType() to hook the Python type functions in the Python type object.
2. A call to the `PYTYPE_READY()` macro (standard Python).
3. Registering the type into the `__cs` object, with `addType()`. The arguments are self explanatory, save for the last which is a boolean to tell if this is a *derived* class or not.
4. Adding the type object (`PyTypeLibrary`) into the dictionnary of the module itself. This allow to mimic closely the C++ syntax:

   ```
   import Hurricane
   lib = Hurricane.Library.create( db, 'root' )
   ```

4. Case 2 - Hierarchy of DBo Derived Classes

Now we want to export the following C++ class hierarchy into Python:

```
PyEntity <-- PyComponent <-+- PyContact
                           +- PySegment <-+- PyHorizontal
                                          +- PyVertical
```

4.1 Base Class Header

Remark: this is only a partial description of the tree for the sake of clarity.

One important fact to remember is that `PyEntity` and `PyComponent` being related to C++ abstract classes, no objects of those types will be created, only `PyContact`, `PyHorizontal` or `PyVertical` will.

The consequence is that there is no `PyEntity_Link()` like in 3.1 but instead two functions:

1. `PyEntity_NEW()` which create the relevant `PyEntity` *derived* object from the `Entity` one. For example, if the `Entity*` given as argument is in fact a `Horizontal*`, then the function will return a `PyHorizontal*`.

2. `EntityCast()` do the reverse of `PyEntity_NEW()` that is, from a `PyEntity`, return the C++ *derived* object. Again, if the `PyEntity*` is a `PyHorizontal*`, the function will cast it as a `Horizontal*` *then* return it as an `Entity*`.

```
#ifndef ISOBAR_PY_ENTITY_H
#define ISOBAR_PY_ENTITY_H

#include "hurricane/isobar/PyHurricane.h"
#include "hurricane/Entity.h"

namespace  Isobar {
  extern "C" {

    typedef struct {
        PyObject_HEAD
        Hurricane::Entity* _object;
    } PyEntity;

    extern  PyObject*      PyEntity_NEW          ( Hurricane::Entity* entity );
    extern  void           PyEntity_LinkPyType  ();
    extern  PyTypeObject   PyTypeEntity;
    extern  PyMethodDef    PyEntity_Methods[];


#define IsPyEntity(v)      ( (v)->ob_type == &PyTypeEntity )
#define PYENTITY(v)        ( (PyEntity*)(v)  )
#define PYENTITY_O(v)      ( PYENTITY(v)->_object )

  }  // extern "C".

  Hurricane::Entity*  EntityCast  ( PyObject* derivedObject );

}  // Isobar namespace.

#endif  // ISOBAR_PY_ENTITY_H
```

4.2 Base Class File

Changes from 3.2 Class Associated File are:

1. No call to `DBoLinkCreateMethod()` because there must be no `PyEntity_Link()`, but the definitions of `PyEntity_NEW()` and `EntityCast`.
2. For defining the `PyTypeEntity` Python type, we call a different macro: `PyTypeRootObjectDefinitions`, dedicated to base classes.

```
#include "hurricane/isobar/PyCell.h"
#include "hurricane/isobar/PyHorizontal.h"
#include "hurricane/isobar/PyVertical.h"
#include "hurricane/isobar/PyContact.h"

namespace Isobar {
  using namespace Hurricane;

  extern "C" {

#if defined(__PYTHON_MODULE__)

#define  METHOD_HEAD(function)   GENERIC_METHOD_HEAD(Entity,entity,function)

    DBoDestroyAttribute(PyEntity_destroy ,PyEntity)

    static PyObject* PyEntity_getCell ( PyEntity *self )
    {
      Cell* cell = NULL;
      HTRY
        METHOD_HEAD( "Entity.getCell()" )
        cell = entity->getCell();
      HCATCH
      return PyCell_Link( cell );
    }

    PyMethodDef PyEntity_Methods[] =
      { { "getCell", (PyCFunction)PyEntity_getCell, METH_NOARGS
                   , "Returns the entity cell." }
      , { "destroy", (PyCFunction)PyEntity_destroy, METH_NOARGS
                   , "Destroy associated hurricane object, the python object remains." }
      , {NULL, NULL, 0, NULL}  /* sentinel */
      };


    DBoDeleteMethod(Entity)
    PyTypeObjectLinkPyType(Entity)

#else  // End of Python Module Code Part.

    PyObject* PyEntity_NEW ( Entity* entity )
    {
      if (not entity) {
        PyErr_SetString ( HurricaneError, "Invalid Entity (bad occurrence)" );
        return NULL;
      }

      Horizontal* horizontal = dynamic_cast<Horizontal*>(entity);
      if (horizontal) return PyHorizontal_Link( horizontal );

      Vertical* vertical = dynamic_cast<Vertical*>(entity);
```

```
      if (vertical) return PyVertical_Link( vertical );

      Contact* contact = dynamic_cast<Contact*>(entity);
      if (contact) return PyContact_Link( contact );

      Py_RETURN_NONE;
    }

    PyTypeRootObjectDefinitions(Entity)

#endif  // Shared Library Code Part (1).

}  // extern "C".


#if !defined(__PYTHON_MODULE__)

Hurricane::Entity* EntityCast ( PyObject* derivedObject ) {
  if (IsPyHorizontal(derivedObject)) return PYHORIZONTAL_O(derivedObject);
  if (IsPyVertical  (derivedObject)) return PYVERTICAL_O(derivedObject);
  if (IsPyContact   (derivedObject)) return PYCONTACT_O(derivedObject);
  return NULL;
}

#endif  // Shared Library Code Part (2).

}  // Isobar namespace.
```

4.3 Intermediate Class Header

Changes from 3.1 Class Associated Header File are:

1. As for `PyEntity`, and because this is still an abstract class, there is no `PyComponent_Link()` function.
2. The definition of the `PyComponent` **struct** is differs. There is no `PyObject_HEAD` (it is a Python *derived* class). The only field is of the base class type `PyEntity` and for use with Coriolis macros, **it must** be named `_baseObject` (note that this is *not* a pointer but a whole object).

```
#ifndef ISOBAR_PY_COMPONENT_H
#define ISOBAR_PY_COMPONENT_H

#include "hurricane/isobar/PyEntity.h"
#include "hurricane/Component.h"

namespace  Isobar {
  extern "C" {

    typedef struct {
        PyEntity  _baseObject;
    } PyComponent;

    extern  PyTypeObject  PyTypeComponent;
    extern  PyMethodDef   PyComponent_Methods[];
    extern  void          PyComponent_LinkPyType ();

#define IsPyComponent(v)  ((v)->ob_type == &PyTypeComponent)
#define PYCOMPONENT(v)    ((PyComponent*)(v))
#define PYCOMPONENT_O(v)  (static_cast<Component*>(PYCOMPONENT(v)->_baseObject._object))

  }  // extern "C".
}  // Isobar namespace.

#endif
```

4.4 Intermediate Class File

Changes from 3.2 Class Associated File are:

1. Redefinition of the default macros `ACCESS_OBJECT` and `ACCESS_CLASS`.
   - The pointer to the C++ encapsulated object (attribute `_object`) is hold by the base class `PyEntity`. The `ACCESS_OBJECT` macro which is tasked to give access to that attribute is then `_baseObject._object` as `PyComponent` is a direct derived class of `PyEntity`.
   - `ACCESS_CLASS` is similar to `ACCESS_OBJECT` for accessing the base class, that is a pointer to `PyEntity`.

2. For defining the `PyTypeComponent` Python type, we call a yet different macro: `PyTypeInheritedObjectDefinitions()`, dedicated to derived classes. For this this macro we need to give as argument the derived class and the base class.

```
#include "hurricane/isobar/PyComponent.h"
#include "hurricane/isobar/PyNet.h"

namespace  Isobar {
  using namespace Hurricane;

  extern "C" {

#undef   ACCESS_OBJECT
#undef   ACCESS_CLASS
#define  ACCESS_OBJECT              _baseObject._object
#define  ACCESS_CLASS(_pyObject)   &(_pyObject->_baseObject)
#define  METHOD_HEAD(function)     GENERIC_METHOD_HEAD(Component,component,function)

#if defined(__PYTHON_MODULE__)

    DirectGetLongAttribute(PyComponent_getX,getX,PyComponent,Component)
    DirectGetLongAttribute(PyComponent_getY,getY,PyComponent,Component)
    DBoDestroyAttribute(PyComponent_destroy,PyComponent)

    static PyObject* PyComponent_getNet ( PyComponent *self )
    {
      Net* net = NULL;
      HTRY
        METHOD_HEAD( "Component.getNet()" )
        net = component->getNet( );
      HCATCH
      return PyNet_Link( net );
    }

    PyMethodDef PyComponent_Methods[] =
      { { "getX"    , (PyCFunction)PyComponent_getX    , METH_NOARGS
                    , "Return the Component X value." }
      , { "getY"    , (PyCFunction)PyComponent_getY    , METH_NOARGS
                    , "Return the Component Y value." }
      , { "getNet" , (PyCFunction)PyComponent_getNet , METH_NOARGS
                    , "Returns the net owning the component." }
      , { "destroy", (PyCFunction)PyComponent_destroy, METH_NOARGS
                    , "destroy associated hurricane object, the python object remains." }
      , {NULL, NULL, 0, NULL}     /* sentinel */
      };

    DBoDeleteMethod(Component)
    PyTypeObjectLinkPyType(Component)

#else  // Python Module Code Part.

    PyTypeInheritedObjectDefinitions(Component, Entity)

#endif  // Shared Library Code Part.

  }  // extern "C".
}  // Isobar namespace.
```

4.5 Terminal Class Header

The contents of this file is almost identical to 4.3 Intermediate Class Header, save for the presence of a `PyContact_Link()` function. She is present at this level because the class is a concrete one and can be instanciated.

```
#ifndef ISOBAR_PY_CONTACT_H
#define ISOBAR_PY_CONTACT_H

#include "hurricane/isobar/PyComponent.h"
#include "hurricane/Contact.h"

namespace  Isobar {
  extern "C" {

    typedef struct {
      PyComponent _baseObject;
    } PyContact;

    extern PyTypeObject  PyTypeContact;
    extern PyMethodDef   PyContact_Methods[];
    extern PyObject*     PyContact_Link       ( Hurricane::Contact* object );
    extern void          PyContact_LinkPyType ();

#define IsPyContact(v)     ( (v)->ob_type == &PyTypeContact )
#define PYCONTACT(v)       ( (PyContact*)(v) )
#define PYCONTACT_O(v)     ( PYCONTACT(v)->_baseObject._baseObject._object )

  }  // extern "C".
}  // Isobar namespace.

#endif  // ISOBAR_PY_CONTACT_H
```

4.6 Terminal Class File

Changes from 4.4 Intermediate Class File are:

1. As previously, we have to redefine the macros `ACCESS_OBJECT` and `ACCESS_CLASS`. But, as we are one level deeper into the hierarchy, one more level of indirection using `_baseObject` must be used.
   - `ACCESS_OBJECT` becomes `_baseObject._baseObject._object`.
   - `ACCESS_CLASS` becomes `&(_pyObject->_baseObject._baseObject)`.
2. For defining the `PyTypeContact` Python type, we call again `PyTypeInheritedObjectDefinitions()`. It is the same whether the class is terminal or not.
3. And, this time, as the Python class is concrete, we call the macro `DBoLinkCreateMethod()` to create the `PyContact_Link()` function.

```
#include "hurricane/isobar/PyContact.h"

namespace  Isobar {
  using namespace Hurricane;

  extern "C" {

#undef   ACCESS_OBJECT
#undef   ACCESS_CLASS
#define ACCESS_OBJECT            _baseObject._baseObject._object
#define ACCESS_CLASS(_pyObject)  &(_pyObject->_baseObject._baseObject)
```

```
#define METHOD_HEAD(function)   GENERIC_METHOD_HEAD(Contact,contact,function)

#if defined(__PYTHON_MODULE__)

    DirectGetLongAttribute(PyContact_getWidth , getWidth , PyContact,Contact)
    DirectGetLongAttribute(PyContact_getHeight, getHeight, PyContact,Contact)
    DBoDestroyAttribute(PyContact_destroy, PyContact)

    static PyObject* PyContact_create ( PyObject*, PyObject *args )
    {
      Contact* contact = NULL;
      HTRY
        // Usual signature then arguments parsing.
      HCATCH
      return PyContact_Link(contact);
    }

    PyMethodDef PyContact_Methods[] =
      { { "create"    , (PyCFunction)PyContact_create    , METH_VARARGS|METH_STATIC
                      , "Create a new Contact." }
      , { "destroy"   , (PyCFunction)PyContact_destroy   , METH_NOARGS
                      , "Destroy associated hurricane object, the python object remains." }
      , { "getWidth"  , (PyCFunction)PyContact_getWidth  , METH_NOARGS
                      , "Return the contact width." }
      , { "getHeight", (PyCFunction)PyContact_getHeight, METH_NOARGS
                      , "Return the contact height." }
      , {NULL, NULL, 0, NULL}   /* sentinel */
      };

    DBoDeleteMethod(Contact)
    PyTypeObjectLinkPyType(Contact)

#else  // Python Module Code Part.

    DBoLinkCreateMethod(Contact)
    PyTypeInheritedObjectDefinitions(Contact, Component)

#endif  // Shared Library Code Part.

  }  // extern "C".
}  // Isobar namespace.
```

4.8 Python Module

```
DL_EXPORT(void) initHurricane ()
{
  PyEntity_LinkPyType();  // step 1.
  PyComponent_LinkPyType();
  PyContact_LinkPyType();

  PYTYPE_READY( Entity )   // step 2.
  PYTYPE_READY_SUB( Component, Entity )
  PYTYPE_READY_SUB( Contact  , Component )

  __cs.addType( "ent"     , &PyTypeEntity    , "<Entity>"    , false ); // step 3.
  __cs.addType( "comp"    , &PyTypeComponent, "<Component>", false, "ent"  );
  __cs.addType( "contact", &PyTypeContact   , "<Contact>"   , false, "comp" );

  PyObject* module = Py_InitModule( "Hurricane", PyHurricane_Methods );
  if (module == NULL) {
```

```
      cerr << "[ERROR]\n"
           << "  Failed to initialize Hurricane module." << endl;
      return;
    }

    Py_INCREF( &PyTypeContact );                                              // step 4.
    PyModule_AddObject( module, "Contact", (PyObject*)&PyTypeContact ); // step 4.
  }
```

5. Case 3 - Non-DBo Standalone Classe

Let's have a look at the encapsulation of `Hurricane::Point`.

Non-BDo derived classes do not support the bi-directionnal communication. So each Python object is associated with one C++ object. The C++ object is created and deleted along with the Python one. This behavior implies that the C++ object is *copy constructible* (which should be the case).

5.1 Class Header

Changes from 3.1 Class Associated Header File:

- There is no `PyPoint_Link()` function, as it's related to the bi-directional communication mechanism.

> **Note**
>
> **About the _object attribute** of the PyPoint. As the C++ object life span (`Point`) is linked to the Python (`PyPoint`) one, we may have used a value instead of a pointer. It is best to keep a pointer as the macros written for `DBo` derived classes will remain usables.

```
#ifndef ISOBAR_PY_POINT_H
#define ISOBAR_PY_POINT_H

#include "hurricane/isobar/PyHurricane.h"
#include "hurricane/Point.h"

namespace  Isobar {
  extern "C" {

    typedef struct {
        PyObject_HEAD
        Hurricane::Point* _object;
    } PyPoint;

    extern  PyTypeObject  PyTypePoint;
    extern  PyMethodDef   PyPoint_Methods[];
    extern  void          PyPoint_LinkPyType();

#define IsPyPoint(v)     ( (v)->ob_type == &PyTypePoint )
#define PYPOINT(v)       ( (PyPoint*)(v) )
#define PYPOINT_O(v)     ( PYPOINT(v)->_object )

  }  // extern "C".
}  // Isobar namespace.

#endif  // ISOBAR_PY_POINT_H
```

5.2 Class File

Changes from 3.2 Class Associated File:

- As there is no `PyPoint_Link()` function, there is no call to any flavor of the `DBoLinkcreatemethod()` macro (obvious as it's *not* a `DBo`).
- To use the standard Python constructor, we have to define `PyPoint_NEW()` and `PyPoint_Init()` functions, I'm not absolutely certain that the later needs to be defined (that part is still not clear to me from the Python doc).
- As it's not a `DBo` there is no `destroy()` method, so no call to `DirectDestroyMethod()`
- Lastly, as this object has a `PyPoint_NEW()` (field `tp_new`) and a `PyPoint_Init()` (field `tp_init`) we have to use the macro `PyTypeObjectLinkPyTypeNewInit()` to define `PyPoint_LinkPyType()`.

```
#include "hurricane/isobar/PyPoint.h"

namespace Isobar {
  using namespace Hurricane;

  extern "C" {

#define METHOD_HEAD(function)   GENERIC_METHOD_HEAD(Point,point,function)

#if defined(__PYTHON_MODULE__)

    static PyObject* PyPoint_NEW ( PyObject* module, PyObject *args )
    {
      Point* point = NULL;
      HTRY
        PyObject* arg0 = NULL;
        PyObject* arg1 = NULL;

        __cs.init( "Point.Point" );
        if (not PyArg_ParseTuple( args,"|O&O&:Point.Point"
                                , Converter,&arg0
                                , Converter,&arg1 )) {
          PyErr_SetString ( ConstructorError
                          , "invalid number of parameters for Point constructor." );
          return NULL;
        }

        if      (__cs.getObjectIds() == "")
                { point = new Point()); }
        else if (__cs.getObjectIds() == ":point")
                { point = new Point( *PYPOINT_O(arg0) ); }
        else if (__cs.getObjectIds() == ":int:int")
                { point = new Point( PyAny_AsLong(arg0), PyAny_AsLong(arg1) ); }
        else {
          PyErr_SetString ( ConstructorError
                          , "invalid number of parameters for Point constructor." );
          return NULL;
        }

        PyPoint* pyPoint = PyObject_NEW( PyPoint, &PyTypePoint );
        if (pyPoint == NULL) { delete point; return NULL; }
        pyPoint->_object = point;
      HCATCH
```

```
    return (PyObject*)pyPoint;
  }

  static int  PyPoint_Init ( PyPoint* self, PyObject* args, PyObject* kwargs )
  { return 0; }

  DirectGetLongAttribute(PyPoint_getX,getX,PyPoint,Point)
  DirectGetLongAttribute(PyPoint_getY,getY,PyPoint,Point)
  DirectSetLongAttribute(PyPoint_SetX,setX,PyPoint,Point)
  DirectSetLongAttribute(PyPoint_SetY,setY,PyPoint,Point)

  PyMethodDef PyPoint_Methods[] =
    { { "getX"   , (PyCFunction)PyPoint_getX     , METH_NOARGS
                 , "Return the Point X value." }
    , { "getY"   , (PyCFunction)PyPoint_getY     , METH_NOARGS
                 , "Return the Point Y value." }
    , { "setX"   , (PyCFunction)PyPoint_SetX     , METH_VARARGS
                 , "Modify the Point X value." }
    , { "setY"   , (PyCFunction)PyPoint_SetY     , METH_VARARGS
                 , "Modify the Point Y value." }
    , {NULL, NULL, 0, NULL}  /* sentinel */
    };

  DirectDeleteMethod(PyPoint_DeAlloc,PyPoint)
  PyTypeObjectLinkPyTypeNewInit(Point)

#else  // Python Module Code Part.

  PyTypeObjectDefinitions(Point)

#endif  // Shared Library Code Part.

  }  // extern "C".
}  // Isobar namespace.
```

5.2 Class File

To put it bluntly, there is no difference in the Python module for a standalone `DBo` class and a non-`DBo` class.

6. Encapsulating DbU

While `Hurricane::DbU` is a class, the `Hurricane::DbU::Unit` is only a `typedef` over `uint64_t`. The `DbU` class only provides a set of static methods to manipulate and convert to and from other units. At Python level, `DbU::Unit` will be stored in plain `long long`.

When a `DbU::Unit` argument is expected in a Python functions, just use the `DbU::Unit PyAny_AsLong( PyObject* )` function to convert it.

For example, if we explicit the expension of:

```
DirectSetLongAttribute(PyPoint_SetX,setX,PyPoint,Point)
```

We would get:

```
static PyObject* PyPoint_setX ( PyPoint *self, PyObject* args )
{
  Point* cobject = static_cast<Point*>( self->_object );
  if (cobject == NULL) {
    PyErr_SetString( ProxyError
                   , "Attempt to call Point.setX() on an unbound Hurricane object" );
    return NULL;
  }

  HTRY
    PyObject* arg0 = NULL;
    if (not PyArg_ParseTuple( args, "O:Point.setX()", &arg0 ))
      return ( NULL );
    cobject->setX( Isobar::PyAny_AsLong(arg0) );
  HCATCH
  Py_RETURN_NONE;
}
```

For the other way around, use `PyObject* PyDbU_FromLong( DbU::Unit )`.

```
DirectGetLongAttribute(PyPoint_GetX,getX,PyPoint,Point)
```

We would get:

```
static PyObject* PyPoint_GetX ( PyPoint *self, PyObject* args )
{
  Point* cobject = static_cast<Point*>( self->_object );
  if (cobject == NULL) {
    PyErr_SetString( ProxyError
                   , "Attempt to call Point.getX() on an unbound Hurricane object" );
    return NULL;
  }
  return Isobar::PyDbU_FromLong(cobject->getX());
}
```

7. No C++ Hurricane::Name encapsulation

To be written.