Proposition

Python Mapping for OpenAccess

LSI Logic Corp.,
December 7, 2004

1 Introduction

This document describes the mapping for Python to the OpenAccess 2.2 API. There are 2 main goals of the mapping. The first goal is to make each Python call as similar as possible to the C++ API call. Thus, OpenAccess class names and member function names shall be preserved (when ever possible). The second goal is to make the API calls natural to Python (were possible), this include using native object types and native syntax as appropriate.

From this mapping and the OpenAccess documentation it should be possible to determine the proper Python statement for each API call.

The Python for OpenAccess API is implemented as module in Python. All functions, classes, and constants shall be a Python module named **openaccess22**. This module is accessed using the native module **import** statement in Python.

This mapping is very similar to the Python for OpenAccess 2.0 mapping. The only major change is the way static members have been named.

2 Classes

The OpenAccess API is classes based. Since Python is also class based, the Python mapping will take advantage of this and be class based.

2.1 Native Python classes

Typedef names in OpenAccess that correspond to fundamental types in C++ (such as int, float, double, etc.) will have a Python object type associated with it. This Python object type will be a new unique type in Python (referred to here as the Python-OpenAccess object type), different from the native Python object types. Introducing new Python object types for these OpenAccess types allows direct mapping of all function parameters and return values to these types.

These Python-OpenAccess object types are true Python types, but with only a limited number of member functions. These types do not support math operation directly. To use them in math operations, they must be converted to the native Python types. Most of the time, this conversion will happen automatically for you. A specific type conversion is only needed if both operands in a math expression are Python-OpenAccess object types. Given below is the list of member functions available on Python-OpenAccess object types.

Member Function	Description
assign(value)	Assign a new value to the object. This

	changes the value of the object. This differs from the = operator, that just creates a new reference an existing object.
int()	Available for oaBoolean, oaInt1, oaInt2, oaInt4, oaUInt2, oaUInt4, oaLayerNum, oaPurposeNum. It returns the equivalent value as a native integer value.
long()	Available for oaInt4, oaInt8, oaUInt4, oaUInt8. It returns the equivalent value as a native long integer value.
complex()	Available for oaComplex_oaDouble and oaComplex_oaFloat. It returns the equivalent value as a native complex value.
str()	Available for oaChar and oaByte. It returns the equivalent value as a native string value.

These Python-OpenAccess object types have modifiable values. So they differ from the native Python types that are immutable. This is required, so that pass by reference parameters can be supported. The consequence of this is that using these values in assignment statements does not generate a new copy of the value, but an additional reference to the same value.

When these typedef names are used in as function parameter type, the value passed from Python should be one of these Python-OpenAccess object types. If the Python-OpenAccess object type as an equivalent native Python object, then a native Python object can be passed, and it will be automatically converted to the Python-OpenAccess object type. Note, that since a new object is created in the process, this only works for input type parameters, parameters that are pass by reference will not modify the original native Python object; only the temporary converted value gets modified.

Given below is a table of the OpenAccess typedefs and the equivalent native Python objects that get automatically converted.

OpenAccess typedef	Python-OpenAccess type name	Python Equivalent
oaString class	oaString	Native string
oaByte	oaByte	Native string of length 1.
oaByte*	oaByte_Array	Native string
oaChar	oaChar	Native string of length 1.
oaChar*	oaChar_Array	Native string
oaBoolean, oaInt1, oaInt2, oaInt4, oaUInt2, oaUInt4, oaCoord, oaDist	oaBoolean, oaInt1, oaInt2, oaInt4, oaUInt2, oaUInt4, oaCoord, oaDist	Native integer with the proper range checking on the value.
oaInt8, oaUInt8	oaInt8, oaUInt8	Native long-integer.

oaDouble, oaFloat	oaDouble, oaFloat	Native float value
oaComplex<oaDouble>, oaComplex<oaFloat>	aaComplex_oaDouble, oaComplex_oaFloat	Native complex value
oaTime	oaTime	Native float value. (Python functions that deal with time similarly to time_t use floats.)
oaLayerNum, oaPurposeNum		Native integer

2.2 Python classes

Each OpenAccess class (except for a few special cases) will have a corresponding Python object type associated with it. The following classes are exceptions:

Class	Comment
oaPcellEvalObserver, oaPcellObserver, oaTextDataCallback	No user access is provided for these types. These are used internally to provide Pcells implemented in Python.
oaEnumIterState, oaTextDisplayIterState	No user access is proved for these types. Corresponding iterators are provided.
oaMemory	No user access is provided for these types.
oaException, oaDBException, oaDBError, oaDBFatal, oaError, oaFatal, oaOSError, oaTcException, oaTCError, oaTCFatal	No user access is provided for these types. C++ exceptions are caught, and turned into Python exceptions.
oaPlugIn, oaPlugInError, oaPlugInMgr, oaPlugInMgrCleaner	No user access is provided for this base types.

2.3 Object ownership

OpenAccess C++ classes fall into two categories: utility objects, and managed objects. For utility objects, the application owns the object (and its memory), for managed objects the database owns the object (and the related memory). Applications never create managed objects, they just use pointer to the objects that are obtained from other API calls. Memory management of the objects is all handled within the Python API for the user. At the Python level all object access is treated as objects and not pointers. Both utility objects, and managed object appear the same. Note that there are no constructors for managed objects.

The wrappers for managed object act as if the object is the dereferenced pointer. Thus dereferencing is not required (there is no Python equivalent anyways).

Examples:

C++ Code	Python Code
`oaBox bx;` `bx.set(0,1,2,3);`	`bx=oaBox();` `bx.set(0,1,2,3,4);`

`oaCellView`
`*cv=oaCellView::open(…);`
`cv->isModified();`
	`cv=oaCellView.open(…)`
	`cv.isModified()`

2.4 Member functions

Public non-static member functions in C++ map to the same member function name in Python. All public non-static member functions of base classes are available as member functions in Python.

In addition to the OpenAccess member function, each class supports the following member functions:

Member Function	Description
assign(value)	Available for all objects that support operator=. Assigns a new value to the existing object. This changes the value of the object. This differs from the = operator, that just creates a new reference an existing object. The returned value is the existing object. For managed objects this just copies the pointer value.
isNull	Available for managed objects. Returns true if the object is a NULL pointer.

2.5 Static Member functions

Static Member functions map to regular functions in the Python. To make these look like static member functions, they are created as special attributes of the Python type object. This will make them callable as members of the object type. Note that in Python they can't be called from an object, only from the type (See examples below). The names of static member functions are prefixed with 'static_', this is done to avoid name conflicts with non-static members of the same name.

Examples:

C++ Code	Python Code	Comment

`oaCellView *cv=`
 	
`oaCellView::open(…);`
	`cv=oaCellView.static_open(…)`	Valid Python Code

`oaCellView *cv=…;` `cv=cv->open(…);`	`cv=…` `cv=cv.static_open(…)`	Invalid Python Code

2.6 Constructors

C++ constructors map to calls to the equivalent Python constructor calls. In Python these are really calls to the object type with parameters.

The constructors for managed objected create a NULL object. This object can be used for calls that take a reference to pointer parameter.

Examples:

C++ Code	Python Code	Comment
`oaBox bx(0,1,2,3);`	`bx=oaBox(0,1,2,3)`	Valid Python Code
`oaBox bx;`	`bx=oaBox()`	The empty parameter list is required for Python.

`oaCellView* cv=…;`
`oaLayerNum num=…;`
`oaTech* tech=NULL;`
`oaLayer* layer=`
`  oaLayer::find(`
`  cv,num,tech)`
	`cv=…`
	`num=oaLayerNum(…)`
	`tech=oaTech()`
	`layer=oaLayer.find(`
	`  cv,num,tech)`
		Create an NULL oaTech, for use in the pointer-reference parameter to oaLayer::find.

2.7 Destructors

C++ destructors are called for utility objects when the corresponding Python object gets deleted (which normally happens when a variable goes out of scope).

2.8 Base Object Types

For OpenAccess functions that return a pointer to some base object type (of a managed object) the Python call will always return the object that is of the true objects type (as apposed to the base object type). For example, if oaRegionQuery is used to return all the oaShapes in a given area, then in Python the user will be back objects such as oaPath and oaRect, and not oaShape objects.

3 Functions

Functions calls in C++ map to function calls in Python. This applies to global functions, member functions, static member functions, and class constructor calls.

3.1 Parameters

C++ calls support pass by reference, which allows output (or return value) parameters. Python's parameter passing mechanism is a pass by object, which is compatible with the C++ pass by reference mechanism. However, some Python object types are immutable, in that their values can't be changed. Immutable object types include the basic types of integer, float, and string. Thus, objects of these types can't be used in a pass by reference call for a return value. Other extension languages, such as TCL would have a similar problem. To work around this limitation, object types are created for all the OpenAccess typedef types (like the oaInt types). These are created as mutable types, and can be used in pass by reference. If the C++ call is pass by value or const reference, the equivalent native Python type can be used, and will be converted to the required

type automatically. [Note this happens on pass by reference calls also, but the modified value is lost].

Some parameters are pointers to utility objects. For these objects, the pointer of the object will be passed to the routine. If a NULL parameters is to be specified, the Python value None can be passed for these parameters.

3.2 Function Return Values

The resulting Python output from a function call will be the function return value as an OpenAccess type.

Examples:

C++	Python	Comment
`oaFile file(…);` `file.remove();`	`file=oaFile(…)` `file.remove()`	Void return value in C++. **None** is return in Python.
`oaFile file(…);` `oaBoolean b=file.exits();`	`file=oaFile(…)` `exits=file.exists()`	Single return value in C++. Single return value in Python.

3.3 Function overloading

In C++ many of the functions are overloaded with different parameters. All of these cases can be determined based on the parameter types. For these cases, Python will do dynamic type checking to determine the proper case to call.

3.4 Function returning references and pointers

Some OpenAccess calls return reference values (or pointers) that are intended to allow the application to modify values within a class. For example, the oaPoint class has member functions x() and y() that return references to the internal x and y values. Most (*all?*) of these cases have 2 versions of the function: one that returns a constant reference and one the returns a non-constant reference. Only the reference return value function is mapped to python. The constant reference return value function is not mapped. Changes to the return value object will change the original object.

C++	Python	Comment
`oaPoint pt;` `value=pt.x()*2;`	`pt=oaPoint()` `value=pt.x()*2`	Read access to the x value.
`oaPoint pt;` `pt.x()=7;`	`pt=oaPoint()` `pt.x().assign(7)`	Write access to the x value.

3.5 Operator functions

Only some C++ operator functions in OpenAccess are mapped to Python. The operator[] function is mapped to the list lookup operation in Python. This results in a similar Python syntax. In OpenAccess most of the operator[] functions have two

functions: one that returns a reference, and one that returns a const reference. Only the non-const reference return value function is mapped to Python. Thus the Python call will return a reference object, that can be used for both read and write access.

C++	Python

`oaPointArray pta(….);`
`oaPoint pt;`
`pt=pta[1];`
`pt.x()=7;`
`pta[2]=pt;`
	`pta=oaPointArray(…)`
	`pt=oaPoint()`
	`pt=pta[1]`
	`pt.x().assign(7)`
	`pta[2].assign(pt)`

For the class oaPoint, the Python math operators -, +, -=, +=, and **abs** are allowed. These are mapped to the corresponding oaPoint functions (such as operator+ and operator-).

Other operator functions, such as the compare operators on oaString are not supported. The native compare operators, on the native object types can be used instead.

3.6 Array parameters

Some OpenAccess calls take a pointer to an array of objects and a count (2 parameters). For these functions Python will require a single parameter that is a special Array type. The type of the object is named with the base type with an extension of *_Array*. These are special types introduced into the mapping.

These array types have access members that act like a python list (operator[], and len). The constructors for these objects can be called in 1 of 3 ways:

1: With and existing _Array object, this will result in a new copy.
2: With an integer value, this will create an array of the specified.
3: With a list of objects, this will copy all the values from list.

Parameters that take an _Array object, can be passed an object of type _Array type, or a python list of objects. If a Python list is given, it is converted to a temporary _Array type object. Auto conversion is only useful if the array is in input parameter that will not be modified.

C++	Python

`oaPoint pts[5];`
`pts[0]=….;`
 	
`oaPointArray pta(pts,5);`
	`pts=opPonit_Array(5)`
	`pts[0]=…`
	
	`pta=oaPointArray(pts)`

The _Array type objects support the following member functions:

Member Function	Description
assign(value)	Assigns a new array value to the existing object. This changes the value of the object. This differs from the = operator, that just creates a new reference an existing object. The returned value is the existing object. For managed objects this just copies the

	pointer value.
list()	Returns a Python list consisting of the objects in the _Array object. This is a list of references to the objects in the _Array object, so the _Array can be modified.

3.7 Array return values

Some OpenAccess return a pointer to an array of objects along with an additional member function that returns the count. For these functions Python will return a _Array object (see Array Parameters).

C++	Python

`oaPointArray pta(…);`
`oaUInt4 size=pta.getSize();`
`oaPoint* points=pta.getPoints();`
`oaUInt4 index;`
`for(index=0;index<size;index++) {`
`  oaPoint pt=points[index];`
`  ..`
`}`
	`pta=oaPointArray(…)`
	`points=pta.getPoints()`
	`size=len(points)`
	
	`for index in range(size):`
	`  pt=points[index]`

4 Templated Classes

The OpenAccess API uses many templated classes. Python does not have templated classes. To support these in Python, typedefs are introduced for all valid template parameters. These types are named by concatenating the typeplate name and the templated parameters separated by underscore(_). Each of these specific templated cases is then implemented as a Python type just like a normal class is. A Python variable with the name of the original template class is created as a dictionary. This dictionary is indexed by the templated type and returns the full expanded type. This results in a Python syntax that is close to the C++ templated calls:

Examples:

C++	Python
`oaInter<oaNet> ext(…..);`	`ext=oaIntExtension[oaNet](…)`

5 Collections and Iterators

The OpenAccess API uses the templated oaCollection and oaIter classes. These can be used directly using the templated classes interface. To assist in using these more naturally in Python, the oaIter class (all templated versions) has a **next** member added so that the class behaves properly as a Python iterator class. Also, for any class member function that returns an oaCollection, there is an additional member function added (named with the original name plus a suffix of **Iter**) that returns an oaIter object directly. This greatly simplifies using iterators in Python.

Examples:

C++	Python
`// Sample C++ code`	`# Direct Python coding`

oaCellView* cv=….;
oaIter<oaNet> itn(cv->getNets());
oaNet* net;
while (net=itn.getNext()) {
  …
}
	cv=…. # some cellview
	itn=oaIter[oaNet](cv.getNets())
	net=itn.getNext()
	while (net!=None):
	  ….
	  net=itn.getNext()

	# With added the added **next**
	# member in oaIter class
	cv=…. # some cellview
	itn=oaIter[oaNet](cv.getNets)
	for net in itn:
	  …

	# With **Iter** class members
	cv=…. # some cellview
	itn=cv.getNetsIter()
	for net in itn:
	  …

6 Exceptions

Any exception thrown by an OpenAccess call is caught and turned into a Python exception. The name of the python exception is **openaccess**.**error**.

? What to do with Python exceptions returned to OpenAccess?

7 Virtual Functions

Some OpenAccess classes have virtual functions that are intended to be overridden by the user in a derived class. Those classes with virtual function that are available from Python, allow the user to derived a user class from the built in Python class, and then implement function with the same name as the virtual function. These Python functions will be called as the virtual functions from OpenAccess.

The following classes support virtual functions in Python:

OpenAccess classes with virtual functions that are supported by Python	Comment
oaRegionQuery	Used for area searching
All oa*Callback classes	All the callback classes.

8 Pcell Interface

?Access functions for Pcells and EvalText are TBD?

10 Limitations in current implementation

Functions that return const pointers to objects are currently not supported.