

Article

[Robert Cemper](#) · Jun 20 7m read

## Working with Globals in Embedded Python

My major interest is Working with Globals in Embedded Python.  
So I checked the available official documentation.

### #1 [Introduction to Globals](#)

an attempt of a generic description of what a global is. Pointing to

### #2 [A Closer Look at ObjectScript](#)

But where is Embedded Python ?

Way down you see

### #3 Embedded Python

#### 3.1 [Embedded Python Overview](#)

##### 3.1.1 [Work with Globals](#)

Great if you have never seen a Global before

Otherwise a shocking primitive example

#### 3.2 [Using Embedded Python](#)

Last hope: >>> but there is just NOTHING visible.

This is more than just disappointing! Even IRIS Native API for Python is more detailed.

To be clear about what I expect:

## SET, GET, KILL of a Global node

[Native API: Fundamental Node Operations](#) and

## Navigation with \$DATA(), \$ORDER(), \$QUERY()

[Native API: Iteration with nextSubscript\(\) and isDefined\(\)](#)

So I had to investigate, reverse engineer it and experiment myself.

## And these are my findings:

All examples are shown in Python Shell as found in IRIS for Windows (x86-64) 2022.1 (Build 209U)  
making intensive use of the implicit print() function.

## The Global

Whatever you plan to do you need to start with the class `iris.gref` to create a reference object for the Global.

The Global name is passed as string directly or as variable similar to Indirection in COS/ISOS.

The initial caret (^) is not required as it is clear that we just deal with Globals !

```
>>> globalname='rcc'  
>>> nglob=iris.gref(globalname)  
>>> glob=iris.gref('rcc')  
>>> cglob=iris.gref('^rcc')
```

These are 3 Global references to the same Global.  
Just a reference but no indication of this global exists

Interactive doc: `print(glob.__doc__)`

InterSystems IRIS global reference object.

Use the `iris.gref()` method to obtain a reference to a global

## SUBSCRIPTS

Any global subscript is passed as a Py list [sub1,sub2]. No big difference to COS/ISOS  
Just the top-level needs special treatment.

To signal No Subscript it is not an empty list but this [None]

## SET

To set a Global we may do it 'directly' as we would in COS/ISOS.

```
>>> glob[1,1]=11
```

or use method `gref.set()`

```
>>> glob.set([1,3],13)
```

Interactive doc: `print(glob.set.__doc__)`

Given the keys of a global, sets the value stored at that key of the global.

Example: `g.set([i,j], 10)` sets the value of the node at key `i,j` of global `g` to 10

To access the content of a Global node we may do it 'directly' as we would in COS/ISOS.

```
>>> glob[1,3]  
13
```

or use method `gref.get()`

```
>>> glob.get([1,1])  
11
```

Interactive doc: `print(glob.get.__doc__)`

Given the keys of a global, returns the value stored at that node of the global.

Example: `x = g.get([i,j])` sets `x` to the value stored at key `i,j` of global `g`.

Attention: This is NOT `$GET()` as you may know from COS/ISOS

```
>>> glob.get([1,99])  
Traceback (most recent call last):  
File "<input>", line 1, in <module>  
KeyError: 'Global Undefined'  
>>>
```

But using it directly it acts as \$GET() in COS/ISOS

```
>>> x=glob[1,99]
>>> print(x)
None
>>>
```

This None signals what SQL expresses as NULL. It will show up later again.

## KILL

There is only the method `glob.kill()` to achieve the expected result.

```
>>> glob.kill([1,3])
>>> y=glob[1,3]
>>> print(y)
None
>>>
```

Interactive doc: `print(glob.kill.__doc__)`

Given the keys of a global, kills that node of the global and its subtree.

Example: `g.kill([i,j])` kills the node stored at key `i,j` of global `g` and any descendants.

## \$DATA()

The related method is `glob.data()`

Interactive doc: `print(glob.data.__doc__)`

Given the keys of a global, returns the state of that.

Example: `x = g.data([i,j])` sets `x` to 0,1,10,11

0-if undefined, 1-defined, 10-undefined but has descendants, 11-has value and descendants

It works as expected.

```
>>> glob.data()
10
>>> glob.data([None])
10
>>> glob[None]=9
>>> glob.data([None])
11
>>> glob.data([1,1])
1
>>> glob.data([1,3])
0
>>>
```

## \$ORDER()

For this example I have added a few nodes to the Global `^rcc`:

```
>zw ^rcc
^rcc=9
^rcc(1,1)=11
```

```
^rcc(1,2)=12
^rcc(2,3,4)=234
^rcc(2,3,5)=235
^rcc(2,4,4)=244
^rcc(7)=7
```

The related method is `gref.order()`

[Interactive doc](#): `print(glob.order.__doc__)`

Given the keys of a global, returns the next key of the global.

Example: `j = g.order([i,j])` sets `j` to the next second-level key of global `g`.

So we see:

```
>>> print(glob.order([]))
1
>>> print(glob.order([1]))
2
>>> print(glob.order([2]))
7
>>> print(glob.order([7]))
None
>>> print(glob.order([1, '']))
1
>>> print(glob.order([1,1]))
2
>>> print(glob.order([2,3,]))
4
>>> print(glob.order([2,3,""]))
4
>>> print(glob.order([2,3,4]))
5
>>> print(glob.order([2,4,4]))
None
>>>
```

Here a missing subscript as reference or an empty string are equivalent.

## \$QUERY()

The related method is `gref.query()`

[Interactive doc](#): `print(glob.query.__doc__)`

Traverses a global starting at the specified key, returning each key and value as a tuple.

Example: `for (key, value) in g.query([i,j])` traverses `g` from key `i,j`, returning each key and value in turn

The behavior of this method differs from COS/ISOS.

- It returns ALL nodes after the starting node
- It includes the stored content
- It returns also virtual nodes with NO content indicated as `None`. Our small example looks like this (wrapped for readability):

```
>>> print(list(glob.query()))
([['1'], None), (['1', '1'], 11), (['1', '2'], 12), (['2'], None),
 (['2', '3'], None), (['2', '3', '4'], 234), (['2', '3', '5'], 235),
 (['2', '4'], None), (['2', '4', '4'], 244), (['7'], 7)]
>>>
```

or more readable:

```
>>> for (key, value) in glob.query():
...   print(key, ''.ljust(20-len(str(list(key))), '> '), value)
...
['1'] >>>>>>>>>>>>>>>>>>>>>> None
['1', '1'] >>>>>>>>>>>>>>>>>>>>> 11
['1', '2'] >>>>>>>>>>>>>>>>>>>>> 12
['2'] >>>>>>>>>>>>>>>>>>>>>> None
['2', '3'] >>>>>>>>>>>>>>>>>>>>> None
['2', '3', '4'] >>>>>>>>>>>>>>>>>>>>>> 234
['2', '3', '5'] >>>>>>>>>>>>>>>>>>>>>> 235
['2', '4'] >>>>>>>>>>>>>>>>>>>>>> None
['2', '4', '4'] >>>>>>>>>>>>>>>>>>>>>> 244
['7'] >>>>>>>>>>>>>>>>>>>>>>>>>>>>> 7
>>>
```

It is definitely not ZWRITE !

Another option is to get the subscripts only using `gref.keys()`

[Interactive doc: print\(glob.keys.\\_\\_doc\\_\\_\)](#)

Traverses a global starting at the specified key, returning each key in the global.

Example: for key in `g.keys([i, j])` traverses `g` from key `i,j`, returning each key in turn. >>>

```
>>> list(glob.keys())
[['1'], ['1', '1'], ['1', '2'], ['2'], ['2', '3'], ['2', '3', '4'],
 ['2', '3', '5'], ['2', '4'], ['2', '4', '4'], ['7']]
>>>
```

And then I found `gref.orderiter()` with this

[Interactive doc: print\(glob.orderiter.\\_\\_doc\\_\\_\)](#)

Traverses a global starting at the specified key, returning the next key and value as a tuple.

Example: for (key, value) in `g.orderiter([i,j])` traverses `g` from key `i,j`, returning the next key and value.

It acts like `$ORDER()` also fetching the content and providing the next sub-node down with it'S content like `$QUERY()` see it:

```
>>> list(glob.orderiter([]))
[[('1', None), ('1', '1'), 11]]
>>> list(glob.orderiter([1]))
[[('2', None), ('2', '3'), None], ('2', '3', '4'), 234]]
>>> list(glob.orderiter([2]))
[[('7', 7)]
>>>
```

Finally, there is a method `gref.getAsBytes()`

[Interactive doc: print\(glob.getAsBytes.\\_\\_doc\\_\\_\)](#)

Given the keys of a global, returns a string stored at that node of the global, as bytes.

Example: `x = g.getAsBytes([i,j])` sets `x` to the value stored at key `i,j` of global `g`, as bytes.

It fails for numeric values. But likes strings:

```
>>> glob[5]="robert"
>>> glob.get([5])
```

```
'robert '  
>>> glob.getAsBytes([5])  
b'robert '
```

And if I run in COS/ISOS: `set ^rcc(9)=$IB(99,"robert")`  
I can get this:

```
>>> glob[9]  
'\x03\x04c\x08\x01robert '  
>>> glob.getAsBytes([9])  
b'\x03\x04c\x08\x01robert '  
>>>
```

How did I detect all these methods:

```
>>> for meth in glob.__dir__():  
... meth  
...  
'__len__'  
'__getitem__'  
'__setitem__'  
'__delitem__'  
'__new__'  
'data'  
'get'  
'set'  
'kill'  
'getAsBytes'  
'order'  
'query'  
'orderiter'  
'keys'  
'__doc__'  
'__repr__'  
'__hash__'  
'__str__'  
'__getattr__'  
'__setattr__'  
'__delattr__'  
'__lt__'  
'__le__'  
'__eq__'  
'__ne__'  
'__gt__'  
'__ge__'  
'__init__'  
'__reduce_ex__'  
'__reduce__'  
'__subclasshook__'  
'__init_subclass__'  
'__format__'  
'__sizeof__'  
'__dir__'  
'__class__'  
>>>
```

I hope this makes life easier if you require direct access to Globals from Embedded Python  
My personal learning: There is mostly a documentation . . . . somewhere.  
You just have to dig and explore it.

[Video Demo](#)

[Traduction française](#)

[#Embedded Python #Globals #Python #InterSystems IRIS](#)

Source URL: <https://community.intersystems.com/post/working-globals-embedded-python>