InterSystems™
IRIS Data Platform

Using the Native API for Python

Version 2020.3
2021-02-04

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

Table of Contents

About This Book

InterSystems IRIS® provides a lightweight *Native API for Python* for direct access to the native data structure of InterSystems databases. *Globals* are the tree-based sparse arrays used to implement the InterSystems multidimensional storage model. These native data structures provide very fast, flexible storage and retrieval. InterSystems IRIS uses globals to make data available as objects or relational tables, but you can use the Native API to implement your own data structures.

The following chapters discuss the main features of the Native API:

- Introduction to the Native API — demonstrates how to create an instance of the main Native API class, open a connection, and perform some simple database operations.
- Working with Global Arrays — describes how to create, change, or delete nodes in a multidimensional global array, and demonstrates methods for iteration, transactions, and locking.
- Calling ObjectScript Methods and Functions — describes a set of methods that allow an application to call user defined ObjectScript class methods and functions on the server.
- Native API Quick Reference — provides a brief description of each method in the Native API.

There is also a detailed Table of Contents.

More information about globals

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

- *Using the InterSystems Native API for Java*
- *Using the InterSystems Native API for .NET*
- *Using the InterSystems Native API for Node.js*

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

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

1

Introduction to the Native API

The *Native API for Python* is a lightweight interface to the native multidimensional storage data structures that underlie the InterSystems IRIS® object and SQL interfaces. The Native API allows you to implement your own data structures by providing direct access to *global arrays*, the tree-based sparse arrays that form the basis of the multidimensional storage model.

This chapter discusses the following topics:

- Introduction to Global Arrays — introduces global array concepts and provides a simple demonstration of how the Native API is used.
- Glossary of Native API Terms — defines some important terms used in this book.
- Global Naming Rules — lists the rules for naming global arrays and subscripts.

1.1 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*. In InterSystems ObjectScript globals notation, the two nodes with values would be:

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

The global name (`root`) is followed by a comma-delimited *subscript list* in parentheses. Together, they specify the entire path to the node.

This global array could be created by two calls to the Native API **set()** method:

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

Global array `root` is created when 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. For details, see "Creating, Updating, and Deleting Nodes" in the next chapter.

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

The NativeDemo Program

```
# Import the Native API module
import irisnative

# Open a connection to the server
args = {'hostname':'127.0.0.1',
        'port':51773,
        'namespace':'USER',
        'username':'_SYSTEM',
        'password':'SYS'}
conn = irisnative.createConnection(**args)

# Create an iris object
iris = irisnative.createIris(conn)

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

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

# Delete the global array and terminate
iris.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, *irisnative* is the Native API module. Native API methods perform the following actions:

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

The next chapter provides detailed explanations and examples for all of these methods.

1.2 Glossary of Native API 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*.

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 data, 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, including 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).

Target address

Many Native API 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 any supported type. A node with no child nodes must contain a value; a node that has child nodes can be valueless.

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.

1.3 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" in *Using Globals*).
- 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 a Unicode string, Integer/Long, Float, or Byte string. String subscripts are case-sensitive, and can include characters of all types. Length is limited only by the 511 character maximum for the total node address.

2
Working with Global Arrays

This chapter covers the following topics:

- Creating, Updating, and Deleting Nodes — describes how to create a global array and set or change node values.
- Finding Nodes in a Global Array — describes iteration methods that allow rapid access to the nodes of a global array.

Note: The examples in this chapter assume that an iris object named *iris* already exists and is connected to the server. The following code was used to create it:

```
import irisnative
conn = irisnative.createConnection('127.0.0.1', 51773, 'USER', '_SYSTEM', 'SYS')
iris = irisnative.createIris(conn)
```

For more information, see the Quick Reference entries for **createConnection()** and **createIris()**.

2.1 Creating, Updating, and Deleting Nodes

This section describes the Native API methods used to create, update, and delete nodes. **set()**, **increment()**, and **kill()** are the only methods that can create a global array or alter its contents. The following examples demonstrate how to use each of these methods.

Setting and changing node values

Iris.**set()** takes a *value* argument and stores the value at the specified address.

If no node exists at that address, a new one is created.

The **set()** method can assign values of any supported datatype. 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`.

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

set() can create and change values of any supported datatype, as demonstrated in this example.

Incrementing node values

Iris.**increment()** takes an integer *number* argument, increments the node value by that amount, and returns the incremented value. The initial target node value can be any supported numeric type, but the incremented value will be an integer. If there is no node at the target address, the method creates one and assigns the *number* argument as the value. This method uses a thread-safe atomic operation to change the value of the node, so the node is never locked.

In the following example, the first call to **increment()** creates new subnode *myGlobal('B')* with value `-2`, and assigns the returned value to *total*. The next two calls each increment by `-2` and assign the new value to *total*, and the loop exits when the node value is `-6`.

```
done = False
while not done:
  total = iris.increment(-2,'myGlobal','B')
  if (total <= -6): done = True
print('total = ', total, " and myGlobal('B') = ", iris.get('myGlobal','B'))
# Prints: total = -6 and myGlobal('B') = -6
```

Note: ***Naming rules***

A global name can include letters, numbers, and periods (`'.'`). The name must begin with a character and may not end with a period. A subscript can be an Integer/Long, Float, Byte string, or Unicode string (case-sensitive, not restricted to alphanumeric characters). See "Global Naming Rules" for more information.

Deleting a node or group of nodes

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:

```
iris.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:

```
iris.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.

2.2 Finding Nodes in a Global Array

The Native API provides ways to iterate over part or all of a global array. The following topics describe the various iteration methods:

- Iterating Over a Set of Child Nodes — describes how to iterate over all child nodes under a given parent node.

- Iteration with next() — describes methods that provide more control over iteration.
- Testing for Child Nodes and Node Values — describes how to find all subnodes regardless of node level, and identify which nodes have values.

2.2.1 Iterating Over a Set of Child Nodes

Child nodes are sets of nodes immediately under the same parent node. Any child node address can be defined by appending one subscript to the subscript list of the parent. For example, the following global array has four child nodes under parent node *heroes('dogs')*:

The heroes global array

This global array uses the names of several heroic dogs (plus a reckless boy and a pioneering sheep) as subscripts. The values are birth years.

```
heroes                                           // root node,    valueless, 2 child nodes
   heroes('dogs')                                // level 1 node, valueless, 4 child nodes
      heroes('dogs','Balto') = 1919              // level 2 node, value=1919
      heroes('dogs','Hachiko') = 1923            // level 2 node, value=1923
      heroes('dogs','Lassie') = 1940             // level 2 node, value=1940, 1 child node
         heroes('dogs','Lassie','Timmy') = 1954  // level 3 node, value=1954
      heroes('dogs','Whitefang') = 1906          // level 2 node, value=1906
   heroes('sheep')                               // level 2 node, valueless, 1 child node
      heroes('sheep','Dolly') = 1996             // level 2 node, value=1996
```

The following methods are used to create an iterator, define the direction of iteration, and set the starting point of the search:

- Iris.**iterator()** returns an instance of Iterator for the child nodes of the specified target node.
- Iterator.**reversed()** — toggles direction of iteration between forward and reverse collation order.
- Iterator.**startFrom()** sets the iterator's starting position to the specified subscript. The subscript is an arbitrary starting point, and does not have to address an existing node.

Read child node values in reverse order

The following code iterates over child nodes of *heroes('dogs')* in reverse collation order, starting with subscript V:

```
# Create an iterator for child nodes of heroes('dogs')
  iterDogs = iris.iterator('heroes','dogs').reversed().startFrom('V')
# Iterate in reverse collation order, starting at address heroes('dogs','V')
  output = '\nDog birth years: '
  for key,value in iterDogs.items():
    output +=  '%s:%i  ' % (key, value)
  print(output)
```

This code prints the following output:

```
Dog birth years: Lassie:1940  Hachiko:1923  Balto:1919
```

In this example, two subnodes of *heroes('dogs')* are ignored:

- Child node *heroes('dogs','Whitefang')* will not be found because it is outside of the search range (`Whitefang` is higher than V in collation order).
- Level 3 node *heroes('dogs','Lassie','Timmy')* will not be found because it is a child of `Lassie`, not `dogs`.

See the last section in this chapter (“Testing for Child Nodes and Node Values”) for a discussion of how to iterate over multiple node levels.

Note: **Collation Order**

The order in which nodes are retrieved depends on the *collation order* of the subscripts. When a node is created, it is automatically stored it in the collation order specified by the storage definition. In this example, the child nodes of *heroes('dogs')* would be stored in the order shown (`Balto`, `Hachiko`, `Lassie`, `Whitefang`) regardless of the order in which they were created. For more information, see "Collation of Global Nodes" in *Using Globals*.

2.2.2 Iteration with next()

The Native API also supports the **next()** method and return type iterator methods:

- Iterator.**next()** — positions the iterator at the next child node (if one exists) and returns 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.
- Iterator.**items()** — sets return type to an array containing both the subscript and the value of the child node. For example, the returned value for node *heroes(,'dogs','Balto')* would be `['Balto',1919]`.
- Iterator.**subscripts()** — sets return type to return only the subscript.
- Iterator.**values()** — sets return type to return only the node value.

In the following example, iterDogs is set to iterate over child nodes of *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 = iris.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
```

2.2.3 Testing for Child Nodes and Node Values

In the previous examples, the scope of the search is restricted to child nodes of *heroes('dogs')*. The iterator fails to find two values in global array heroes because they are under different parents:

- Level 3 node *heroes('dogs','Lassie','Timmy')* will not be found because it is a child of `Lassie`, not `dogs`.
- Level 2 node *heroes('sheep','Dolly')* is not found because it is a child of `sheep`, not `dogs`.

To search the entire global array, we need to find all of the nodes that have child nodes, and create an iterator for each set of child nodes. The **isDefined()** method provides the necessary information:

- Iris.**isDefined()** — can be used to determine if a 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 = (iris.isDefined(root,subscripts) > 0)       # returned 1, 10, or 11
hasValue = (iris.isDefined(root,subscripts)%10 > 0)  # returned 1 or 11
hasChild = (iris.isDefined(root,subscripts) > 9)     # returned 10 or 11
```

The following example consists of two methods:

- **findAllHeroes()** iterates over child nodes of the current node, and calls **testNode()** for each node. Whenever **testNode()** indicates that the current node has child nodes, **findAllHeroes()** creates a new iterator for the next level of child nodes.
- **testNode()** will be called for each node in the *heroes* global array. It calls **isDefined()** on the current node, and returns a boolean value indicating whether the node has child nodes. It also prints node information for each node.

Method findAllHeroes()

This example processes a known structure, and traverses the various levels with simple nested calls. In the less common case where a structure has an arbitrary number of levels, a recursive algorithm could be used.

```
def findLostHeroes():
  root = 'heroes'
  iterRoot = iris.iterator(root)
  hasChild = False

  # Iterate over children of root node heroes
  for sub1,value in iterRoot:
    hasChild = testNode(value,root,*[sub1])

    # Iterate over children of heroes(sub1)
    if hasChild:
      iterOne = iris.iterator(root,sub1)
      for sub2,value in iterOne:
        hasChild = testNode(value,root,*[sub1,sub2])

      # Iterate over children of heroes(sub1,sub2)
        if hasChild:
          iterTwo = iris.iterator(root,sub1,sub2)
          for sub3,value in iterTwo:
            testNode(value,root,*[sub1,sub2,sub3]) #no child nodes below level 3
```

Method testNode()

```
def testNode(value, root, *subs):
  # Test for values and child nodes
  state = iris.isDefined(root,*subs)
  hasValue = (state%10 > 0) # has value if state is 1 or 11
  hasChild = (state > 9)    # has child if state is 10 or 11

  # format the node address output string
  level = len(subs)-1
  address = "  %s%s('%s')" % ("  "*level, root, "','".join(subs))

  # Add node value and note special cases
  if (hasValue):  # ignore valueless nodes
    address += ' = %i' % value
    for name in ['Timmy','Dolly']:
      if name == subs[level]:
        address += ' (not a dog!)'
  print(address)
  return hasChild
```

This method will write the following lines:

```
  heroes('dogs')
    heroes('dogs','Balto') = 1919
    heroes('dogs','Hachiko') = 1923
    heroes('dogs','Lassie') = 1940
      heroes('dogs','Lassie','Timmy') = 1954 (not a dog!)
    heroes('dogs','Whitefang') = 1906
  heroes('sheep')
    heroes('sheep','Dolly') = 1996 (not a dog!)
```

The output of **testNodes()** includes some nodes that were not found in previous examples because they are not child nodes of *heroes('dogs')*:

- *heroes('dogs','Lassie','Timmy')* is a child of `Lassie`, not `dogs`.
- *heroes('sheep','Dolly')* is a child of `sheep`, not `dogs`.

3

Calling ObjectScript Methods and Functions

This chapter describes methods of class iris that allow an application to call class methods and functions from user defined ObjectScript classes and routines.

- Class Method Calls — demonstrates how to call ObjectScript class methods.
- Function Calls — demonstrates how to call ObjectScript functions and procedures.
- Sample ObjectScript Methods and Functions — lists the ObjectScript methods and functions called by these examples.

Note: Although these methods can also be used with InterSystems classes defined in the Class Library, best practice is to call them indirectly, from within a user defined class or routine.

Many class methods return only a status code, passing the actual results back in an argument (which cannot be accessed by the Native API). System defined functions (listed under ObjectScript Functions in the *ObjectScript Reference*) cannot be called directly.

3.1 Class Method Calls

The following Native API methods call a specified ObjectScript class method.

- iris.**ClassMethodValue()** — calls a class method and gets the return value.
- iris.**ClassMethodVoid()** — calls a class method and discards any returned value.

These methods take string arguments for *className* and *methodName*, plus `0` or more method arguments. Trailing arguments may be omitted in argument lists, causing default values to be used for those arguments, either by passing fewer than the full number of arguments, or by passing `null` for trailing arguments. An exception will be thrown if a non-null argument is passed to the right of a null argument.

The code in the following example calls class methods of several datatypes from an ObjectScript test class named User.NativeTest. (For a listing, see "Sample ObjectScript Methods and Functions" at the end of this chapter).

Calling class methods of User.NativeTest

In this example, assume that *iris* is an existing instance of class iris, and is currently connected to the server.

```
className = "User.NativeTest"
cmValue = ""

cmValue = iris.classMethodValue(className,"cmBoolean",False)
print(className,".cmBoolean() returned value: ", cmValue)

cmValue = iris.classMethodValue(className,"cmBytes","byteArray")
print(className,".cmBytes() returned value: ", cmValue)

cmValue = iris.classMethodValue(className,"cmString","Test String")
print(className,".cmString() returned value: ", cmValue)

cmValue = iris.classMethodValue(className,"cmLong",7,8)
print(className,".cmLong() returned value: ", cmValue)

cmValue = iris.classMethodValue(className,"cmDouble",7.56)
print(className,".cmDouble() returned value: ", cmValue)

try:
  iris.classMethodVoid(className,"cmVoid",67)
  cmValue = iris.get("cm1")
except:
  cmValue = "method failed."
print(className,".cmVoid() set global array cm1 to value: ", cmValue)
```

3.2 Function Calls

The following methods call user-defined ObjectScript functions or procedures (see “Callable User-defined Code Modules” in *Using ObjectScript*).

- iris.**Function()** — calls a user-defined function and gets the return value.
- iris.**Procedure()** calls a user-defined procedure or function and discards any returned value.

These methods take string arguments for *functionName* and *routineName*, plus `0` or more function arguments. Trailing arguments may be omitted in argument lists, causing default values to be used for those arguments, either by passing fewer than the full number of arguments, or by passing `null` for trailing arguments. An exception will be thrown if a non-null argument is passed to the right of a null argument.

The code in the following example calls functions of several datatypes from an ObjectScript test routine named NativeRoutine. (For a listing of NativeRoutine.mac, see “Sample ObjectScript Methods and Functions” at the end of this chapter).

Calling functions of NativeRoutine.mac

In this example, assume that *iris* is an existing instance of class iris, and is currently connected to the server.

```
routineName = 'NativeRoutine'
fnValue = ""

fnValue = iris.function('funcBoolean',routineName,False)
print(routineName, '.funcBoolean() returned value: ', fnValue)

fnValue = iris.function('funcBytes',routineName,'byteArray')
print(routineName, '.funcBytes() returned value: ', fnValue)

fnValue = iris.function("funcString",routineName,"Test String")
print(routineName, '.funcString() returned value: ', fnValue)

fnValue = iris.function('funcLong',routineName,7,8)
print(routineName, '.funcLong() returned value: ', fnValue)

fnValue = iris.function('funcDouble',routineName,7.56)
print(routineName, '.funcDouble() returned value: ', fnValue)

try:
  iris.procedure("funcProcedure",routineName,67)
  fnValue = iris.get("fn1")
except:
  fnValue = 'procedure failed.'
print(routineName, '.funcVoid() set global array fn1 to value: ', fnValue)
```

3.3 Sample ObjectScript Methods and Functions

To run the examples, these ObjectScript class methods and functions must be compiled and available on the server:

ObjectScript Class User.NativeTest

```
Class User.NativeTest Extends %Persistent
{
  ClassMethod cmBoolean(cm1 As %Boolean) As %Boolean
  {
   quit 0
  }
  ClassMethod cmBytes(cm1 As %String) As %Binary
  {
   quit $C(65,66,67,68,69,70,71,72,73,74)
  }
  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 * 100
  }
  ClassMethod cmVoid(cm1 As %Integer)
  {
   set ^cm1=cm1
   quit
  }
}
```

ObjectScript Routine NativeRoutine.mac

```
funcBoolean(fn1) public {
    quit 0
}
funcBytes(fn1) public {
    quit $C(65,66,67,68,69,70,71,72,73,74)
}
funcString(fn1) public {
    quit "Hello "_fn1
}
funcLong(fn1,fn2) public {
    quit fn1+fn2
}
funcDouble(fn1) public {
    quit fn1 * 100
}
funcProcedure(fn1) public {
    set ^fn1=fn1
    quit
}
```

4 Native API Quick Reference for Python

This chapter is a quick reference for module irisnative, which contains the following classes:

- Class connection provides a connection to the server.
- Class iris provides the main functionality.
- Class iterator provides methods to navigate a global array.

Note: This chapter is intended as a convenience for readers of this book, but it is not the definitive reference for the Native API. For the most complete and up-to-date information on these classes, see the online API documentation.

4.1 List of Methods by Usage

Module irisnative

The irisnative module provides classes iris, connection, and iterator, plus the following methods:

- irisnative.**createConnection()** — returns a new open server connection.
- irisnative.**createIris()** — returns a new iris object that uses the specified connection.

Class Connection

The connection class encapsulates a connection to the server. Instances of connection are created and connected to the server by irisnative method **createConnection()**.

- connection.**close()** — closes the connection.
- connection.**isClosed()** — returns `True` if the connection is closed.
- connection.**isUsingSharedMemory()** — returns `True` if the connection is using shared memory.

Class Iris

The iris class provides most of the Native API functionality. Instances of iris are created by irisnative.**createIris()**. Methods are listed below, organized by usage:

Class iris: Create, access, and delete nodes

- iris.**get()** — returns the specified node value.
- iris.**getBoolean()** — returns a node value as a Boolean.

- iris.**getBytes()** — returns a node value as a Byte string.
- iris.**getLong()** — returns a node value as an Integer/Long.
- iris.**getFloat()** — returns a node value as a Float.
- iris.**getString()** — returns a node value as a Unicode string.
- iris.**increment()** — increments the value of a global node by the specified amount.
- iris.**kill()** — deletes the global node including any descendants.
- iris.**set()** — sets a node to a value, creating a new node if necessary.

Class iris: Transactions and locking

- iris.**lock()** — performs an incremental lock on the global, returns `True` on success.
- iris.**unlock()** — performs an immediate or deferred unlock on the global.
- iris.**releaseAllLocks()** — releases all locks associated with the session.
- iris.**getTLevel()** — returns the current transaction level (`0` if not in a transaction).
- iris.**tCommit()** — commits the current transaction and decrements the transaction level.
- iris.**tRollback()** — rolls back all open transactions in the session.
- iris.**tRollbackOne()** — rolls back the current level transaction only.
- iris.**tStart()** — starts a transaction and increments the transaction level.

Class iris: ClassMethod and function calls

- iris.**classMethodValue()** — calls a user defined ObjectScript method and gets the returned value.
- iris.**classMethodVoid()** — calls a user defined ObjectScript method, ignoring any returned value.
- iris.**function()** — calls a function of a user defined ObjectScript routine and gets the returned value.
- iris.**procedure()** — calls a procedure of a user defined ObjectScript routine.

Class iris: Global iteration and utilities

- iris.**getClientVersion()** — returns the version string for this version of the Native API.
- iris.**getServerVersion()** — returns the version string for the currently connected server.
- iris.**isDefined()** — tests whether a global node exists, and if it contains data.
- iris.**iterator()** — returns an instance of iterator.

Class Iterator

The iterator class provides methods to iterate over a set of nodes. Instances of iterator are created by iris.**iterator()**.

- iterator.**next()** — returns a tuple containing the subscript and value of the next node in the iteration.
- iterator.**startFrom()** — sets the iterator to the specified starting address and returns it, after which **next()** can be used to advance the iterator to the next node in collating sequence.
- iterator.**reversed()** — toggles direction of iteration between forward and reverse collation order.
- iterator.**items()** — sets return type to an array containing subscript and node value.
- iterator.**subscripts()** — sets return type to return the node subscript only.

- iterator.**values()** — sets return type to return the node value only.

4.2 Module irisnative

The irisnative module provides classes iris, connection, and iterator, plus methods to create new instances of iris and connection. See "Introduction to Global Arrays" for examples.

createConnection()

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 (see connection.**isClosed()**).

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

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` — (connection object) — object that provides the server connection

4.3 Class connection

Class connection is a member of module irisnative. Instances of connection are created by calling irisnative.**createConnection()**. See “Introduction to Global Arrays” for more details and examples.

close()

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

```
connection.close()
```

isClosed()

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

```
connection.isClosed()
```

returns: Boolean

isUsingSharedMemory()

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

```
connection.isUsingSharedMemory()
```

returns: Boolean

Properties

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

4.4 Class iris

Class iris is a member of module irisnative. Instances of iris are created by calling irisnative.**createIris()**. See “Introduction to Global Arrays” for more details and examples.

classMethodValue()

iris.**classMethodValue()** calls a class method, passing zero or more arguments and returning the value returned by the class method. Throws a `<COMMAND>` exception if no value is returned.

```
iris.classMethodValue(className,methodName,*args)
```

returns: None, Integer/Long, Unicode string, or Float (depending on the result of the class method)

parameters:

- `className` — fully qualified name of the class to which the called method belongs.
- `methodName` — name of the class method.
- `args` — zero or more method arguments of types None, Integer/Long, Unicode string, Float, or Byte string. Trailing arguments may be omitted (see “Calling ObjectScript Methods and Functions”).

classMethodVoid()

iris.**classMethodVoid()** calls a class method, passing zero or more arguments. This method will ignore any values returned by the class method.

```
iris.classMethodVoid(className,methodName,*args)
```

parameters:

- `className` — fully qualified name of the class to which the called method belongs.
- `methodName` — name of the class method.
- `*args` — zero or more method arguments of types Integer/Long, Unicode string, Float, Byte string, or None. Trailing arguments may be omitted (see "Calling ObjectScript Methods and Functions").

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 just be ignored.

function()

iris.**function()** calls a function, passing zero or more arguments and returning the value returned by the function (also see iris.**procedure()**).

```
iris.function(routineName,functionName,*args)
```

returns: None, Integer/Long, Unicode string, or Float (depending on the result of the function)

parameters:

- `routineName` — name of the routine containing the function.
- `functionName` — name of the function to call.
- `*args` — zero or more function arguments of types None, Integer/Long, Unicode string, Float, or Byte string. Trailing arguments may be omitted (see "Calling ObjectScript Methods and Functions").

iterator()

iris.**iterator()** returns an iterator that traverses the immediate children of the specified node. See "Class iterator" for a list of methods that can modify the direction of iteration and the type of return value.

```
iris.iterator(globalName,*subscripts)
```

returns: an instance of iterator

parameters:

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

get()

iris.**get()** returns the value of the global node as Integer/Long, Unicode string, or Float.

```
iris.get(globalName,*subscripts)
```

returns: Integer/Long, Unicode string, or Float. Returns None if the node is valueless or does not exist.

parameters:

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

getBoolean()

iris.**getBoolean()** gets the value of the node as a Boolean.

```
iris.getBoolean(globalName,*subscripts)
```

returns: Boolean (or None if node is valueless or does not exist)

parameters:

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

getBytes()

iris.**getBytes()** gets the value of the node as a Byte string. Throws an exception if the node value is not a byte or ASCII string.

```
iris.getBytes(globalName,*subscripts)
```

returns: Byte string (or `b''` if empty string, None if node does not exist)

parameters:

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

getClientVersion()

iris.**getClientVersion()** returns the version string for this version of the Native API.

```
iris.getClientVersion()
```

returns: Unicode string

getFloat()

iris.**getFloat()** gets the value of the node as a Float. Returns `0.0` if node value is an empty string, or None if the node does not exist.

```
iris.getFloat(globalName,*subscripts)
```

returns: Float (or `0.0` node is valueless, None if node does not exist)

parameters:

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

getLong()

iris.**getLong()** gets the value of the node as a Long. Returns `0` if node value is an empty string, or None if the node does not exist.

```
iris.getLong(globalName,*subscripts)
```

returns: Long (or `0` if node is valueless, None if node does not exist)

parameters:

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

getServerVersion()

iris.**getServerVersion()** returns the version string for the currently connected server.

```
iris.getServerVersion()
```

returns: Unicode string

getString()

iris.**getString()** gets the value of the global as a Unicode string. Returns None if the node is valueless or does not exist.

```
iris.getString(globalName,*subscripts)
```

returns: Unicode string (or None if node is valueless or does not exist)

parameters:

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

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.

```
iris.getTLevel()
```

returns: Integer/Long

increment()

iris.**increment()** increments the global node with the *value* argument. If there is no existing node at the specified address, a new node is created with the specified value. Returns the new value of the global node.

```
iris.increment(value,globalName,*subscripts)
```

returns: Integer/Long or Float

parameters:

- `value` — (Integer/Long or Float) amount to increment by
- `globalName` — global name
- `*subscripts` — zero or more subscripts specifying the target node

isDefined()

iris.**isDefined()** returns a value that indicates whether a node has a value, a child node, or both. See “Testing for Child Nodes and Node Values” for details and examples.

```
iris.isDefined(globalName,*subscripts)
```

returns: one of the following Integer/Long values:

- 0 if the node does not exist.
- 1 if the node has a value but no descendants.
- 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:

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

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.

```
iris.kill(globalName,*subscripts)
```

parameters:

- `globalName` — 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 "LOCK" in the ObjectScript Reference for detailed information on locks.

```
iris.lock(lockMode,timeout,globalName,*subscripts)
```

parameters:

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

procedure()

iris.**procedure()** calls an ObjectScript procedure or function, passing zero or more arguments and returning nothing (also see iris.**function()**).

```
iris.procedure(routineName,procedureName,*args)
```

parameters:

- `routineName` — name of the routine containing the procedure.
- `procedureName` — name of the procedure to call.

- `*args` — zero or more function arguments of types None, Integer/Long, Unicode string, Float, or Byte string. Trailing arguments may be omitted (see "Calling ObjectScript Methods and Functions").

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 just be ignored.

releaseAllLocks()

iris.**releaseAllLocks()** releases all locks associated with the session.

```
iris.releaseAllLocks()
```

set()

iris.**set()** assigns *value* as the current node value. The new value may be a Unicode string, Byte string, Float, Integer/Long, or None.

```
  iris.set(value,globalName,*subscripts)
```

parameters:

- `value` — new value of the global node (Integer/Long, Unicode string, byte string, Float, or None)
- `globalName` — global name
- `*subscripts` — zero or more subscripts specifying the target node

tCommit()

iris.**tCommit()** commits the current transaction.

```
   iris.tCommit()
```

tRollback()

iris.**tRollback()** rolls back all open transactions in the session.

```
   iris.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.

```
   iris.tRollbackOne()
```

tStart()

iris.**tStart()** starts or opens a transaction.

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

```
   iris.unlock(lockMode,globalName,*subscripts)
```

parameters:

- `lockMode` — a string containing a combination of 0 or more the following characters:
  - `"I"` for immediate unlock, `"D"` for deferred unlock, or `""` for neither.
  - `"S"` to unlock a shared lock, `"E"` to unlock an escalating lock, or `"SE"` for both.

  For example, to unlock a Shared, Escalating lock Immediately, you would specify the string `"SEI"` as the *lockMode*. The characters may be in any order.

  An empty string is the default mode (unshared, non-escalating, always defers releasing an unlocked lock to the end of the current transaction).
- `globalName` — global name
- `*subscripts` — zero or more subscripts specifying the target node

4.5 Class iterator

Class iterator is a member of module irisnative. Instances of iterator are created by calling iris.**iterator()**. See “Finding Nodes in a Global Array” for more details and examples.

Note: All iterator methods except **next()** return the calling iterator (not a copy) to enable method chaining. For example, the following code would return an iterator for children of node *myGlobal('parentNode')*. It will iterate in reverse collation order, and is positioned to find the first child node after subscript `"V"`:

```
iter = iris.iterator('myGlobal','parentNode').reversed().startFrom('V')
```

next()

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. See “Iteration with next()” for more details and examples.

```
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()

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()

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()

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

```
iterator.subscripts()
```

returns: same instance of iterator

values()

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

```
iterator.values()
```

returns: same instance of iterator

items()

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