Article
José Pereira · Apr 5, 2022 4m read

Open Exchange

Serializing Python objects in globals

Motivation

This project was thought of when I was thinking of how to let Python code deal naturally with the scalable storage and efficient retrieving mechanism given by IRIS globals, through Embedded Python.

My initial idea was to create a kind of Python dictionary implementation using globals, but soon I realized that I should deal with object abstraction first.

So, I started creating some Python classes that could wrap Python objects, storing and retrieving their data in globals, i.e., serializing and deserializing Python objects in IRIS globals.

How does it work?

Like ObjectScript %DispatchGetProperty(), %DispatchSetProperty() and %DispatchMethod(), Python has methods to delegate object ’ s properties and methods calls.

When you set or get an object property, the Python interpreter lets you intercept this operation by methods setattr(self, name, value) and getattr(self, name).

Check out this pretty basic example:

```
>>> class Test:
...         def __init(self, prop1):
...                 self.prop1 = prop1
...         def __setattr__(self, name, value):
...                 print(f"setting property {name} to value {value}")
...         def __getattr__(self, name):
...                 print(f"getting property {name}")
...
>>> obj = Test()
>>> obj.prop1 = "test"
setting property prop1 to value test
>>> obj.prop1
getting property prop1
>>> obj.prop2
getting property prop2
>>> obj.prop2
getting property prop2
>>>
```

Note that the methods setattr() and getattr() were called indirectly by set and get operation on objects of class Test - which implements those methods. Even a non-declared property, like prop2 in this example, issues calls to them.

That mechanism is the heart of the serialization test that I tried in my project python-globals-serializer-example. With it, you can intercept set/get operations and store/retrieve data from IRIS globals.

Object model

Globals offer a highly customizable structure. By using their hierarchical model for information access, which is pretty similar to JSON objects and Python dictionaries, we can store objects ’ properties data and metadata.

Here is how I used global to create a simple object model to serialize Python objects:

For each serialized object, its class name is serialized into a node labeled as “ class ” :

```
^test(1,"class")="<class 'employee.SalaryEmployee'>"
```

The first index is an incremental number which is used as reference to this object in the model. So, in the above example, an object of class employee.SalaryEmployee is stored with the reference value 1.

For properties of primitive data types, their type and value are stored. For instance:

```
^test(1,"name","type")="<class 'str'>"
^test(1,"name","value")="me"
```

That structure is interpreted as the object referred to by index 1, has a property called name, with value equals ‘ me ’ .

For properties referencing objects the model is slightly different, because unlike JSON objects or Python dictionaries, globals are intended to store only primitive data type data. So another “ class ” node is created to this referenced object and its node index (i.e. its reference) is stored in the property node:

```
^test(1,"company","oref")=2
^test(1,"company","type")="<class 'iris_global_object.IrisGlobalObject'>"
```

```
^test(2,"class")="<class 'employee.Company'>"
^test(2,"name","type")="<class 'str'>"
^test(2,"name","value")="Company ABC"
```

Those structures means that the object 1 has a property called company, which its values are stored in the index 2 - note the value for ^test(1,"company","oref").

Serialization/Deserialization process

When you create a wrapper to serialize or deserialize Python objects, you need to define the name of the global which stores the object.

After that, the serialization process is done when a set operation is executed. The setattr() method sets the property value and type in the global defined to store the object, using the simple object model explained before.

In the opposite direction, a deserialization is done by the getattr_method, when a get operation is performed.

For primitive data types, this process is straightforward: just get the value stored in the global and returns.

But for objects, the process needs to instantiate their class data type and set all their properties as well, so this way a restored Python object could be used, including calls to its methods.

Future work

As said in the beginning of this post, this project was born as a simplification of a way to let Python code use global as a natural storage engine, and aims to be just a proof of concept.

Serializing/deserializing objects is just the beginning of this goal. So, a lot of effort needs to be done to get this idea mature.

I hope this post lets you understand the purpose of my work in this project, and may inspire you to think in new ways of how to use IRIS globals to bring Python even closer to IRIS.

#Data Model #Embedded Python #Globals #Python #InterSystems IRIS #Open Exchange
Check the related application on InterSystems Open Exchange