Introduction to Embedded Python

Version 2024.2
2024-09-05

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

Introduction to Embedded 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

Introduction to Embedded Python

Embedded Python allows you to use Python side-by-side with ObjectScript, the native programming language of the InterSystems IRIS data platform. When you write a method in an InterSystems IRIS class using Embedded Python, the Python source code is compiled into object code that runs on the server, along with the compiled ObjectScript code. This allows for tighter integration than is possible using a Gateway or the Native SDK for Python. You can also import Python packages, whether they are custom or publicly available, and use them from within your ObjectScript code. Python objects are first class citizens in ObjectScript and vice versa.

This article introduces you to Embedded Python and gives several examples of how you can use it. In particular, this document covers the following scenarios:

- Using a Python package from ObjectScript — This scenario assumes you are an ObjectScript developer and you want to harness the power of the numerous Python packages that are available to the Python developer community.
- Calling the InterSystems IRIS APIs from Python — This scenario assumes you are a Python developer who is new to InterSystems IRIS and you want to know how to access the InterSystems APIs.
- Using ObjectScript and Python together — This scenario assumes you are on a mixed team of ObjectScript and Python developers and want to know how to use the two languages together.

To use this article, you will need a running InterSystems IRIS instance of version 2021.2 or later, as well as some prerequisites depending on your operating system. You also need to know how to access the ObjectScript shell, the InterSystems IRIS command-line tool.

Some of the examples in this document use classes from the Samples-Data repository on GitHub: https://github.com/intersystems/Samples-Data. InterSystems recommends that you create a dedicated namespace called SAMPLES and load samples into that namespace. If you would like to view or modify the sample code, you will need to set up an integrated development environment (IDE). Visual Studio Code is recommended.

This article does not attempt to provide a thorough overview of Embedded Python or programming with InterSystems IRIS. Use the sources listed at the end of this document to continue your exploration.

Note: Before using Embedded Python for the first time, read the Introduction and Prerequisites section of Using Embedded Python to make sure you have installed an appropriate version of Python for your system and have configured any necessary InterSystems IRIS settings.

1 Use a Python Package from ObjectScript

Using Embedded Python gives ObjectScript developers an easy way to use any of the numerous available Python libraries (commonly known as *packages*), right from InterSystems IRIS, eliminating the need to develop custom libraries to duplicate existing functionality. InterSystems IRIS looks in the <installdir>/mgr/python directory for installed Python packages

Preparing a Python package for use from ObjectScript is a two-step process:

1. From the command line, install the desired package from the Python Package Index (or another index).
2. In ObjectScript, import the installed package to load the package and return it as an object. Then, you can use the object as you would an instantiated ObjectScript class.

You can browse the Python Package Index at https://pypi.org.

1.1 Install a Python Package

Install Python packages from the command line before using them with Embedded Python. The command you use differs depending on whether you are on Windows or a UNIX-based system. InterSystems recommends that you install packages into the directory <installdir>/mgr/python.

On UNIX-based systems (except AIX), use the command `python3 -m pip install --target <installdir>/mgr/python <package>`.

For example, you can install the `numpy` package on a Linux machine as follows:

```
$ python3 -m pip install --target /InterSystems/IRIS/mgr/python numpy
```

Note: If it is not installed already, first install the package `python3-pip` with your system's package manager.

Note: On Windows, the **irispip** command was removed in InterSystems IRIS 2024.2. If you are running an earlier version, see the installation method described in the 2024.1 documentation.

On Windows, use the command: `python -m pip install --target <installdir>\mgr\python <package>`.

For example, you can install the `numpy` package on a Windows machine as follows:

```
C:\>python -m pip  install --target C:\InterSystems\IRIS\mgr\python numpy
```

If you are using the Python launcher, you can use:

```
C:\>py -m pip install --target C:\InterSystems\IRIS\mgr\python numpy
```

If you are running InterSystems IRIS in a container, see Install Python Packages in a Container.

If you are running InterSystems IRIS on AIX, see Install Python Packages on AIX.

1.2 Import a Python Package

The %SYS.Python class contains the functionality you need to use Python from ObjectScript. You can use %SYS.Python in any ObjectScript context, such as classes, Terminal sessions, or SQL. See the Class Reference for a full list of methods.

To import a Python package or module from ObjectScript, use the **%SYS.Python.Import()** method.

For example, the following command imports the math module in Terminal, assuming you are in the USER namespace.:

```
USER>set pymath = ##class(%SYS.Python).Import("math")
```

The `math` module comes packaged with the standard Python release, so you don't need to install it before importing it. By calling **zwrite** on the `pymath` object, you can see it is an instance of the built-in `math` module:

```
USER>zwrite pymath
pymath=1@%SYS.Python  ; <module 'math' (built-in)>  ; <OREF>
```

Note: A package is a collection of Python modules, but when you import a package, the object created is always of type module.

Now, you can access the `math` module properties and methods much the same as you would for any ObjectScript object:

```
USER>write pymath.pi
3.141592653589793116
USER>write pymath.factorial(10)
3628800
```

1.3 Example

This example uses the `geopy` package to access OpenStreetMap's Nominatim geocoding tool. Geocoding is the process of taking a text-based description of a location, such as an address or the name of a place, and returning geographic coordinates, such as latitude and longitude, to pinpoint the location on the Earth's surface.

First, install `geopy` from the command line, as in this Windows example (InterSystems IRIS 2024.2 or later):

```
C:\InterSystems\IRIS\bin>python -m pip install --target C:\InterSystems\IRIS\mgr\python geopy
Collecting geopy
  Using cached geopy-2.2.0-py3-none-any.whl (118 kB)
Collecting geographiclib<2,>=1.49
  Using cached geographiclib-1.52-py3-none-any.whl (38 kB)
Installing collected packages: geographiclib, geopy
Successfully installed geographiclib-1.52 geopy-2.2.0
```

If you are using the Python launcher, substitute the `py` command for the `python` command.

On a UNIX-based system (except AIX), use:

```
$ python3 -m pip install --target /InterSystems/IRIS/mgr/python geopy
```

If you are on Windows and are running a version of InterSystems IRIS earlier than 2024.2, see the installation method described in the 2024.1 documentation.

If you are running InterSystems IRIS in a container, see Install Python Packages in a Container.

If you are running InterSystems IRIS on AIX, see Install Python Packages on AIX.

Then run the following commands in terminal to import and use the module:

```
USER>set geopy = ##class(%SYS.Python).Import("geopy")

USER>set args = { "user_agent": "Embedded Python" }

USER>set geolocator = geopy.Nominatim(args...)

USER>set flatiron = geolocator.geocode("175 5th Avenue NYC")

USER>write flatiron.address
Flatiron Building, 175, 5th Avenue, Flatiron District, Manhattan, New York County, New York, 10010,
United States
USER>write flatiron.latitude _ "," _ flatiron.longitude
40.74105919999999514,-73.98964162240997666
USER>set cityhall = geolocator.reverse("42.3604099,-71.060181")

USER>write cityhall.address
Government Center, Cambridge Street, Downtown Crossing, West End, Boston, Suffolk County, Massachusetts,
 02203, United States
```

This example imports the `geopy` module into ObjectScript. It then uses the `Nominatim` module to create a `geolocator` object. The example uses the **geocode()** method of the geolocator to find a location on Earth, given a string. It then calls the **reverse()** method to find an address, given a latitude and longitude.

One thing to note is that **Nominatim()** takes a named keyword argument, a construct that is not directly supported in ObjectScript. The solution is to create a JSON object containing an argument list (which in this case sets the `user_agent` keyword to the value `"Embedded Python"`) and then pass it to the method using the `args...` syntax.

In contrast to the `math` module imported in the previous example, calling **zwrite** on the `geopy` object shows it is an instance of the `geopy` package installed in C:\InterSystems\iris\mgr\python:

```
USER>zwrite geopy
geopy=2@%SYS.Python  ; <module 'geopy' from 'c:\\intersystems\\iris\\mgr\\python\\geopy\\__init__.py'>
  ; <OREF>
```

2 Call the InterSystems IRIS APIs from Python

If you are using Embedded Python and need to interact with InterSystems IRIS, you can use the `iris` module from the Python shell or from a method written in Python in an InterSystems IRIS class. To follow the examples in this section, you can start the Python shell from a Terminal session using the ObjectScript command `do ##class(%SYS.Python).Shell()`.

When you start a Terminal session, you are placed in the USER namespace in InterSystems IRIS, and you will see the prompt USER>. However, if you have loaded the sample classes from GitHub, you will need to be in the SAMPLES namespace to access them.

In Terminal, change to the SAMPLES namespace and then launch the Python shell, as follows:

```
USER>set $namespace = "SAMPLES"
SAMPLES>do ##class(%SYS.Python).Shell()

Python 3.9.5 (default, Jul 19 2021, 17:50:44) [MSC v.1927 64 bit (AMD64)] on win32
Type quit() or Ctrl-D to exit this shell.
>>>
```

When you launch the Python shell from a Terminal session, the Python shell inherits the same context as Terminal, for example, the current namespace and user. Local variables are not inherited.

2.1 Work with Classes

To access an InterSystems IRIS class from Python, use the `iris` module to instantiate the class you want to use. Then, you can use access its properties and methods much as you would a Python class.

Note: You may be used to importing a module in Python before using it, for example:

```
>>> import iris
```

However, you do not need to import the `iris` module explicitly when running the Python shell using the Shell() method of the %SYS.Python class. Just go ahead and use the module.

The following example uses the **ManagerDirectory()** method of the system class %Library.File to print the path to the InterSystems IRIS manager directory:

```
>>> lf = iris.cls('%Library.File')
>>> print(lf.ManagerDirectory())
C:\InterSystems\IRIS\mgr\
```

This example uses the **Dump()** method of the system class %SYSTEM.CPU to display information about the server on which the instance of InterSystems IRIS is being run:

```
>>> cpu = iris.cls('%SYSTEM.CPU')
>>> cpu.Dump()

-- CPU Info for node MYSERVER ----------------------------------------------
          Architecture: x86_64
                 Model: Intel(R) Core(TM) i7-7600U CPU @ 2.80GHz
                Vendor: Intel
          # of threads: 4
            # of cores: 2
            # of chips: 1
 # of threads per core: 2
   # of cores per chip: 2
          MT supported: 1
            MT enabled: 1
                   MHz: 2904
-----------------------------------------------------------------------------
```

This example uses the Sample.Company class from the Samples-Data repository on GitHub. While you can use classes that start with a percent sign (%), like %SYS.Python or %Library.File, from any namespace, to access the Sample.Company class, you must be in the SAMPLES namespace, as mentioned earlier.

The class definition for Sample.Company is as follows:

```
Class Sample.Company Extends (%Persistent, %Populate, %XML.Adaptor)
{

/// The company's name.
Property Name As %String(MAXLEN = 80, POPSPEC = "Company()") [ Required ];

/// The company's mission statement.
Property Mission As %String(MAXLEN = 200, POPSPEC = "Mission()");

/// The unique Tax ID number for the company.
Property TaxID As %String [ Required ];

/// The last reported revenue for the company.
Property Revenue As %Integer;

/// The Employee objects associated with this Company.
Relationship Employees As Employee [ Cardinality = many, Inverse = Company ];

}
```

This class extends %Library.Persistent (often abbreviated as %Persistent), which means objects of this class can be persisted in the InterSystems IRIS database. The class also has several properties, including Name and TaxID, both of which are required for the object to be saved.

Though you will not see them in the class definition, persistent classes come with a number of methods for manipulating objects of this class, such as **%New()**, **%Save()**, **%Id()**, and **%OpenId()**. However, percent signs (%) are not allowed in Python method names, so the underscore (_) is used instead.

The code below creates a new Company object, sets the required Name and TaxID properties, and then saves the company in the database:

```
>>> my_company = iris.cls('Sample.Company')._New()
>>> my_company.Name = 'Acme Widgets, Inc.'
>>> my_company.TaxID = '123456789'
>>> status = my_company._Save()
>>> print(status)
1
>>> print(my_company._Id())
22
```

The code above uses the **_New()** method to create an instance of the class and **_Save()** to save the instance in the database. The **_Save()** method returns a status code. In this case, a 1 indicates that the save was successful. When you save an object, InterSystems IRIS assigns it a unique ID that you can use to retrieve the object from storage at a later time. The **_Id()** method returns the ID of the object.

Use the **_OpenId()** method of the class to retrieve an object from persistent storage into memory for processing:

```
>>> your_company = iris.cls("Sample.Company")._OpenId(22)
>>> print(your_company.Name)
Acme Widgets, Inc.
```

As noted previously, many methods of InterSystems IRIS classes return status codes. Let's say that you change the name of your_company in the example above and try to save the object. A status of 1 indicates a successful save, but if an error occurs during the save, the status contains an encoded error string.

```
>>> your_company.Name = 'The Best Company'
>>> status = your_company._Save()
>>> print(status)
0
0Sample.Company%SaveData=1^Sample.Company.1^USER#e^%SaveData=1^Sample.Company.1^7e^%SerializeObject=7^Sample.Company.1^2e^%Save=8^Sample.Company.1^5e^%Shell=47^%SYS.Python.1^1d^^0
```

Instead of trying to print the contents of a status, you can use the method **iris.check_status()** to check the status and throw a Python exception if it contains an error, as in the following example:

```
>>> try:
...     iris.check_status(status)
... except Exception as ex:
...     print(ex)
...
ERROR #5803: Failed to acquire exclusive lock on instance of 'Sample.Company'
```

Here, someone else must have obtained an exclusive lock on this company, so the save fails. See Object Concurrency Options for more information.

If you are familiar with the %SYSTEM.Status class, you can also use its methods to work with status codes, for example:

```
>>> if iris.cls('%SYSTEM.Status').IsError(status):
...     iris.cls('%SYSTEM.Status').DisplayError(status)
...

ERROR #5803: Failed to acquire exclusive lock on instance of 'Sample.Company'1
```

The method **IsError()** checks to see if the status contains an error, and **DisplayError()** prints the error message.

Note: Effective with InterSystems IRIS 2024.2, an optional shorter syntax for referring to an InterSystems IRIS class from Embedded Python has been introduced. Either the new form or the traditional form are permitted.

2.2 Work with SQL

Classes in InterSystems IRIS are projected to SQL, allowing you to access the data using a query, in addition to using class methods or direct global access. The `iris` module provides you with two different ways for you to run SQL statements from Python.

The following example uses the **iris.sql.exec()** to run an SQL SELECT statement to find all instances of the Sample.Company class where the name of the company starts with `"Comp"`, returning a result set that includes each the name and mission statement of each company. Here, the class Sample.Company projects to SQL as a table of the same name.

```
>>> rs = iris.sql.exec("SELECT Name, Mission FROM Sample.Company WHERE Name %STARTSWITH 'Comp'")
```

The following example uses the **iris.sql.prepare()** to prepare an SQL query object and then executes the query, passing in `"Comp"` as a parameter:

```
>>> stmt = iris.sql.prepare("SELECT Name, Mission FROM Sample.Company WHERE Name %STARTSWITH ?")
>>> rs = stmt.execute("Comp")
```

In either case, you can iterate through the result set as follows, and the output is the same:

```
>>> for idx, row in enumerate(rs):
...     print(f"[{idx}]: {row}")
...
[0]: ['CompuDynamics.com', 'Post-sale services for disruptive optical virtualized productivity tools
for our long-term clients.']
[1]: ['CompuCalc Holdings Inc.', 'Developers of enhanced seven-sigma forecasting technologies for the
 Entertainment industry.']
[2]: ['Compumo Gmbh.', 'Resellers of just-in-time database gaming for the Fortune 5.']
```

If a query results in an error, InterSystems returns an exception that is an instance of the type irisbuiltins.SQLError. The example below shows what it looks like if you try to insert a new row, but you are missing a required field.

```
>>> stmt = iris.sql.prepare("INSERT INTO Sample.Company (Mission, Name, Revenue, TaxID) VALUES (?, ?,
 ?, ?)")
>>> try:
...     rs = stmt.execute("We are on a mission", "", "999", "P62")
... except Exception as ex:
...     print(ex.sqlcode, ex.message)
...
-108 'Name' in table 'Sample.Company' is a required field
```

2.3 Work with Globals

In the InterSystems IRIS database, all data is stored in *globals*. Globals are arrays that are persistent (meaning they are stored on disk), multidimensional (meaning they can have any number of subscripts), and sparse (meaning that the subscripts do not have to be contiguous). When you store objects of a class or rows in a table, this data is actually stored in globals, though you typically access them through methods or SQL and never touch the globals directly.

Sometimes it can be useful to store persistent data in globals, without setting up a class or an SQL table. In InterSystems IRIS, a global looks much like any other variable, but it is denoted with a caret (^) in front of the name. The following example stores the names of the workdays in the global `^Workdays` in your current namespace.

```
>>> my_gref = iris.gref('^Workdays')
>>> my_gref[None] = 5
>>> my_gref[1] = 'Monday'
>>> my_gref[2] = 'Tuesday'
>>> my_gref[3] = 'Wednesday'
>>> my_gref[4] = 'Thursday'
>>> my_gref[5] = 'Friday'
>>> print(my_gref[3])
Wednesday
```

The first line of code, `my_gref = iris.gref('^Workdays')`, gets a *global reference* (or *gref*) to a global called `^Workdays`, which may or may not already exist.

The second line, `my_gref[None] = 5`, stores the number of workdays in `^Workdays`, without a subscript.

The third line, `my_gref[1] = 'Monday'`, stores the string `Monday` in the location `^Workdays(1)`. The next four lines store the remaining workdays in the locations `^Workdays(2)` through `^Workdays(5)`.

The final line, `print(my_gref[3])`, shows how you can access the value stored in a global, given its gref.

For a more detailed example, see Globals.

3 Use ObjectScript and Python Together

InterSystems IRIS makes it easy for mixed teams of ObjectScript and Python programmers to work together. For example, some of the methods in a class can be written in ObjectScript and some in Python. Programmers can choose to write in the language they are most comfortable with, or the language that is more suitable for the task at hand.

3.1 Create Mixed InterSystems IRIS Classes

The following version of the Sample.Company class has a **Print()** method written in Python and a **Write()** method written in ObjectScript, but they are functionally equivalent, and either method can be called from Python or ObjectScript.

```
Class Sample.Company Extends (%Persistent, %Populate, %XML.Adaptor)
{

/// The company's name.
```

```
Property Name As %String(MAXLEN = 80, POPSPEC = "Company()") [ Required ];

/// The company's mission statement.
Property Mission As %String(MAXLEN = 200, POPSPEC = "Mission()");

/// The unique Tax ID number for the company.
Property TaxID As %String [ Required ];

/// The last reported revenue for the company.
Property Revenue As %Integer;

/// The Employee objects associated with this Company.
Relationship Employees As Employee [ Cardinality = many, Inverse = Company ];

Method Print() [ Language = python ]
{
    print(f"\nName: {self.Name} TaxID: {self.TaxID}")
}

Method Write() [ Language = objectscript ]
{
    write !, "Name: ", ..Name, " TaxID: ", ..TaxID, !
}
}
```

This Python code sample shows how to open the Company object with %Id=2 and call both the **Print()** and **Write()** methods.

```
>>> company = iris.cls("Sample.Company")._OpenId(2)
>>> company.Print()

Name: IntraData Group Ltd. TaxID: G468
>>> company.Write()

Name: IntraData Group Ltd. TaxID: G468
```

This ObjectScript code sample shows how to open same Company object and call both methods.

```
SAMPLES>set company = ##class(Sample.Company).%OpenId(2)

SAMPLES>do company.Print()

Name: IntraData Group Ltd. TaxID: G468

SAMPLES>do company.Write()

Name: IntraData Group Ltd. TaxID: G468
```

3.2 Pass Data Between Python and ObjectScript

While Python and ObjectScript are compatible in many ways, they have many of their own data types and constructs, and sometimes it is necessary to do some data conversion when passing data from one language to another. You saw one example earlier, in the example of passing named arguments from ObjectScript to Python.

The **Builtins()** method of the %SYS.Python class provides a handy way for you to access Python's built-in functions, which can help you create objects of the type expected by a Python method.

The following ObjectScript example creates two Python arrays, `newport` and `cleveland`, each of which contain the latitude and longitude of a city:

```
USER>set builtins = ##class(%SYS.Python).Builtins()

USER>set newport = builtins.list()

USER>do newport.append(41.49008)

USER>do newport.append(-71.312796)

USER>set cleveland = builtins.list()

USER>do cleveland.append(41.499498)

USER>do cleveland.append(-81.695391)

USER>zwrite newport
newport=11@%SYS.Python  ; [41.49008, -71.312796]  ; <OREF>

USER>zwrite cleveland
cleveland=11@%SYS.Python  ; [41.499498, -81.695391]  ; <OREF>
```

The code below uses the `geopy` package, which you saw in the earlier example, to calculate the distance between Newport, RI, and Cleveland, OH. It creates a route using the **geopy.distance.distance()** method, passing the arrays as parameters, and then prints the `miles` property of the route.

```
USER>set distance = $system.Python.Import("geopy.distance")

USER>set route = distance.distance(newport, cleveland)

USER>write route.miles
538.3904453677205311
```

Note: The **geopy.distance.distance()** method actually expects the parameters to be of the Python tuple data type, but arrays also work.

3.3 Run an Arbitrary Python Command from ObjectScript

When you are developing or testing something, sometimes it can be useful to run a line of Python code from ObjectScript to see what it does or if it works. In cases like this, you can use the **%SYS.Python.Run()** method, as is shown in the example below:

```
USER>do ##class(%SYS.Python).Run("print('hello world')")
hello world
```

3.4 Run an Arbitrary ObjectScript Command from Embedded Python

Conversely, sometimes it can be useful to run a line of ObjectScript code from Embedded Python. In cases like this, you can use the **iris.execute()** method, as is shown in the example below:

```
>>> iris.execute('write "hello world", !')
hello world
```

4 Learn More About Embedded Python

Use the resources listed below to learn more about Embedded Python.

- What Is Embedded Python? — A short introductory video on Embedded Python.

- Parsing Images and Charting Data with Embedded Python — A longer hands-on exercise that shows you how to use ObjectScript and Python interchangeably as you write an application.
- Using Embedded Python — A deeper look at Embedded Python, including some nuts and bolts to help you use Embedded Python and ObjectScript together.
- Python Built-In Functions — List of functions built in to the Python interpreter, including those that can be handy when converting data to a Python data type.

Additional documentation and online training materials are in development and will be added to this list.

5 Learn More About Programming in InterSystems IRIS

Use the resources listed below for general information about programming in InterSystems IRIS.

- Writing Python Applications with InterSystems — A learning path that runs through all of the options for using Python with InterSystems IRIS, including pyodbc, Embedded Python, and the External Language Server.
- InterSystems IRIS Basics: Globals — A hands-on introduction to the underlying InterSystems IRIS storage mechanism.
- ObjectScript Tutorial — An interactive introduction to the ObjectScript programming language.
- Orientation Guide for Server-Side Programming — The essentials for programmers who write server-side code using InterSystems products, with both ObjectScript and Python examples.