IT 09 036
Examensarbete 30 hp
November 2009

Python Integration with a Functional DBMS

Hanzheng Zou

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Python Integration with a Functional DBMS

Hanzheng Zou

Python is an Object Oriented programming language and widely used nowadays. This report describes how to extend a functional database system Amos II for integration with Python. Several possibilities are analyzed to combine the Amos II C external interfaces with those of Python. Based on these discussions, new functionality has been added to the Python language by implementing a Python C external module. A basic API called PyAmos, interfacing Python and Amos II, is proposed and implemented in this work. To utilize object oriented nature of Python, some new Python classes are also defined for Amos II database access. The performance of PyAmos interface is also evaluated. This work shows how Python can be integrated with a functional DBMS.

Handledare: Tore Risch
Ämnesgranskare: Tore Risch
Examinator: Anders Jansson
IT 09 036
Tryckt av: Reprocentralen ITC

Index

1. Introduction

Amos II [25] is a functional database system where functions are used both to represent data, and to query and update the database. The database queries are expressed in AmosQL [24], a functional query language similar to the object-oriented (OO) parts of SQL-99. Amos II can be used as a stand-alone main-memory functional DBMS using AmosQL. The system can furthermore execute functional queries over federations of other databases distributed over the internet.

The purpose of this project is to design and implement an API between the programming language Python and Amos II, called *PyAmos*. Python is scripting language for building, e.g., interactive web pages and scientific applications. Python may, e.g., run inside a Python-enabled web server but can also run as a stand-alone interpreter. Python can be extended by calling different kinds of external systems. For example, APIs between Python and various relational DBMSs have been defined [21]. In this project the ability of Python to call other systems is used for tightly extending Python with new functionality for calling Amos II functions. The functional nature of Amos II makes is straight forward to enable transparent calling of Amos II database functions as Python functions.

There are two kinds of interfaces between Amos II and the programming languages C and Java called the *callin* and the *callout* interfaces [23]. Generally speaking, the *callin* interface allows systems written in C or Java to call Amos II, while the *callout* interface allows the programmer to define *foreign* Amos II functions in Java or C. The foreign functions can then be freely used in database queries. The external interfaces permit the development of access modules to external data, called *wrappers*. For example, wrappers to relational databases, web forms, XML documents, MIDI files, and web browsers have been implemented, and they are all available in the Amos II project wrapper overview page [1].

The interface between Amos II and other systems can either be based on client-server communication using TCP-IP or a *tight connection* where both Amos II and the external system run in the same address space. The client-server interface is slower since TCP-IP communication is involved. Similarly, PyAmos allows having either a *tight* connection between Python and Amos II, where the Amos II database is running inside the Python engine as a subsystem in the same address space, or a more conventional client-server connection to a separate Amos II database server.

Since Amos II is an object-oriented system it possible to pass DBMS managed OIDs between the Python-based application and the Amos II database. It is investigated how to manage such OIDs being passed to and from the embedded Amos II system.

The functional nature of Amos II is exposed in Python. The dynamic nature of Python makes it possible to design an interface on a much higher level than the interface between C or Java and

Amos II. For example, functional programming constructs in Python such as lambda expressions and map functions can be used for operating over large query results from Amos II.

Performance is an issue, in particular when transporting high volumes of data between Python and Amos II. Careful choice of Python-provided data structures for basic data elements is important. Measurements are made to evaluate the best performance and how the interface performance scales when the volume of interchanged data increases.

Both Python and Amos II have automatic garbage collectors that must interoperate so that all temporary memory used by the embedded Amos II is released when no longer referenced from Python. Furthermore, the error management of Amos II is integrated with Python's error management.

The rest of this report is organized as follows.

In chapter Two, the technical background is discussed, including comparisons of Python and PHP, Python and Java, and introduction to the Amos II system.

The PyAmos system is presented in Chapter Three. The chapter includes a description of the architecture of PyAmos, including how to pass OIDs, implement Amos II types as Python types, handle errors, and integrate the garbage collectors.

Chapter Four is about performance evaluations of the PyAmos interface. Finally chapter Five summarizes and discusses future work.

2. Background

The technical background is discussed in this chapter, including relevant DBMS technologies, a Python language introduction, comparison between Python and the similar language PHP, and the functional DBMS Amos II.

2.1 Database Management Systems

A Database Management System (DBMS) is a set of software programs that are used to store, update and retrieve a database. Some of the most popular DBMS solutions in the markets are Microsoft SQL server [13], DB2 [9], Oracle [17], and MySQL [14]. The DBMS Amos II [25] is used in this project.

2.1.1 Data Model, Schema and Query language.

A *data model* of a database is a language for describing how a database is structured. The most

common data model is the relational data model. In the relational data model data is represented as tables.

The description of the contents of a database is called the database *schema*. For a relational database it is a set of meta-data that describes the tables storing the database. A database schema is usually specified when the database is designed and it can be changed as the database evolved. The data model provides the primitives to represent the schema. For example, in relational databases the data model consists of tables, in the object-oriented data model it consists of classes and methods, and in the Amos II functional data model it consists of type and function definitions.

Query languages are very high level computer languages used to make queries to databases. The most known common query language for relational databases is SQL [6]. Amos II uses the query language AmosQL [23], which is a functional query language based on OSQL [18] and DAPLEX [5] with some extensions.

2.1.2 Database Interfaces

Database Application Programming Interfaces (APIs) are used for interfacing application programs with a DBMS. Examples of DBMS APIs are ODBC [16] and JDBC [10]. Database APIs provide *database connections* to let a client program establish a session with the database server. It is usually specified by a connection string, which is addressing a specific database or server. A *database cursor* or a *scan* comprises a structure for the traversal and processing of records in a result set from a query sent to the DBMS through the API. User can get, put, and delete database records by using cursors. Both ODBC and JDBC provide standard software API for using DBMS. *Prepared statements* in ODBC and JDBC provide a way to dynamically compile SQL statements to save query optimization time if the same query is executed repeatedly.

Amos II provides the JavaAmos interface [4] between the programming language Java and Amos II. Similarly, the PHP-Amos interface [3] provides a simple Amos II API for the PHP language. The PyAmos system provides a similar API for interfacing Python programs with Amos II databases. Both the JavaAmos, PHP-Amos, and PyAmos interfaces are based on the C native interface of Amos II [23]. By making use of the dynamic features and the system libraries of Python, the PyAmos is as simple as PHP-Amos, more dynamic than JavaAmos, and almost as fast as the C native interface. The performance of PyAmos compared to the other Amos II APIs will be elaborated in the coming chapters.

2.2 Python

Python [20] is a general purpose high level programming language. It is quite simple to use, and the code is easy to read since indentation is used as block delimiters. With a large and comprehensive standard, Python has become a popular programming language.

2.2.1 Python Dynamic Nature

Python is a multi paradigm programming language. The programmers are not forced to adopt a particular programming style. Not only object-oriented programming and structured programming are supported, it also permits developers to use functional programming and aspect-oriented programming. In the present work mainly the functional and object-oriented Python styles are used.

Python is as simple to use as UNIX shell scripts or Windows batch files, but it is much more powerful than these primitive scripting languages. It is a complete programming language, which offers structures, functions, and modularization to support building large programs. Python furthermore offers as fully dynamic features as a scripting language and dynamic error handling. As a high level language, some high level data types such as flexible arrays and dictionaries are built in.

Python is an interpreted language, which saves the programmer considerable development time since there is no need for compilation and linking. It is quite easy to make experiments with different features of the language by using the interpreter interactively.

As a high level programming language, Object-Oriented features are also fully supported. Python *classes* are used to define new data types matching the real world. All Python symbols are objects with attributes. All the modules are also objects, which could contain other objects inside; thus the Python's namespace is nested. All the Python objects are automatically reclaimed by an automatic garbage collector when no longer needed, similar with other modern languages like Java, Perl [11], PHP [19], or Ruby [2].

Each Python code object is documented by a comment string that is placed as the first non-remark declaration in a module, class or function. The comment string can be accessed at run time from Python as '__doc__' attributes of objects, modules, classes, or function objects.

An important feature utilized in this project is that Python is quite extensible. New modules and language extensions can be easily written in C or C++. It makes Python a tool for system integration and lets Python make use of C and C++ legacy code. Some modules that need complex algorithms or calculations can be efficiently implemented in C or C++ and then easily integrated into Python. This will elaborated later in this report, when the PyAmos interface is described.

2.2.2 Using a DBMS in Python

Python has API modules to interface relational databases [22], the current version called DBAPI-2.0. Python also has database interfaces modules for some specific relational database systems, such as IBM DB2, MySQL, Microsoft SQL server, and Oracle.

This is an example of using DBAPI-2.0 to call SQL:

```
>>> import psycopg
>>> conn = psycopg.connect("dbname=test user=test")
>>> curs = conn.cursor("DROP TABLE atable")
>>> conn.commit()
```

By using the standardized database API, the Python code is more portable across relational databases. PyAmos provides a similar functional database API utilizing the functional nature of both Python and Amos II to query and update Amos II databases and data sources accessible through Amos II.

2.2.3 Python vs. PHP

Compared with PHP, which is a web-oriented programming language that stands for 'Hypertext Processor', Python is a general purpose programming language with fewer web-specific features. 'Will Python be used instead of PHP in the future?' This is a much disputed question and has drawn many developers' attentions. In principle, we can say 'yes' because in most cases where PHP can be used, Python can be used instead.

Python and PHP have the following in common:

- Both Python and PHP are interpreted languages, usually called 'scripting languages'. They are high level languages with dynamic typing. In most cases they can be run in all operating systems without recompilation.
- Both languages are open source languages supported by large developer communities.
- Both languages are easy to learn.
- Both languages are extensible and the developer can easily extend them using C, C++, or Java.
- Python is an Object oriented language. OO features have also been recently added to PHP.

When it comes to Web applications, Python still has a long way to go to 'beat' PHP:

- Python is not a template language, which means that you cannot easily mix Python code with HTML, as is easy to do in PHP.
- Up to now, most of the inexpensive web-hosters support PHP, but not Python. This situation might be solved with the development of new hardware, but still needs time.
- There are many more pre-written Web scripts in PHP than in Python.

Despite of the features stated above, both Python and PHP have low entry barriers and are easy to learn. One special thing in Python is to use indentations as a form of syntax. This makes the code look clean and tidy, but the developers from other language like C, C++ or Java might think Python is kind of 'neat freak'. Wrong indentations will cause syntax errors or even bugs in the code. Especially if there are nested loops in the code, wrong code indentations may lead to totally different execution logic. Therefore Python's indentation syntax is not as clear as the '{}' used in C++ or Java code.

Python is an OO language and it supports both procedural coding and class definitions. PHP also has OO support in the new versions, but the big problem is that PHP has quite awful backward compatibility [15] which is never existed in Python.

Python also have some features that PHP doesn't have. Python supports namespaces and modules, which are currently in the PHP development plan. Python has threading support, unlike PHP. For general purpose development Python is a more powerful language than PHP. The biggest limitation of using Python in web development is current lack of web-hosters, but this will perhaps be not a problem with development of hardware and web-hoster technologies.

2.2.4 Python vs. Java

Python and Java do have some features in common. Both Python and Java are general purpose programming language; both of them support object-oriented features and have large numbers of useful libraries. The difference is that Java is a static typed language, while Python is dynamic typed as PHP. Java is called a static typed language, since in Java, all the variables must be explicitly declared. You can't declare an integer variable and later assign it a string value. Exceptions are raised if wrong type assignments occur. By contrast Python is known as dynamic typed language where the user doesn't need to declare anything. Any assignment will bind a variable name to a Python object, and the object could be of any type. You can also assign a variable to a type and later assign it to some other different object type.

Python also supports variable number of function parameters. It means you can pass a dynamic number of parameters to a Python function. During run-time, the Python interpreter can check how many parameters are passed and then interpret them dynamically. This is used in PyAmos to provide convenient database access. By contrast, in Java the method argument declarations are static and resolved by the compiler.

Finally, Java doesn't support lambda functions and a functional programming style, while Python does, which will be elaborated later.

2.3 Amos II

Amos II (Active Mediator Object System) is a light-weight and extensible database management system (DBMS) with an object oriented and functional data model. Amos II uses AmosQL, a relationally complete functional query language, to make queries and views to the database.

Amos II provides an intermediate system level between different kinds of data sources and applications. A *wrapper* is a program module that can query data from different classes of external data sources, such as a Web services [12] or relational databases [8], in terms of the functional data model in Amos II. There are interfaces between several programming languages and Amos II, e.g. Lisp [23], C/C++ [23], Java [4] or PHP [3]. In this project the external C interface

is mainly used. It allows Python programs query and update Amos II databases.

Amos II can access data stored in a local Amos II database as well as data in wrapped external data sources. The facilities to access data in wrapped data sources make Amos II extensible. Through the wrappers new applications oriented data types can easily be accessed.

The basic data model of Amos II provides types, functions, and objects. The model is functional but supports OO representations such as inheritance, instantiation, dynamic binding and so on.

Types

To match with the object oriented nature, the types in Amos II are similar to the concept of 'class' in OO programming languages. The types are organized in a multiple inheritance, super-type and sub-type hierarchy. Figure 2 illustrates the basic type hierarchy that is offered by Amos II.

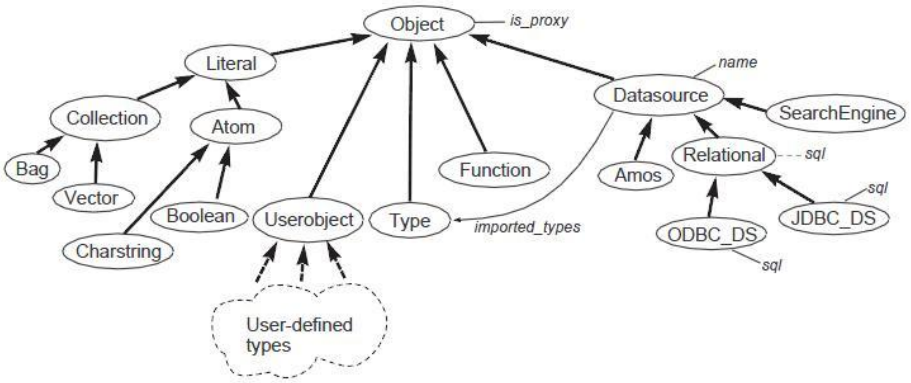


Figure 2.1: Amos II system type hierarchy

Users can define their own types. The user-defined types may have properties represented as functions, analogous to attributes of class definitions, for example:

```
create type Person properties (name Charstring,birthyear Integer);
create type Supervisor under Person;
create type Student under Person properties (class Charstring, su Supervisor);
```

User-defined types like *Person*, *Supervisor* and *Student* are always subtypes of the system type *Userobject*.

Functions

Functions model properties of objects, computations over objects, and relationships between objects. They are instances of system built in type named *Function*. They provide the basic primitives in functional queries and views. Basically, there are two parts in a function, the *signature* and the *implementation*. The signature defines the function name, the arguments and results types; and the implementation specifies how to do the computations of the function.

There are four types of Function in Amos II.

- *Stored functions*, represent properties of objects similar to attributes of a class. For example,

  create function name(Student) -> Charstring name as stored;.

- *Derived functions* are functions defined in terms of functional queries over other functions. For example,

  create function topstudents(Student p) -> Charstring nm
  as select name(p) where grade(p)<90;

- *Database procedures* are defined by a procedural sublanguage of AmosQL. For example,

  create function helloworld (Charstring name)->Boolean as
  begin
  print('Hello world '); print(name); result true;
  end;

- *Foreign functions* provide the possibility to define Amos II functions in other programming languages, such as JAVA, C/C++, or Lisp. This also provides the basis for wrapping external systems. For example,

  create function myforeignfunc(Charstring x) -> Real as foreign 'myforeignfuncbf';

Amos II functions can furthermore be overloaded on the argument types. Each different overloaded implementation is called a *resolvent*.

Objects

Objects model all entities in the database. The object-oriented nature makes Amos objects fundamental to the system. There are two main types of object in Amos system: *literals* and *surrogates*. Literal objects are system maintained object which have no explicit OIDs while surrogates are managed by the user using OIDs. All user-define objects, types, and function entities are surrogates. The instances of types or functions are all surrogate objects.

Queries

Generally speaking, AmosQL queries are formulated through the *select* statement. The syntax is:

```
select <result>
from <type extents>
where <condition>
```

For example:

```
select name(p), birthyear(p)
from Person p
where birthyear(p) > 1970;
```

The *from* keyword specifies the types whose instances you want to make a query to. The condition of the queries (the *where* clause) define the search condition as a restriction of the cartesian product of the extents of the types in the *from* clause.

3. The PyAmos Interface

As a modern interpreted programming language, new built-in modules can be added to Python by C programming through the Python C APIs [21]. The Python extension modules are quite powerful: you can implement new built-in object types, call C library functions, or make use of legend C code. New functionalities can be added to Python by creating interfaces to existing code.

The Amos II system has analogous C-based *callin* and *callout*-interfaces [23], which make it possible to interface Amos II with Python by calling the Amos II C interfaces in the extension modules. This is how PyAmos is implemented.

In this chapter, the technical details of PyAmos is discussed, including how to extend Python using C code, how to pass OIDs between Python and Amos II, how to transform Amos II types like *integers*, *strings*, and *vectors* into corresponding Python types, how to integrate the error management, and how to integrate the automatic garbage collectors of Python and Amos II.

3.1 System Overview

Figure 6 shows the architecture of PyAmos:

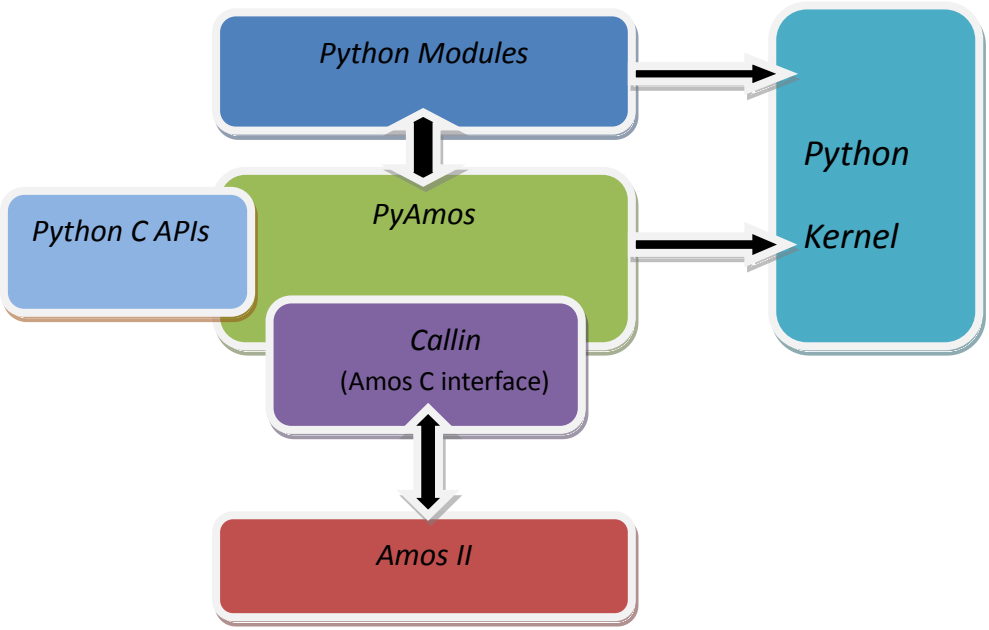


Figure 3.1: PyAmos system architecture

PyAmos consists of the following components:

- The *Amos II* kernel works in the backend of the system, and the Python interpreter connects to Amos II through the PyAmos interface.

- The *callin* interface is called by the PyAmos extension module to execute user calls to Amos II from Python.
- The *PyAmos* module is the core of the project. It makes use of the Python C APIs to define an extension module of Python, and calls the Amos C callin interface to interact with Amos II. The binary file is compiled and built as a dynamic object library 'amospy.pyd'. It can be called directly from the Python interpreter.
- The *Python modules* are a part of PyAmos written in Python to realize some features like map functions.
- The *Python kernel* provides the interpreter, compiler, and run time system for Python. Both the Python modules and the C extension module are executed by the Python kernel.
- The *Python C API* will be elaborated later.

Basically all the functions from the C *callin* interface of Amos II are supported by PyAmos. The following is an example of how to run PyAmos from the Python top loop:

```
>>> import amospy
>>> conn = amospy.amos_connect("")
>>> scan = amospy.amos_execute(conn,"select name(t) from type t;")
>>> while amospy.amos_eos(scan)!= True :
        row = amospy.amos_getrow(scan)
        print row
        amospy.amos_next(scan)
```

The above code snippet first loads the '.pyd' extension module into the Python interpreter, then uses the *amos_connect()* function to connect to a local Amos II database running in main memory inside Python. After that a query to get all type names in the local database is executed by the *amos_execute()* function. The result is returned as a scan, and then the while loop will iterate through the scan and print the type names. The following picture shows how the script runs:

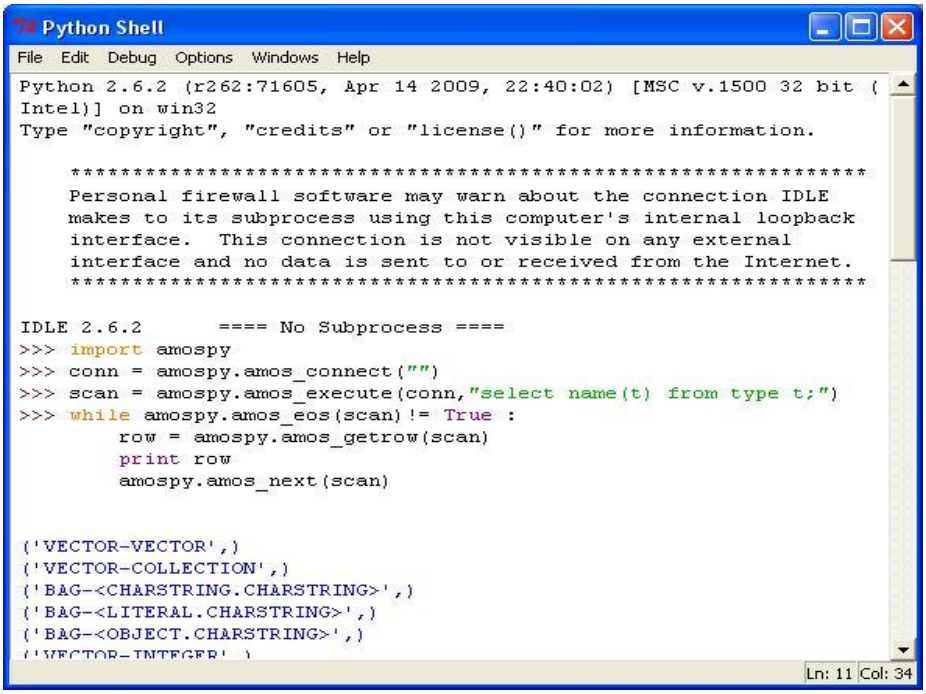


Figure 3.2: PyAmos running in the Python top loop IDLE

Table 3.1 shows how Amos II types matches corresponding C, Python, PHP, and Java types in the corresponding APIs.

Amos II types	C interface	PyAmos	PHP-Amos	JavaAmos
Interger	INTEGERTYPE(int)	Py_Integer	ZVAL_LONG	Int
Real	REALTYPE(double)	Py_Double	ZVAL_DOUBLE	Double
CharString	STRINGTYPE(char*)	Py_String	ZVAL_STRING	String
Boolean	SYMBOLTYPE (a_true/a_false)	Py_Ture/Py_False	ZVAL_BOOL (TRUE/FALSE)	Boolean (TRUE/FALSE)
nil	SYMBOLTYPE(NULL)	Py_None	ZVAL_NULL	Null
vector	ARRAYTYPE(a_tuple)	Py_Tuple	PHP Array	Tuple (user-define java class)
OID	OIDTYPE(oidtype)	AmosOID (user-define Python Type)	ZVAL_STRING (“#[OID <oidnum>]”)	Oid (user-define java class)
Connection	a_connnection	AmosConn (user-define Python Type)	scan_resource	Connection (user-define java class)
Scan	a_scan	AmosScan (user-define Python Type)	connection_resou rce	Scan (user-define java class)

Table3.1. Amos II type mappings

3.2 Extending Python

Developer can make contributions to Python by implementing their own C extensions.

Python C API

The Python C API gives C/C++ programmers access to the Python interpreter at several levels. Python itself is built internally by using the same API. There are two ways of using the API:

- Write Python extension modules for specific usages.
- Use Python as a component in a large application.

The approach of using Python as a component in a large application is to embed the Python interpreter in C/C++ code [21] and expose the flexibility of Python in the application. By contrast, in this project we focus on writing a Python extension module to access and Amos II database.

To illustrate how to extend Python in C, we start with a 'hello world' example. Here is Python Code:

```
def hello_world(name):
    "say hello world to somebody."
    print "Hello %s!" % name
```

The corresponding Python C extension Code, *hello.c*, looks like this:

```
#include <Python.h>

static PyObject* hello_world(PyObject* self, PyObject* args)
{
    const char* name;
    if (!PyArg_ParseTuple(args, "s", &name))
        return NULL;
    printf("Hello World, %s!\n", name);
    Py_RETURN_NONE;
}

static PyMethodDef HelloMethods[] =
{
     {"say_hello", hello_world, METH_VARARGS, "Say hello world to somebody."},
     {NULL, NULL, 0, NULL}
};

PyMODINIT_FUNC inithello(void)
{
     (void) Py_InitModule("hello", HelloMethods);
}
```

The three main parts in the extension module besides the necessary include files are:

- The C functions that will be called from Python
- The definition of a *method mapping table,* which maps Python interface names to the corresponding C function entries.
- An *initialize* function for the extension module.

The C function *hello_world()* will run the actual user extension. In general, the implementations

of the C functions can have three formats: functions with arguments, functions with keywords arguments, and functions with no arguments. They are defined as follows:

```
static PyObject *TheCFunction( PyObject *self, PyObject *args );
static PyObject *TheCFunctionWithKeywords(PyObject *self, PyObject *args, PyObject *kw);
static PyObject *TheCFunctionWithNoArgs( PyObject *self );
```

The functions mapping table maps the C function names to the interface name that will be used in Python. That means that once you imported the extension module, you can call the *say_hello* interface, and it will execute the *hello_world()* function. The *PyMethodDef* structure is defined as follows:

```
struct PyMethodDef {
    char *ml_name;
    PyCFunction ml_meth;
    int ml_flags;
    char *ml_doc;
};
```

with attributes meaning:

**ml_name*: The function name that will be used in the Python code.
**ml_meth*: A pointer to the corresponding C function defined in the extension module.
**ml_flags*: A set of flags, normally set to METH_VARARGS. If you want to allow keyword arguments in the function you will need to use METH_KEYWORDS, while you can use METH_NOARGS if you don't want to accept any arguments at all.
**ml_doc*: A documentation string that describes the function.

The other important part of the extension module is the *initialize* function. It will initialize the module. The initialize function is named as 'init*Module_Name*', and thus the *hello* module will have a initialize function called 'init*hello*'. In the initialize function, the *Py_InitModule()* will be called initially to make the module ready to be used in Python.

Passing Parameters from Python to extension functions

In the example we use *PyArg_ParseTuple()* to parse a Python argument tuple to several C variables. It acts like the function *sscanf()* in C. The 'hello_world' example uses METH_VARAGRS in the function mapping table to handle variable number of arguments. The following code snippet shows how it works:

```
static PyObject *my_func(PyObject *self, PyObject *args) {
char * name;
int i;
double d;
PObject* p;
if (!PyArg_ParseTuple(args,"sido",&name,&I,&d,&p))
        return NULL;
```

}

In Python the function *my_func* is called like this:

```
my_func('Derek',11,20.5,PY_TUPLE)
# PY_TUPLE is an Python Tuple object defined as PY_TUPLE=("string",1000,"sample")
```

The function uses the following system C function to retrieve the actual arguments in the function call:

int ***PyArg_ParseTuple****(PyObject *args, const char *format, ...)*

The *format* here is a C string that describes the types of the arguments.

Format string	**Parsing to C type**	**From Python**
s	char *	Convert Python string to char *
c	Char	Convert Python char to C char
d	Double	Convert Python double to C double
f	Float	Convert Python float type to C float
i	Int	Convert Python int to C int
l	Long	Convert Python long to C long
o	PObject *	Convert a Python object reference to a PObject * pointer in C

Table 3.2 Parse types from Python to C

Return value to Python

In the extension module, we may also want to return a value to Python. In this case, one needs to convert C types back to Python types. This is done by the C function *Py_BuildValue()*. It is defined as:

PyObject* **Py_BuildValue**(*const char *format, ...*)

This function does the inverse work of what the *PyArg_ParseTuple()* function does.

Format string	**C Type**	**Description**
s	Char *	Uses *char * to build a Python string object, or *Py_None* will be built in case of a null pointer.
c	Char	A C character will be converted to a string in Python with length one.

d	double	Convert C *double* to Python *float*.
f	Float	Convert C *float* to Python *float*
i	Int	Convert C *int* to Python *int*
l	Long	Convert C *long* to Python *long*
o	PObject *	Return a Python Object and increase the reference counter.
N	PObject*	Return a Python Object without increasing the reference counter.

Table 3.3 Return Values from C to Python

Here are some examples how you can use the *Py_BuildValue()* function to create Python objects of different kinds in C code.

```
Py_BuildValue("s", "x") --> "x"
Py_BuildValue("i", 17) --> 17
Py_BuildValue("{si,si}", "x", 17, "y", 2) --> {"x":17, "y":2}
Py_BuildValue("(isi)", 17, "x", 23) --> (17, "x", 23)
Py_BuildValue("{si,s(ii)}", "x", 17, "y", 2, 3) --> {"x":17, "y":(2,3)}
Py_BuildValue("") --> None
```

Py_RETURN_NONE is used when you want to return null value to Python, and this is the same as the statement *return Py_None*. It is important to use *Py_INCREF* to increase the reference counter before returning a newly created Python object back to Python from C; otherwise the application will crash. The same holds for *Py_None.*

How to build the extension

The *distutils* package is used to distribute both pure Python and C extension modules. You can build and install the modules by running a setup script called 'setup.py', e.g.:

```
from distutils.core import setup, Extension
setup(name='hello', version='1.0',    \
        description = 'This is Hello extension module written in C'
        ext_modules=[Extension('hello', ['hello.c'])])
```

You also need to compile and link the module by running the following command:

$ python setup.py install

In Linux, this command will make up a module named 'hello.so'.

In Windows the Microsoft Visual C++ IDE can be used to compile the extension [21] as a file in the Python Dynamic Module (pyd) format, which is similar to a DLL file.

Running the extension

When running the newly built Python extension, you need to change your working directory to where the file 'hello.so' or 'hello.pyd' file resides. If you are using Windows, you can also put 'hello.pyd' in the '*Python_BIN/DLLs*' directory (*PythonBIN* is Python's installation path). The module can then be used like:

```
>>> import hello
>>> hello.say_hello("Derek")
Hello World,Derek!
```

In a similar way, user defined types can be implemented in extension modules.

3.3 Passing Database OID

The Amos II system supports objects, which means everything in Amos II is represented as either system or user-defined objects.

Literals and surrogates are two main representations in the system. OIDs are object identifiers which are associated with surrogate objects. The surrogate types match real-world entities and are instances of the meta-type named *Type* in Amos II. The literal objects are self-described system maintained objects. They usually don't have explicit OIDs. Numbers and strings are literal objects. The literals are managed by an automatically garbage collector and will be deleted if they are no longer referenced in the system.

Here is an example of how to create an object using surrogates:

```
create Person instances :zou;
```

The statement returns a new object, e.g. printed as '#[OID 1122]'. The OIDs are unique for all surrogate objects.

When programming in C, all Amos II objects are referenced to through *object handles* which are references to any kind of data stored in the Amos II system. The object handles can also refer to literal objects and collections, which have no OID numbers. An object handle is basically a logical pointer to a data structure in the Amos II database image. An object handle is represented as a C type named *oidtype* declared as '*typedef unsigned int oidtype;*'. The unsigned integer holds the offset from the start of the database image to the C structure representing the Amos II object. A new object handle *obj* is declared by *dcl_oid.* The value of a handle can be initialized using the macro *a_let().* A new value can be assigned to a handle by calling the *a_assign(<location>,<new value>)* function. You are suggested to use *a_assign()* and *a_let()* instead of the '=' operator so that proper reference counting is maintained. The object handles must be released by calling the function '*free_oid(<location>)*' after which the system will deallocate object handle if no other object references it. If you don't call *free_oid()* when you exit the code block, a memory leak will happen.

To make all kinds of objects available in Python it is needed to represent object handles as Python objects. Surrogates, literals, and collections have to be passed between Amos II and Python in some way. One alternative is to pass the surrogate objects as strings in '#[OID <oidnum>]' format. This approach is used by the PHP-Amos interface [3]. By that approach, a C function first would translate an object handle referencing a surrogate object into a string '#[OID <oidnum>]', and would then use *Py_buildValue()* to return a Python string. When Python wants to pass an OID back to Amos II, the string would first be checked if it matches the '#[OID <oidnum>]' syntax. If so the string is parsed to get the OID number. With this solution, the surrogate objects would be a special string in Python. The user could not use a string with syntax '#[OID <oidnum>]', since it will be parsed into an Amos II OID. This solution works fine for the PHP-Amos interface, because PHP is mainly for WEB applications, and passing an OID in a string is easy to handle by a web client.

Another alternative is to convert Amos II object handles to Python user defined objects. This is the PyAmos solution. Common literal objects such as numbers, strings, and vectors are represented using corresponding built-in Python types. We define a new Python type called *AmosOID* to represent Amos II objects in Python. The *AmosOID* class will hold an object handle (*oidtype)* value as member. When passing an object handle from Amos to Python, the handle will be assigned to a member variable of an *AmosOID* object, and then the object is returned to Python. In the other direction, Amos II will pick up the *oidtype* value from the *AmosOID* object and send it to C *callin* interface function. A Python member function *toString()* is also needed for the *AmosOID* class when the Python user want to output an OID value; it formats the object handle into a string, e.g. '#[OID <oidnum>]' for surrogate objects, and return a Python string. Here is an example showing how the AmosOID works in PyAmos:

```
>>> conn = amos_connect("")
>>> fct = amos_getfunction(conn,"plus")
>>> fct
<AmosOID object at 0x00B08190>
>>> fct.toString()
'#[OID 131]'
```

This solution is definitely better than translating OIDs into strings. The *AmosOID* objects are Python objects that will be managed by the Python garbage collector. This allows for PyAmos to connect the garbage collectors in Python and Amos II so that Amos II object no longer referenced from Python will be automatically deallocated if they have no other reference. How to create Python user defined types and how to use connect the garbage collectors will be elaborated next.

3.4 Tuples in Amos II and Python

The Python data structure *Tuple* holds a sequence of objects indexable by integers. For example:

```
>>> tup = 12345, 54321, 'hello!'
>>> tup[0]
```

```
12345
>>> tup
(12345, 54321, 'hello!')
```

In the Amos II C interface, there is a similar representation of tuples that can be initialized by using the macro *dcl_tuple*:

```
dcl_tuple(tpl);
```

The macro will declare and initialize the C variable *tpl* to be a tuple, declared by a structure named *a_tuple*. When integrating Python with Amos II, we need to have functions to convert between Python tuple objects and Amos II *a_tuple* structures.

PyAmos converts every element in an *a_tuple* structure, using *Py_BuildValue(),* into the corresponding Python data representation. For example, if an element is an Amos II object a corresponding *AmosOID* object should be created. If an element is of type *Vector* in Amos II a Python tuple is constructed recursively. Thus the *Vector* type in Amos II is translated to be Python's *Tuple* type.

3.5 Creating New Types in Python

Previously the methods for converting Amos II types like numbers, strings, OIDs, and tuples to Python types have been discussed. Now we will take a look at how to convert a C structure used by the Amos II *callin* interface into a corresponding Python user define type.

In the C *callin* interface of Amos II, the data structures for interfacing the database are using connections and scans. The structure *a_connection* represents a connection to the database, and the *a_scan* structure stores results from Amos II function calls and queries as a scan. In the *callin* interface connections and scans are declared as followed:

```
typedef struct a_connection_rec
{
    int hasbeeninitialized;
    char *name;
    oidtype servid;
    oidtype port;
    int status;
    oidtype result;
    a_scan primscan;
} *a_connection;

typedef struct a_scan_rec
{
    int hasbeeninitialized;
```

```
    oidtype here;
    oidtype row;
    int stopafter;
    int status;
} *a_scan;
```

When PyAmos passes the *a_connection* and *a_scan* structures between the Python interpreter and Amos II they have to be converted to corresponding Python objects. There are advantages to create them as two different Python types without worrying about memory leaks by utilizing the garbage collectors of Python and Amos II. Furthermore, Python types are easy to use and can be passed as parameters of other functions.

In general, there are five steps to create a user defined Python type in C:

- First, define your corresponding C structure for the Python type.
- Second, make an *initialize* and *deallocate* function for the new type.
- Third, write and declare the *member functions* of the new type.
- Fourth, declare all the *components* of the type.
- Fifth, make the type ready for the module.

In Pyamos, there are three new C structures for creating correspond Python types: *Py_AmosOIDObject*, *Py_AmosConnObject*, and *Py_AmosScanObject*. For example, the *Py_AmosConnObject* has the name *AmosConn* in Python. The *Py_AmosConnObject* structure is used to represent the structures *a_connection* in Python. It is created as follows.

Defining the C structure

Since an *a_connection* is defined as a C data structure we should make the C structure definition as a *Py_AmosConnObject* having the same data fields as an *a_connection* structure. One way is to copy all the data fields from the *a_connection* structure to the *Py_AmosConnObject* structure, another way is to 'wrap' a pointer the *a_connection* structure, i.e. keep the data in Amos II and just store references to the Amos II data objects. PyAmos keeps the data in Amos II which is faster as it requires less copying.

```
typedef struct {
      PyObject_HEAD
       a_connection _conn;
} Py_AmosConnObject;
```

The first line inside the *Py_AmosConnObject* structure definition is a macro called *PyObject_HEAD* that defines the initial fields in the structure and makes the structure be usable in Python. Every Python C structure must have this macro in its definition. The second field is a pointer to an *a_connection* structure.

Initialize and deallocate functions

When writing a class in python, you may have an *__init__* method do the initialization work for

creating new instances of the class and the same holds for the Python C extension, e.g.:

```
static int AmosConnObject_init(Py_AmosConnObject *self, PyObject *args, PyObject *kwds){
        /*    AmosConn object initialize code    */
        return 0;
}
```

The *AmosConnObject_init* function will initialize the structure members. It will be called when a new *AmosConn* object is created. The *self* pointer associates this function with the new *AmosConnObject*.

In Python, in most cases you don't need to worry about the deallocation function of a class. However, there is an optional *__del__* method that defines customized deallocation functions for Python class instances, e.g.:

```
static void AmosConnObject_dealloc
            (Py_AmosConnObject* self) {    /*    deallocate code for AmosConn Object    */    }
```

Later in the definition of *AmosConnType*, the destructor *AmosConnObject_dealloc* has been assigned to the *tp_dealloc* field and then function *AmosConnObject_dealloc* is called automatically when the Python garbage collector wants to deallocate this Python object.

Write the member function and declare the members

Now we can start to write the member function of the new object. As we know the *AmosConnObject* is used to handle connections to Amos II databases. The member function definition of *connectTo* uses *Py_AmosConnObject ** instead of *PyObject ** for the *self* argument:

```
static PyObject * AmosConn_connectTo(Py_AmosConnObject *self, PyObject *args){
    char *dbname;
    static int isAmosInitialized = 0;
    if (!PyArg_ParseTuple(args, "s", &dbname)) {
            printf("error: wrong args in funcition amos_connect \n");
            PyErr_BadArgument();
           return NULL;
      };
      // some other work to do
     return Py_None;
}
```

You need to declare *connectTo* and make it a member function of the *PyAmosConnObject* class:

```
static PyMethodDef AmosConn_methods[] = {
     { "connectTo",        (PyCFunction) AmosConn_connectTo, METH_VARARGS,
                    "Connect to The database server with serverName" },
```

```
    // some other function declarations
    { NULL }
};
```

Declare all the components of the type

Now we can combine the pieces together and initialize the *PyAmosConnObject* type structure:

```
static PyTypeObject AmosConnType = {
    PyObject_HEAD_INIT(NULL)
    0,                          /* ob_size */
    "AmosConn",                 /* tp_name */
    sizeof(Py_AmosConnObject), /* tp_basicsize */
    0,                          /* tp_itemsize */
    (destructor)AmosConnObject_dealloc, /* tp_dealloc */
    0,                          /* tp_print */
    0,                          /* tp_getattr */
    0,                          /* tp_setattr */
    0,                          /* tp_compare */
    0,                          /* tp_repr */
    0,                          /* tp_as_number */
    0,                          /* tp_as_sequence */
    0,                          /* tp_as_mapping */
    0,                          /* tp_hash */
    0,                          /* tp_call */
    0,                          /* tp_str */
    0,                          /* tp_getattro */
    0,                          /* tp_setattro */
    0,                          /* tp_as_buffer */
    Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE, /* tp_flags*/
    "AmosConn object",          /* tp_doc */
    0,                          /* tp_traverse */
    0,                          /* tp_clear */
    0,                          /* tp_richcompare */
    0,                          /* tp_weaklistoffset */
    0,                          /* tp_iter */
    0,                          /* tp_iternext */
    AmosConn_methods,           /* tp_methods */
    0,                          /* tp_members */
    0,                          /* tp_getset */
    0,                          /* tp_base */
    0,                          /* tp_dict */
    0,                          /* tp_descr_get */
    0,                          /* tp_descr_set */
    0,                          /* tp_dictoffset */
```

```
    (initproc)AmosConnObject_init,    /* tp_init */
    0,                                /* tp_alloc */
    0,                                /* tp_new */
};
```

As you can see, most the fields of the structure are assigned to 0 and actually omitted by Python. You just need to fill in the parts you need and provide your own functions. The *PyAmosConnObject* structure is defined as the Python class named *AmosConn* specified by setting the value of the *tp_name* field.

Make the type ready for the module

Now the Python type definitions are done. The only thing remaining is to make it available in the PyAmos extension module:

```
AmosConnType.tp_new = PyType_GenericNew;
if (PyType_Ready(&AmosConnType) < 0) {
        return;
}
```

The code snippet should be added to the extension module function. Then this new type can be used in Python just as other Python classes.

For some special cases in PyAmos, an *AmosConnType* instance must be available in C. This happens, e.g., when the user calls *amos_connect()*in Python code to get an *AmosConnType* instance, the following code will run inside the extension module:

```
Py_AmosConnObject* connObj;
AmosConnType.ob_type = &PyType_Type;
connObj = PyObject_New(Py_AmosConnObject, &AmosConnType);
```

The *connObj* will hold a pointer referencing to an *AmosConn* type instance.

3.6 Error Mangement and Execptions

Amos II and Python have their own error managements and the PyAmos implementation integrates them.

First let's take a look at the following functions of the Amos II external interface:

```
int a_initialize(char *image, int catcherror);
int a_connectto(a_connection c, char *peer, char *hostid, int catcherror);
```

As these functions, most Amos II C interface functions have the parameter *catcherror.* When *catcherror* is assigned to *FALSE*(=0), a failure of the call will cause a fatal system error; if the *catcherror* is *TRUE*, the call will return the error number to the calling program.

There are three C global variables used to keep track of the returned errors:

```
int a_errno; /* The Amos II error number, same as the error indication */
char *a_errstr; /* A string explaining the error */
oidtype a_errform; /* A reference to an Amos II object */
```

If an error occurs during the call, the *a_errno* will store the error number not equal to 0, and the error string will be stored in the *a_errstr*. The *a_errform* is an object handle holding a reference to the Amos II object associated with the failure. Then you can call *a_print()* to print it on standard output.

```
if (a errorflag != 0) {
printf("(1) Error %d: %s ",a_errno, a_errstr);
a_print(a_errform);
/* do some error handling here */
}
```

The functions *a_print()* and *printf()* can only print out errors on standard output, but the Python won't be aware of the error occurring in Amos II. Thus integration between the Amos error handling and Python error management is needed.

In Python you need to be aware that errors might occur in either sides of Python's C extension [21]. The common case is that when an error occurs in the code, a NULL value should be returned to flag it. The exceptions are stored in the global static variables for addressing errors, which are *sys.exc_type*, *sys.exc_value* and *sys.exc_traceback* in Python.

There are several functions in Python's C API to deal with errors, among which *PyErr_SetString()* is the most commonly used. The first argument is the exception type, which is normally one of the standard exceptions, for instance *PyExc_RuntimeError* or *PyExc_MemoryError*. The second argument is a string containing the error message.

void **PyErr_SetString**(*PyObject *type, const char *message*)

You can get the current exception by calling the function *PyErr_Occurred()*. It will return NULL if no exceptions occurred. The function *PyErr_Format(PyObject *exception, const char *format, ...)* will set the current error indicator and print a string containing format codes. Another important thing is that if you use *malloc()* or *realloc()* to allocate memory in C, you should always throw an exception when it fails. In this case *PyErr_NoMemory()* should be called and a NULL value be returned. The exceptions can also be ignored by using the function *PyErr_Clear()*, but this only happens when the errors are handled in C code blocks and the developer don't want to pass the exception to the Python interpreter.

In PyAmos we integrate the Amos II and the Python error handling by defining the following

macros:

```
#define ILLEGAL_PARAM    PyErr_BadArgument();
#define AMOS_ERROR (a_errform==nil?PyErr_Format(PyExc_RuntimeError, "a_errstr:%s", a_errstr):PyErr_Format(PyExc_RuntimeError, "%s : %s", a_errstr, a_to_string(a_errform)))
#define AMOS_WARNING (a_errform==nil?PyErr_Format(PyExc_RuntimeError, "a_errstr:%s", a_errstr):PyErr_Format(E_WARNING, "%s : %s", a_errstr, a_to_string(a_errform)))
#define CHECK_AMOS_ERROR    if(a_errorflag) AMOS_ERROR
```

In the implementation, we check the errors by doing:

```
// check the error when passing the parameters
if (!PyArg_ParseTuple(args, "s", &dbname)) {
        printf("error: wrong args in funcition amos_connect \n");
        ILLEGAL_PARAM;
         return NULL;
    };
```

The above code will report bad parameters error by calling *PyErr_BadArgument()*. To check for whether an error has occurred in a call to Amos II the macro *CHECK_AMOS_ERROR* is used, e.g.:

```
a_commit(self->_conn,TRUE);
CHECK_AMOS_ERROR;
```

The *catcherror* parameter should always be set to be *TRUE* after which the C macro *CHECK_AMOS_ERROR* checks if whether errors have occurred. If an error has occurred, the macro will throw a *PyExc_RuntimeError*, and print the *a_errstr* string in Python to provide error information from the Amos II error handling mechanism.

3.7 Integrating the garbage collectors

Both Python and Amos II have their own automatic garbage collectors, unlike C. In PyAmos the GCs of Python and Amos II interoperate so that all temporary memory used by the embedded Amos II is released when no longer referenced from Python.

Like many other garbage collectors, both Python and Amos II uses a reference counting scheme to keep track of object references. In Python each object has a private member called '*Py_ssize_t ob_refcnt'* carrying the number of places where it is referenced. When *ob_refcnt* equals zero the Python garbage collector will give this object's memory space back to the system automatically.

You can get the object reference count in Python by doing:

```
>>>import sys
>>>sys.getrefcount(conn)
```

3

Normally the garbage collector manages the reference counting behind the scenes, so that you don't need to worry about memory leaks when you are using Python. When it comes to the C extending modules, the C code is responsible for managing the reference counts of the Python objects it uses. System functions in the Python C-APIs are used to increase or decrease the reference counting of the Python object. The functions *Py_INCREF()* and *Py_XINCREF()* are used to increment the reference counting, while the *Py_DECREF()* and *Py_XDECREF()* are used to decrement. The difference is *Py_XINCREF()* and *Py_XDECREF()* can be used on *null* pointers, but the *Py_INCREF()* and *Py_DECREF()* can't.

When Python objects are created in the C code, the reference counter of an object must be increased before given back to C. In most cases, argument objects and the returned objects don't need to change the reference counter.

Similar to the reference counting in Python, Amos II has its own reference counting mechanism. The *a_assign()* function will decrease the reference count of the old value stored in a location and the object will be de-allocated if the reference counter becomes zero, i.e. if no other location references the old value. The reference counter of the object to assign to the location will be increased.

In PyAmos, there are two cases needed to be considered in integrating the garbage collectors.

1. The reference counter of an Amos II object handle to be handed over to Python must be increased so that the object is not deallocated by Amos II. The macro *a_let()* increases the the reference counter of an object and is used to initialize the *oidtype* reference in the python object. Example as followed:

```
Py_AmosOIDObject* py_oid = new_AmosOIDObject();
a_let(py_oid->_oid,fct);
return (PyObject*)py_oid;
```

2. In PyAmos the Python user defined objects *AmosOID, AmosConn* and *AmosScan* have *deallocate* functions, which are called by the Python garbage collection system to free the corresponding Amos II interface structures when they are deallocated. For example, for object handles (OIDs represented by Python type *AmosOID*), if Python decides to deallocate an *AmosOID* object referencing an Amos II object handle, the Amos II system macro *a_free()* is called to decrease the Amos II reference counter and deallocate the Amos II object if not referenced anywhere else.

Now let's see how an *AmosOID* object can be destroyed:

```
static void AmosOIDObject_dealloc(Py_AmosOIDObject* self){
        if(self->_oid)
                a_free(self->_oid);
```

```
        PyObject_Del(self);
    }
```

When the reference counting of a Python *AmosOID* object goes to zero, the *deallocate* function is called. It will first check if its *oidtype* member is available. If yes then *a_free()* is called on the Amos II object handle to deallocate the Amos object. Later the *PyObject_Del(self)* will destroy this Python object itself. The memory deallocation will be managed by Amos II and Python automatically.

3.8 The Python Modules

This section describes parts of the PyAmos system written in Python, including how to use map functions and lambda functions.

Python supports anonymous functions at runtime by using *lambda functions*. A Python lambda function in principle can be used wherever functions are required, thus we can say the Python lambda is just an unnamed function definition.

Here is a simple lambda function in Python:

```
>>> lambda a,b : a+b
```

Basically, the syntax of a lambda function is as follows:

```
lambda   [parameter_list]   :   expression .
```

The *parameter_list* is a list of parameters that will be used in the expression. However, you can't use, e.g., the '*print*' statement in lambda functions, you must use *sys.stdout.write()* instead, because the body of a lambda function has to be an expression, not a statement, while '*print'* is a statement. It is important to remember that only expressions are allowed in a lambda function, which means assigning variables is not allowed.

Python's *map* function

Python provides a '*map'* function, and lambda functions are often used in conjunction with *map* functions as callback functions. The function *'map (fn, seq1, seq2,…)'* applies the function *fn* on each items in the sequences *seq1…* and returns a list of the returned values.

```
>>> a = range(4)
>>> map( lambda x,y:(x,y) , a,    ('one','two','three','four') )
[(0, 'one'), (1, 'two'), (2, 'three'), (3, 'four')]
>>>
```

I In the example, the lambda expression '*lambda x,y:(x,y)*' is applied on each element in the two lists. In PyAmos, you can also make use of lambda expressions and map functions to simplify your

coding. For example, you can use a map function to add the scan results with '*atrName'*.

```
>>> row = scan.getrow()
>>> map(lambda x,y:(x,y), ['atrname1','atrname2','atrname3'] , row)
```

Mastering how to use the map function can simplify the code. In particular there is a *mapCall()* function defined in PyAmos interface using Python, which will call the Python function for each element of the results of an Amos II function call.

```
>>> printf = lambda x: sys.stdout.write("%s\n"%x)
>>>amos_mapCall(conn,printf,"iota",0,100)
```

The *mapCall()* takes a Python lambda function as the first parameter. The lambda function *'printf'* will be a callback function called by PyAmos's map function. Inside the *mapCall()* function, the lambda function works on each element of the result list which is returned by calling *'iota'* in Amos II. Here the function *iota* returns each integer from 0 to 100. The map function is applied by *mapCall* on each result returned from Amos II. Thus in this case the number 0 to 100 are printed on Python's standard output.

The map functions for PyAmos are easily implemented using Python as follows:

```
def amos_mapCallPy(pyFunc,*tup):
        scan = amos_call(*tup)
        while amos_eos(scan)!= True:
                row = amos_getrow(scan)
                pyFunc(row)
                amos_next(scan)

def amos_mapQueryPy(pyFunc,*tup):
      scan = amos_execute(*tup)
      while amos_eos(scan)!= True:
                row = amos_getrow(scan)
                pyFunc(row)
                amos_next(scan)
```

Example of usage:

```
        conn = amos_connect("")
        printf = lambda x:sys.stdout.write("%s\n"%x)
        amos_mapQueryPy(printf,conn,"select name(t) from type t;");
       amos_mapCallPy(printf,conn,"iota",0,10)
```

In the above example, the Python function *amos_mapCallPy()* and *amos_mapQueryPy()* apply Python functions, either a normal Python function or a lambda function, on the elements of the

scan returned by calling an Amos II functions or executing an Amos II query.

4. Performance Measurements

What is the performance of the PyAmos interface? How long is the execution time of making an Amos II call from PyAmos? How long does it take to pass different kinds of primitive data between Python and Amos II, and how much time is spent to iterate over large sets of different primitive data? Now we are to find out answers to these questions.

First we declare the PyAmos interface testing environments. The performance were tested by using:

- Amos II Release 12,v3
- Python 2.6.2
- Window XP professional sp3.

The test machine we used is a personal computer: Lenovo Thinkpad T61, which has 2.0GH Intel Core 2 Duo CPU and 2GB RAM memory. When investigating the performance of the client-server interface, the Amos II server runs on the same machine as the Python client.

Time of calling an Amos II function from PyAmos

To check the overhead of a PyAmos call to an Amos II function, a function called *'dummy'* is defined in Amos II without arguments and returning a Boolean:

```
create function dummy()-> Boolean ;
```

The *'dummy'* function is not returning any value. We called this function 10000 times, and compared it with the Amos II interfaces, *C callin* [23]*, PHP*-Amos [3], and *JavaAmos* [4].

The code snippets for testing calling the dummy function in C, JavaAmos, PHPAmos, and PyAmos are as follows:

```
// C code using callin interface.

t0 = clock();
f1= a_getfunction(“dummy”,FALSE);
(a_getFunction or something in external.pdf)
for (i= 0; i<10000;i++)      {
        a_setarity(argl,0);
        a_callfunction(c, s, f1, argl, FALSE);
}
t= clock()-t0;
```

```
// Java code using JavaAmos interface

long time3 = System.currentTimeMillis();
f1 = theConnection.getFunction("dummy");
arg1 = new Tuple(0);
for (int i =0;i<10000;i++){
theConnection.callFunction(f1, arg1);
}
time3 = System.currentTimeMillis() - time3;
```

```
// PyAmos interface

t0 = time.time()
for i in range(1,10000):
        res = amos_call(conn,"dummy")
t1 = time.time() -t0t= clock()-t0;
```

```
// PHP-Amos interface
<?php

$starttime = microtime();
for($counter=1; $counter<=10000; $counter++){
      $scan = amos_call($con, "dummy");
} $endtime = microtime();
$duration=$endtime-$starttime;
?>
```

The average time in seconds for 10000 calls to the dummy function is shown in the table 5.1.

Average seconds for 10000 calls	Dummy call from PyAmos	Dummy call from JavaAmos	Dummy call from PHP-Amos	Dummy call from C callin
Tight Connection	0.188000	0.266000	0.220000	0.172000
Client Server	1.406000	1.796000	1.579000	1.359000

Table4.1. Time of calling a dummy function using PyAmos , JavaAmos, PHP-Amos, and C callin interfaces

Since all interfaces are based on the C callin interface, it can be concluded that the overhead of PyAmos is 9.3% with the tight connection, and 3.5% with the client-server connection. In particular, PyAmos is much faster than JavaAmos and PHP-Amos.

Time of passing primitive data between Python and Amos II

In Amos II there are primitive data types such as integers, strings, reals, vectors, and OIDs. It was investigated how much time is used to pass these data types using PyAmos.

For testing the time of sending and receiving different types of data, the *call_1()* function is used. Unlike *amos_call()*, which returns a scan object, *amos_call_1()* returns the calling result as a single value. It is the fast-path for simple calls that return a scan with a single value such as an

integer, string etc.

Take passing strings as an example. The following Amos II function definitions test sending and receiving strings, respectively:

```
create function sendString(Charstring str)->Boolean as select true;
create function receiveString()->Charstring as select 'A receive string';
```

The *'sendString'* function is used for measuring the time of PyAmos translating Python strings into Amos II, and *'receiveString'* is measuring the time to translate Amos II strings to Python strings. The two functions are compared with the '*dummy*' function to calculate the extra cost of converting strings between Amos II and Python. The Python testing scripts are:

```
t0 = time.time()
for i in range(1,10000):
        res = conn.call_1("sendstring","A Test String")
t1 = time.time() -t0
print "\n t1 call1 func sendstring :%f\n"%t1

t0 = time.time()
for i in range(1,10000):
        res = conn.call_1("receivestring")
t1 = time.time() -t0
print "\n t1 call1 func receivestring :%f\n"%t1
```

The time difference of calling '*sendString*' and '*dummy*' function is the time of converting the Python string to an Amos II string. For 10000 calls using the tight connection the measured times were:

$t_{send} = 0.219$, $t_{dummy} = 0.188$, $t_{diff} = t_{send} - t_{dummy} = 0.031$

Thus the time of transmitting and converting a string from Python to Amos II is 0.031 second for 10000 calls. Analogous measurements were done for the other interfaced data types integers, reals, vectors and OIDs.

Time in seconds for transmitting different data types 10000 times		**Tight Connection**	**Client Server**
String('A testing String',length 16)	Send	0.031(of 0.219)	0.282(of 1.688)
	Receive	0.034(of 0.222)	0.253(of 1.657)
Integer	Send	0.031(of 0.219)	0.281(of 1.687)
	Receive	0.021(of 0.203)	0.264(of 1.672)
Real	Send	0.031(of 0.219)	0.313(of 1.719)
	Receive	0.030(of 0.218)	0.312(of 1.718)

OID	Send	0.046(of 0.234)	0.329(of 1.735)
	Receive	0.047(of 0.235)	0.329(of 1.735)
Vector(vector {0,1,2,3}, size 4,)	Send	0.237(of 0.515)	0.547(of 1.953)
	Receive	0.062(of 0.250)	0.266(of 1.672)
Vector (vector {0…7}, size 8)	Send	0.432(of 0.610)	0.884(of 2.292)
	Receive	0.122(of 0.310)	0.744(of 2.150)

Table4.2. Time of transmitting primitive data types between Amos II and Python.

Table 4.2 shows the comparisons between cost of passing different data types through the PyAmos interface using the tight-connection and the client-server connection. The full Amos II function definitions and Python test script can be seen in the Appendix 2.

We can see that sending or receiving a *'vector'* of size eight, converted to a *Tuple* in Python, will definitely take more time than other types since it holds several (8) other objects. The time increases when vector size gets larger. The reason is that before sending and receiving a vector, all the vector elements are converted into corresponding Python types.

Time of iterating through large result sets

The next measurements will focus on the scalability when the size of results set increases. We will call an Amos II function that returns a set of results as a scan, and then test the execution time when the size of the results set increases.

For the tests, Amos II functions are defined to return sets of results for different types. For example, returning bags of strings and integers are defined as followed:

```
create function StringResult(Integer size)-> Bag of Charstring nm as
    select “the string” from Integer i where i in iota(1,size)

create function IntResult(Integer size)-> Bag of Integer nm as
    select 1 from Integer i where i in iota(1,size);
```

Analogous functions are defined for reals, vectors, and OIDs. The number of returned objects is varied by changing the parameter *size*.

The test focuses on the time of iterating through the results coming back from the Amos II calls. For example:

```
amos_execute (conn,"create function IntResult(Integer size)-> Bag of Integer nm as \
                    select 1 from Integer i where i in iota(1,size);");
t0= time.time()
scan = amos_call(conn,"IntResult",10000)
```

```
while amos_eos(scan)!= True :
        row = amos_getrow(scan)
        amos_next(scan)
t1 = time.time() -t0
```

The previous Python test scripts is testing the time of iterating through result sets of 10000 *integers* when using the tight connection; the time was *t1* = 16 milliseconds on average. Table 4.3 shows the time durations for difference types.

Average time/Millisecond	10000	20000	30000	60000	80000	100000	200000	400000
Integer(Tight)	16	31	62	140	155	233	437	891
Integer(C/S)	31	62	94	171	234	296	578	1171
Real (Tight)	16	32	62	139	171	219	437	890
Real(C/S)	62	110	156	328	437	593	1093	2171
String(Tight)	15	46	62	141	172	235	485	983
String(C/S)	46	62	125	218	296	391	780	1625
Vector(Tight)	31	62	93	202	265	358	765	1703
Vector(C/S)	62	108	156	344	453	593	1218	2562
OID(Tight)	16	47	78	156	203	266	516	1031
OID(C/S)	63	125	188	360	453	563	1173	2360

Table 4.3 Time of iterating large result sets.

We also compare the tight connection with client-server connection when the results set of different types increase in size. The *client server overhead*, *Co,* is defined as:

$$Co = (t_{(client\text{-}server)} - t_{(tight)}) / t_{(tight)} * 100\%$$

Co measures how much slower the client-server interface is compared with the tight connection.

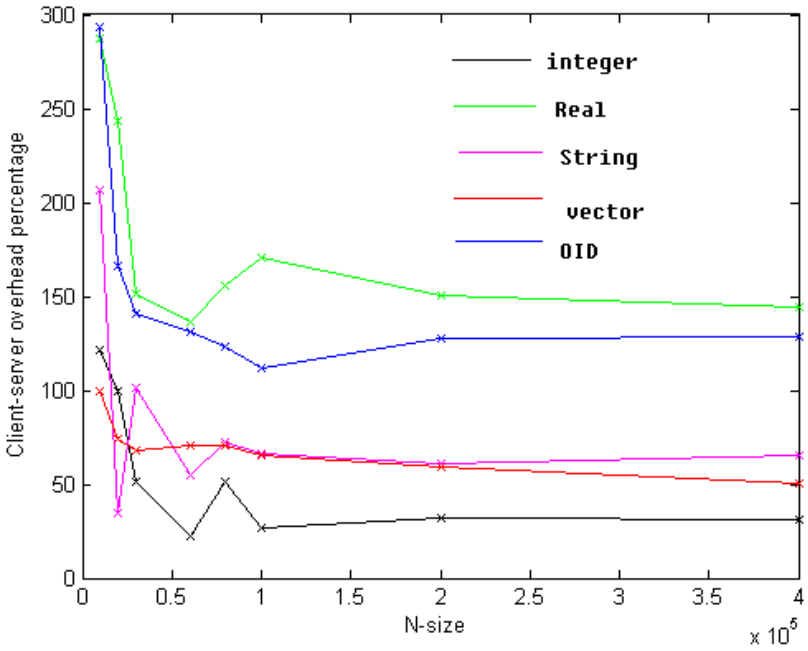


Figure 4.1: Client-server overhead percentage when result set size increases.

In figure 4.1, the client-server overhead percentage is measured. We notice that when the results set size is small, the client-server overhead is high (up to nearly 300% for OIDs), and it gets lower for large result sets.

The setup time for the TCP/IP is quite high compared to the tight connection. Thus, when the number of data elements is small, the setup time for TCP/IP is relatively expensive, so the curves in figure 4.3 all start from a high percentage. When the number of returned data elements increases the setup time for TCP/IP gets relatively cheaper so the curves in the figure the just show the overhead of sending more data elements by using TCP/IP compared to local connection.

5. Conclusion and future work

This report has described the implementation of the PyAmos database interface, which makes the functional DBMS Amos II callable from Python. By using PyAmos, Python user can make use of Amos II either by connecting to an embedded database inside Python via a tight connection, or connecting to database servers running Amos II. The approaches of how to interface Python and Amos II in C were discussed. The performance of the PyAmos interface for various parameters were made and compared with other Amos II APIs.

Functions in PyAmos can call Amos II functions and send ad hoc queries to Amos II. The result of functions and queries are represented as a new Python data type storing scans. There are furthermore functions used to make conversion between Amos II data types and Python types, thus the Amos II objects can be easily used in Python.

There are some other possibilities to continue working on this project in the future:

- Make it possible to implement Amos II foreign functions in Python. This could be done by interfacing the Amos II *callout* interface, as well as using the methods of calling Python from C code.
- Implement tight type integration between Amos II and Python. Python supports dynamic type creation. It could be possible to create Python objects dynamically that correspond to Amos II objects retrieved from the database, including their type names and data fields. If Amos II has a type named 'STUDENT', a corresponding 'STUDENT' type would be created in Python. Creating a 'STUDENT' instance in Python will then also create an object in the Amos II database.

Acknowledgement

I would like to thank Professor *Tore Risch* for giving me the interesting project. Prof. *Tore Risch* was my project supervisor and examiner; he gave me quite a lot of good ideas about the implementations and testing during the project. I'm grateful for the time he spent in discussing details of the project for getting me over the obstacles. Under his support, I learned a lot of things about database technology and enjoyed working on the project.

References

[1] AMOS II Wrapper, *UDBL whitepaper, Dept. of Information Technology, Uppsala University, Sweden.* http://user.it.uu.se/~udbl/amos/wrappers.html .

[2] B.Pharr: Getting to know Ruby. J. Comput. Small Coll. 21, 5 (May. 2006), 181-182.

[3] C.Werner: PHP Integration with object relational DBMS, UBDL wrapper, Dept. of information Technology, Uppsala, Uppsala University, Sweden. http://user.it.uu.se/%7Eudbl/Theses/ChristianWernerMSc.pdf .

[4] D.Elin, T.Risch: Amos II Java Interfaces. *UDBL Technical Report, Dept. of Information Technology, Uppsala University, Sweden*, 2000, http://user.it.uu.se/~torer/publ/javaapi.pdf.

[5] D. Shipman: The Functional Data Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, 6(1), 140-173, 1981.

[6] Elmasri, R. A. and Navathe, S. B. 1999 *Fundamentals of Database Systems*. 3rd. Addison-Wesley Longman Publishing Co., Inc.

[7] Embedding and extending Python, Python, http://docs.python.org/extending/ , last viewed 2009-10-18.

[8] G. Fahl and T. Risch: Query Processing over Object Views of Relational Data, *The VLDB Journal* , Vol. 6 No. 4, November 1997, pp 261-281.

[9] IBM DB2, IBM cooperation, http://www-01.ibm.com/software/data/db2/ , last viewed 2009-10-09.

[10] Java Database Connectivity (JDBC), Sun Microsystems, http://java.sun.com/javase/technologies/database/ , last viewed 2009-12-08.

[11] Liang, C. 2004: Programming language concepts and Perl. J. Comput. Small Coll. 19, 5 (May. 2004), 193-204.

[12] M.Sabesan and T.Risch: Web Service Mediation Through Multi-level Views, Proc. *International Workshop on Web Information Systems Modeling (WISM 2007)*, Trondheim, Norway, June 12, 2007.

[13] Microsoft SQL server, Microsoft, www.microsoft.com/sqlserver/2008/en/us/default.aspx, last viewed 2009-09-10.

[14] MySQL Database, Sun Microsystems, http://www.mysql.com/ ,last viewed 2009-12-08.

[15] N.Cholakov: On some drawbacks of the PHP platform. ACM International Conference Proceeding Series, Vol. 374, ISBN 978-954-9641-52-3, 2008.

[16] Open Database Connectivity (ODBC), Microsoft, http://msdn.microsoft.com/en-us/library/ms710252%28VS.85%29.aspx , last viewed 2009-12-08.

[17] Oracle Database, Oracle, http://www.oracle.com/database/index.html , last view 2009-09-11.

[18] P. Lyngbaek: *OSQL: A Language for Object Databases*, Tech. Report, HP Labs, HPL-DTD-91-4, 1991.

[19] PHP, OpenSource, http://php.net/index.php , last viewed 2009-12-08

[20] Python, Python Software foundation, http://www.python.org/ , last viewed 2009-10-08.

[21] Python C API document, Python software foundation, http://docs.python.org/c-api/ , last viewed 2009-10-12.

[22] Python DBAPI-2.0, http://www.python.org/dev/peps/pep-0249/ , last view 2009-09-12.

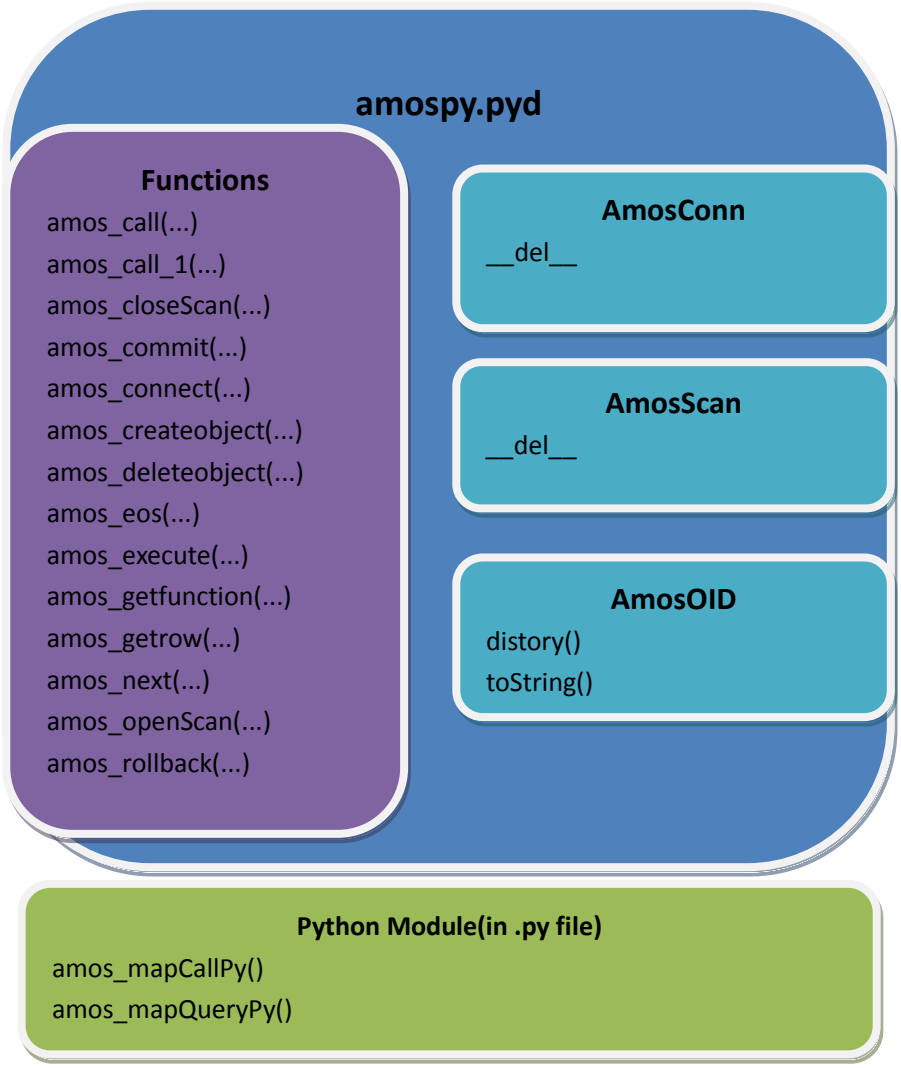
[23] T.Risch: Amos II External Interfaces. *UDBL Technical Report,Dept. of Information Technology, Uppsala University, Sweden*, 2001, http://user.it.uu.se/~torer/publ/external.pdf .

[24] T.Risch, V.Josifovski, and T.Katchaounov: Functional Data Integration in a Distributed Mediator System, in P.Gray, L.Kerschberg, P.King, and A.Poulovassilis (eds.): Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data, Springer, ISBN 3-540-00375-4, 2003.

[25] S.Flodin, M.Hansson, V.Josifovski, T.Katchaounov, T.Risch, and M.Skold: Amos II Release 12 User's Manual. *UDBL Technical Report, Dept. of Information Technology, Uppsala University, Sweden*, Nov 3, 2009, http://user.it.uu.se/~udbl/amos/doc/amos_users_guide.html.

Appendix 1 PyAmos Function References

PyAmos Overview



- amos_call(connection, function, a1,....,an) -> scan
  The *connection* is an AmosII connection object
  The parameter *function* is the Amos II function to call
  The parameters *a1,a2....* are parameters passed to the called Amos II function.
  amos_call() calls an Amos II function and returns the result as an *AmosScan* object.

- amos_call_1(connection, function, a1,....,an) -> scan
  The *connection* is an AmosII connection object
  The *function* is the Amos II function to call
  The parameters *a1,a2....* are passed to the called Amos II function
  amos_call_1() calls an Amos II function and returns a single value.

- amos_closeScan(scan)-> None
  amos_closeScan() closes an *AmosScan* Object

- amos_commit(conn) -> None
  Commits the changes

- amos_connect (dbName) -> *AmosConn*
  dbName : This is a string indicate the Amos II database.
  amos_connect() establishes a connection to an Amos II database.
  The *tight connection* is used if *dbName* is an empty string.
  The function returns an *AmosConn* object

- amos_createobject(conn,ObjectName) -> *AmosOID*
  The function will take an *AmosConn* object and a string representing the name of a type as parameters, and return a new *AmosOID* object.

- amos_deleteobject(conn,*AmosOID*)->None
  The function will take an *AmosConn* object and *AmosOID* object as parameter and delete the corresponding Amos II object.

- amos_disconnect(conn)->None
  Disconnect the session.

- amos_eos(scan)->None
  scan: an *AmosScan* object
  Return: Will return True or False.
  If the scan point to the end of a scan the function returns *True,* otherwise *False*.

- amos_execute(connection,queryString)-> *AmosScan*
  connection: Reference of *AmosConn* object created by *amos_connect()* function.
  queryString: string object,the query string that you want to execute
  amos_execute() executes an AmosQL query and returns an *AmosScan* object.

- amos_getfunction(connection,functionName,errorTrack) -> *AmosOID*
  connection: a connection object created by *amos_connect ()*
  functionName: a function name string
  The function returns an *AmosOID* of Amos II function

- amos_getrow(scan) ->Py_Tuple
  scan: a scan object
  The function returns the current row of the scan as a Python Tuple.

- amos_mapCallPy(pycallbackFunc, conn, amosFunc, a1,a2,……)->None

pycallbackFunc: A Python callback function. It can be a normal Python function or a *lambda* function.
conn : *AmosConn* object
amosFunc: Amos II function, e.g. “iota”
a1,a2,…an : parameters that are used by Amos Function.
For example: amos_mapCallPy(printFunc,conn, "iota", 0,15) # printFunc is Python function

- amos_mapQueryPy(pycallbackFunc, conn, amosQueryString)->None
  pycallbackFunc: A Python callback function. It can be a normal Python function or a *lambda* function.
  conn : *AmosConn* object
  amosQueryString: Amos II Query, e.g. “select name(t) from type t; ”
  For example: amos_mapQueryPy(printFunc,conn, "select name(t) from type t;")

- amos_next(scan)->None
  scan:the reference of scan object
  This function advances the current element of a scan to the next element.

- amos_rollback(conn)->None
  Rollback the execution.

Setup the PyAmos

The script ‘compile.cmd’ will compile the Python extension module and copy it to *%Python_BIN%DLLs.* The file ‘PyAmos.py’ is copied to *%Python_BIN%Libs.* Then you can try the system by running the ‘demo.py’ scripts in Python.

PyAmos Demonstration scripts:demo.py

```
import sys
from PyAmos import *

# use amos_connect()
conn = amos_connect("")

# use amos_execute()
scan = amos_execute(conn,"select name(t) from type t;")
while amos_eos(scan)!= True :
        row = amos_getrow(scan)
        print row
        amos_next(scan)
print "\n FROM .PY ******while-loop done\n"

# use amos_getfunction()
fct = amos_getfunction(conn,"charstring.typenamed->type")
```

```
# use amos_call()
scan1 = amos_call(conn,fct,'FUNCTION')
row1 = amos_getrow(scan1)
print row1

## use amos_call_1()
print "\nTest amos_call fucntion PLUS(3,8)\n"
res = amos_call_1(conn,"PLUS",3,8)
print res

amos_execute(conn,"create type newtype1;")
amos_rollback(conn)

amos_execute(conn,"create type newtype2;")
amos_execute(conn,"create type newtype3;")
amos_commit(conn)

amos_execute(conn,"create type newtype4;")
amos_rollback(conn)

scan = amos_execute(conn,"select name(t) from type t;")
while amos_eos(scan)!= True :
        row = amos_getrow(scan)
        print row
        amos_next(scan)
print "\n FROM .PY ******while-loop done\n"

# use amos_createobject() and amos_deleteobject()
obj2 = amos_createobject(conn,"newtype2")
print obj2.toString()
obj3 = amos_createobject(conn,"newtype3")
print obj3.toString()

amos_deleteobject(conn,obj2)

print "Testing tuple arguments and results in Amos II functions:\n"
a = (1.1,None,2,"2",3,True,False,obj3,(1,2,obj3) )
tpl = amos_getrow(amos_call(conn,'id', a));
print tpl

print "\nTesting OID, true, false, null values:\n"
tpl = amos_getrow(amos_call(conn, 'id', obj3))
print tpl
```

```
print tpl[0].toString()+ "\n"

tpl = amos_getrow(amos_call(conn, 'id', True));
print tpl

tpl = amos_getrow(amos_call(conn, 'id', False));
print tpl

tpl = amos_getrow(amos_call(conn, 'id', None));
print tpl

printf = lambda x:sys.stdout.write("%s\n"%x)
print "\nTest for amos_mapQueryPy() function\n"
amos_mapQueryPy(printf,conn,"select name(t) from type t;");
print "\nTest for amos_mapCallPy() function\n"
amos_mapCallPy(printf,conn,"iota",0,10)

amos_closeScan(scan)
amos_disconnect(conn)amos_disconnect(conn)
```

Appendix 2 Test Scripts of Performance Evaluation.

Test2.amosql

```
create function sendString(Charstring str)->Boolean as select true;
create function receiveString()->Charstring as select 'A receive string';
create function sendInt(Integer i)->Boolean as select true;
create function receiveInt()->Integer as select 11111;
create function sendReal(Real r)->Boolean as select true;
create function receiveReal()->Real as select 12.3456;
create function sendVector(Vector v)->Boolean as select true;
create function receiveVector()->Vector as select {0,1,2,3};
create type Person;
createobject('Person');
create function receiveObject()->Object o as
   select p from Person p;
```

Test2.py

```
import sys,time
from amospy import *
conn = amos_connect("")
################################################################################
amos_execute(conn,"< 'test2.amosql';")

t0 = time.time()
for i in range(1,10000):
        res = amos_call_1(conn,"sendstring","A Test String...")
t1 = time.time() -t0
print "\n t1 call1 func sendstring :%f\n"%(t1)

t0 = time.time()
for i in range(1,10000):
        res = amos_call_1(conn,"receivestring")
t1 = time.time() -t0
print "\n t1 call1 func receivestring :%f\n"%(t1)
################################################################################
t0 = time.time()
for i in range(1,10000):
        res = amos_call_1(conn,"sendInt",11111)
```

```
t1 = time.time() -t0
print "\n t1 call1 func sendInt :%f\n"%(t1)

t0 = time.time()
for i in range(1,10000):
        res = amos_call_1(conn,"receiveInt")
t1 = time.time() -t0
print "\n t1 call1 func receiveInt :%f\n"%(t1)
################################################################################
t0 = time.time()
for i in range(1,10000):
        res =amos_call_1(conn,"sendReal",12.3456)
t1 = time.time() -t0
print "\n t1 call1 func sendReal :%f\n"%(t1)

t0 = time.time()
for i in range(1,10000):
        res =amos_call_1(conn,"receiveReal")
t1 = time.time() -t0
print "\n t1 call1 func receiveReal :%f\n"%(t1)

################################################################################
t0 = time.time()
for i in range(1,10000):
        res =amos_call_1(conn,"sendVector",(0,1,2,3) )
t1 = time.time() -t0
print "\n t1 call1 func sendVector length 4:%f\n"%(t1)

t0 = time.time()
for i in range(1,10000):
        res =amos_call_1(conn,"sendVector",(0,1,2,3,4,5,6,7) )
t1 = time.time() -t0
print "\n t1 call1 func sendVector length 8 :%f\n"%(t1)

t0 = time.time()
for i in range(1,10000):
        res =amos_call_1(conn,"receiveVector")
t1 = time.time() -t0
print "\n t1 call1 func receiveVector :%f\n"%(t1)

################################################################################
oid = amos_createobject(conn,"Person")
print oid.toString()
t0 = time.time()
```

```
for i in range(1,10000):
        res =amos_call_1(conn,"sendObject",oid )
t1 = time.time() -t0
print "\n t1 call1 func sendObject :%f\n"%(t1)

t0 = time.time()
for i in range(1,10000):
        res = amos_call(conn,"receiveObject")
t1 = time.time() -t0
print "\n t1 call1 func RecieveObject :%f\n"%(t1)
```

Test3.amosql

```
create function RealResult(Integer size)-> Bag of Real nm as
    select 1.0 from Integer i where i in iota(1,size);

create function StringResult(Integer size)-> Bag of Charstring nm as
    select 'A Returning String result' from Integer i where i in iota(1,size);

create function VectorResult(Integer size)-> Bag of Vector nm as
    select {1,2,3,4} from Integer i where i in iota(1,size);
```