Using Python with Caché

Version 2018.1
2018-12-07

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

Using Python with Caché
Caché Version 2018.1 2018-12-07

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

About This Book

This book is a guide to the Caché Python Language Binding.

This book contains the following sections:

- The Caché Python Binding
- Using the Python Binding
- Python Client Class Reference

There is also a detailed Table of Contents.

For general information, see *Using InterSystems Documentation*.

1

The Caché Python Binding

The Caché Python binding provides a simple, direct way to manipulate Caché objects from within a Python application. It allows Python programs to establish a connection to a database on Caché, create and open objects in the database, manipulate object properties, save objects, run methods on objects, and run queries. All Caché datatypes are supported.

The Python binding offers complete support for object database persistence, including concurrency and transaction control. In addition, there is a sophisticated data caching scheme to minimize network traffic when the Caché server and the Python applications are located on separate machines.

This document assumes a prior understanding of Python and the standard Python modules. Caché does not include a Python interpreter or development environment.

1.1 Python Binding Architecture

The Caché Python binding gives Python applications a way to interoperate with objects contained within a Caché server. The Python binding consists of the following components:

- *The intersys.pythonbind module* — a Python C extension that provides your Python application with transparent connectivity to the objects stored in the Caché database.
- *The Caché Object Server* — a high performance server process that manages communication between Python clients and a Caché database server. It communicates using standard networking protocols (TCP/IP), and can run on any platform supported by Caché. The Caché Object Server is used by all Caché language bindings, including Python, Perl, C++, Java, JDBC, and ODBC.

The basic mechanism works as follows:

- You define one or more classes within Caché. These classes can represent persistent objects stored within the Caché database or transient objects that run within a Caché server.
- At runtime, your Python application connects to a Caché server. It can then access instances of objects within the Caché server. Caché automatically manages all communications as well as client-side data caching. The runtime architecture consists of the following:
  - A Caché database server (or servers).
  - The Python interpreter (see Python Client Requirements).
  - A Python application. At runtime, the Python application connects to Caché using either an object connection interface or a standard ODBC interface. All communications between the Python application and the Caché server use the TCP/IP protocol.

1.2 Quick Start

Here are examples of a few basic functions that make up the core of the Python binding:

- *Create a connection and get a database*

```
conn = intersys.pythonbind.connection()
conn.connect_now(url,user,password, None)
database = intersys.pythonbind.database(conn)
```

  `database` is your logical connection to the namespace specified in `url`.

- *Open an existing object*

```
person =  database.openid("Sample.Person",str(id),-1,-1)
```

  `person` is your logical connection to a `Sample.Person` object on the Caché server.

- *Create a new object*

```
person =  database.create_new("Sample.Person", None)
```

- *Set or get a property*

```
person.set("Name","Doe, Joe A")
name = person.get("Name")
```

- *Run a method*

```
answer = person.run_obj_method("Addition",[17,20])
```

- *Save an object*

```
person.run_obj_method("%Save",[])
```

- *Get the id of a saved object*

```
id = person.run_obj_method("%Id",[])
```

- *Run a query*

```
sqlstring ="SELECT ID, Name, DOB, SSN \
            FROM SAMPLE.PERSON \
            WHERE Name %STARTSWITH ?"
query = intersys.pythonbind.query(database)
query.prepare(sqlstring)
query.set_par(1,"A")
query.execute();
while 1:
   cols = query.fetch([None])
   if len(cols) == 0: break
   print cols
```

1.3 Installation and Configuration

The standard Caché installation places all files required for Caché Python binding in <cachesys>/dev/Python. (For the location of <cachesys> on your system, see Default Caché Installation Directory in the *Caché Installation Guide*). You should be able to run any of the Python sample programs after performing the following installation procedures.

1.3.1 Python Client Requirements

Caché provides client-side Python support through the intersys.pythonbind module, which implements the connection and caching mechanisms required to communicate with a Caché server.

This module requires the following environment:

- Python version 2.7 or Python 3.0+. For Windows, Intersystems supports only the ActiveState distribution, ActivePython© (www.activestate.com).
- A C++ compiler to generate the Python C extension. On Windows, you need Visual Studio .NET 2008 or higher (required by the ActiveState distribution). On UNIX®, you need GCC.

  The bitness of both your Python distribution and your compiler must match the bitness of Caché: 64-bit systems require the 64-bit versions of Python and compiler, and 32-bit systems require the 32-bit versions of Python and compiler.
- On RedHat Linux, the `python-devel` package must be installed in order to compile the python sample.
- Your PATH must include the <cachesys>\bin directory. (For the location of <cachesys> on your system, see Default Caché Installation Directory in the *Caché Installation Guide*).
- Set up your environment variables to support C compilation and linking, as described in the following sections.

1.3.2 UNIX® Installation

- Make sure <cachesys>/bin is on your PATH and in your LD_LIBRARY_PATH. (For the location of <cachesys> on your system, see Default Caché Installation Directory in the *Caché Installation Guide*).

  For example:

  ```
  export PATH=/usr/cachesys/bin:$PATH
  export LD_LIBRARY_PATH=/usr/cachesys/bin:$LD_LIBRARY_PATH
  ```

  Note: Mac OS X uses DYLD_LIBRARY_PATH instead of LD_LIBRARY_PATH. For example:

  ```
  export DYLD_LIBRARY_PATH=/usr/cachesys/bin:$DYLD_LIBRARY_PATH
  ```

- Run setup.py (located in <cachesys>/dev/python):

  ```
  python setup.py  install
  ```

  The following prompt is displayed:

  ```
  enter directory where you installed Cache'
  ```

- At the prompt, supply the location of <cachesys>. For example:

  ```
  /usr/cachesys
  ```

  The resulting lib and include paths will be displayed:

  ```
  libdir=/usr/cachesys/bin
  include dir=/usr/cachesys/dev/cpp/include
  ```

- Run test.py (located in dev/python/samples) to test the installation:

  ```
  python test.py
  ```

  Do not run test programs from <cachesys>/dev/python or the test program will not be able to find the pythonbind module. The python path is relative and you will pick up files from the intersys subdirectory instead.

1.3.3 Windows Installation

- Make sure your path and environment is setup to run the Microsoft C/C++ compiler. Follow the Microsoft instructions. For example from the command line, run:

  ```
  vsvars32.bat
  ```

  for 32-bit systems, or:

  ```
  vcvarsall.bat x64
  ```

  for 64-bit systems. These files set up the path and environment variables for using the Microsoft C/C++ compiler. Please read your Microsoft documentation to determine the location of these .bat files, since the location varies depending on the version of Visual Studio you are using.

- Run setup.py, located in <cachesys>/dev/python (for the location of <cachesys> on your system, see Default Caché Installation Directory in the *Caché Installation Guide*):

  ```
  python setup.py install
  ```

  The following prompt is displayed:

  ```
  enter directory where you installed Cache'
  ```

- At the prompt, supply the location of <cachesys>. For example:

  ```
  C:\Intersystems\Cache
  ```

  The resulting lib and include paths will be displayed:

  ```
  libdir=C:\Intersystems\Cache\dev\cpp\lib
  include dir=C:\Intersystems\Cache\dev\cpp\include
  ```

- Run test.py (located in <cachesys>\dev\python\samples) to test the installation:

  ```
  python test.py
  ```

  Do not run test programs from <cachesys>\dev\python or the test program will not be able to find the pythonbind module. The python path is relative and you will pick up files from the intersys subdirectory instead.

1.3.4 Caché Server Configuration

Very little configuration is required to use a Python client with a Caché server. The Python sample programs provided with Caché should work with no change following a default Caché installation. This section describes the server settings that are relevant to Python and how to change them.

Every Python client that wishes to connect to a Caché server needs the following information:

- A URL that provides the server IP address, port number, and Caché namespace.
- A username and password.

By default, the Python sample programs use the following connection information:

- URL: `"localhost[1972]:Samples"`
- username: `"_SYSTEM"`
- password: `"SYS"`

Check the following points if you have any problems:

- Make sure that the Caché server is installed and running.
- Make sure that you know the IP address of the machine on which the Caché server is running. The Python sample programs use `"localhost"`. If you want a sample program to default to a different system you will need to change the connection string in the code.
- Make sure that you know the TCP/IP port number on which the Caché server is listening. The Python sample programs use `"1972"`. If you want a sample program to default to a different port, you will need change the number in the sample code.
- Make sure that you have a valid username and password to use to establish a connection. (You can manage usernames and passwords using the Management Portal). The Python sample programs use the administrator username `"_SYSTEM"` and the default password `"SYS"` or `"sys"`. Typically, you will change the default password after installing the server. If you want a sample program to default to a different username and password, you will need to change the sample code.
- Make sure that your connection URL includes a valid Caché namespace. This should be the namespace containing the classes and data your program uses. The Python samples connect to the SAMPLES namespace, which is pre-installed with Caché.

1.4 Sample Programs

The standard Caché installation contains a set of sample programs that demonstrate the use of the Caché Python binding. These samples are located in:

<cachesys>/dev/Python/samples/

(For the location of <cachesys> on your system, see Default Caché Installation Directory in the *Caché Installation Guide*)

The following sample programs are provided:

- CPTest2.py — Get and set properties of an instance of Sample.Person.
- CPTest3.py — Get properties of embedded object Sample.Person.Home.
- CPTest4.py — Update embedded object Sample.Person.Home.
- CPTest5.py — Process datatype collections.
- CPTest6.py — Process the result set of a ByName query.
- CPTest7.py — Process the result set of a dynamic SQL query.
- CPTest8.py — Process employee subclass and company/employee relationship.

All of these applications use classes from the Sample package in the SAMPLES namespace (accessible in Atelier).

Arguments

The sample programs are controlled by various switches that can be entered as arguments to the program on the command line. A default value is supplied if you don't enter an argument.

For example, CPTest2.py accepts the following optional arguments:

- `-user` — the username you want to login under (default is `"_SYSTEM"`).
- `-password` — the password you want to use (default is `"SYS"`).
- `-host` — the host computer to connect to (default is `"localhost"`).

- `-port` — the port to use (default is `"1972"`).

A `-user` argument would be specified as follows:

```
python CPTest2.py -user _MYUSERNAME
```

The CPTest7.py sample accepts a `-query` argument that is passed to an SQL query:

```
python CPTest7.py -query A
```

This query will list all Sample.Person records containing names that start with the letter A.

2

Using the Python Binding

This chapter provides concrete examples of Python code that uses the Caché Python binding. The following subjects are discussed:

- Python Binding Basics — the basics of accessing and manipulating Caché database objects.
- Passing Parameters by Reference — some Python-specific ways to access Python bindings.
- Using Collections — iterating through Caché lists and arrays.
- Using Relationships — manipulating embedded objects.
- Using Queries — running Caché queries and dynamic SQL queries.
- Using %Binary Data — moving data between Caché %Binary and Python list of integers.
- Handling Exceptions — handling Python exceptions and error messages from the Python binding.

Many of the examples presented here are modified versions of the sample programs. The argument processing and error trapping (try/catch) statements have been removed to simplify the code. See Sample Programs for details about loading and running the complete sample programs.

2.1 Python Binding Basics

A Caché Python binding application can be quite simple. Here is a complete sample program:

```
import codecs, sys
import intersys.pythonbind

# Connect to the Cache' database
url = "localhost[1972]:Samples"
user = "_SYSTEM"
password = "SYS"

conn = intersys.pythonbind.connection()
conn.connect_now(url,user,password, None)
database = intersys.pythonbind.database(conn)

# Create and use a Cache' object
person =  database.create_new("Sample.Person", None)
person.set("Name","Doe, Joe A")
print "Name: " + str(person.get("Name"))
```

This code imports the intersys.pythonbind module, and then performs the following actions:

- Connects to the Samples namespace in the Caché database:
  - Defines the information needed to connect to the Caché database.

  - Creates a Connection object (`conn`).
  - Uses the Connection object to create a Database object (`database`).

- Creates and uses a Caché object:
  - Uses the Database object to create an instance of the Caché Sample.Person class.
  - Sets the Name property of the Sample.Person object.
  - Gets and prints the Name property.

The following sections discuss these basic actions in more detail.

2.1.1 Connecting to the Caché Database

The basic procedure for creating a connection to a namespace in a Cache database is as follows:

- Establish the physical connection:

```
conn = intersys.pythonbind.connection()
conn.connect_now(url,user,password, timeout)
```

  The **connect_now()** method creates a physical connection to a namespace in a Caché database. The `url` parameter defines which server and namespace the Connection object will access. The Connection class also provides the **secure_connect_now()** method for establishing secure connections using Kerberos. See Connections for a detailed discussion of both methods.

- Create a logical connection:

```
database = intersys.pythonbind.database(conn)
```

  The Connection object is used to create a Database object, which is a logical connection that lets you use the classes in the namespace to manipulate the database.

  The following code establishes a connection to the Samples namespace:

```
address ="localhost" # server TCP/IP address ("localhost" is 127.0.0.1)
port = "1972" # server TCP/IP port number
namespace = "SAMPLES" # sample namespace installed with Cache'

url = address + "[" + port + "]:" + namespace
user = "_SYSTEM";
password = "SYS";

conn = intersys.pythonbind.connection()
conn.connect_now(url,user,password, None)
database = intersys.pythonbind.database(conn)
```

2.1.2 Using Caché Database Methods

The **intersys.pythonbind.database** class allows you to run Caché class methods and connect to Caché objects on the server. Here are the basic operations that can be performed with the Database class methods:

- *Create Objects*

  The **create_new()** method is used to create a new Caché object. The syntax is:

```
object = database.create_new(class_name, initial_value)
```

  where `class_name` is the name of a Caché class in the namespace accessed by `database`. For example, the following statement creates a new instance of the Sample.Person class:

```
person =  database.create_new("Sample.Person", None)
```

In this example, the initial value of `person` is undefined.

- *Open Objects*

  The **openid()** method is used to open an existing Caché object. The syntax is:

  ```
  object = database.openid(class_name, id, concurrency, timeout)
  ```

  For example, the following statement opens the Sample.Person object that has an id with a value of 1:

  ```
  person =  database.openid("Sample.Person",str(1),-1,-1)
  ```

  Concurrency and timeout are set to their default values.

- *Run Class Methods*

  You can run class methods using the following syntax:

  ```
  result = database.run_class_method(classname,methodname,[LIST])
  ```

  where `LIST` is a list of method arguments. For example, the `database.openid()` example shown previously is equivalent to the following code:

  ```
  person = database.run_class_method("Sample.Person","%OpenId",[str(1)])
  ```

  This method is the analogous to the Caché `##class` syntax. For example, the following code:

  ```
  list = database.run_class_method("%ListOfDataTypes","%New",[])
  list.run_obj_method("Insert",["blue"])
  ```

  is exactly equivalent to the following ObjectScript code:

  ```
  set list=##class(%ListOfDataTypes).%New()
  do list.Insert("blue")
  ```

2.1.3 Using Caché Object Methods

The **intersys.pythonbind.object** class provides access to Caché objects. Here are the basic operations that can be performed with the Object class methods:

- *Get and Set Properties*

  Properties are accessed through the **set()** and **get()** accessor methods. The syntax for these methods is:

  ```
  object.set(propname,value)
  value = object.get(propname)
  ```

  For example:

  ```
  person.set("Name","Doe, Joe A")
  name = person.get("Name")
  ```

  Private and multidimensional properties are not accessible through the Python binding.

- *Run Object Methods*

  You can run object methods by calling **run_obj_method()**:

  ```
  answer = object.run_obj_method(MethodName,[LIST]);
  ```

  For example:

  ```
  answer = person.run_obj_method("Addition",[17,20])
  ```

This method is useful when calling inherited methods (such as **%Save** or **%Id**) that are not directly available.

- *Save Objects*

  To save an object, use **run_obj_method()** to call **%Save**:

  ```
  object.run_obj_method("%Save",[])
  ```

  To get the id of a saved object, use **run_obj_method()** to call **%Id**:

  ```
  id = object.run_obj_method("%Id",[])
  ```

2.2 Passing Parameters by Reference

It is possible to pass arguments by reference with the Python binding. Since Python does not have a native mechanism for passing by reference, the binding passes all arguments as a list that can be modified by the called function. For example, assume the following Caché class:

```
Class Caudron.PassByReference Extends %Persistent [ ProcedureBlock ] {
   ClassMethod PassByReference(ByRef Arg1 As %String) {
      set Arg1="goodbye"
   }
}
```

The following code passes the argument by reference:

```
list = ["hello"]
print "passed to method PassByReference = " + str(list[0])
database.run_class_method("Caudron.PassByReference","PassByReference",list)
print "returned by reference = " + str(list[0])
```

The print statements will produce the following output:

```
passed to method PassByReference = hello
returned by reference = goodbye
```

2.3 Using Collections and Lists

Caché %Collection objects are handled like any other Python binding object. Caché %List variables are mapped to Python array references. The following sections demonstrate how to use both of these items.

2.3.1 %Collection Objects

Collections are manipulated through object methods of the Caché %Collection class. The following example shows how you might manipulate a Caché %ListOfDataTypes collection:

```
# Create a %ListOfDataTypes object and add a list of colors
newcolors = database.create_new("%ListOfDataTypes", None)
color_list = ['red', 'blue', 'green']
print "Adding colors to list object:"
for i in range(len(color_list)):
   newcolors.run_obj_method("Insert",[color_list[i]])
   print " added >"+ str(color_list[i]) +"<"

# Add the list to a Sample.Person object.
person = database.openid("Sample.Person",str(1),-1,0)
person.set("FavoriteColors",newcolors)

# Get the list back from person and print it out.
colors = person.get("FavoriteColors")
```

```
print "\nNumber of colors in 'FavoriteColors' list: %d"\
   % (colors.get("Size"))
index = [0]
while (index[0] != None):
   color = colors.run_obj_method("GetNext",index)
   if (index[0] != None):
      print " Color #%d = >%s<" % (index[0], str(color))

# Remove and replace the second element
index = 2
if (colors.get("Size") > 0):
   colors.run_obj_method("RemoveAt",[index])
colors.run_obj_method("InsertAt",['purple',index])
newcolor = colors.run_obj_method("GetAt",[index])
print "\nChanged color #%d to %s." % (index, str(newcolor))
```

2.3.2 %List Variables

The Python binding maps Caché %List variables to Python array references.

CAUTION: While a Python array has no size limit, Caché %List variables are limited to approximately 32KB. The actual limit depends on the datatype and the exact amount of header data required for each element. Be sure to use appropriate error checking (as demonstrated in the following examples) if your %List data is likely to approach this limit.

The examples in this section assume the following Caché class:

```
Class Sample.List Extends %Persistent
{
Property CurrentList As %List;

Method InitList() As %List {
   q $ListBuild(1,"hello",3.14) }

Method TestList(NewList As %List) As %Integer {
   set $ZTRAP="ErrTestList"
   set ItemCount = $ListLength(NewList)
   if (ItemCount = 0) {set ItemCount = -1}
   q ItemCount
ErrTestList
   set $ZERROR = ""
   set $ZTRAP = ""
   q 0 }
}
```

The **TestList()** method is used to test if a Python array is a valid Caché list. If the list is too large,the method traps an error and returns `0` (Python false). If the list is valid, it returns the number of elements. If a valid list has `0` elements, it returns `-1`.

Example 1: Caché to Python

The following code creates a Sample.List object, gets a predefined Caché list from the **InitList()** method, transforms it into a Python array, and displays information about the array:

```
listobj = database.create_new("Sample.List",None)
mylist = listobj.run_obj_method("InitList",[])

print "Initial List from Cache:"
print "array contents = " + str(mylist)
print "There are %d elements in the list:" % (len(mylist))
for i in range(len(mylist)):
   print "   element %d = [%s]" % (i,mylist[i])
```

This code produces the following output:

```
Initial List from Cache:
array contents = [1, 'hello', 3.1400000000000001]
There are 3 elements in the list:
   element 0 = [1]
   element 1 = [hello]
   element 2 = [3.14]
```

In element 3, the `null` list element in Caché corresponds to value of `None` in the Python array.

Example 2: Python to Caché and Back Again

The following code passes a list in both directions. It creates a small Python array, stores it in the Caché object's CurrentList property, then gets it back from the property and converts it back to a Python array.

```
# Generate a small Python list, pass it to Cache and get it back.
oldarray = [1, None, 2.78,"Just a small list."]
listobj.set("CurrentList",oldarray)
newarray = listobj.get("CurrentList")
print "\nThe 'CurrentList' property now has %d elements:"\
   % (len(newarray))
for i in range(len(newarray)):
   print "  element %d = [%s]" % (i,newarray[i])
```

This code produces the following output:

```
The 'CurrentList' property now has 4 elements:
   element 0 = [1]
   element 1 = [None]
   element 2 = [2.78]
   element 3 = [Just a small list.]
```

Example 3: Testing List Capacity

It is important to make sure that a Cache %List variable can contain the entire Python array. The following code creates a Python array that is too large, and attempts to store it in the CurrentList property.

```
# Create a large array and print array information.
longitem = "1022 character element" + ("1234567890" * 100)
array =["This array is too long."]
cache_list_size = len(array[0])
while (cache_list_size < 32768):
   array.append(longitem)
   cache_list_size = cache_list_size + len(longitem)
print "\n\nNow for a HUGE list:"
print "Total bytes required by Cache' list: more than %d" % (cache_list_size)
print "There are %d elements in the ingoing list.\n" % (len(array))

# Check to see if the array will fit.
bool = listobj.run_obj_method("TestList",[array])
print "TestList reports that "
if (bool):
   print "the array is OK, and has %d elements.\n" % bool
else:
   print "the array will be damaged by the conversion.\n"

# Pass the array to Cache', get it back, and display the results
listobj.set("CurrentList",array)
badarray = listobj.get("CurrentList")
print "There are %d elements in the returned array:\n" % (len(badarray))
for i in range(len(badarray)):
   line = str(badarray[i])
   if (len(line)> 80):
      # long elements are shortened for readability.
      line = line[:][:9] + "..." + line[:][-50:]
   print "   element %d = [%s]\n" % (i, line)
```

The printout shortens undamaged sections of the long elements to make the output more readable. The following output is produced on a unicode system:

```
Now for a HUGE list:
Total bytes in array: at least 33884
There are 34 elements in the ingoing array.
~
TestList reports that the array will be damaged by the conversion.
There are 17 elements in the returned array :
   element 1 = [This list is too long.]
   element 2 = [1022 chara...90123456789012345678901234567890]
   element 3 = [1022 chara...90123456789012345678901234567890]
   element 4 = [1022 chara...90123456789012345678901234567890]
   element 5 = [1022 chara...90123456789012345678901234567890]
   element 6 = [1022 chara...90123456789012345678901234567890]
   element 7 = [1022 chara...90123456789012345678901234567890]
   element 8 = [1022 chara...90123456789012345678901234567890]
   element 9 = [1022 chara...90123456789012345678901234567890]
   element 10 = [1022 chara...90123456789012345678901234567890]
```

```
element 11 = [1022 chara...90123456789012345678901234567890]
element 12 = [1022 chara...90123456789012345678901234567890]
element 13 = [1022 chara...90123456789012345678901234567890]
element 14 = [1022 chara...90123456789012345678901234567890]
element 15 = [1022 chara...90123456789012345678901234567890]
element 16 = [1022 chara...90123456789012345678901234567890]
element 17 = [1022 chara...901234567st is too long.ï´ È‡1022 c]
```

The damaged list contains only 17 of the original 34 elements, and element 17 is corrupted.

2.4 Using Relationships

Relationships are supported through the relationship object and its methods.

The following example generates a report by using the one/many relationship between the `company` object and a set of `employee` objects. The relationship object `emp_relationship` allows the code to access all `employee` objects associated with the `company` object:

```
company = database.openid("Sample.Company",str(1),-1,0)
emp_relationship = company.get("Employees")
index = [None]
print "Employee list:\n"
while 1:
   employee = emp_relationship.run_obj_method("GetNext",index)
   # "GetNext" sets index[0] to the next valid index, or
   # to None if there are no more records.
   if (employee != None):
      i = str(index[0])
      name = str(employee.get("Name"))
      title = str(employee.get("Title"))
      company = employee.get("Company")
      compname = str(company.get("Name"))
      SSN = str(employee.get("SSN"))
      print "   employee #%s:" % i
      print "      name=%s SSN=%s" % (name, SSN)
      print "      title=%s companyname=%s\n"% (title, compname)
   else:
      break
```

The following code creates a new employee record, adds it to the relationship, and automatically saves the employee information when it saves `company`.

```
new_employee = database.create_new("Sample.Employee","")
new_employee.set("Name",name)
new_employee.set("Title",title)
new_employee.set("SSN",SSN)
emp_relationship.run_obj_method("Insert", new_employee)
company.run_obj_method("%Save")
```

2.5 Using Queries

The basic procedure for running a query is as follows:

- *Create the query object*

  ```
  query = intersys.pythonbind.query(database)
  ```

  where `database` is an **intersys.pythonbind.database** object and `query` is an **intersys.pythonbind.query** object.

- *Prepare the query*

  An SQL query uses the **prepare()** method:

  ```
  sql_code = query.prepare(sql_query)
  ```

where `sql_query` is a string containing the query to be executed.

A Caché class query uses the **prepare_class()** method:

```
sql_code = query.prepare_class(class_name, query_name)
```

- *Assign parameter values*

```
query.set_par(idx, val)
```

The **set_par()** method assigns value `val` to parameter `idx`. The value of an existing parameter can be changed by calling **set_par()** again with the new value. The Query class provides several methods that return information about an existing parameter.

- *Execute the query*

```
sql_code = query.execute()
```

The **execute()** method generates a result set using any parameters defined by calls to **set_par()**.

- *Fetch a row of data*

```
data_row = query.fetch([None])
```

Each call to the **fetch()** method retrieves a row of data from the result set and returns it as a list. When there is no more data to be fetched, it returns an empty list. The Query class provides several methods that return information about the columns in a row.

Here is a simple SQL query that retrieves data from Sample.Person:

```
# create a query
sqlstring ="SELECT ID, Name, DOB, SSN \
            FROM SAMPLE.PERSON \
            WHERE Name %STARTSWITH ?"
query = intersys.pythonbind.query(database)
query.prepare(sqlstring)
query.set_par(1,"A")
query.execute();

# Fetch each row in the result set, and print the
# name and value of each column in a row:
while 1:
   cols = query.fetch([None])
   if len(cols) == 0: break
   print "\nRow %s ==================" % str(cols[0])
   print "   %s: %s, %s: %s, %s: %s" \
      % (str(query.col_name(2)), str(cols[1]), \
         str(query.col_name(3)), str(cols[2]), \
         str(query.col_name(4)), str(cols[3]))
```

For more information on the intersys.pythonbind.Query class, see Queries in the Python Client Class Reference chapter. For information about queries in Caché, see Defining and Using Class Queries in *"Using Caché Objects"*.

2.6 Using %Binary Data

The Python binding uses the Python **pack()** and **unpack()** functions to convert data between a Caché %Binary and a Python list of integers. Each byte of the Caché binary data is represented in Python as an integer between 0 and 255.

In this example, the binary initially contains the string "hello". The following code changes the binary to "hellos":

```
   binary = reg.get("MyByte")
   for c in binary:
       print "%c" % c
   print type(ord("s"))
   binary.append(ord("s"))
   reg.set("MyByte",binary)
   binary = reg.get("MyByte")
   for c in binary:
       print "%c" % c
```

2.7 Handling Exceptions

The Python binding uses Python exceptions to return errors from the C binding and elsewhere. Here is an example using Python exceptions:

```
try:
   #some code
except intersys.pythonbind.cache_exception, err:
    print "InterSystems Cache' exception"
    print sys.exc_type
    print sys.exc_value
    print sys.exc_traceback
    print str(err)
```

2.7.1 Error reporting

When processing an argument or a return value, error messages from the C binding are specially formatted by the Python binding layer. This information can be used by InterSystems WRC to help diagnose the problem.

Here is a sample error message:

```
   file=PythonBIND.xs line=71 err=-1 message=cbind_variant_set_buf()
   cpp_type=4 var.cpp_type=-1 var.obj.oref=1784835886
   class_name=%Library.RelationshipObject mtd_name=GetNext argnum=0
```

The error message components are:

message	meaning
`file=PythonBIND.xs`	file where the failure occurred.
`line=71`	line number in the file.
`err=-1`	return code from the C binding.
`message= cbind_variant_set_buf()`	C binding error message.
`cpp_type=4`	cpp type of the method argument or return type.
`var.cpp_type=-1`	variant cpp type.
`var.obj.oref=1784835886`	variant oref.
`class_name= %Library.RelationshipObject`	class name of the object on which the method is invoked.
`mtd_name=GetNext`	method name.
`argnum=0`	argument number. 0 is the first argument and -1 indicates a return value.

3

Python Client Class Reference

This chapter describes how Caché classes and datatypes are mapped to Python code, and provides details on the classes and methods supported by the Caché Python binding. The following subjects are discussed:

- Datatypes — %Binary data.
- Connections — methods used to create a physical connection to a namespace in a Caché database.
- Database — methods used to open or create Caché objects, create queries, and run Caché class methods.
- Objects — methods used to manipulate Caché objects by getting or setting properties, running object methods, and returning information about the objects.
- Queries — methods used to run queries and fetch the results.
- Times and Dates — methods used to access the Caché %Time, %Date, or %Timestamp datatypes.
- Locale and Client Version — methods that provide access to Caché version information and Windows locale settings.

3.1 Datatypes

All Caché datatypes are supported. See the following sections for information on specific datatypes:

- The Caché %Binary datatype corresponds to a Python list of integers. See Using %Binary Data for examples.
- Collections such as %Library.ArrayOfDataTypes and %Library.ListOfDataTypes are handled through object methods of the Caché %Collection classes. See %Collection Objects for examples.
- Caché %Library.List variables are mapped to Python lists. A list can contains strings, ordinary or unicode, integers, None, and doubles. See %List Variables for examples.
- Caché %Time, %Date, and %Timestamp datatypes are supported by corresponding classes in the Python binding. See Times and Dates for a description of these classes.

3.2 Connections

Methods of the intersys.pythonbind.connection package create a physical connection to a namespace in a Caché database. A Connection object is used only to create a Database object, which is the logical connection that allows Python binding

applications to manipulate Caché objects. See Connecting to the Caché Database for information on how to use the Connection methods.

Here is a complete listing of connection methods:

connect_now()

```
conn = intersys.pythonbind.connection()
conn.connect_now(url,user,password, timeout)
```

See Connection Information later in this section for a detailed discussion of the parameters.

secure_connect_now()

```
conn = intersys.pythonbind.connection()
conn.secure_connect_now(url, srv_principal, security_level, timeout)
```

Connection.secure_connect_now() returns the connection proxy that is used to get the proxy for the Caché namespace identified by `url`. This method takes the following parameters:

- `url` — See Connection Information later in this section for a detailed description of the URL format.
- `srv_principal` — A Kerberos "principal" is an identity that is represented in the Kerberos database, has a permanent secret key that is shared only with the Kerberos KDCs (key distribution centers), can be assigned credentials, and can participate in the Kerberos authentication protocol.
  - A "user principal" is associated with a person, and is used to authenticate to services which can then authorize the use of resources (for example, computer accounts or Caché services).
  - A "service principal" is associated with a service, and is used to authenticate user principals and can optionally authenticate itself to user principals.
  - A "service principal name" (such as `srv_principal_name`) is the string representation of the name of a service principal, conventionally of the form:

    <service>/<instance>@<REALM>

    For example:

    cache/turbo.iscinternal.com@ISCINTERNAL.COM

  On Windows, The KDCs are embedded in the domain controllers, and service principal names are associated with domain accounts.

  See your system's Kerberos documentation for a detailed discussion of principals.
- `security_level` — Sets the "Connection security level", which is an integer that indicates the client/server network security services that are requested or required. Security level can take the following values:
  - `0` — None.
  - `1` — Kerberos client/server mutual authentication, no protection for data.
  - `2` — As `1`, plus data source and content integrity protection.
  - `3` — As `2`, plus data encryption.
- `timeout` — Number of seconds to wait before timing out.

3.2.1 Connection Information

The following information is needed to establish a connection to the Caché database:

- *URL* — The URL specifies the server and namespace to be accessed as a string with the following format:

  ```
  <address>[<port>]:<namespace>
  ```

  For example, the sample programs use the following connection string:

  ```
  "localhost[1972]:Samples"
  ```

  The components of this string are:

  – `<address>` — The server TCP/IP address or Fully Qualified Domain Name (FQDN). The sample programs use `"localhost"` (`127.0.0.1`), assuming that both the server and the Python application are on the same machine.

  – `<port>` — The server TCP/IP port number for this connection. Together, the IP address and the port specify a unique Caché server.

  – `<namespace>` — The Caché namespace containing the objects to be used. This namespace must have the Caché system classes compiled, and must contain the objects you want to manipulate. The sample programs use objects from the SAMPLE namespace.

- *username* — The username under which the connection is being made. The sample programs use `"_SYSTEM"`, the default SQL System Manager username.
- *password* — The password associated with the specified username. Sample programs use the default, `"SYS"`.

3.3 Database

Database objects provide a logical connection to a Caché namespace. Methods of the intersys.pythonbind.Database package are used to open or create Caché objects, create queries, and run Caché class methods. Database objects are created by calling `database = intersys.pythonbind.database(conn)`, where *conn* is a intersys.pythonbind.connection object. See Connecting to the Caché Database for more information on creating a Database object.

Here is a complete listing of Database methods:

create_new()

```
obj = database.create_new(type, init_val)
```

Creates a new Caché object instance from the class named by `type`. Normally, `init_val` is None. See Objects for details on the objects created with this method.

open()

```
obj = database.open(class_name, oid, concurrency, timeout, res)
```

Opens a Caché object instance using the class named by `class_name` and the oid of the object. The `concurrency` argument has a default value of `-1`. `timeout` is the ODBC query timeout.

openid()

```
obj = database.openid(class_name, id, concurrency, timeout)
```

Opens a Caché object instance using the class named by `class_name` and the id of the object. The `concurrency` argument has a default value of `-1`. `timeout` is the ODBC query timeout.

run_class_method()

```
value = database.run_class_method(class_name, method_name, [LIST])
```

Runs the class method `method_name`, which is a member of the `class_name` class in the namespace that `database` is connected to. Arguments are passed in `LIST`. Some of these arguments may be passed by reference depending on the class definition in Caché. Return values correspond to the return values from the Caché method.

3.4 Objects

Methods of the intersys.pythonbind.object package provide access to a Caché object. An Object object is created by the intersys.pythonbind.database **create_new()** method (see Database for a detailed description). See Using Caché Object Methods for information on how to use the Object methods.

Here is a complete listing of Object methods:

get()

```
value = object.get(prop_name)
```

Returns the value of property `prop_name` in Caché object `object`.

run_obj_method()

```
value = object.run_obj_method(method_name, [LIST])
```

Runs method `method_name` on Caché object `object`. Arguments are passed in `LIST`. Some of these arguments may be passed by reference depending on the class definition in Caché. Return values correspond to the return values from the Caché method.

set()

```
object.set(prop_name, val)
```

Sets property `prop_name` in Caché object `object` to `val`.

3.5 Queries

Methods of the intersys.pythonbind.query package provide the ability to prepare a query, set parameters, execute the query, and and fetch the results. See Using Queries for information on how to use the Query methods.

A Query object is created as follows:

```
query = intersys.pythonbind.query(database)
```

Here is a complete listing of Query methods:

3.5.1 prepare query

prepare()

```
query.prepare(string)
```

Prepares a query using the SQL string in `string`.

prepare_class()

```
query.prepare_class(class_name, query_name)
```

Prepares a query in a class definition

3.5.2 set parameters

set_par()

```
query.set_par(idx, val)
```

Assigns value `val` to parameter `idx`. The method can be called several times for the same parameter. The previous parameter value will be lost, and the new value can be of a different type. The **set_par()** method does not support by-reference parameters.

is_par_nullable()

```
nullable = query.is_par_nullable(idx)
```

Returns `1` if parameter `idx` is nullable, else `0`.

is_par_unbound()

```
unbound = query.is_par_unbound(idx)
```

Returns `1` if parameter `idx` is unbound, else `0`.

num_pars()

```
num = query.num_pars()
```

Returns number of parameters in query.

par_col_size()

```
size = query.par_col_size(idx)
```

Returns size of parameter column.

par_num_dec_digits()

```
num = query.par_num_dec_digits(idx)
```

Returns number of decimal digits in parameter.

par_sql_type()

```
type = query.par_sql_type(idx)
```

Returns sql type of parameter.

3.5.3 execute query

execute()

```
query.execute()
```

Generates a result set using any parameters defined by calls to **set_par()**.

3.5.4 fetch results

fetch()

```
data_row = query.fetch([None])
```

Retrieves a row of data from the result set and returns it as a list. When there is no more data to be fetched, it returns an empty list.

col_name()

```
name = query.col_name(idx)
```

Returns name of column.

col_name_length()

```
length = query.col_name_length(idx)
```

Returns length of column name.

col_sql_type()

```
sql_type = query.col_sql_type(idx)
```

Returns sql type of column.

num_cols()

```
num_cols = query.num_cols()
```

Returns number of columns in query.

3.6 Times and Dates

The PTIME_STRUCTPtr, PDATE_STRUCTPtr, and PTIMESTAMP_STRUCTPtr packages are used to manipulate Caché %TIME, %DATE, or %TIMESTAMP datatypes.

3.6.1 %TIME

Methods of the PTIME_STRUCTPtr package are used to manipulate the Caché %DATE data structure. Times are in `hh:mm:ss` format. For example, 5 minutes and 30 seconds after midnight would be formatted as `00:05:30`. Here is a complete listing of Time methods:

new()

```
time = PTIME_STRUCTPtr.new()
```

Create a new Time object.

get_hour()

```
hour = time.get_hour()
```

Return hour

get_minute()

```
minute = time.get_minute()
```

Return minute

get_second()

```
second = time.get_second()
```

Return second

set_hour()

```
time.set_hour(hour)
```

Set hour (an integer between 0 and 23, where 0 is midnight).

set_minute()

```
time.set_minute(minute)
```

Set minute (an integer between 0 and 59).

set_second()

```
time.set_second(second)
```

Set second (an integer between 0 and 59).

toString()

```
stringrep = time.toString()
```

Convert the time to a string: `hh:mm:ss`.

3.6.2 %DATE

Methods of the PDATE_STRUCTPtr package are used to manipulate the Caché %DATE data structure. Dates are in `yyyy-mm-dd` format. For example, December 24, 2003 would be formatted as `2003-12-24`. Here is a complete listing of Date methods:

new()

```
date = PDATE_STRUCTPtr.new()
```

Create a new Date object.

get_year()

```
year = date.get_year()
```

Return year

get_month()

```
month = date.get_month()
```

Return month

get_day()

```
day = date.get_day()
```

Return day

set_year()

```
date.set_year(year)
```

Set year (a four-digit integer).

set_month()

```
date.set_month(month)
```

Set month (an integer between 1 and 12).

set_day()

```
date.set_day(day)
```

Set day (an integer between 1 and the highest valid day of the month).

toString()

```
stringrep = date.toString()
```

Convert the date to a string: `yyyy-mm-dd`.

3.6.3 %TIMESTAMP

Methods of the PTIMESTAMP_STRUCTPtr package are used to manipulate the Caché %TIMESTAMP data structure. Timestamps are in `yyyy-mm-dd`<space>`hh:mm:ss.fffffffff`. format. For example, December 24, 2003, five minutes and 12.5 seconds after midnight, would be formatted as:

```
2003-12-24 00:05:12:500000000
```

Here is a complete listing of TimeStamp methods:

new()

```
timestamp = PTIMESTAMP_STRUCTPtr.new()
```

Create a new Timestamp object.

get_year()

```
year = timestamp.get_year()
```

Return year in `yyyy` format.

get_month()

```
month = timestamp.get_month()
```

Return month in `mm` format.

get_day()

```
day = timestamp.get_day()
```

Return day in dd format.

get_hour()

```
hour = timestamp.get_hour()
```

Return hour in hh format.

get_minute()

```
minute = timestamp.get_minute()
```

Return minute in mm format.

get_second()

```
second = timestamp.get_second()
```

Return second in ss format.

get_fraction()

```
fraction = timestamp.get_fraction()
```

Return fraction of a second in fffffffff format.

set_year()

```
timestamp.set_year(year)
```

Set year (a four-digit integer).

set_month()

```
timestamp.set_month(month)
```

Set month (an integer between 1 and 12).

set_day()

```
timestamp.set_day(day)
```

Set day (an integer between 1 and the highest valid day of the month).

set_hour()

```
timestamp.set_hour(hour)
```

Set hour (an integer between 0 and 23, where 0 is midnight).

set_minute()

```
timestamp.set_minute(minute)
```

Set minute (an integer between 0 and 59).

set_second()

```
timestamp.set_second(second)
```

Set second (an integer between 0 and 59).

set_fraction()

```
timestamp.set_fraction(fraction)
```

Set fraction of a second (an integer of up to nine digits).

toString()

```
stringrep = timestamp.toString()
```

Convert the timestamp to a string `yyyy-mm-dd hh:mm:ss.fffffffff`.

3.7 Locale and Client Version

Methods of the intersys.pythonbind. default package provide access to Caché version information and Windows locale settings. Here is a complete listing of these methods:

get_client_version()

```
clientver = intersys.pythonbind.get_client_version();
```

Identifies the version of Caché running on the Python client machine.

setlocale()

```
newlocale = intersys.pythonbind.setlocale(category, locale)
```

Sets the default locale and returns a locale string for the new locale. For example:

```
newlocale = intersys.pythonbind.setlocale(0, "Russian") # 0 stands for LC_ALL
```

would set all locale categories to Russian and return the following string:

```
Russian_Russia.1251
```

If the `locale` argument is an empty string, the current default locale string will be returned. For example, given the following code:

```
intersys.pythonbind.setlocale(0, "English")
mylocale = intersys.pythonbind.setlocale(0, ""),"\n";
```

the value of `mylocale` would be:

```
English_United States.1252
```

For detailed information, including a list of valid `category` values, see the MSDN library (http://msdn.microsoft.com/library) entry for the **setlocale()** function in the Visual C++ runtime library.

set_thread_locale()

```
intersys.pythonbind.set_thread_locale(lcid)
```

Sets the locale id (LCID) for the calling thread. Applications that need to work with locales at runtime should call this method to ensure proper conversions.

For a listing of valid LCID values, see the "Locale ID (LCID) Chart" in the MSDN library (http://msdn.microsoft.com/library). The chart can be located by a search on "LCID Chart". It is currently located at:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/script56/html/vsmsclcid.asp

For detailed information on locale settings, see the MSDN library entry for the **SetThreadLocale()** function, listed under "National Language Support".