Article
Robert Cemper · Juil 1, 2022 8m de lecture

Travailler avec les Globales dans Embedded Python

Je m'intéresse particulièrement à l'utilisation des **Globales** avec **Embedded Python**.
Alors, j'ai commencé à consulter la documentation officielle.

#1 Introduction to Globals
Une tentative de description générique de ce qu'est une Globale. Pointant ensuite vers:

#2 A Closer Look at ObjectScript
Mais où puis-je trouver Embedded Python ?
Plus bas, se trouve:

#3 Embedded Python

3.1 Embedded Python Overview
3.1.1 Work with Globals
Idéal si vous n'avez jamais vu une Globale.
Sinon ce n'est qu'un exemple primitif choquant
3.2 Using Embedded Python
Dernier espoir: >>> Mais, absolument **RIEN** de visible.

Une déception! Même IRIS Native API for Python est plus détaillé.
Pour être clair sur ce que j'attends :

SET, GET, KILL pour un node de Global

Native API: Fundamental Node Operations et

Navigation avec $DATA(), $ORDER(), $QUERY()

Native API: Iteration with nextSubscript() and isDefined()
Alors, Il me faut d' enquêter, faire du reverse engineering
et expérimenter moi-même.

Voilà ce que j'ai trouvé:

Tous les exemples sont présentés dans Python Shell comme fourni de
IRIS for Windows (x86-64) 2022.1 (Build 209U)
faisant usage intensif de la fonction implicite print().

La Globale

Quoi que vous envisagiez de faire, vous devez commencer par la classe **iris.gref** pour créer un objet de référence pour la Globale.
Le nom de la Globale est transmis directement (%String) ou par une variable similaire à l'*indirection* en COS/ISOS.
Le caret initial (**^**) n'est pas nécessaire, parce qu' il est clair que nous ne traitons que des Globales !

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

Ce sont 3 references de la même Globale.
C'est seulement une référence, il n'y aucune indication sur l'existance de la globale.
Interactive doc: print(glob.doc)
InterSystems IRIS global reference object.
Use the iris.gref() method to obtain a reference to a global

SUBSCRIPTS

Tous Subscripts des Globales sont passés par Pylist **[sub1,sub2]**. Pas de grande difference par rapport à COS/ISOS
Seul la racine, sans aucun Subscript a besoin d'un traitement spécial.
Vous devez utiliser cette construction spéciale **[None]** pour indiquer que vous faites référence à la racine.

SET

Pour définir une Globale, nous pouvons le faire "directement" comme nous le ferions dans COS/ISOS.

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

ou utiliser la méthode **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

Pour accéder au contenu d'un noeud, nous pouvons le faire "directement" comme nous le ferions dans COS/ISOS.

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

ou utiliser la méthode **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: Ce n' est **pas $GET()** comme vous le connaissez en COS/ISOS

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

```
KeyError: 'Global Undefined'
>>>
```

Mais en l'utilisant "directement", il agit comme $GET() en COS/ISOS

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

Ce *None* signale ce que SQL exprime comme NULL.Il réapparaîtra plus tard.

KILL

Il -y-a seulement la méthod **gref.kill()** pour arriver au résultat escompté.

```
>>> 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()

La méthode est **gref.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

Cela fonctionne comme prévu.

```
>>> 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()

Pour cet exemple, j'ai ajouté quelques noeuds à la globale ^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
```

La méthode est **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.

Donc, nous voyons:

```
>>> 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
>>>
```

Ici, une référence Subscript manquante un %String vide sont équivalents.

$QUERY()

La méthode associée est **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

Le comportement de cette méthode est différent de COS/ISOS

- ça retourne **TOUS les noeuds** apres le noeud de référence
- ça comprend les valeurs stockées
- ça retourne aussi les noeuds virtuelles SANS valeurs et c'est indiqué par *None*. Notre exemple ressemble à ceci (enveloppé pour plus de lisibilité):

```
>>> 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)]
>>>
```

ou plus lisible :

```
>>> 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
>>>
```

Ce n'est certainement pas **ZWRITE** !

Une autre option consiste à obtenir les Subscripts uniquement à l'aide de **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'], see it here
             ['2', '3', '5'], ['2', '4'], ['2', '4', '4'], ['7']]
>>>
```

Et ensuite, j'ai trouvé **gref.orderiter()** avec:
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.

Cela agit comme **$ORDER(),** récupérant également les valeurs et
fournis le sous-Node suivant avec sa valeur comme le $QUERY()
voici:

```
>>> 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)]
>>>
```

Finallement, il-y-a une méthode **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.

Il échoue pour les valeurs numériques. Mais travailes pour %String:

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

Et si j'exécute en COS/ISOS: *set ^rcc(9)=$lB(99,"robert")*
Je peux avoir le résultat suivant:

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

Comment ai-je détecté toutes ces méthodes:

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

```
'__sizeof__'
'__dir__'
'__class__'
>>>
```

J'espère que cela vous facilitera la vie si vous avez besoin d'un accès direct aux Globales dans Embedded Python
Mon apprentissage personnel : Il y a surtout une documentation . . . . *quelque part*.
Il suffit de creuser et d'explorer.

Video Demo

Merci d'avance pour votre vote au **Concours d'articles techniques d'InterSystems : Édition Python**

#Embedded Python #Globals #Python #InterSystems IRIS

URL de la source:https://fr.community.intersystems.com/post/travailler-avec-les-globales-dans-embedded-python