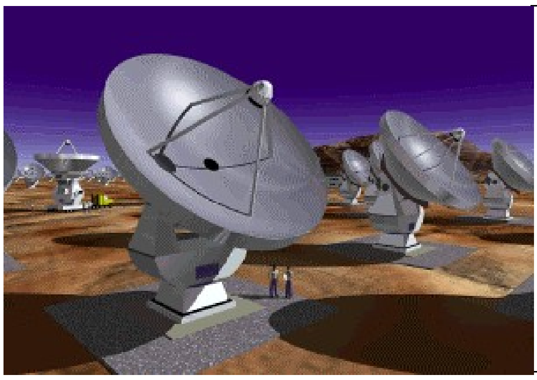


Atacama Large Millimeter	ALMA-SW-NNNN Revision: 4.1 2005-06-20

ALMA Common Software and Python

Design and Tutorial

David Fugate
University of Calgary

Keywords: ACS, acspy, Python, CORBA, IDL, Client, Servant

Author Signature: David Fugate	Date: 2003-11-19
Approved by: Institute:	Signature: Date:
Released by: Institute:	Signature: Date:

Change Record

REVISION	DATE	AUTHOR	SECTIONS/PAGES AFFECTED
	REMARKS		
3.0	2003-11-19	D. Fugate	All
	Created		
3.1	2004-05-10	D. Fugate	All
	Updated for ACS 3.1		
4.1.0	2005-06-20	D. Fugate	All
	Updated for ACS 4.1		

Table of Contents

1 Overview

This document describes in detail the ALMA Common Software Python API and how to use it. For those unfamiliar with Python, it is a high-level, object-oriented, interpreted programming language that is both dynamically-typed and scoped. C and C++ extensions to Python are very easy to create and it runs incredibly fast for an interpreted language. Python is great for scripting, prototyping, web development, and high-level applications. It's also a wonderful language for non-programmers to use because of its simplicity. Most importantly for ALMA, OMG has defined an IDL to Python CORBA mapping and several vendors offer free ORBs. This implies that anything done in C++ or Java can also be coded in Python! ALMA uses the omniORBPy ORB because of its stability and performance.

Specifically, this document intends to help the reader in four core areas: the IDL to Python mapping, ACS Python utility and helper packages, Python CORBA clients, and Python CORBA servants.

Short when compared to other IDL language mappings, **it's still vital to learn the complete IDL to Python mapping**. The easiest way to do this is using OMG's specification document ([R05]), although studying the entire specification is overkill.

Python helper packages have been contributed by a diverse group including ALMA Control, APEX, and ACS developers. In short, they generally provide the same (if not more) functionality as the C++/Java APIs and are developed in a much shorter time span. These subpackages can be found in the *Acspy* package located in the *src* directory of the acspy module (module in the sense of an ALMA Software Engineering module). They range from functions designed to find files located in the ALMA directory structure to classes capable of reading/parsing XML files from the ACS Configuration Database.

Client and servant development run hand-in-hand so to speak. This document provides step-by-step tutorials on developing ACS Python clients and servants by using helper classes. These helper classes are extremely similar to what's been developed for C++ and Java because the helpers are built on common IDL interfaces.

Developers without Python programming experience are highly encouraged to read the first ten chapters of Python Essential Reference ([R03]). This may sound like a large undertaking, but should take most individuals under a couple of hours to read.

Finally, every effort is made to ensure this document is kept up-to-date for each ACS release. However, sometimes things are overlooked and one should always trust the pydoc for the *Acspy* package over material presented here as it is guaranteed to be more recent ([R09]).

1.1 Abbrevations

ACS	ALMA Common Software
acspy	ALMA Common Software Python module
acspyexmpl	ALMA Common Software Python example module
ALMA	Atacama Large Millimeter Array
APEX	Atacama Pathfinder Experiment
BACI	Basic Access and Control Interfaces

CORBA	Common Object Request Broker Architecture
IDL	Interface Definition Language
MACI	Management and Control Interface
OMG	Object Management Group
ORB	Object Request Broker

1.2 References

[R01] ACS Architecture 5.0

(http://www.eso.org/projects/alma/develop/acs/OnlineDocs/ACSArchitectureNL.pdf)

[R02] ACS CVS Software Modules: “acspy” and “acsexmpl”

(ACS/LGPL/CommonSoftware/*)

[R03] Python Essential Reference (Beazley)

[R04] CORBA

(http://www.omg.org/technology/documents/formal/corba_iiop.htm)

[R05] OMG IDL-Python Mapping

(http://www.omg.org/technology/documents/formal/python.htm)

[R06] ALMA Python Coding Standards

(http://www.eso.org/projects/alma/develop/alma-se/reference/CodingStandards.html)

[R07] Notification Channel (Design & Tutorial)

(http://www.eso.org/~gchiozzi/AlmaAcs/OnlineDocs/Notification_Channel_Module_Software_Design.pdf)

[R08] BACI Specifications Document (http://www.eso.org/~gchiozzi/AlmaAcs/OnlineDocs/ACS_Basic_Control_Interface_Specification.pdf)

[R09] Acspy Pydoc (http://www.eso.org/projects/alma/develop/acs/OnlineDocs/ACS_docs/py/index.html)

[R10] ACS Error System Document (http://www.eso.org/projects/alma/develop/acs/OnlineDocs/ACS_Error_System.pdf)

[R11] ACS Time System Document (http://www.eso.org/projects/alma/develop/acs/OnlineDocs/ACSTime.pdf)

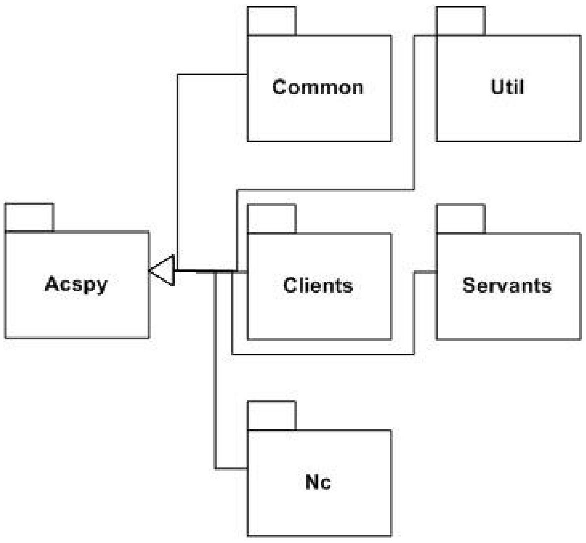
2 Design

2.1 Requirements

- The *Acspy* package shall abide by the standards set fourth in the **ALMA Python Coding Standards** ([R06]).
- Features implemented in C++ and Java shall be reimplemented in Python where feasible and applicable.
- XML support shall be provided in some form.

2.2 Package Structure

All ALMA Common Software Python packages reside in one core package – *Acspy*.



Common

The *Common* subpackage provides Python modules which are of use to all developers. This consists of:

- **Callbacks.py** – Callback helper classes useful for clients accessing *ACS::CharacteristicComponent*'s
- **CDBAccess.py** – provides read-only access into the ACS Configuration Database.

- **Err.py** – provides the superclass for ACS Error System generated exceptions and completions.
- **Log.py** – ACS Logging System helper class.
- **TimeHelper.py** – ACS Time System helper class.
- **EpochHelper.py** – provides a wrapper for the *acstime::Epoch* IDL struct.
- **DurationHelper.py** – provides a wrapper for the *acstime::Duration* IDL struct.
- **QoS.py** – sets ORB timeouts for COBRA method calls

Util

The *Util* subpackage provides Python modules which are generally of use only to ACS developers. Classes and functions found here provide low-level access to CORBA and the host's filesystem. The entire package can be safely ignored by most developers.

Clients

Clients consists of a Python module, **SimpleClient.py**, used by developers to manipulate remote components and access manager services. Also, see **ContainerServices.py**

Servants

The *Servants* subpackage provides Python modules containing helper classes for coding Python CORBA servants:

- **ACSComponent.py** – an implementation of the *ACS::ACSComponent* IDL interface.
- **ComponentLifecycle.py** – a concrete implementation of the Lifecycle interface defined by the High-level Architecture team.
- **ContainerServices.py** – common services provided by the container for Python servants and clients.
- **CharacteristicComponent.py** – a prototype implementation of the *ACS::CharacteristicComponent* IDL interface. Use at your own risk.

Nc package

Nc subpackage contains Python modules to manipulate notification channels. A full description of all the classes it contains can be found in the **Notification Channel Design and Tutorial** ([R07]).

3 IDL to Python Mapping

3.1 Guidelines

- Always define IDL types within a module. Otherwise the global IDL types will be mapped to a package that is ORB-vendor specific.
- Try to stay away from giving your IDL files the same name as standard Python packages. There's a slim chance this can cause problems.
- Do not name IDL modules names that are also standard Python packages or ALMA Python packages. For example, defining "module Acspy" in some IDL file would wreak major havoc.
- It is recommended that definitions of constants, structures, enumerations, etc. be placed in the IDL module and not within the definition of an IDL interface. This suggestion can make your life much easier.
- If an enumeration has been declared within an IDL module, treat the values for that enumeration as if they were reserved words throughout the rest of the module. Failure to do so can result in all sorts of fun problems.
- The following are reserved keywords in Python and shall not be used in your IDL:

And	assert	break	class	continue
Def	del	elif	else	except
Exec	finally	for	from	global
If	import	in	is	lambda
Not	or	pass	print	raise
return	try	while	none	

3.2 Simple Types

OMG IDL	Python
octet	`Integer (<type 'int'>)`
short	`Integer`
Long	`Integer`
unsigned short	`Integer`
unsigned long	`Long integer(<type 'long int'>)`
Long long	`Long integer`
unsigned long long	`Long integer`
float	`Floating Point Number (<type 'float'>)`
Double	`Floating Point Number`
Long double	`CORBA.long_double`
boolean	`Integer (0 or 1)`
Char	`string of length 1`
wchar	`Wide string of length 1`

3.3 Complex Types

OMG IDL	Python
((Un)bounded) string	`String`
(Sequences/arrays)	`Lists or tuples`
enum	An enumeration is mapped into a number of constant objects in the

	namespace where the enumeration is defined. Please do not use the native Python copy methods on CORBA enumerations.
Struct	An IDL struct definition is mapped into a Python class or type. For each field in the struct, there is a corresponding attribute in the class with the same name as the field. The constructor of the class expects the field values, from left to right.
(Constants)	An IDL constant definition maps to a Python variable initialized with the value of the constant.
(Exceptions)	An IDL exception is translated into a Python class derived from CORBA.UserException. System exceptions are derived from CORBA.SystemException. Both base classes are derived from CORBA.Exception. The parameters of the exception are mapped in the same way as the fields of a struct definition. When raising an exception, a new instance of the class is created and the constructor expects the exception parameters.

3.4 *in*, *out*, and *inout* Parameters

in parameters are mapped as one would expect in Python. That is if some interface *I* contains a method, *void methodA(in string parameterX)*, a client would invoke it like this – *I. methodA("not interesting").* However, things get much more interesting with the other two cases.

Let's see another method of the *I* interface which incorporates *in*, *out*, and *inout* parameters in addition to a return value:

double methodB(in string parameterX, inout boolean isOK, out long someValue);

A client would invoke *methodB* like this:

(methodBReturnValue, boolOutValue, longValue) = I.methodB("…", 0);

Now for the explanation – since every type in Python is an object which is really just a pointer, the concept of references to objects does not really exist. In short, parameters passed to Python functions and methods are complete copies of the objects (i.e., pass by value). To deal with this, the CORBA mapping for Python specifies that *out* parameters will be returned via a tuple. The first object in this tuple is the return value if the method has one, followed by *inout/out* parameters in the order they were specified from left to right. To make this clearer, the implementation of *methodB* in a Python servant could be the following:

```
def methodB(self,  parameterX, isOK):
   # 3.14 corresponds to the return double value
   # 1 is for 'inout Boolean isOK'
   # 34 is 'out long someValue'
   return (3.14, 1, 34)
```

3.5 Narrowing CORBA Objects to their Real Type

Because ALMA has chosen to use **omniORBPy** as our Python ORB, 99 out of 100 times this is not necessary and the ORB does it automatically. The only time this needs to be done is when an IDL interface uses complex inheritance and the object needs to be narrowed to a base interface. If this is ever the case, all CORBA objects have a *"_narrow(I)"* method where *I* is the Python CORBA stub class for the interface you want to narrow the object to.

4 Utility and Helper Packages

Precise examples in this section have yet to be determined. In the meantime, you can look at the test scripts located in the ALMA CVS repository which should be pretty self-explanatory.

4.1 ACS Configuration Database

See ACS/LGPL/CommonSoftware/acspy/test/acspyTestCDB.py in the ALMA CVS repository.

4.2 ACS Error System

See the ACS Error System document ([R10]).

4.3 ACS Logging System

See ACS/LGPL/CommonSoftware/acspy/test/acspyTestLogging.py in the ALMA CVS repository.

4.4 ACS Time System

See the ACS Time System document ([R11]).

5 Client Tutorial

All ACS Python clients must use the *Acspy.Clients.SimpleClient* module. This module defines a single class, *PySimpleClient*, which implements the *Client* IDL interface and is responsible for logging into manager and providing ACS services to the developer.

Some PySimpleClient Methods/Attributes Available to Developers

Method	Description
availableComponents()	Gets a list of *ComponentInfo* structures (see *maci.idl* for a definition) about all known components.

	Parameters: None Returns: a list consisting of *ComponentInfo* structures for every component manager knows of or None if this method fails. Raises: ???
getComponent(name)	Get a component reference from manager. It's important to note this method narrows the reference for the developer and even imports the proper IDL Python stub. Parameters: name is the component's name Returns: a narrowed reference to the component or None if that reference cannot be obtained. Raises: ???
releaseComponent(name)	Release the component defined by name. Parameters: name is the component's name Returns: the number of clients/components attached to the component or None on failure. Raises: ???
token.h	Security token for this client. If *PySimpleClient* does not provide access to some manager method that is needed, you can always use the *getManager* function of the *ACSCorba* module directly.

An Example

The example depicted here consists of a client which logs into manager, accesses an *ACS::CharacteristicComponent*, and then creates a monitor for one of that component's BACI properties. Please note error handling has been omitted.

```
#!/usr/bin/env python

from Acspy.Clients.SimpleClient import PySimpleClient
```

```
4:  import ACS, ACS__POA
5:  #------------------------------------------------------------------------------
6:  class MyMonitor(ACS__POA.CBdouble):
7:      def working(self, value, completion, desc):
8:          print "Working…the value is: ", str(value)
9:      def done(self, value, completion, desc):
10:         print "Done.  The value is: ", str(value)
11:     def negotiate (self, time_to_transmit, desc):
12:         return 1
13: #------------------------------------------------------------------------------
14: simpleClient = PySimpleClient()
15:
16: mount = simpleClient.getComponent("MOUNT1")
17: actAzProperty = mount._get_actAz()
18:
19: cbMon = MyMonitor()
20: cbMonServant = simpleClient.activateOffShoot(cbMon)
21: desc = ACS.CBDescIn(0L, 0L, 0L)  # Create the real monitor registered with MOUNT1
22: actMon = actAzProperty.create_monitor(cbMonServant, desc)
23:
24: actMon.set_timer_trigger(10000000)
25: from time import sleep
26: sleep(10)
27: actMon.destroy()
28:
29: simpleClient.releaseComponent("MOUNT1")
30: simpleClient.disconnect()
```

3 *PySimpleClient* is the standard Python client capable of logging into manager.

4 Import the Python CORBA stubs for the BACI module. These are only necessary because callbacks are being used.

6-15 Since an asynchronous monitor will be created for a BACI double property, an implementation of the double callback IDL interface must be defined. The *Acspy.Common.Callbacks* module could have been used, but where's the fun in that?

6 Our class must be derived from *ACS__POA.CBdouble* which is generated by the IDL to Python compiler. *MyMonitor* is actually a servant!

7-8 The *working* method is invoked when a value changes for on-change monitors or when the timing trigger has been executed. Developers override this method to do the real monitoring of data values. The *value* parameter is the real data value (*completion* and *desc* parameters are not too useful except for timestamps).

9-10 *done* is essentially identical to the *working* method except for the fact that done is invoked when the monitor is destroyed.

11-12 The only useful thing to know about negotiate is that it should normally return a true value (i.e., 1). For developers really interested in understanding this method, please see the **BACI Specifications** ([R08]).

14 Standard way of instantiating a client which logs into manager to access common services and components.

16 *getComponent* method of *PySimpleClient* is quite literally used to get a named component, activating it if necessary. Please take special note we never even had to import the CORBA Python stubs for the *MOUNT1* component. *PySimpleClient* does this automatically!

17 Under the Python mapping, references to attributes of interfaces are obtained by prepending "_get_" to the attribute's name and invoking that method on the interface.

19-20 An instance of the callback class is created. On line 20, the subclass becomes a CORBA object by utilizing the *activateOffShoot ContainerService* method.

21 Explaining *CBDescIn* and similar IDL structs/interfaces is beyond the scope of this document. This line of code can be simply copied without really understanding what it does. Feel free to review the **BACI Specifications** document ([R08]) if you're really interested though.

24 Override the default monitor time using this method. It now monitors the property once per second.

25-27 Allow the monitor to run for ten seconds and then destroy it.

29 Since we're done with this component, release it. Depending on whether or not other clients/components reference *MOUNT1*, it may or may not be deactivated.

30 Must always disconnect from manager when we're through.

6 Servant Tutorial

ACS provides a full implementation of the *Container* IDL interface in Python implying that any IDL interface derived from *ACS::ACSComponent* can be implemented in Python. Furthermore, specific support classes have been placed in the *Acspy.Servants* package to make the developers life easier. In this section, the process of creating a Python CORBA servant is covered from the ground up.

Developing an IDL Interface

Regardless of servant implementation language, the first thing that needs to be done is to define an IDL interface. It's assumed the developer has some background in CORBA and IDL so we will not go into too much detail here. Besides the guidelines presented in the IDL to Python mapping section, there is **only one rule which must be followed**: developer-defined IDL interfaces must be derived from the *ACS::ACSComponent* interface in some form. This stipulation is imposed on all programming languages ACS supports! Now an example IDL will be presented which will be used throughout the rest of the servant tutorial.

```
1:  #ifndef _HELLODEMO_IDL_
2:  #define _HELLODEMO_IDL_
3:
4:  #include <acscomponent.idl>
5:
6:  #pragma prefix "alma"
7:
8:  module demo
9:  {
10:     interface HelloDemo : ACS::ACSComponent
11:     {
12:         string sayHello();
13:         string sayHelloWithParameters(in string inString,
14:                                       inout double inoutDouble,
15:                                       out long outInt);
16:     };
17: };
18: #endif
```

1, 2, & 13	It is critical that IDL files use include guards.
4	Used to access the *ACSComponent* interface.
6	This line is standard for **all** ALMA IDL interfaces.
8	The module's name is arbitrary except that it should not conflict with Python package names (i.e., "module sys" or "module Acspy" are extremely bad names).
10	Choose any name you want for the interface. Just make sure it is derived from *ACSComponent*.

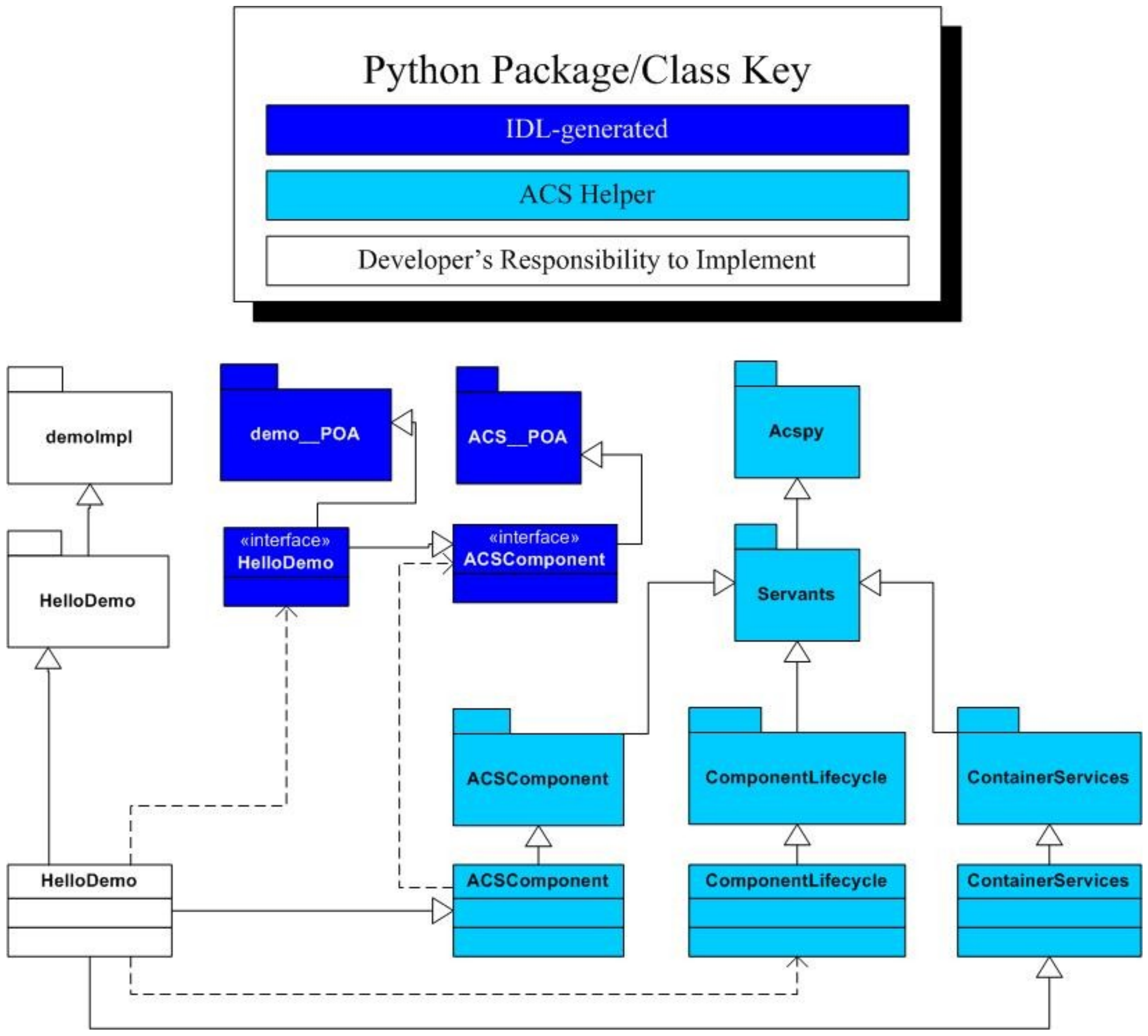
12-15 Two methods are defined for this simple example. Both return strings and the difference between the two is that one uses parameters.

The *Servants* Package

Servant helper classes can be found in *Acspy.Servants*. They are designed to provide services to servants, give the developer an implementation of the *ACSComponent* interface, as well as provide methods which only the container can invoke when the component is started, stopped, or ends up in some error state (*LifeCycle* interface). For details, see the pydoc.

A Python Implementation for the IDL Interface

So far an IDL interface has been defined in the "demo" module. Well the Python mapping specifies the IDL compiler will generate a Python package for that IDL module and name it "demo__POA". By convention, we will create a Python package which just consists of "Impl" appended to the IDL module's name. In our case, this will be the "demoImpl" package. Also by convention, we place the real implementation of the IDL interface inside a Python module of an identical name. This must sound pretty confusing, but this class diagram should hopefully help some:



Now the *src* directory you're working in should hopefully contain the following developer-defined Python package - *demoImpl/HelloDemo.py*. Now we can finally see what the real implementation looks like:

```
import demo__POA
from Acspy.Servants.ContainerServices     import ContainerServices
from Acspy.Servants.ComponentLifecycle  import ComponentLifecycle
from Acspy.Servants.ACSComponent        import ACSComponent
#---------------------------------------------------------------------------
class HelloDemo(demo__POA.HelloDemo,
                ACSComponent,
                ContainerServices,
                ComponentLifecycle):
    #---------------------------------------------------------------------------
    def __init__(self):
        ACSComponent.__init__(self)
        ContainerServices.__init__(self)
    #--Override ComponentLifecycle methods----------------------------
```

```
15:    def initialize(self):
16:        try:
17:            lamp = self.getComponent("LAMP1")
18:            self.brightness = lamp._get_brightness()
19:        except Exception, e:
20:            print "LAMP1 unavailable:", e
21:    #---------------------------------------------------------------------------
22:    def cleanUp(self):
23:        self.releaseComponent("LAMP1")
24:    #--Implementation of IDL methods-------------------------------------
25:    def sayHello(self):
26:        return "hello"
27:    #---------------------------------------------------------------------------
28:    def sayHelloWithParameters(self, inString, inoutDouble):
29:        self.getLogger().logInfo("sayHello called with arguments inString="
30:                                + inString
31:                                + "; inoutDouble="
32:                                + str(inoutDouble)
33:                                + ". Will return 'hello'..."
34:                                         )
35:        return ("hello", inoutDouble, 23)
36:
37: #--Main defined only for generic testing-------------------------------
38: if __name__ == "__main__":
39:    g = HelloDemo()
```

1 Must import the package created by the IDL compiler for the IDL interface we will be developing a servant implementation for.

2-4 ACS servant helper modules are imported.

5-35 Class *HelloDemo* is the concrete implementation of the *HelloDemo* IDL interface. It should be noted that the container requires the implementing Python class to have the same name as the IDL interface (i.e., *HelloDemo*).

6 The implementing class must be derived from the CORBA-generated class defining the interface's methods and attributes.

7 *ACSComponent* provides an implementation of the *ACS::ACSComponent* interface and it is mandatory that components be derived from this class.

8 Derivation from *ContainerServices* is highly recommended as this class provides access to other components, ACS services, etc.

9 The *ComponentLifecycle* class contains methods (to be overridden) that are only accessible by the Container. Components should not invoke *ComponentLifecycle* methods on themselves.

10-13 If a developer defines a constructor for the servant implementation, it must call *ACSComponent* and *ContainerService*'s constructors!

14-23 Two *ComponentLifecycle* methods are overridden and show usage of a couple of *ContainerServices* methods. These are called at object creation and destruction times by the container.

15-20 In the *initialize* method, a reference to the "LAMP1" component is obtained and from that we get the "brightness" BACI property.

22-23 A properly written servant/client will always release other components when done using them.

24-35 Implementations of the various IDL methods that were defined for the *HelloDemo* interface. While this should be pretty straightforward, please review the section on parameter ordering in the Python mapping chapter if any of this seems confusing.

37-39 Its good practice to ensure your component can at least be instantiated as a Python object. These three lines of code have saved the author an incredible amount of debugging time and should prove valuable to other developers.

Unlike C++ and Java servants, there are no macros or helper classes that must be created by the developer to make an instance of the servant. In essence, these 38 lines consist of the only Python code that needs to be created!

Editing the Configuration Database

Specific instructions on creating an ACS Configuration Database can be found in various other documents, so we will assume only modifications to an existing CDB need to be made. There is only one file that needs to be modified - *$ACS_CDB/CDB/MACI/Components/Components.xml*. Here, we just add one entry of the "_" element per component we wish to create:

```
1:  <_ Name="HELLODEMO1"
2:     Code="demoImpl.HelloDemo"
3:     Type="IDL:alma/demo/HelloDemo:1.0"
4:     Container="aragornContainer"/>
```

1 Name of the component.

2 Name of the Python package containing an implementation of the IDL interface defined on line 3.

3 The interface repository location of the servant's IDL interface determined from the IDL file itself. In this case:

- *alma* corresponds to *#pragma prefix "alma"*
- *demo* comes from *module demo*
- *HelloDemo* is the IDL interface's name.
- For all intensive purposes, the version number will always be *1.0*.

4 The name of the Python container responsible for the lifecyle of this component.

Makefile Support

There are several Makefile targets dealing with IDL files, Python packages, etc. developers should familiarize themselves with:

Target	Description
IDL_FILES	List of IDL files (without their '.idl' postfix) separated by spaces that can be found in the *../idl* directory. Specifying this target means the Python CORBA stubs will be created when doing a 'make all' from the *src* directory.
PYTHON_PACKAGES	List of Python packages in the *src* directory. Byte-compiled by the 'all' target.
PYTHON_MODULES	List of Python modules in the *src* directory. Byte-compiled by the 'all' target.
PYTHON_PACKAGES_L & PYTHON_MODULES_L	Same as the previous two descriptions except that the 'install' target does not install them into *$INTROOT*.

7 Known Problems and Issues

Please see the latest ACS Release Notes referenced from the ACS Homepage for this information.

8 Appendix

ALMA CVS Repository

ACS/LGPL/CommonSoftware/acspyexmpl/src/acspyexmplMountCallback.py

ACS/LGPL/CommonSoftware/acspyexmpl/src/demoImpl/HelloDemo.py

ACS/LGPL/CommonSoftware/jcontexmpl/idl/HelloDemo.idl