Using the Native SDK for Python

Version 2024.2
2024-09-05

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

Using the Native SDK for Python
PDF generated on 2024-09-05
InterSystems IRIS® Version 2024.2

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

List of Figures

1

Introduction to the Native SDK for Python

See the Table of Contents for a detailed listing of the subjects covered in this document. See the Python Native SDK Quick Reference for a brief description of Native SDK classes and methods.

The InterSystems *Native SDK for Python* is a lightweight interface to powerful InterSystems IRIS® resources that were once available only through ObjectScript:

- Call ObjectScript Methods and Functions — call any embedded language classmethod from your Python application as easily as you can call native Python methods.
- Control Database Objects from Python — use Python proxy objects to control embedded language class instances. Call instance methods and get or set property values as if the instances were native Python objects.
- Work with Global Arrays — directly access globals, the tree-based sparse arrays used to implement the InterSystems multidimensional storage model.
- Use InterSystems Transactions and Locking — use Native SDK implementations of embedded language transactions and locking methods to work with InterSystems databases.
- Python DB-API Support — use the InterSystems implementation of the PEP 249 version 2.0 Python Database API for relational database access.

Native SDKs for other languages

Versions of the Native SDK are also available for Java, .NET, and Node.js:

- *Using the Native SDK for Java*
- *Using the Native SDK for .NET*
- *Using the Native SDK for Node.js*

More information about globals

The following book is highly recommended for developers who want to master the full power of global arrays:

- *Using Globals* — describes how to use globals in ObjectScript, and provides more information about how multidimensional storage is implemented on the server.

2

Calling Database Methods and Functions from Python

This section describes methods of class iris.IRIS that allow you to call ObjectScript class methods and functions directly from your Python application. See the following sections for details and examples:

- Calling Class Methods from Python — demonstrates how to call ObjectScript class methods.
- Calling Functions and Procedures from Python — demonstrates how to call functions and procedures.
- Passing Arguments by Reference — demonstrates how to pass arguments with the IRISReference class. .

2.1 Calling Class Methods from Python

The **classMethodValue()** and **classMethodVoid()** methods will work for most purposes, but if a specific return type is needed, the following IRIS. typecast methods are also available: **classMethodBoolean()**, **classMethodBytes()**, **classMethodDecimal()**, **classMethodFloat()**, **classMethodIRISList()**, **classMethodInteger()**, **classMethodObject()**, and **classMethodString()**.

These methods all take string arguments for *class_name* and *method_name*, plus `0` or more method arguments.

The code in the following example calls class methods of several datatypes from an ObjectScript test class named User.NativeTest. (see listing "ObjectScript Class User.NativeTest" at the end of this section).

Python calls to ObjectScript class methods

The code in this example calls class methods of each supported datatype from ObjectScript test class User.NativeTest (listed immediately after this example). Assume that variable *irispy* is a previously defined instance of class iris.IRIS and is currently connected to the server (see "Creating a Connection in Python").

```
className = 'User.NativeTest'

comment = ".cmBoolean() tests whether arguments 2 and 3 are equal: "
boolVal = irispy.classMethodBoolean(className,'cmBoolean',2,3)
print(className + comment + str(boolVal))

comment = ".cmBytes returns integer arguments 72,105,33 as a byte array (string value 'Hi!'):
"
byteVal = irispy.classMethodBytes(className,'cmBytes',72,105,33) #ASCI 'Hi!'
print(className + comment + str(byteVal))

comment = ".cmString() concatenates 'Hello' with argument string 'World': "
stringVal = irispy.classMethodString(className,'cmString','World')
print(className + comment + stringVal)
```

```
  comment = ".cmLong() returns the sum of arguments 7+8: "
  longVal = irispy.classMethodInteger(className,'cmLong',7,8)
  print(className + comment + str(longVal))

  comment = ".cmDouble() multiplies argument 4.5 by 1.5: "
  doubleVal = irispy.classMethodFloat(className,'cmDouble',4.5)
  print(className + comment + str(doubleVal))

  comment = ".cmList() returns a $LIST containing arguments 'The answer is ' and 42: "
  listVal = irispy.classMethodIRISList(className,"cmList","The answer is ",42);
  print(className + comment+listVal.get(1)+str(listVal.get(2)))

  comment = ".cmVoid assigns argument value 75 to global node ^cmGlobal: "
  try:
    irispy.kill('cmGlobal') # delete ^cmGlobal if it exists
    irispy.classMethodVoid(className,'cmVoid',75)
    nodeVal = irispy.get('cmGlobal');  #get current value of ^cmGlobal
  except:
    nodeVal = 'FAIL'
  print(className + comment + str(nodeVal))
```

This example omits **classMethodValue()** (which returns an untyped value), **classMethodDecimal()** (which differs from **classMethodFloat()** primarily in support for higher precision), and **classMethodObject()** (which is demonstrated in "Controlling Database Objects from Python").

ObjectScript Class User.NativeTest

To run the previous example, this ObjectScript class must be compiled and available on the server:

```
Class User.NativeTest Extends %Persistent
  {

  ClassMethod cmBoolean(cm1 As %Integer, cm2 As %Integer) As %Boolean
  {
     Quit (cm1=cm2)
  }

  ClassMethod cmBytes(cm1 As %Integer, cm2 As %Integer, cm3 As %Integer) As %Binary
  {
     Quit $CHAR(cm1,cm2,cm3)
  }

  ClassMethod cmString(cm1 As %String) As %String
  {
     Quit "Hello "_cm1
  }

  ClassMethod cmLong(cm1 As %Integer, cm2 As %Integer) As %Integer
  {
     Quit cm1+cm2
  }

  ClassMethod cmDouble(cm1 As %Double) As %Double
  {
     Quit cm1 * 1.5
  }

  ClassMethod cmVoid(cm1 As %Integer)
  {
     Set ^cmGlobal=cm1
     Quit
  }

  ClassMethod cmList(cm1 As %String, cm2 As %Integer)
  {
     Set list = $LISTBUILD(cm1,cm2)
     Quit list
  }
}
```

You can test these methods by calling them from the Terminal. For example:

```
USER>write ##class(User.NativeTest).cmString("World")
Hello World
```

2.2 Calling Functions and Procedures from Python

Note: **Procedural Code Support**

On older InterSystems database platforms, code consisted of modules containing functions and procedures, rather than object-oriented classes and methods (see "Callable User-defined Code Modules" in *Using ObjectScript*). Functions are frequently necessary for older code bases, but new code should use object oriented method calls if possible.

The **function()** and **procedure()** methods will work for most purposes, but if a specific return type is needed, the following IRIS. typecast methods are also available: **functionBoolean()**, **functionBytes()**, **functionDecimal()**, **functionFloat()**, **functionIRISList()**, **functionObject()**, **functionInteger()**, and **functionString()**.

These methods take string arguments for *functionLabel* and *routineName*, plus `0` or more function arguments, which may be bool, bytes, bytearray, Decimal, float, int, str or **IRISList**.

The code in the following example calls functions from an ObjectScript test routine named NativeRoutine (listed immediately after the example).

Note: **Built-in ObjectScript $ system functions are not supported**

These methods are designed to call functions in user-defined routines. ObjectScript system functions (which start with a $ character. See "ObjectScript Functions" in the *ObjectScript Reference*) cannot be called directly from your Python code. However, you can call a system function indirectly by writing an ObjectScript wrapper function that calls the system function and returns the result. For example, the **fnList()** function (at the end of this section in ObjectScript Routine NativeRoutine.mac) calls $LISTBUILD.

Python calls to ObjectScript routines

The code in this example calls functions of each supported datatype from the ObjectScript routine NativeRoutine (File NativeRoutine.mac, listed immediately after this example). Assume that *irispy* is an existing instance of class iris.IRIS, and is currently connected to the server (see "Creating a Connection in Python").

```
routineName = 'NativeRoutine'

comment = ".fnBoolean() tests whether arguments 2 and 3 are equal: "
boolVal = irispy.functionBoolean('fnBoolean',routineName,2,3)
print(routineName + comment + str(boolVal))

comment = ".fnBytes returns integer arguments 72,105,33 as a byte array (string value 'Hi!'):
"
byteVal = irispy.functionBytes('fnBytes',routineName,72,105,33)  #ASCI 'Hi!'
print(routineName + comment + str(byteVal))

comment = ".fnString() concatenates 'Hello' with argument string 'World': "
stringVal = irispy.functionString("fnString",routineName,"World")
print(routineName + comment + stringVal)

comment = ".fnLong() returns the sum of arguments 7+8: "
longVal = irispy.functionInteger('fnLong',routineName,7,8)
print(routineName + comment + str(longVal))

comment = ".fnDouble() multiplies argument 4.5 by 1.5: "
doubleVal = irispy.functionFloat('fnDouble',routineName,4.5)
print(routineName + comment + str(doubleVal))

comment = ".fnList() returns a $LIST containing arguments 'The answer is ' and 42: "
listVal = irispy.functionIRISList("fnList",routineName,"The answer is ",42);
print(routineName + comment + listVal.get(1)+str(listVal.get(2)));

comment = ".fnProcedure() assigns argument value 66 to global node ^fnGlobal: "
try:
  irispy.kill('fnGlobal') # delete ^fnGlobal if it exists
  irispy.procedure('fnProcedure',routineName,66)
  nodeVal = irispy.get('fnGlobal') # get current value of node ^fnGlobal
```

```
except:
  nodeVal = 'FAIL'
print(routineName + comment + str(nodeVal))
```

This example omits **function()** (which returns an untyped value), **functionDecimal()** (which differs from **functionFloat()** primarily in support for higher precision), and **functionObject()** (functionally identical to the **classMethodObject()** method demonstrated in "Controlling Database Objects from Python").

ObjectScript Routine NativeRoutine.mac

To run the previous example, this ObjectScript routine must be compiled and available on the server:

```
fnBoolean(fn1,fn2) public {
   quit (fn1=fn2)
}
fnBytes(fn1,fn2,fn3) public {
    quit $CHAR(fn1,fn2,fn3)
}
fnString(fn1) public {
    quit "Hello "_fn1
}
fnLong(fn1,fn2) public {
    quit fn1+fn2
}
fnDouble(fn1) public {
    quit fn1 * 1.5
}
fnProcedure(fn1) public {
    set ^fnGlobal=fn1
    quit
}
fnList(fn1,fn2) public {
    set list = $LISTBUILD(fn1,fn2)
    quit list
}
```

You can test these functions by calling them from the Terminal. For example:

```
USER>write $$fnString^NativeRoutine("World")
Hello World
```

2.3 Passing Arguments by Reference

Most of the classes in the InterSystems Class Library use a calling convention where methods only return a %Status value. The actual results are returned in arguments passed by reference. The Native SDK supports pass by reference for both methods and functions by assigning the argument value to an instance of class IRISReference and passing that instance as the argument:

```
ref_object = iris.IRISReference(None); // set inital value to None
irispy.classMethodObject("%SomeClass","SomeMethod",ref_object);
returned_value = ref_object.getValue;  // get the method result
```

The following example calls a standard Class Library method:

Using pass-by-reference arguments

This example calls %SYS.DatabaseQuery.**GetDatabaseFreeSpace()** to get the amount of free space (in MB) available in the iristemp database.

Python

```
  dir = "C:/InterSystems/IRIS/mgr/iristemp" # directory to be tested
  value = "error"
  status = 0

  freeMB = iris.IRISReference(None) # set inital value to 0
  print("Variable freeMB is type"+str(type(freeMB)) + ", value=" + str(freeMB.getValue()))
  try:
    print("Calling %SYS.DatabaseQuery.GetDatabaseFreeSpace()... ")
    status = irispy.classMethodObject("%SYS.DatabaseQuery","GetDatabaseFreeSpace",dir,freeMB)
    value = freeMB.getValue()
  except:
    print("Call to class method GetDatabaseFreeSpace() returned error:")
  print("(status=" + str(status) + ") Free space in " + dir + " = " + str(value) + "MB\n")
```

prints:

```
Variable freeMB is type<class 'iris.IRISReference'>, value=None
Calling %SYS.DatabaseQuery.GetDatabaseFreeSpace()...
(status=1) Free space in C:/InterSystems/IRIS/mgr/iristemp = 10MB
```

3

Controlling Database Objects from Python

The Native SDK works together with InterSystems External Servers, allowing your external Python application to control instances of database classes written in either ObjectScript or Embedded Python. Native SDK *inverse proxy objects* can use External Server connections to create a target database object, call the target's instance methods, and get or set property values as easily as if the target were a native Python object.

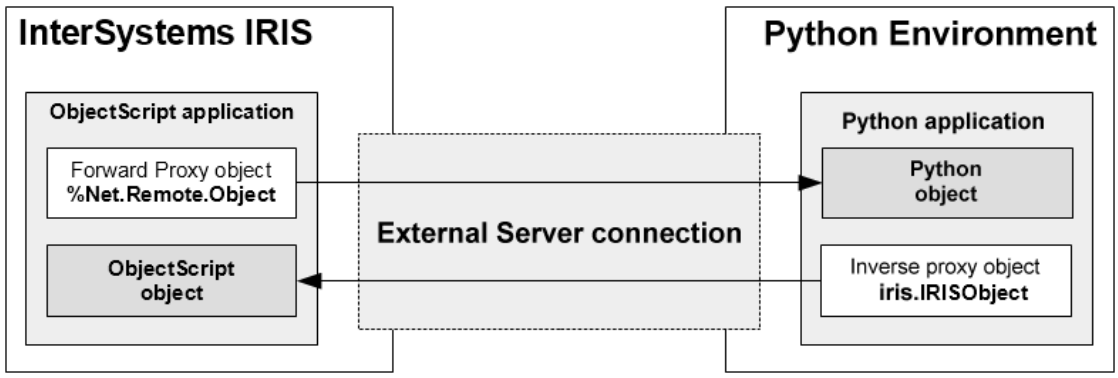
This section covers the following topics:

- Introducing the Python External Server — provides a brief overview of External Servers.
- Creating Python Inverse Proxy Objects — describes methods used to create inverse proxy objects.
- Controlling Database Objects with IRISObject — demonstrates how inverse proxy objects are used.

3.1 Introducing the Python External Server

The Python External Server allows InterSystems IRIS embedded language objects and Python Native SDK objects to interact freely, using the same connection and working together in the same context (database, session, and transaction). External Server architecture is described in detail in *Using InterSystems External Servers*, but for the purposes of this discussion you can think of it as a simple black box connecting proxy objects on one side to target objects on the other:

Figure 3–1: Python External Server System



As the diagram shows, a *forward proxy* is a database object that controls an external Python target object. The corresponding Native SDK object is an *inverse proxy* that controls a database target object from an external Python application, inverting the normal flow of control.

3.2 Creating Python Inverse Proxy Objects

You can create an inverse proxy object by obtaining the OREF of a database class instance (for example, by calling the **%New()** method of an ObjectScript class). The IRIS.**classMethodObject()** and IRIS.**functionObject()** methods will return an IRISObject instance if the call obtains a valid OREF. The following example uses IRIS.**classMethodObject()** to create an inverse proxy object:

Creating an instance of IRISObject

```
proxy = irispy.classMethodObject("User.TestInverse","%New");
```

- **classMethodObject()** calls the **%New()** method of an ObjectScript class named User.TestInverse
- The call to **%New()** creates a database instance of User.TestInverse.
- **classMethodObject()** returns inverse proxy object *proxy*, which is an instance of IRISObject mapped to the database instance.

This example assumes that *irispy* is a connected instance of IRIS (see "Creating a Connection in Python"). See "Calling Database Methods and Functions from Python" for more information on how to call class methods.

The following section demonstrates how *proxy* can be used to access methods and properties of the ObjectScript User.TestInverse instance.

3.3 Controlling Database Objects with IRISObject

The iris.IRISObject class provides several methods to control the database target object: **invoke()** and **invokeVoid()** call an instance method with or without a return value, and accessors **get()** and **set()** get and set a property value. .

This example uses an ObjectScript class named User.TestInverse, which includes declarations for methods **initialize()** and **add()**, and property *name*:

ObjectScript sample class TestInverse

```
Class User.TestInverse Extends %Persistent {
  Method initialize(initialVal As %String = "no name") {
    set ..name = initialVal
    return 0
  }
  Method add(val1 As %Integer, val2 As %Integer) As %Integer {
   return val1 + val2
  }
  Property name As %String;
}
```

The first line of the following example creates a new instance of User.TestInverse and returns inverse proxy object *proxy*, which is mapped to the database instance. The rest of the code uses *proxy* to access the target instance. Assume that a connected instance of IRIS named *irispy* already exists (see "Creating a Connection in Python").

Using inverse proxy object methods in Python

```
# Create an instance of User.TestInverse and return an inverse proxy object for it
proxy = irispy.classMethodObject("User.TestInverse","%New");

# instance method proxy.initialize() is called with one argument, returning nothing.
proxy.invokeVoid("initialize", "George");
print("Current name is "+ proxy.get("name"));   # display the initialized property value

# instance method proxy.add() is called with two arguments, returning an int value.
print("Sum of 2 plus 3 is " + str(proxy.invoke("add",2,3)));

# The value of property proxy.name is displayed, changed, and displayed again.
proxy.set("name", "Einstein, Albert");  # sets the property to "Einstein, Albert"
print("New name is "+ proxy.get("name"));   # display the new property value
```

This example uses the following methods to access methods and properties of the User.TestInverse instance:

- IRISObject.**invokeVoid()** invokes the **initialize()** instance method, which initializes a property but does not return a value.
- IRISObject.**invoke()** invokes instance method **add()**, which accepts two integer arguments and returns the sum as an integer.
- IRISObject.**set()** sets the *name* property to a new value.
- IRISObject.**get()** returns the value of property *name*.

This example used the variant **get()**, and **invoke()** methods, but the IRISObject class also provides datatype-specific Typecast Methods for supported datatypes:

IRISObject get() typecast methods

In addition to the variant **get()** method, IRISObject provides the following IRISObject. typecast methods: **getBytes()**, **getDecimal() getFloat() getInteger()**, **getString()**, **getIRISList()**, and **getObject()**.

IRISObject invoke() typecast methods

In addition to **invoke()** and **invokeVoid()**, IRISObject provides the following IRISObject. typecast methods: **invokeBytes()**, **invokeDecimal()**, **invokeInteger()**, **invokeString()**, **invokeIRISList()**, and **invokeObject()**.

All of the **invoke()** methods take a str argument for *methodName* plus `0` or more method arguments. The arguments may be either Python objects, or values of supported datatypes. Database proxies will be generated for arguments that are not supported types.

invoke() can only call instance methods of an IRISObject instance. See "Calling Database Methods and Functions from Python" for information on how to call class methods.

4

Accessing Global Arrays with Python

The *Native SDK for Python* provides mechanisms for working with global arrays. The following topics are covered here:

- Introduction to Global Arrays — introduces global array concepts and provides a simple demonstration of how the Native SDK is used.
- Fundamental Node Operations — demonstrates how use **set()**, **get()**, and **kill()** to create, access, and delete nodes in a global array.
- Iteration with nextSubscript() and isDefined() — demonstrates methods that emulate ObjectScript-style iteration.

Note: **Creating a Database Connection in Python**

The examples in this chapter assume that iris.IRIS object *irispy* already exists and is connected to the server. The following code is used to create and connect *irispy*:

```
import iris
conn = iris.connect('127.0.0.1', 51773, 'USER', '_SYSTEM', 'SYS')
irispy = iris.createIRIS(conn)
```

For more information, see the Quick Reference entries for iris package functions **connect()** and **createIRIS()**.

4.1 Introduction to Global Arrays

A global array, like all sparse arrays, is a tree structure rather than a sequential list. The basic concept behind global arrays can be illustrated by analogy to a file structure. Each *directory* in the tree is uniquely identified by a *path* composed of a *root directory* identifier followed by a series of *subdirectory* identifiers, and any directory may or may not contain *data*.

Global arrays work the same way: each *node* in the tree is uniquely identified by a *node address* composed of a *global name* identifier and a series of *subscript* identifiers, and a node may or may not contain a *value*. For example, here is a global array consisting of six nodes, two of which contain values:

```
root -->|--> foo --> SubFoo='A'
        |--> bar --> lowbar --> UnderBar=123
```

Values could be stored in the other possible node addresses (for example, root or root->bar), but no resources are wasted if those node addresses are *valueless*. Unlike a directory structure, all nodes in a global array *must* have either a value or a subnode with a value. In InterSystems ObjectScript globals notation, the two nodes with values would be:

```
root('foo','SubFoo')
root('bar','lowbar','UnderBar')
```

In this notation, the global name (`root`) is followed by a comma-delimited *subscript list* in parentheses. Together, they specify the entire *node address* of the node.

This global array could be created by two calls to the Native SDK **set()** method. The first argument is the value to be assigned, and the rest of the arguments specify the node address:

```
irispy.set('A', 'root', 'foo', 'SubFoo')
irispy.set(123, 'root', 'bar', 'lowbar', 'UnderBar')
```

Global array `root` is does not exist until the first call assigns value `'A'` to node *root('foo','SubFoo')*. Nodes can be created in any order, and with any set of subscripts. The same global array would be created if we reversed the order of these two calls. The valueless nodes are created automatically, and will be deleted automatically when no longer needed.

The Native SDK code to create this array is demonstrated in the following example. An IRISConnection object establishes a connection to the server. The connection will be used by an instance of iris.IRIS named *irispy*. Native SDK methods are then used to create a global array, read the resulting persistent values from the database, and delete the global array.

The NativeDemo Program

```
# Import the Native SDK module
import iris

# Open a connection to the server
args = {'hostname':'127.0.0.1', 'port':52773,
    'namespace':'USER', 'username':'_SYSTEM', 'password':'SYS'
}
conn = iris.connect(**args)
# Create an iris object
irispy = iris.createIRIS(conn)

# Create a global array in the USER namespace on the server
irispy.set('A', 'root', 'foo', 'SubFoo')
irispy.set(123, 'root', 'bar', 'lowbar', 'UnderBar')

# Read the values from the database and print them
subfoo_value = irispy.get('root', 'foo', 'SubFoo')
underbar_value = irispy.get('root', 'bar', 'lowbar', 'UnderBar')
print('Created two values: ')
print('   root("foo","SubFoo")=', subfoo_value)
print('   root("bar","lowbar","UnderBar")=', underbar_value)

# Delete the global array and terminate
irispy.kill('root') # delete global array root
conn.close()
```

NativeDemo prints the following lines:

```
Created two values:
   root('foo','SubFoo')="A"
   root('bar','lowbar','UnderBar')=123
```

In this example, Native SDK iris package methods are used to connect to the database and to create *irispy*, which is the instance of iris.IRIS the contains the connection object. Native SDK methods perform the following actions:

- iris.**connect()** creates a connection object named *conn*, connected to the database associated with the USER namespace.
- iris.**createIRIS()** creates a new instance of iris.IRIS named *irispy*, which will access the database through server connection *conn*.
- iris.IRIS.**set()** creates new persistent nodes in database namespace USER.
- iris.IRIS.**get()** returns the values of the specified nodes.
- iris.IRIS.**kill()** deletes the specified root node and all of its subnodes from the database.
- iris.IRISConnection.**close()** closes the connection.

See "iris Package Methods" for details about connecting and creating an instance of iris.IRIS. See "Fundamental Node Operations" for more information about **set()**, **get()**, and **kill()**.

This simple example doesn't cover more advanced topics such as iteration. See "Class iris.IRISGlobalNode" for information on how to create and iterate over complex global arrays.

4.1.1 Glossary of Global Array Terms

See the previous section for an overview of the concepts listed here. Examples in this glossary will refer to the global array structure listed below. The *Legs* global array has ten nodes and three node levels. Seven of the ten nodes contain values:

```
Legs                      # root node, valueless, 3 child nodes
  fish = 0                # level 1 node, value=0
  mammal                  # level 1 node, valueless
    human = 2             # level 2 node, value=2
    dog = 4               # level 2 node, value=4
  bug                     # level 1 node, valueless, 3 child nodes
    insect = 6            # level 2 node, value=6
    spider = 8            # level 2 node, value=8
    millipede = Diplopoda # level 2 node, value="Diplopoda", 1 child node
      centipede = 100     # level 3 node, value=100
```

Child node

The nodes immediately under a given parent node. The address of a child node is specified by adding exactly one subscript to the end of the parent subscript list. For example, parent node *Legs('mammal')* has child nodes *Legs('mammal','human')* and *Legs('mammal','dog')*.

Global name

The identifier for the root node is also the name of the entire global array. For example, root node identifier `Legs` is the global name of global array *Legs*. Unlike subscripts, global names can only consist of letters, numbers, and periods (see Global Naming Rules).

Node

An element of a global array, uniquely identified by a namespace consisting of a global name and an arbitrary number of subscript identifiers. A node must either contain a value, have child nodes, or both.

Node level

The number of subscripts in the node address. A 'level 2 node' is just another way of saying 'a node with two subscripts'. For example, *Legs('mammal','dog')* is a level 2 node. It is two levels under root node *Legs* and one level under *Legs('mammal')*.

Node address

The complete namespace of a node, consisting of the global name and all subscripts. For example, node address *Legs('fish')* consists of root node identifier `Legs` plus a list containing one subscript, `'fish'`. Depending on context, *Legs* (with no subscript list) can refer to either the root node address or the entire global array.

Root node

The unsubscripted node at the base of the global array tree. The identifier for a root node is its global name with no subscripts.

Subnode

All descendants of a given node are referred to as *subnodes* of that node. For example, node *Legs('bug')* has four different subnodes on two levels. All nine subscripted nodes are subnodes of root node *Legs*.

Subscript / Subscript list

All nodes under the root node are addressed by specifying the global name and a list of one or more subscript identifiers. (The global name plus the subscript list is the node address). Subscripts can be bool, bytes, bytearray, Decimal, float, int, or str.

Target address

Many Native SDK methods require you to specify a valid node address that does not necessarily point to an existing node. For example, the **set()** method takes a *value* argument and a target address, and stores the value at that address. If no node exists at the target address, a new node is created.

Value

A node can contain a value of type bool, bytes, bytearray, Decimal, float, int, str, IRISList, or None (see Typecast Methods and Supported Datatypes). A node that has child nodes can be valueless, but a node with no child nodes *must* contain a value.

Valueless node

A node must either contain data, have child nodes, or both. A node that has child nodes but does not contain data is called a valueless node. Valueless nodes only exist as pointers to lower level nodes.

4.1.2 Global Naming Rules

Global names and subscripts obey the following rules:

- The length of a node address (totaling the length of the global name and all subscripts) can be up to 511 characters. (Some typed characters may count as more than one encoded character for this limit. For more information, see "Maximum Length of a Global Reference").
- A global name can include letters, numbers, and periods (`'.'`), and can have a length of up to 31 significant characters. It must begin with a letter, and must not end with a period.
- A subscript can be bool, bytes, bytearray, Decimal, float, int, or str. String subscripts are case-sensitive, and can contain any character (including non-printing characters). Subscript length is restricted only by the maximum length of a node address.

4.2 Fundamental Node Operations

This section demonstrates how to use the **set()**, **get()**, and **kill()** methods to create, access, and delete nodes. These methods have the following signatures:

```
set (value, globalName, subscripts)
get (globalName, subscripts)
kill (globalName, subscripts)
```

- *value* can be bool, bytes, bytearray, Decimal, float, int, str, IRISList, or None.
- *globalName* can only include letters, numbers, and periods (`'.'`), must begin with a letter, and cannot end with a period.
- *subscripts* can be bool, bytes, bytearray, Decimal, float, int, or str. A string subscript is case-sensitive and can include non-printing characters.

All of the examples in this section assume that a connected instance of IRIS named *irispy* already exists (see "Creating a Connection in Python").

Setting and changing node values

iris.IRIS.**set()** takes *value*, *globalname*, and **subscripts* arguments and stores the value at the specified node address. If no node exists at that address, a new one is created.

In the following example, the first call to **set()** creates a new node at subnode address *myGlobal('A')* and sets the value of the node to string `'first'`. The second call changes the value of the subnode, replacing it with integer `1`.

```
irispy.set('first','myGlobal','A')  # create node myGlobal('A') = 'first'
irispy.set(1,'myGlobal','A')        # change value of myGlobal('A') to 1.
```

Retrieving node values with get()

iris.IRIS.**get()** takes *globalname* and **subscripts* arguments and returns the value stored at the specified node address, or None if there is no value at that address.

```
irispy.set(23,'myGlobal','A')
value_of_A = irispy.get('myGlobal','A')
```

The **get()** method returns an untyped value. To return a specific datatype, use one of the IRIS.get() typecast methods. The following methods are available: **getBoolean()**, **getBytes()**, **getDecimal() getFloat() getInteger()**, **getString()**, **getIRISList()**, and **getObject()**.

Deleting a node or group of nodes

iris.IRIS.**kill()** — deletes the specified node and all of its subnodes. The entire global array will be deleted if the root node is deleted or if all nodes with values are deleted.

Global array *myGlobal* initially contains the following nodes:

```
myGlobal = <valueless node>
  myGlobal('A') = 0
    myGlobal('A',1) = 0
    myGlobal('A',2) = 0
  myGlobal('B') = <valueless node>
    myGlobal('B',1) = 0
```

This example will delete the global array by calling **kill()** on two of its subnodes. The first call will delete node *myGlobal('A')* and both of its subnodes:

```
irispy.kill('myGlobal','A')    # also kills myGlobal('A',1) and myGlobal('A',2)
```

The second call deletes the last remaining subnode with a value, killing the entire global array:

```
irispy.kill('myGlobal','B',1)  # deletes last value in global array myGlobal
```

- The parent node, *myGlobal('B')*, is deleted because it is valueless and now has no subnodes.
- Root node *myGlobal* is valueless and now has no subnodes, so the entire global array is deleted from the database.

4.3 Iteration with nextSubscript() and isDefined()

In ObjectScript, the standard iteration methods are $ORDER and $DATA. The Native SDK provides corresponding methods **nextSubscript()** and **isDefined()** for those who wish to emulate the ObjectScript methods.

The IRIS.**nextSubscript()** method (corresponds to `$ORDER`) is a much less powerful iteration method than **node()**, but it works in much the same way, iterating over a set of nodes under the same parent. Given a node address and direction of iteration, it returns the subscript of the next node under the same parent as the specified node, or `None` if there are no more nodes in the indicated direction.

The IRIS.**isDefined()** method (corresponds to `$DATA`) can be used to determine if a specified node has a value, a subnode, or both. It returns one of the following values:

- `0` — the specified node does not exist
- `1` — the node exists and has a value
- `10` — the node is valueless but has a child node
- `11` — the node has both a value and a child node

The returned value can be used to determine several useful boolean values:

```
exists = (irispy.isDefined(root,subscripts) > 0)
hasValue = (irispy.isDefined(root,subscripts) in [1,11]) # [value, value+child]
hasChild = (irispy.isDefined(root,subscripts) in [10,11]) # [child, value+child]
```

Find sibling nodes and test for children

The following code uses **nextSubscript()** to iterate over nodes under *heroes('dogs')*, starting at *heroes('dogs',chr(0))* (the first possible subscript). It tests each node with **isDefined()** to see if it has children.

```
direction = 0 # direction of iteration (boolean forward/reverse)
next_sub = chr(0) # start at first possible subscript
while next_sub != None:
  if (irispy.isDefined('heroes','dogs',next_sub) in [10,11]): # [child, value+child]
    print('   ', next_sub, 'has children')
  next_sub = irispy.nextSubscript(direction,'heroes','dogs',next_sub)
  print('next subscript = ' + str(next_sub) )
```

Prints:

```
next subscript = Balto
next subscript = Hachiko
next subscript = Lassie
    Lassie has children
next subscript = Whitefang
next subscript = None
```

5

Managing Transactions and Locking with Python

The *Native SDK for Python* provides transaction and locking methods that use the InterSystems transaction model, as described in the following sections:

- Processing Transactions — describes how transactions are started, nested, rolled back, and committed.
- Concurrency Control — describes how to use the various lock methods.

For information on the InterSystems transaction model, see "Transaction Processing" in *Using ObjectScript*.

5.1 Processing Transactions in Python

The iris.IRIS class provides the following methods for transaction processing:

- IRIS.**tCommit()** — commits one level of transaction.
- IRIS.**tStart()** — starts a transaction (which may be a nested transaction).
- IRIS.**getTLevel()** — returns an int value indicating the current transaction level (0 if not in a transaction).
- IRIS.**increment()** — increments or decrements a node value without locking the node.
- IRIS.**tRollback()** — rolls back all open transactions in the session.
- IRIS.**tRollbackOne()** — rolls back the current level transaction only. If this is a nested transaction, any higher-level transactions will not be rolled back.

The following example starts three levels of nested transaction, storing a different global node value at each transaction level. All three nodes are printed to prove that they have values. The example then rolls back the second and third levels and commits the first level. All three nodes are printed again to prove that only the first node still has a value. The example also increments two counters during the transactions to demonstrate the difference between the **increment()** method and the += operator.

Note: This example uses global arrays as a convenient way to store values in the database without having to create a new storage class. Global array operations that are not directly relevant to this example are isolated in utility functions listed immediately after the main example.

Controlling Transactions: Creating and rolling back three levels of nested transaction

Assume that *irispy* is a connected instance of iris.IRIS (see "Creating a Connection in Python").

Global array utility functions *store_values()*, *show_values()*, *start_counters()*, and *show_counters()* are listed immediately after this example (see "Global array utility functions").

```
tlevel = irispy.getTLevel()
counters = start_counters()
action = 'Initial values:'.ljust(18,' ') + 'tLevel='+str(tlevel)
print(action + ', ' + show_counters() + ', ' + show_values() )

print('\nStore three values in three nested transaction levels:')
while tlevel < 3:
  irispy.tStart()   # start a new transaction, incrementing tlevel by 1
  tlevel = irispy.getTLevel()
  store_values(tlevel)
  counters['add'] += 1                              # increment with +=
  irispy.increment(1,counters._global_name, 'inc')  # call increment()
  action = '  tStart:'.ljust(18,' ') + 'tLevel=' + str(tlevel)
  print(action + ', ' + show_counters() + ', ' + show_values() )

print('\nNow roll back two levels and commit the level 1 transaction:')
while tlevel > 0:
  if (tlevel>1):
    irispy.tRollbackOne()         # roll back to level 1
    action = '  tRollbackOne():'
  else:
    irispy.tCommit()              # commit level 1 transaction
    action = '  tCommit():'
  tlevel = irispy.getTLevel()
  action = action.ljust(18,' ') + 'tLevel=' + str(tlevel)
  print(action + ', ' + show_counters() + ', ' + show_values() )
```

Prints:

```
Initial values:   tLevel=0, add=0/inc=0, values=[]

Store three values in three nested transaction levels:
  tStart:         tLevel=1, add=1/inc=1, values=["data1"]
  tStart:         tLevel=2, add=2/inc=2, values=["data1", "data2"]
  tStart:         tLevel=3, add=3/inc=3, values=["data1", "data2", "data3"]

Now roll back two levels and commit the level 1 transaction:
  tRollbackOne(): tLevel=2, add=2/inc=3, values=["data1", "data2"]
  tRollbackOne(): tLevel=1, add=1/inc=3, values=["data1"]
  tCommit():      tLevel=0, add=1/inc=3, values=["data1"]
```

Global array utility functions

This example uses global arrays as a convenient way to store values in the database without having to create a new storage class (see "Accessing Global Arrays with Python"). The following functions use two IRISGlobalNode objects named *data_node* and *counter_node* to store and retrieve persistent data.

The *data_node* object will be used to store and retrieve transaction values. A separate child node will be created for each transaction, using the level number as the subscript.

```
irispy.kill('my.data.node') # delete data from previous tests
data_node = irispy.node('my.data.node')  # create IRISGlobalNode object

def store_values(tlevel):
  ''' store data for this transaction using level number as the subscript '''
  data_node[tlevel] = '"data'+str(tlevel)+'"'  # "data1", "data2", etc.

def show_values():
  ''' display values stored in all subnodes of data_node '''
  return 'values=[' + ", ".join([str(val) for val in data_node.values()]) + ']'
```

The **increment()** method is typically called before attempting to add a new entry to a database, allowing the counter to be incremented quickly and safely without having to lock the node. The value of the incremented node is not affected by transaction rollbacks.

To demonstrate this, the *counter_node* object will be used to manage two counter values. The *counter_node('add')* subnode will be incremented with the standard += operator, and the *counter_node('inc')* subnode will be incremented with the **increment()** method. Unlike the *counter_node('add')* value, the *counter_node('inc')* value will retain its value after rollbacks.

```
# counter_node object will manage persistent counters for both increment methods
irispy.kill('my.count.node') # delete data left from previous tests
counter_node = irispy.node('my.count.node')  # create IRISGlobalNode object

def start_counters():
  ''' initialize the subnodes and return the IRISGlobalNode object '''
  counter_node['add'] = 0  # counter to be incremented by += operator
  counter_node['inc'] = 0  # counter to be incremented by IRIS.increment()
  return counter_node

def show_counters():
  ''' display += and increment() counters side by side: add=#/inc=#  '''
  return 'add='+str(counter_node['add'])+'/inc='+str(counter_node['inc'])
```

5.2 Concurrency Control with Python

Concurrency control is a vital feature of multi-process systems such as InterSystems IRIS. It provides the ability to lock specific elements of data, preventing the corruption that would result from different processes changing the same element at the same time. The Native SDK transaction model provides a set of locking methods that correspond to ObjectScript commands (see "LOCK" in the *ObjectScript Reference*).

The following methods of class iris.IRIS are used to acquire and release locks:

- IRIS.**lock()** — locks the node specified by the *lockReference* and **subscripts* arguments. This method will time out after a predefined interval if the lock cannot be acquired.
- IRIS.**unlock()** — releases the lock on the node specified by the *lockReference* and **subscripts* arguments.
- IRIS.**releaseAllLocks()** — releases all locks currently held by this connection.

parameters:

```
lock(lockMode, timeout, lockReference, *subscripts)
unlock(lockMode, lockReference, *subscripts)
releaseAllLocks()
```

- *lockMode* — str specifying how to handle any previously held locks. Valid arguments are, S for shared lock, E for escalating lock, or SE for shared and escalating. Default is empty string (exclusive and non-escalating).
- *timeout* — number of seconds to wait before timing out when attempting to acquire a lock.
- *lockReference* — str starting with a circumflex (^) followed by the global name (for example, `^myGlobal`, *not* just `myGlobal`).

  Important: the `lockReference` parameter *must* be prefixed by a circumflex, unlike the `globalName` parameter used by most methods. Only **lock()** and **unlock()** use `lockReference` instead of `globalName`.
- *subscripts* — zero or more subscripts specifying the node to be locked or unlocked.

In addition to these methods, the IRISConnection.**close()** method is used to release all locks and other connection resources.

Tip: You can use the Management Portal to examine locks. Go to System Operation > View Locks to see a list of the locked items on your system.

Note: A detailed discussion of concurrency control is beyond the scope of this document. See the following articles for more information on this subject:

- "Transaction Processing" and "Lock Management" in *Using ObjectScript*
- "Locking and Concurrency Control" in the *Orientation Guide for Server-Side Programming*
- "LOCK" in the *ObjectScript Reference*

6

Using the Python DB-API

The InterSystems Python DB-API driver is a fully compliant implementation of the PEP 249 version 2.0 Python Database API specification. The following sections list all required implementation features, indicate the level of support for each one, and describe all InterSystems-specific features in detail:

- Usage

  Describes how to make a connection to InterSystems IRIS and get a Cursor object.

- PEP 249 Implementation Reference

  Lists all PEP 249 requirements and provides implementation details in the following subsections:

  – Globals lists values for required global constants *apilevel*, *threadsafety*, and *paramstyle*.

  – Connection Object describes Connection methods connect(), close(), commit(), rollback(), and cursor().

  – Cursor Object describes the following Cursor members:

    - Attributes arraysize, description, and rowcount.
    - Standard methods callproc(), close(), execute(), executemany(), fetchone(), fetchmany(), fetchall(), nextset(), scroll(), setinputsizes(), and setoutputsize().
    - InterSystems extension methods isClosed() and stored_results().

- SQLType enumeration values

  Lists valid SQLType enumeration constants.

Note: **DB-API Driver Installation**

The DB-API is available when you install InterSystems IRIS. If you do not have the InterSystems DB-API driver (for example, if you are connecting from a host on which InterSystems IRIS is not installed), you can download it from the InterSystems IRIS Drivers page and install it with:

```
pip install intersystems_irispython-<version>.whl
```

6.1 Usage

The following example makes a connection to the InterSystems IRIS database, creates a cursor associated with the connection, sets up to make some DB-API calls, and then shuts down.

See "Connection Object" and "Cursor Object" in the following section for detailed documentation on all available DBAPI methods.

Connecting to the DB-API driver and getting a cursor

Python

```
import iris

def main():
  connection_string = "localhost:1972/USER"
  username = "_system"
  password = "SYS"

  connection = iris.connect(connection_string, username, password)
  cursor = connection.cursor()

  try:
    pass  # do something with DB-API calls
  except Exception as ex:
    print(ex)
  finally:
    if cursor:
      cursor.close()
    if connection:
      connection.close()

if __name__ == "__main__":
  main()
```

See iris.connect(), Connection.close(), Connection.cursor(), and Cursor.close() for more information on the methods called in this example.

Connecting Your Application to InterSystems IRIS also provides instructions, including sample code, for connecting to an InterSystems IRIS server from a Python application using DB-API.

6.2 PEP 249 Implementation Reference

This section lists all required implementation features described in the PEP 249 version 2.0 Python Database API specification, indicates the level of support for each one, and describes all InterSystems-specific features in detail.

6.2.1 Globals

These are required implementation-specific constants. In the InterSystems implementation, these globals are set to the following values:

apilevel

`"2.0"` — specifies compliance with PEP 249 version 2.0.

threadsafety

`0` — threads may not share the module.

paramstyle

This is a global in the _DBAPI.py file that indicates the format of the input parameters when parameterized calls to **execute()**, **executemany()**, and **callproc()** are invoked with SQL statements. The following values are supported:

- `"qmark"` — query parameters use question mark style (for example: `WHERE name=?`).
- `"named"` — query parameters use named style (for example: `WHERE name=:name`)).

The default value is `qmark`. The following example sets it to `named`:

```
import iris
iris.dbapi._DBAPI.paramstyle = "named"
```

Examples using "qmark"

An input parameter is indicated by a question mark (?). The input parameters are provided as a Python list.

```
sql = "Select * from Sample.Person where id = ? and name = ?"
params = [1, 'Jane Doe']
cursor.execute(sql, params) // same for direct_execute

sql = "Insert into Sample.Person (name, phone) values (?, ?)"
params = [('ABC', '123-456-7890'), ('DEF', '234-567-8901'), ('GHI', '345-678-9012')]
cursor.executemany(sql, params) // batch update

proc = "{ ? = Proc1(?) }" // parameter modes: RETURN_VALUE, INPUT
params = [1]
return_args = cursor.callproc(proc, params) // stored procedure

proc = "{ CALL Proc2 (?, ?) }" // parameter modes: INPUT, OUTPUT
params = ['abc']
return_args = cursor.callproc(proc, params) // stored procedure
```

Examples using "named"

An input parameter is indicated by a variable name preceded by a colon. The input parameters are provided as a Python dictionary where the keys indicate the variable names used in the SQL and the values for the keys hold the actual data for the corresponding variables/parameters. Named parameters will work even if *paramstyle* is not set.

```
sql = "SELECT * FROM Sample.Person WHERE firstname = :fname AND lastname = :lname"
params = {'fname' : 'John', 'lname' : 'Doe'}
cursor.execute(sql, params)

sql = "INSERT INTO Sample.Person(name) VALUES (:name)"
params = [{'name' : 'John'}, {'name' : 'Jane'}]
cursor.executemany(sql, params)
```

6.2.2 Connection Object

This section describes how to use iris.connect() to create a Connection object, and provides implementation details for required Connection methods close(), commit(), rollback(), and cursor().

6.2.2.1 Creating a Connection Object

DB-API Connection objects are created by calls to the InterSystems iris.**connect()** method:

connect()

iris.**connect()** returns a new Connection object and attempts to create a new connection to an instance of InterSystems IRIS. The object will be open if the connection was successful, or closed otherwise (see Cursor.**isClosed**()).

```
iris.connect(hostname,port,namespace,username,password,timeout,sharedmemory,logfile)
iris.connect(connectionstr,username,password,timeout,sharedmemory,logfile)
```

The *hostname*, *port*, *namespace*, *timeout*, and *logfile* from the last successful connection attempt are saved as properties of the connection object.

parameters:

Parameters may be passed by position or keyword.

- `hostname` — str specifying the server URL

- `port` — int specifying the superserver port number
- `namespace` — str specifying the namespace on the server
- The following parameter can be used in place of the `hostname`, `port`, and `namespace` arguments:
  - `connectionstr` — str of the form *`hostname:port/namespace`*.

- `username` — str specifying the user name
- `password` — str specifying the password
- `timeout` (optional) — int specifying maximum number of seconds to wait while attempting the connection. Defaults to 10.
- `sharedmemory` (optional) — specify bool `True` to attempt a shared memory connection when the hostname is `localhost` or `127.0.0.1`. Specify `False` to force a connection over TCP/IP. Defaults to `True`.
- `logfile` (optional) — str specifying the client-side log file path. The maximum path length is 255 ASCII characters.

6.2.2.2 Connection Object Methods

A Connection object can be used to create one or more Cursor objects. Database changes made by one cursor are immediately visible to all other cursors created from the same connection. Rollbacks and commits affect all changes made by cursors using this connection.

close()

Connection.**close()** closes the connection immediately. The connection and all cursors associated with it will be unusable. An implicit rollback will be performed on all uncommitted changes made by associated cursors.

```
Connection.close()
```

A ProgrammingError exception will be raised if any operation is attempted with a closed connection or any associated cursor.

commit()

Connection.**commit()** commits all SQL statements executed on the connection since the last commit/rollback. The rollback affects all changes made by any cursor using this connection. Explicit calls to this method are not required.

```
Connection.commit()
```

rollback()

Connection.**rollback()** rolls back all SQL statements executed on the connection that created this cursor (since the last commit/rollback). It affects all changes made by any cursor using this connection.

```
Connection.rollback()
```

cursor()

Connection.**cursor()** returns a new Cursor object that uses this connection.

```
Connection.cursor()
```

Any changes made to the database by one cursor are immediately visible to all other cursors created from the same connection. Rollbacks and commits affect all changes made by any cursor using this connection.

6.2.3 Cursor Object

This section describes how to create a Cursor object, and provides implementation details for the following required Cursor methods and attributes:

- Attributes arraysize, description, and rowcount.
- Standard methods callproc(), close(), execute(), executemany(), fetchone(), fetchmany(), fetchall(), nextset(), scroll(), setinputsizes(), and setoutputsize().
- InterSystems extension methods isClosed() and stored_results().

6.2.3.1 Creating a Cursor object

A Cursor object is created by establishing a connection and then calling Connection.**cursor()**. For example:

```
connection = iris.connect(connection_string, username, password)
cursor = connection.cursor()
```

Any changes made to the database by one cursor are immediately visible to all other cursors created from the same connection.

Once the cursor is closed, accessing the column data of a DataRow object or any other attributes/functions of the Cursor class will result in an error.

See "Connecting to the DB-API driver and getting a cursor" for a more complete example. See "Creating a Connection Object" for detailed information on creating a connection.

6.2.3.2 Cursor attributes

arraysize

Cursor.**arraysize** is a read/write attribute that specifies the number of rows to fetch at a time with fetchmany(). Default is `1` (fetch one row at a time).

description

Cursor.**description** returns a list of tuples containing information for each result column returned by the last SQL select statement. Value will be *None* if an execute method has not been called, or if the last operation did not return any rows.

Each tuple (column description) in the list contains the following items:

- *name* — column name (defaults to `None`)
- *type_code* — integer SQLType identifier (defaults to `0`). See "SQLType enumeration values" for valid values.
- *display_size* — not used - value set to `None`
- *internal_size* — not used - value set to `None`
- *precision* — integer (defaults to `0`)
- *scale* — integer (defaults to `None`)
- *nullable* — integer (defaults to `0`)

rowcount

Cursor.**rowcount** specifies the number of rows modified by the last SQL statement. The value will be `-1` if no SQL has been executed or if the number of rows is unknown. For example, DDLs like CREATE, DROP, DELETE, and SELECT statements (for performance reasons) return -1.

Batch updates also return the number of rows affected.

6.2.3.3 Cursor methods

callproc()

Cursor.**callproc()** calls a stored database procedure with the given *procname*.

```
Cursor.callproc(procname)
Cursor.callproc(procname, parameters)
```

parameters:

- `procname` – string containing a stored procedure call with parameterized arguments.
- `parameters` – list of parameter values to pass to the stored procedure

Any of the fetch*() methods can be used to access the rows of a result set for a stored procedure that is expected to return result sets. They are expected to behave in the same way as for SELECT queries (non-procedures). For example, after using **callproc()** to call a procedure that is expected to return at least one result set, **fetchone()** will return the first row of the first result set and subsequent calls to **fetchone()** will return the remaining rows one by one. A **fetchall()** call will return all the remaining rows of the current result set.

For example, this code calls stored procedure `Sample.SP_Sample_By_Name`, specifying parameter value `"A"` in a list:

```
cursor.callproc("CALL Sample.SP_Sample_By_Name (?)", ["A"])
row = cursor.fetchone()
while row:
  print(row.ID, row.Name, row.DOB, row.SSN)
  row = cursor.fetchone()
```

Output will be similar to the following:

```
167 Adams,Patricia J. 1964-10-12 216-28-1384
28 Ahmed,Dave H. 1954-01-12 711-67-4091
20 Alton,Samantha E. 2015-03-28 877-53-4204
118 Anderson,Elvis V. 1994-05-29 916-13-245
```

If a stored procedure of `'function'` type is expected to return a result set, then it will be available in the return value of **callproc()** as a tuple at the corresponding placeholder. An internal call to **fetchall()** is made in this specific case, hence, the tuple holds all the rows of the result set. The rows hold all the column data as well.

For example, the stored procedure below has two parameters whose modes are `RETURN_VALUE` and `INPUT`, respectively:

```
proc = "{ ? = MyProc3(?) }"
params = [1]
return_args = cursor.callproc(proc, params)
print(return_args[0]) # tuple of all the result set rows
print(return_args[1] == None)
```

Note: Outputs of **fetch*()** and **callproc()** APIs which returned Python lists in previous releases now return Python tuples. If the output from older version was `[1, "hello"]`, then the new version will return `(1, "hello")`. Python tuples containing one item will have a comma appended. For example, a list such as `[100]` is represented as tuple `(100,)`.

close()

Cursor.**close()** closes the cursor.

```
Cursor.close()
```

A ProgrammingError exception will be raised if any operation is attempted with a closed cursor. Cursors are closed automatically when they are deleted (typically when they go out of scope), so calling this is not usually necessary.

execute()

Cursor.**execute()** executes the query specified in the *operation* parameter. Updates the Cursor object and sets the rowcount attribute to `-1` for a query or `1` for an update.

```
Cursor.execute(operation)
Cursor.execute(operation, parameters)
```

parameters:

- `operation` – string containing SQL statement to be executed
- `parameters` – optional list of values. This must be a Python list (tuples or sets are not acceptable).

examples:

Parameter values are used in positions where the SQL statement contains a `?` (qmark) rather than a literal or constant. If the statement does not contain any qmarks, the *parameters* argument is not required will raise an exception if given.

- `sql = "...(1,2)..."; execute(sql)`
- `sql = "...(?,?)..."; params = [1,2]; execute(sql, params)`
- `sql = "...(1,?)..."; params = [2]; execute(sql, params)`

executemany()

Cursor.**executemany()** is used for batch inserts/updates. It prepares a database operation (query or command) and then executes it against all parameter sequences or mappings found in the sequence *seq_of_parameters*.

```
Cursor.executemany(operation)
Cursor.executemany(operation, seq_of_parameters)
```

parameters:

- `operation` – string containing SQL INSERT or UPDATE statement to be executed
- `seq_of_parameters` – sequence of parameter sequences or mappings

Returns a tuple of integers and/or strings, depending on success or failure of the INSERT or UPDATE operation. Each item in the tuple corresponds to the row in the batch which is a list/tuple of user-provided parameters. (Integers for success, strings for error messages in case of failure). Returns 1 to indicate every successful INSERT or an error message with details in case of a failure. Returns a cardinal number to indicate the number of rows that were successfully modified for an UPDATE or an error message with details in case of failure. The rowcount attribute indicates the number of rows successfully inserted/updated.

fetchone()

Cursor.**fetchone()** returns the pointer to the next ResultSetRow.DataRow object (integer array of data offsets) in the query, or `None` if no more data is available.

```
Cursor.fetchone()
```

Data is fetched only on request, via indexing. The object contains a list of integer offsets that can be used to retrieve the row values. Index values must be positive integers (a value of 1 refers to column 1, and so on).

Column values can be fetched using cardinal values, column name (as a string), and the slice operator, but not via dynamic attributes. For example, `row[:]` fetches all the column data, `row[0]` fetches the data in first column, `row[1]` fetches the data in second column, and `row['<columnName>']` fetches the data from the `<columnName>` column, but `row.columnName` will not work.

A ProgrammingError exception is raised if no SQL has been executed or if it did not return a result set (for example, if it was not a SELECT statement).

fetchmany()

Cursor.**fetchmany()** fetches the next set of rows of a query result, returning a sequence of sequences (a list of tuples). If the *size* argument is not specified, the number of rows to fetch at a time is set by the Cursor.arraysize attribute (default `1`). An empty sequence is returned when no more rows are available.

```
Cursor.fetchmany()
Cursor.fetchmany(size)
```

parameters:

- `size` – optional. Defaults to the current value of attribute Cursor.arraysize.

fetchall()

Cursor.**fetchall()** fetches all remaining rows of a query result.

```
Cursor.fetchall()
```

isClosed() [InterSystems extension method]

Cursor.**isClosed**() is an InterSystems extension method that returns `True` if the cursor object is already closed, `False` otherwise.

```
Cursor.isClosed()
```

nextset() [optional DB-API method]

Cursor.**nextset()** is an optional DB-API method for iterating over multiple result sets. It makes the cursor skip to the next available set, discarding any remaining rows from the current set. If there are no more sets, the method returns None. Otherwise, it returns True and subsequent calls to the **fetch*()** methods will return rows from the next result set.

```
Cursor.nextset()
```

Example:

```
for row in cursor.stored_results():
  row_values = row[0] // data in all columns
  val1 = row[1]       // data in column 1
  cursor.nextset()    // skips to the next result set if multiple result sets
  // does nothing (or breaks out of loop) in case of single result set;
```

scroll() [optional DB-API method]

Cursor.**scroll**() is an optional DB-API method that scrolls the cursor in the result set to a new position and returns the row at that position. This method does not work with stored procedures. It raises an IndexError if scroll operation would leave the result set.

```
Cursor.scroll(value, mode)
```

parameters:

- `value` – integer value specifying the new target position.
  - If *mode* is `relative` (the default) , *value* is a positive or negative offset to the current position in the result set.
  - If *mode* is `absolute`, *value* is an absolute target position (negative values are not valid).
- `mode` – optional. Valid values are `relative` or `absolute`. The use of an empty string for the *mode* argument sets its value to `relative` (for example, `cursor.scroll(3,'')`).

Example:

For each example, assume the result set has a total of `10` rows, and the initial number of rows fetched is `5`. Result set index values are 0–based, so the current position in the result set is `rs[4]` (the 5th row).

```
cursor.execute("select id, * from simple.human where id <= 10")
cursor.fetchmany(5)

# Scroll forward 3 rows, relative to Row 5
datarow = cursor.scroll(3,'relative')      # Row 8
print(datarow[0] == 8)

# Scroll to absolute position 3
datarow = cursor.scroll(3,'absolute')      # Row 3
print(datarow[0] == 3)

# Scroll backward 4 rows, relative to Row 3 (mode defaults to 'relative')
datarow = cursor.scroll(-4,'')             # Row 9
print(datarow[0] == 9)

# Attempt to scroll to absolute position -4
# Error: Negative values with absolute scrolling are not allowed.
datarow = cursor.scroll(-4,'absolute')     # ERROR
print(datarow[0])
```

setinputsizes()

Cursor.**setinputsizes()** is not applicable to InterSystems IRIS, which does not implement or require this functionality. Throws NotImplementedError if called.

setoutputsize()

Cursor.**setoutputsize()** is not applicable to InterSystems IRIS, which does not implement or require this functionality. Throws NotImplementedError if called.

Note: The **stored_results()** method is deprecated and will be removed in a future release.

stored_results() [DEPRECATED InterSystems extension method]

Cursor.**stored_results**() is an InterSystems extension method that returns a list iterator (containing first row of each result set) if the procedure type is `'query'`, and empty list if the procedure type is `'function'`

```
Cursor.stored_results()
```

Example:

```
  for row in cursor.stored_results(): // row is DataRow object for 1st row of result set
    row_values = row[0] // data in all columns
    val1 = row[1] // data in column 1

Incorrect Syntax:
  row = cursor.stored_results() // row values not accessible using row[0] since it is a list
iterator
```

6.3 SQLType enumeration values

Valid values for the Cursor.description attribute.

- BIGINT = -5
- BINARY = -2
- BIT = -7
- CHAR = 1
- DECIMAL = 3
- DOUBLE = 8
- FLOAT = 6
- GUID = -11
- INTEGER = 4
- LONGVARBINARY = -4
- LONGVARCHAR = -1
- NUMERIC = 2
- REAL = 7
- SMALLINT = 5
- DATE = 9
- TIME = 10
- TIMESTAMP = 11
- TINYINT = -6
- TYPE_DATE = 91
- TYPE_TIME = 92
- TYPE_TIMESTAMP = 93
- VARBINARY = -3
- VARCHAR = 12
- WCHAR = -8
- WLONGVARCHAR = -10
- WVARCHAR = -9
- DATE_HOROLOG = 1091

- TIME_HOROLOG = 1092
- TIMESTAMP_POSIX = 1093

7

Native SDK for Python Quick Reference

This is a quick reference for the InterSystems IRIS Native SDK for Python, providing information on the following classes:

- package iris
  - iris Package Methods — connect and create an instance of IRIS.
  - Class IRIS main entry point for the Native SDK
  - Class IRISConnection connects an IRIS instance to the database.
  - Class IRISList provides support for InterSystems `$LIST` serialization.
  - Class iris.IRISGlobalNode navigate global arrays
  - Class iris.IRISGlobalNodeView dictionary-like views of child nodes
  - Class IRISReference passes method arguments by reference.
  - Class IRISObject provides methods for External Server inverse proxy objects.

- Legacy Support Classes
  - Class LegacyIterator iterator methods retained for compatibility with earlier implementations.
  - Class IRISNative connection methods retained for compatibility with earlier implementations.

Typecast Methods and Supported Datatypes

The Native SDK supports datatypes bool, bytes, bytearray, Decimal, float, int, str, IRISList, and None.

In many cases, a class will have a generic method (frequently named **get()**) that returns a default value type, plus a set of typecast methods that work exactly like the generic method but also cast the return value to a specific datatype. For example, IRIS.**get()** is followed by typecast methods **getBoolean()**, **getBytes()**, and so forth.

To avoid lengthening this quick reference with dozens of nearly identical listings, all typecast methods are listed collectively after the generic version (for example, IRIS.get() Typecast Methods).

Sets of typecast methods are available for IRIS.**classMethodValue()**, IRIS.**function()**, IRIS.**get()**, IRISList.**get()**, IRISObject.**get()**, IRISObject.**invoke()**, and IRISReference.**getValue()**.

7.1 iris Package Methods

The iris package includes method **connect()** for creating a connection, and **createIRIS()** for creating a new instance of class IRIS that uses the connection. A connected instance of iris.IRIS is required to use the Native SDK.

7.1.1 Creating a Connection in Python

The following code demonstrates how to open a database connection and create an instance of iris.IRIS named *irispy*. Most examples in this document will assume that a connected instance of *irispy* already exists.

```
import iris

# Open a connection to the server
args = {'hostname':'127.0.0.1', 'port':52773,
    'namespace':'USER', 'username':'_SYSTEM', 'password':'SYS'
}
conn = iris.connect(**args)
# Create an iris object
irispy = iris.createIRIS(conn)
```

See the following section for detailed information about iris.**connect()** and iris.**createIRIS()**.

7.1.2 iris Package Method Details

connect()

iris.**connect()** returns a new IRISConnection object and attempts to create a new connection to the database. The object will be open if the connection was successful. Throws an exception if a connection cannot be established.

```
iris.connect(hostname,port,namespace,username,password,timeout,sharedmemory,logfile)
iris.connect(connectionstr,username,password,timeout,sharedmemory,logfile)
```

The *hostname*, *port*, *namespace*, *timeout*, and *logfile* from the last successful connection attempt are saved as properties of the connection object.

parameters:

Parameters may be passed by position or keyword.

- `hostname` — str specifying the server URL
- `port` — int specifying the superserver port number
- `namespace` — str specifying the namespace on the server
- The following parameter can be used in place of the `hostname`, `port`, and `namespace` arguments:
  - `connectionstr` — str of the form *`hostname:port/namespace`*.
- `username` — str specifying the user name
- `password` — str specifying the password
- `timeout` (optional) — int specifying maximum number of milliseconds to wait while attempting the connection. Defaults to 10000.
- `sharedmemory` (optional) — specify bool `True` to attempt a shared memory connection when the hostname is `localhost` or `127.0.0.1`. Specify `False` to force a connection over TCP/IP. Defaults to `True`.

- `logfile` (optional) — str specifying the client-side log file path. The maximum path length is 255 ASCII characters.

createIRIS()

iris.**createIRIS()** returns a new instance of IRIS that uses the specified IRISConnection. Throws an exception if the connection is closed.

```
iris.createIRIS(conn)
```

returns: a new instance of iris.IRIS

parameter:

- `conn` — an IRISConnection object that provides the server connection

7.2 Class iris.IRIS

To use the Native SDK, your application must create an instance of IRIS with a connection to the database. Instances of IRIS are created by calling iris package method **createIRIS()**.

7.2.1 IRIS Method Details

classMethodValue()

IRIS.**classMethodValue()** calls a class method, passing zero or more arguments and returns the value as a type corresponding to the datatype of the ObjectScript return value. Returns None if the ObjectScript return value is an empty string (`$$$NULLOREF`). See "Calling Class Methods from Python" for details and examples.

```
classMethodValue (className, methodName, args)
```

returns: bool, bytes, bytearray, int, float, Decimal, str, IRISList, or None. See classMethodValue() Typecast Methods for ways to return a specific type.

parameters:

- `class_name` — fully qualified name of the class to which the called method belongs.
- `method_name` — name of the class method.
- `*args` — zero or more method arguments. Arguments of types bool, bytes, Decimal, float, int, str and IRISList are projected as literals. All other types are projected as proxy objects.

Also see **classMethodVoid()**, which is similar to **classMethodValue()** but does not return a value.

classMethodValue() Typecast Methods

All of the IRIS.**classMethodValue()** typecast methods listed below work exactly like IRIS.**classMethodValue()**, but also cast the return value to a specific type. They all return None if the ObjectScript return value is an empty string (`$$$NULLOREF`). See "Calling Class Methods from Python" for details and examples.

```
classMethodBoolean (className, methodName, args)
classMethodBytes (className, methodName, args)
classMethodDecimal (className, methodName, args)
classMethodFloat (className, methodName, args)
classMethodInteger (className, methodName, args)
classMethodIRISList (className, methodName, args)
classMethodString (className, methodName, args)
classMethodObject (className, methodName, args)
```

returns: bool, bytes, bytearray, int, float, Decimal, str, IRISList, Object, or None.

parameters:

- `class_name` — fully qualified name of the class to which the called method belongs.
- `method_name` — name of the class method.
- `*args` — zero or more method arguments of supported types. Arguments of types bool, bytes, Decimal, float, int, str and IRISList are projected as literals. All other types are projected as proxy objects.

Also see **classMethodVoid()**, which is similar to **classMethodValue()** but does not return a value.

classMethodVoid()

IRIS.**classMethodVoid()** calls an ObjectScript class method with no return value, passing zero or more arguments. See "Calling Class Methods from Python" for details and examples.

```
classMethodVoid (className, methodName, args)
```

parameters:

- `class_name` — fully qualified name of the class to which the called method belongs.
- `method_name` — name of the class method.
- `*args` — zero or more method arguments of supported types. Arguments of types bool, bytes, Decimal, float, int, str and IRISList are projected as literals. All other types are projected as proxy objects.

This method assumes that there will be no return value, but can be used to call any class method. If you use **classMethodVoid()** to call a method that returns a value, the method will be executed but the return value will be ignored.

close()

IRIS.**close()** closes the IRIS object.

```
close ()
```

function()

IRIS.**function()** calls a function, passing zero or more arguments and returns the value as a type corresponding to the datatype of the ObjectScript return value. Returns None if the ObjectScript return value is an empty string (`$$$NULLOREF`). See "Calling Functions and Procedures from Python" for details and examples.

```
function (functionName, routineName, args)
```

returns: bool, bytes, bytearray, int, float, Decimal, str, IRISList, Object, or None. See function() Typecast Methods for ways to return a specific type.

parameters:

- `function_name` — name of the function to call.
- `routine_name` — name of the routine containing the function.
- `*args` — zero or more method arguments of supported types. Arguments of types bool, bytes, Decimal, float, int, str and IRISList are projected as literals. All other types are projected as proxy objects.

Also see **procedure()**, which is similar to **function()** but does not return a value.

function() Typecast Methods

All of the IRIS.**function()** typecast methods listed below work exactly like IRIS.**function()**, but also cast the return value to a specific type. They all return None if the ObjectScript return value is an empty string (`$$$NULLOREF`). See "Calling Functions and Procedures from Python" for details and examples.

```
functionBoolean (functionName, routineName, args)
functionBytes (functionName, routineName, args)
functionDecimal (functionName, routineName, args)
functionFloat (functionName, routineName, args)
functionInteger (functionName, routineName, args)
functionIRISList (functionName, routineName, args)
functionString (functionName, routineName, args)
functionObject (functionName, routineName, args)
```

returns: bool, bytes, bytearray, int, float, Decimal, str, IRISList, Object, or None

parameters:

- `function_name` — name of the function to call.
- `routine_name` — name of the routine containing the function.
- `*args` — zero or more method arguments of supported types. Arguments of types bool, bytes, Decimal, float, int, str and IRISList are projected as literals. All other types are projected as proxy objects.

Also see **procedure()**, which is similar to **function()** but does not return a value.

get()

IRIS.**get()** returns the value of the global node as a type corresponding to the ObjectScript datatype of the property. Returns None if the node is an empty string, is valueless, or does not exist.

```
get (globalName, subscripts)
```

returns: bool, bytes, int, float, Decimal, str, IRISList, Object, or None. See get() Typecast Methods for ways to return other types.

parameters:

- `global_name` — global name
- `*subscripts` — zero or more subscripts specifying the target node

get() Typecast Methods

All of the IRIS.**get()** typecast methods listed below work exactly like IRIS.**get()**, but also cast the return value to a specific type. They all return None if the node is an empty string, is valueless, or does not exist.

```
getBoolean (globalName, subscripts)
getBytes (globalName, subscripts)
getDecimal (globalName, subscripts)
getFloat (globalName, subscripts)
getInteger (globalName, subscripts)
getIRISList (globalName, subscripts)
getString (globalName, subscripts)
getObject (globalName, subscripts)
```

returns: bool, bytes, int, float, Decimal, str, IRISList, Object, or None

parameters:

- `global_name` — global name
- `*subscripts` — zero or more subscripts specifying the target node

Also see deprecated typecast method **getLong()**.

getAPIVersion() [static]

IRIS.**getAPIVersion()** returns the version string for this version of the Native SDK. Also see **getServerVersion()**.

```
getAPIVersion ()
```

returns: str

getLong() [deprecated]

IRIS.**getLong()** The Python long type is no longer required in Python 3. Please use IRIS.**get()** typecast method **getInteger()**, which supports the same precision.

getServerVersion()

IRIS.**getServerVersion()** returns the version string for the currently connected InterSystems IRIS server. Also see **getAPIVersion()**.

```
getServerVersion ()
```

returns: str

getTLevel()

IRIS.**getTLevel()** returns the number of nested transactions currently open in the session (1 if the current transaction is not nested, and 0 if there are no transactions open). This is equivalent to fetching the value of the ObjectScript **$TLEVEL** special variable. See "Processing Transactions in Python" for details and examples.

```
getTLevel ()
```

returns: int

increment()

IRIS.**increment()** increments the global node with the *value* argument and returns the new value of the global node. If there is no existing node at the specified address, a new node is created with the specified value. This method uses a very fast, thread-safe atomic operation to change the value of the node, so the node is never locked. See "Processing Transactions in Python" for details and examples.

```
increment (value, globalName, subscripts)
```

returns: float

parameters:

- `value` — int or float number to increment by
- `global_name` — global name
- `*subscripts` — zero or more subscripts specifying the target node

Common usage for $INCREMENT is to increment a counter before adding a new entry to a database. $INCREMENT provides a way to do this very quickly, avoiding the use of the LOCK command. See "$INCREMENT and Transaction Processing" in the *ObjectScript Reference*.

isDefined()

IRIS.**isDefined()** returns a value that indicates whether a node has a value, a child node, or both.

```
isDefined (globalName, subscripts)
```

returns: one of the following int values:

- `0` if the node does not exist.
- `1` if the node has a value but no child nodes.
- `10` if the node is valueless but has one or more child nodes.
- `11` if it has a both a value and child nodes.

parameters:

- `global_name` — global name
- `*subscripts` — zero or more subscripts specifying the target node

iterator() [deprecated]

IRIS.**iterator()** is deprecated; use **node()** instead. See Class for information about continued functionality in existing applications.

kill()

IRIS.**kill()** deletes the specified global node and all of its subnodes. If there is no node at the specified address, the command will do nothing. It will not throw an `<UNDEFINED>` exception.

```
kill (globalName, subscripts)
```

parameters:

- `global_name` — global name
- `*subscripts` — zero or more subscripts specifying the target node

lock()

IRIS.**lock()** locks the global. This method performs an incremental lock (you must call the **releaseAllLocks()** method first if you want to unlock all prior locks). Throws a `<TIMEOUT>` exception if the *timeout* value is reached waiting to acquire the lock. See "Concurrency Control with Python" for more information.

```
lock (lock_mode, timeout, lock_reference, subscripts)
```

parameters:

- `lock_mode` — one of the following strings: `"S"` for shared lock, `"E"` for escalating lock, `"SE"` for both, or `""` for neither. An empty string is the default mode (unshared and non-escalating).
- `timeout` — number of seconds to wait to acquire the lock
- `lock_reference` — a string starting with a circumflex (`^`) followed by the global name (for example, `^myGlobal`, *not* just `myGlobal`).

  NOTE: Unlike the `global_name` parameter used by most methods, the `lock_reference` parameter *must* be prefixed by a circumflex. Only **lock()** and **unlock()** use `lock_reference` instead of `global_name`.
- `*subscripts` — zero or more subscripts specifying the target node

See "LOCK" in the ObjectScript Reference for detailed information on locks.

nextSubscript()

IRIS.**nextSubscript()** accepts a node address and returns the subscript of the next sibling node in collation order. Returns None if there are no more nodes in the specified direction. This method is similar to $ORDER in ObjectScript.

```
nextSubscript (reversed, globalName, subscripts)
```

returns: bytes, int, str, or float (next subscript in the specified direction)

parameters:

- `reversed` — boolean true indicates that nodes should be traversed in reverse collation order.
- `global_name` — global name
- `*subscripts` — zero or more subscripts specifying the target node

Also see **node()**, which returns an object that iterates over children of a specified node (unlike **nextSubscript()**, which allows you to iterate over nodes on the same level as the specified node).

node()

IRIS.**node()** returns an IRISGlobalNode object that allows you to iterate over children of the specified node. An IRISGlobalNode behaves like a virtual dictionary representing the immediate children of a global node. It is iterable, reversable, indexable and sliceable.

```
def  node (self, globalName, subscripts)
```

parameters:

- `global_name` — global name
- `*subscripts` — zero or more subscripts specifying the target node

Also see **nextSubscript()**, which allows you to iterate over nodes on the same level as the specified node (unlike **node()**, which iterates over children of a specified node).

procedure()

IRIS.**procedure()** calls an ObjectScript procedure or function, passing zero or more arguments and returning nothing (also see IRIS.**function()**). See "Calling Functions and Procedures from Python" for details and examples.

```
procedure (procedureName, routineName, args)
```

parameters:

- `procedureName` — name of the procedure to call.
- `routine_name` — name of the routine containing the procedure.
- `*args` — zero or more method arguments. Arguments of types bool, bytes, Decimal, float, int, str and IRISList are projected as literals. All other types are projected as proxy objects.

This method assumes that there will be no return value, but can be used to call any function. If you use **procedure()** to call a function that returns a value, the function will be executed but the return value will be ignored.

releaseAllLocks()

IRIS.**releaseAllLocks()** releases all locks associated with the session. See "Concurrency Control with Python" for more information.

```
releaseAllLocks ()
```

set()

IRIS.**set()** assigns *value* as the current node value. The new value may be bool, bytes, bytearray, Decimal, float, int, str, or IRISList,.

```
set (value, globalName, subscripts)
```

parameters:

- `value` — new value of the global node
- `global_name` — global name
- `*subscripts` — zero or more subscripts specifying the target node

tCommit()

IRIS.**tCommit()** commits the current transaction. See "Processing Transactions in Python" for details and examples.

```
tCommit ()
```

tRollback()

IRIS.**tRollback()** rolls back all open transactions in the session. See "Processing Transactions in Python" for details and examples.

```
tRollback ()
```

tRollbackOne()

IRIS.**tRollbackOne()** rolls back the current level transaction only. This is intended for nested transactions, when the caller only wants to roll back one level. If this is a nested transaction, any higher-level transactions will not be rolled back. See "Processing Transactions in Python" for details and examples.

```
tRollbackOne ()
```

tStart()

IRIS.**tStart()** starts or opens a transaction. See "Processing Transactions in Python" for details and examples.

```
tStart ()
```

unlock()

IRIS.**unlock()** decrements the lock count on the specified lock, and unlocks it if the lock count is 0. To remove a shared or escalating lock, you must specify the appropriate *lockMode* (`"S"` for shared, `"E"` for escalating lock). See "Concurrency Control with Python" for more information.

```
unlock (lock_mode, lock_reference, subscripts)
```

parameters:

- `lock_mode` — one of the following strings: `"S"` for shared lock, `"E"` for escalating lock, `"SE"` for both, or `""` for neither. An empty string is the default mode (unshared and non-escalating).
- `lock_reference` — a string starting with a circumflex (`^`) followed by the global name (for example, `^myGlobal`, *not* just `myGlobal`).

  NOTE: Unlike the `global_name` parameter used by most methods, the `lock_reference` parameter *must* be prefixed by a circumflex. Only **lock()** and **unlock()** use `lock_reference` instead of `global_name`.
- `*subscripts` — zero or more subscripts specifying the target node

7.3 Class iris.IRISList

Class iris.IRISList implements a Python interface for InterSystems $LIST serialization. IRISList is a Native SDK supported type (see "Typecast Methods and Supported Datatypes").

7.3.1 IRISList Constructors

The IRISList constructor takes the following parameters:

```
IRISList(buffer = None, locale = "latin-1", is_unicode = True, compact_double = False)
```

parameters:

- `buffer` — optional list buffer, which can be an instance of IRISList or a byte array in $LIST format (returned by Native SDK methods such as IRIS.**getBytes()**).
- `locale` — optional str indicating locale setting for buffer.
- `is_unicode` — optional bool indicating if buffer is Unicode.
- `compact_double` — optional bool indication if compact doubles are enabled.

Instances of IRISList can be created in the following ways:

- Create an empty IRISList

```
list = IRISList()
```

- Create a copy of another IRISList

```
listcopy = IRISList(myOtherIRISList)
```

- Construct an instance from a $LIST formatted byte array, such as that returned by IRIS.**getBytes()** and numerous other iris methods.

```
globalBytes = myIris.getBytes("myGlobal",1)
listFromByte = IRISList(globalBytes)
```

Many methods in the iris package return IRISList, which is one of the Native SDK supported types.

7.3.2 IRISList Method Details

add()

IRISList.**add()** appends a value to the end of the IRISList and returns the IRISList object.

```
add (value)
```

returns: self (the IRISList object)

parameters:

- `value` — a value of any supported type, or an array or collection of values. Each element of an array or collection will be appended individually. Instances of IRISList are always appended as a single element.

The **add()** method never concatenates two instances of IRISList. However, you can use IRISList.**toArray()** to convert an IRISList to an array. Calling **add()** on the resulting array will append each element separately.

clear()

IRISList.**clear()** resets the list by removing all elements from the list, and returns the IRISList object.

```
clear ()
```

returns: self (the IRISList object)

count()

IRISList.**count()** returns the number of data elements in the list.

```
count ()
```

returns: int

equals()

IRISList.**equals()** compares the specified *irislist2* with this instance of IRISList, and returns true if they are identical. To be equal, both lists must contain the same number of elements in the same order with identical serialized values.

```
equals (irislist2)
```

returns: bool

parameters:

- `irislist2` — instance of IRISList to compare.

get()

IRISList.**get()** moves the list cursor to the specified *index* (if one is specified) and returns the element at the cursor as an Object. Throws IndexOutOfBoundsException if the index is out of range (less then 1 or past the end of the list). Returns None if

```
get (index)
```

returns: bool, bytes, bytearray, int, float, Decimal, str, IRISList, Object, or None

parameters:

- `index` — optional int specifying the index (one-based) of the new cursor location

get() Typecast Methods

All of the IRISList.**get()** typecast methods listed below work exactly like IRISList.**get()**, but also cast the return value to a specific type. They all throw IndexOutOfBoundsException if the index is out of range (less then 1 or past the end of the list).

```
getBoolean (index)
getBytes (index)
getDecimal (index)
getFloat (index)
getInteger (index)
getIRISList (index)
getString (index)
```

returns: bool, bytes, bytearray, int, float, Decimal, str, IRISList, Object, or None

parameters:

- `index` — optional int specifying the index (one-based) of the new cursor location

remove()

IRISList.**remove()** removes the element at *index* from the list and returns the IRISList object.

```
remove (index)
```

returns: self (the IRISList object)

parameters:

- `index` — int specifying the index (one-based) of the element to remove

set()

IRISList.**set()** replaces the list element at *index* with *value* and returns the IRISList object.

If *value* is an array, each array element is inserted into the list, starting at *index*, and any existing list elements after *index* are shifted to make room for the new values.

If *index* is beyond the end of the list, *value* will be stored at *index* and the list will be padded with nulls up to that position. Throws IndexOutOfBoundsException if *index* is less than 1.

```
set (index, value)
```

returns: self (the IRISList object)

parameters:

- `index` — int specifying the index (one-based) of the list element to be set
- `value` — Object value or Object array to insert at *index*. Objects can be any supported type.

size()

IRISList.**size()** returns the byte length of the serialized value for this IRISList.

```
size ()
```

returns: int

7.4 Class iris.IRISConnection

Instances of IRISConnection are created by calling package method iris.**connect()**. See "iris Package Methods" for details about creating and using connections.

close()

IRISConnection.**close()** closes the connection to the iris instance if it is open. Does nothing if the connection is already closed.

```
connection.close()
```

returns: self

isClosed()

IRISConnection.**isClosed()** returns `True` if the connection was successful, or `False` otherwise.

```
connection.isClosed()
```

returns: bool

isUsingSharedMemory()

IRISConnection.**isUsingSharedMemory()** returns `True` if the connection is open and using shared memory.

```
connection.isUsingSharedMemory()
```

returns: bool

Properties

The *hostname*, *port*, *namespace*, *timeout*, and *logfile* from the last successful connection attempt are saved as properties of the connection object.

7.5 Class iris.IRISObject

Class iris.IRISObject provides methods to work with External Server inverse proxy objects (see "Controlling Database Objects from Python" for details and examples).

If the called method or function returns an object that is a valid OREF, an inverse proxy object (an instance of IRISObject) for the referenced object will be generated and returned. For example, **classMethodObject()** will return a proxy object for an object created by **%New()**.

7.5.1 IRISObject Constructor

The IRISObject constructor takes the following parameters:

```
IRISObject(connection, oref)
```

parameters:

- `connection` — an IRISConnection object
- `oref` — the OREF of the database object to be controlled.

7.5.2 IRISObject Method Details

close()

IRISObject.**close()** releases this instance of IRISObject.

```
close ()
```

returns: self

get()

IRISObject.**get()** fetches a property value of the proxy object. Returns None for IRIS empty string (`$$$NULLOREF`); otherwise, returns a variable of a type corresponding to the ObjectScript datatype of the property.

```
get (propertyName)
```

returns: bool, bytes, bytearray, int, float, Decimal, str, IRISList, Object, or None

parameters:

- `property_name` — name of the property to be returned.

get() Typecast Methods

All of the IRISObject.**get()** typecast methods listed below work exactly like IRISObject.**get()**, but also cast the return value to a specific type. They all return None for IRIS empty string (`$$$NULLOREF`), and otherwise return a value of the specified type.

```
getBoolean (propertyName)
getBytes (propertyName)
getDecimal (propertyName)
getFloat (propertyName)
getInteger (propertyName)
getIRISList (propertyName)
getString (propertyName)
getObject (propertyName)
```

returns: bool, bytes, bytearray, int, float, Decimal, str, IRISList, Object, or None

parameters:

- `property_name` — name of the property to be returned.

getConnection()

IRISObject.**getConnection()** returns the current connection as an IRISConnection object.

```
getConnection ()
```

returns: an IRISConnection object

getOREF()

IRISObject.**getOREF()** returns the OREF of the database object mapped to this IRISObject.

```
getOREF ()
```

returns: OREF

invoke()

IRISObject.**invoke()** invokes an instance method of the object, returning a variable of a type corresponding to the ObjectScript datatype of the property.

```
invoke (methodName, args)
```

returns: bool, bytes, bytearray, int, float, Decimal, str, IRISList, Object, or None

parameters:

- `method_name` — name of the instance method to be called.
- `*args` — zero or more arguments of supported types.

Also see **invokeVoid()**, which is similar to **invoke()** but does not return a value.

invoke() Typecast Methods

All of the IRISObject.**invoke()** typecast methods listed below work exactly like IRISObject.**invoke()**, but also cast the return value to a specific type. They all return None for IRIS empty string (`$$$NULLOREF`), and otherwise return a value of the specified type.

```
invokeBoolean (methodName, args)
invokeBytes (methodName, args)
invokeDecimal (methodName, args)
invokeFloat (methodName, args)
invokeInteger (methodName, args)
invokeIRISList (methodName, args)
invokeString (methodName, args)
invokeObject (methodName, args)
```

returns: bool, bytes, bytearray, int, float, Decimal, str, IRISList, Object, or None

parameters:

- `method_name` — name of the instance method to be called.
- `*args` — zero or more arguments of supported types.

Also see **invokeVoid()**, which is similar to **invoke()** but does not return a value.

invokeVoid()

IRISObject.**invokeVoid()** invokes an instance method of the object, but does not return a value.

```
invokeVoid (methodName, args)
```

parameters:

- `method_name` — name of the instance method to be called.
- `*args` — zero or more arguments of supported types.

set()

IRISObject.**set()** sets a property of the proxy object.

```
set (propertyName, propertyValue)
```

parameters:

- `property_name` — name of the property to which *value* will be assigned.
- `value` — property value to assign. Value can be any supported type.

7.6 Class iris.IRISReference

The iris.IRISReference class provides a way to use pass-by-reference arguments when calling database classmethods. See "Passing Arguments by Reference" for details and examples.

7.6.1 IRISReference Constructor

The IRISReference constructor takes the following parameters:

```
IRISReference(value, type = None)
```

parameters:

- `value` — initial value of the referenced argument.
- `type` — optional Python type used as a type hint for unmarshaling modified value of the argument. Supported types are bool, bytes, bytearray, Decimal, float, int, str, IRISList, or None. If None is specified, uses the type that matches the original database type

7.6.2 IRISReference Method Details

get_type() [deprecated]

IRISReference.**get_type()** is deprecated. Use one of the getValue() Typecast Methods to get a return value of the desired type.

```
get_type ()
```

get_value() [deprecated]

IRISReference.**get_value()** is deprecated. Use **getValue()** or the **getObject()** instead.

getValue()

IRISReference.**getValue()** returns the value of the referenced parameter as a type corresponding to the datatype of the database value.

```
getValue ()
```

returns: value of the referenced parameter

getValue() Typecast Methods

All of the IRISReference.**getValue()** typecast methods listed below work exactly like IRISReference.**getValue()**, but also cast the return value to a specific type.

```
getBoolean ()
getBytes ()
getDecimal ()
getFloat ()
getInteger ()
getIRISList ()
getString ()
getObject ()
```

returns: bool, bytes, bytearray, int, float, Decimal, str, IRISList, Object, or None

set_type() [deprecated]

IRISReference.**set_type()** is deprecated. There is an option to set the type when calling the IRISReference.

```
set_type (type)
```

setValue()

IRISReference.**setValue()** sets the value of the referenced argument.

```
setValue (value)
```

parameters:

- `value` — value of this IRISReference object

set_value() [deprecated]

IRISReference.**set_value()** is deprecated. Use **setValue()** instead.

7.7 Class iris.IRISGlobalNode

IRISGlobalNode provides an iterable interface that behaves like a virtual dictionary representing the immediate children of a global node. It is iterable, sliceable, reversible, and indexable, with support for views and membership tests.

- IRISGlobalNode supports an iterable interface:

  For example, the following `for` loop will list the subscripts of all child nodes:

  ```
  for x in node:
      print(x)
  ```

- IRISGlobalNode supports views:

Methods **keys()**, **subscripts()**, **values()**, **items()**, and **nodes()** return IRISGlobalNodeView view objects. They provide specific views on the IRISGlobalNode entries which can be iterated over to yield their respective data, and they support membership tests. For example, the **items()** method returns a view object containing a list of subscript-value pairs:

```
for sub, val in node.items():
    print('subscript:',sub,' value:', val)
```

- IRISGlobalNode is sliceable:

  Through standard Python slicing syntax, IRISGlobalNode can be iterated over a more restricted ranges of subscripts.

  ```
  node[start:stop:step]
  ```

  This results in a new IRISGlobalNode object with the subscript range limited to from start (inclusive) to stop (exclusive). step can be 1 or -1, meaning traversing in forward direction or in reversed direction.

- IRISGlobalNode is reversible:

  If the "step" variable is -1 in the slicing syntax, the IRISGlobalNode will iterate backwards - reversed from the standard order. For example, the following statement will traverse the subscripts from 8 (inclusive) to 2 (exclusive):

  ```
  for x in node[8:2:-1]: print(x)
  ```

- IRISGlobalNode is indexable and supports membership tests

  For example, given a child node with subscript `x`, the node value can be set with `node[x] = y` and returned with `z = node[x]`. A membership test with `x in node` will return a boolean.

7.7.1 IRISGlobalNode Constructor

Instances of IRISGlobalNode can be created by calls to IRIS.**node()**, IRISGlobalNode.**node()**, or IRISGlobalNode.**nodes()**.

7.7.2 IRISGlobalNode Method Details

get()

iris.IRISGlobalNode.**get()** returns the value of a node at the specified *subscript*. Returns *default_value* if the node is valueless or does not exist. .

```
get (subscript, default_value)
```

parameters:

- `subscript` — subscript of the child node containing the value to retrieve
- `default_value` — value to return if there is no value at the specified subscript

items()

iris.IRISGlobalNode.**items()** returns a view yielding subscript-value tuples for all child nodes of this node.

```
items ()
```

returns: IRISGlobalNodeView object yielding subscript-value tuples

keys()

iris.IRISGlobalNode.**keys()** returns a view yielding subscripts for all child nodes of this node. Identical to **subscripts()**.

```
keys ()
```

returns: IRISGlobalNodeView object yielding only subscripts.

node()

iris.IRISGlobalNode.**node()** returns an IRISGlobalNode object representing the child node at the specified subscript.

```
node (subscript)
```

returns: IRISGlobalNode object

parameters:

- `subscript` — subscript of the desired child node

nodes()

iris.IRISGlobalNode.**nodes()** returns a view yielding an IRISGlobalNode object for each child node of this node.

```
nodes ()
```

returns: IRISGlobalNodeView object yielding IRISGlobalNode objects

subscripts()

iris.IRISGlobalNode.**subscripts()** returns a view yielding subscripts for all child nodes of this node. Identical to **keys()**.

```
subscripts ()
```

returns: IRISGlobalNodeView object yielding only subscripts.

values()

iris.IRISGlobalNode.**values()** returns a view yielding values for all child nodes of this node. Returns None if a node does not have a value.

```
values ()
```

returns: IRISGlobalNodeView object yielding only values.

7.8 Class iris.IRISGlobalNodeView

Class iris.IRISGlobalNodeView implements view objects for IRISGlobalNode which can be iterated over to yield data from the child nodes. IRISGlobalNodeView objects are returned by IRISGlobalNode methods **keys()**, **subscripts()**, **values()**, **items()**, and **nodes()**.

7.9 Legacy Support Classes

These classes have been retained to allow legacy applications to run without alteration. They should never be used in new code, and legacy code should work without any changes.

7.9.1 Class iris.LegacyIterator [deprecated]

This class is included only for backward compatibility. Class iris.LegacyIterator allows code that was written with the deprecated irisnative.Iterator class to continue working without any changes.

New code should always use the iris.IRISGlobalNode class instead.

7.9.1.1 LegacyIterator Method Details

next() [deprecated]

iterator.**next()** positions the iterator at the next child node and returns the node value, the node subscript, or a tuple containing both, depending on the currently enabled return type (see below). Throws a StopIteration exception if there are no more nodes in the iteration.

```
iterator.next()
```

returns: node information in one of the following return types:

- `subscript and value (default)` — a tuple containing the subscript and value of the next node in the iteration. The subscript is the first element of the tuple and the value is the second. Enable this type by calling **items()** if the iterator is currently set to a different return type.
- `subscript only` — Enable this return type by calling **subscripts()**.
- `value only` — Enable this return type by calling **values()**.

startFrom() [deprecated]

iterator.**startFrom()** returns the iterator with its starting position set to the specified subscript. The iterator will not point to a node until you call **next()**, which will advance the iterator to the next child node after the position you specify.

```
iterator.startFrom(subscript)
```

returns: calling instance of iterator

parameter:

- `subscript` — a single subscript indicating a starting position.

Calling this method with `None` as the argument is the same as using the default starting position, which is just before the first node, or just after the last node, depending on the direction of iteration.

reversed() [deprecated]

iterator.**reversed()** returns the iterator with the direction of iteration reversed from its previous setting (direction is set to forward iteration when the iterator is created).

```
iterator.reversed()
```

returns: same instance of iterator

subscripts() [deprecated]

iterator.**subscripts()** returns the iterator with its return type set to subscripts-only.

```
iterator.subscripts()
```

returns: same instance of iterator

values() [deprecated]

iterator.**values()** returns the iterator with its return type set to values-only.

```
iterator.values()
```

returns: same instance of iterator

items() [deprecated]

iterator.**items()** returns the iterator with its return type set to a tuple containing both the subscript and the node value. The subscript is the first element of the tuple and the value is the second. This is the default setting when the iterator is created.

```
iterator.items()
```

returns: same instance of iterator

7.9.1.2 Legacy Iteration Example

In the following example, *iterDogs* is set to iterate over child nodes of global array *heroes('dogs')*. Since the **subscripts()** method is called when the iterator is created, each call to **next()** will return only the subscript for the current child node. Each subscript is appended to the *output* variable, and the entire list will be printed when the loop terminates. A StopIteration exception is thrown when there are no more child nodes in the sequence.

Use next() to list the subscripts under node heroes('dogs')

```
# Get a list of child subscripts under node heroes('dogs')
iterDogs = irispy.iterator('heroes','dogs').subscripts()
output = "\nSubscripts under node heroes('dogs'): "
try:
  while True: output += '%s ' % iterDogs.next()
except StopIteration:  # thrown when there are no more child nodes
  print(output + '\n')
```

This code prints the following output:

```
Subscripts under node heroes('dogs'): Balto Hachiko Lassie Whitefang
```

7.9.2 class irisnative.IRISNative [deprecated]

This class is included only for backward compatibility. Class irisnative.IRISNative allows code that was written with the deprecated irisnative connection methods to continue working without any changes.

The old **createConnection()** and **createIRIS()** methods are now exposed as iris package methods **connect()** and **createIRIS()** (see "iris"), which should always be used in new code.

createConnection() [deprecated]

Replace this method with iris package method iris.**connect()**.

IRISNative.**createConnection()** attempts to create a new connection to an IRIS instance. Returns a new connection object. The object will be open if the connection was successful, or closed otherwise).

```
irisnative.createConnection(hostname,port,namespace,username,password,timeout,sharedmemory,logfile)
   irisnative.createConnection(connectionstr,username,password,timeout,sharedmemory,logfile)
```

The *hostname*, *port*, *namespace*, *timeout*, and *logfile* from the last successful connection attempt are saved as properties of the connection object.

returns: a new instance of connection

parameters: Parameters may be passed by position or keyword.

- `hostname` — str specifying the server URL
- `port` — int specifying the superserver port number
- `namespace` — str specifying the namespace on the server
- The following parameter can be used in place of the `hostname`, `port`, and `namespace` arguments:
  - `connectionstr` — str of the form *`hostname:port/namespace`*.
- `username` — str specifying the user name
- `password` — str specifying the password
- `timeout` (optional) — int specifying maximum number of milliseconds to wait while attempting the connection. Defaults to 10000.
- `sharedmemory` (optional) — specify bool `True` to attempt a shared memory connection when the hostname is `localhost` or `127.0.0.1`. Specify `False` to force a connection over TCP/IP. Defaults to `True`.
- `logfile` (optional) — str specifying the client-side log file path. The maximum path length is 255 ASCII characters.

createIris() [deprecated]

Replace this method with iris package method iris.**createIRIS()**.

IRISNative.**createIris()** returns a new instance of IRIS that uses the specified connection. Throws an exception if the connection is closed.

```
irisnative.createIris(conn)
```

returns: a new instance of IRIS

parameter:

- `conn` — object that provides the server connection