InterSystems™
IRIS Data Platform

Using Embedded Python

Version 2025.1
2025-06-03

Using Embedded Python
PDF generated on 2025-06-03
InterSystems IRIS® Version 2025.1
Copyright © 2025 InterSystems Corporation
All rights reserved.

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

1

Introduction and Prerequisites

Embedded Python allows you to use Python as a native option for programming InterSystems IRIS applications. If you are new to Embedded Python, read Introduction to Embedded Python, and then read this document for a deeper dive into Embedded Python.

While this document will be helpful to anyone who is learning Embedded Python, some level of ObjectScript familiarity will be beneficial to the reader. If you are a Python developer who is new to InterSystems IRIS and ObjectScript, also see the Orientation Guide for Server-Side Programming.

1.1 Recommended Python Version

The version of Python recommended when using Embedded Python depends on the platform you are running. In most cases, this is the default version of Python for your operating system. See Other Supported Features for a complete list of operating systems and the corresponding supported version of Python.

Note: On some operating systems, you can override the recommended version of Python using the Flexible Python Runtime feature.

Microsoft Windows does not come with a default version of Python, and as of InterSystems IRIS 2024.2, the InterSystems IRIS installer for Windows no longer installs Python for you. See Flexible Python Runtime Feature for information on setting up Embedded Python for Windows.

Many flavors of UNIX-based operating systems come with Python installed. If you need to install it, use the version recommended for your operating system by your package manager, for example:

- Ubuntu: `apt install python3`

- Red Hat Enterprise Linux or Oracle Linux: `yum install python3`

- SUSE: `zypper install python3`

- macOS: Install Python 3.11 using Homebrew.

    `brew install python@3.11`

    You should also make sure you are using the current version of OpenSSL:

    ```
    brew unlink openssl
    brew install openssl@3
    brew link --force openssl@3
    ```

- AIX: Install Python 3.9.18+ using dnf (dandified yum) from the AIX Toolbox for Open Source Software

If you get an error that says "Failed to load python," it means that you either don't have Python installed or an unexpected version of Python is detected on your system. Check Other Supported Features and make sure you have the required version of Python installed, and if necessary, install it or reinstall it using one of the above methods. Or, override the recommended Python version by using the Flexible Python Runtime feature. (Not available on all platforms.)

If you are on a platform that does not support the Flexible Python Runtime feature, your computer has multiple versions of Python installed, and you try to run Embedded Python from the command line, **irispython** will run the first **python3** executable that it detects, as determined by your path environment variable. Make sure that the folders in your path are set appropriately so that the required version of the executable is the first one found. For more information on using the **irispython** command, see Start the Python Shell from the Command Line.

1.2 Required Service

To prevent `IRIS_ACCESSDENIED` errors while running Embedded Python, enable `%Service_CallIn`. In the Management Portal, go to **System Administration** > **Security** > **Services**, select **%Service_CallIn**, and check the **Service Enabled** box.

1.3 Flexible Python Runtime Feature

The Flexible Python Runtime Feature allows you to choose which version of Python you want to use with Embedded Python. Flexible Python Runtime is not supported on all operating systems. See Other Supported Features for a complete list of platforms that support the feature.

Important: There are known incompatibilities between InterSystems IRIS and Python 3.13, so you are advised to avoid Python 3.13 at this time. This issue will be addressed in an upcoming version of InterSystems IRIS.

On Microsoft Windows, as of InterSystems IRIS 2024.2, the installer no longer installs a default version of Python. You can install Python from https://www.python.org/downloads/. Make sure to do a custom installation and install Python for all users.

Important: The installer for InterSystems IRIS 2024.1 on the Windows platform previously installed OpenSSL 3.1.4 along with Python 3.9.5. However, installing Python 3.9 from Python.org installs OpenSSL 1.1.1n. If you need to use OpenSSL 3 with Embedded Python on InterSystems IRIS 2024.2, install Python 3.11 or higher.

On InterSystems IRIS 2024.2 for Windows, the AutoML feature of IntegratedML requires using Python 3.11.

On other operating systems that support the Flexible Python Runtime feature, you can override the default version of Python. This is useful if you are writing code or using a package that depends on a version of Python other than the default version. You must use a version of Python that is the same or greater than your operating system's default version. For example, Red Hat Enterprise Linux 9 comes with Python 3.9, so on that operating system you must use version 3.9 or higher.

After installing Python, follow the steps below to configure the version of Python to be used by Embedded Python:

1. In the Management Portal, go to **System Administration** > **Configuration** > **Additional Settings** > **Advanced Memory**.

2. On the **Advanced Memory Settings** page, in the **PythonRuntimeLibrary** row, click **Edit**.

3. Enter the location of the Python runtime library you want to use.

   This location will vary based on your operating system, Python version, and other factors.

   Windows example: `C:\Program Files\Python311\python3.dll` (Python 3.11 on Windows)

Linux example: `/usr/lib/x86_64-linux-gnu/libpython3.11.so.1.0` (Python 3.11 on Ubuntu 22.04 on the x86 architecture)

4. Click **Save**.

5. On the **Advanced Memory Settings** page, in the **PythonRuntimeLibraryVersion** row, click **Edit**.

6. Enter the version number of the Python runtime library you want to use.

   For example: `3.11`

7. Click **Save**.

For more information, see PythonRuntimeLibrary and PythonRuntimeLibraryVersion.

Note: If you install a new version of Python and cannot find the Python runtime library, you may need to install it separately. For example, to install the Python 3.11 runtime library on Ubuntu 22.04: `apt install libpython3.11`.

On Microsoft Windows, if you find that the Python runtime library is located within your user directory, for example, C:\Users\<username>\AppData\Local\Programs\Python\Python311, it indicates that you have installed Python for the current user only. InterSystems recommends installing Python for all users to avoid unexpected results.

For more details on how to configure this feature, including step-by-step examples, see Use the Flexible Python Runtime Feature.

2

Install and Import Python Packages

Embedded Python gives you easy access to thousands of useful libraries. Commonly called "packages," these need to be installed into the InterSystems IRIS file system before they can be used. Then they need to imported to load them into memory for use by your code. There are different ways to do this, depending on how you will use Embedded Python.

2.1 Install Python Packages

Install Python packages from the command line before using them with Embedded Python. The command you use differs depending on whether you are using InterSystems IRIS on a UNIX-based system (except AIX), on AIX, on Windows, or in a container.

2.1.1 Install Python Packages on a UNIX-Based System (except AIX)

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

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

For example, the ReportLab Toolkit is an open source package for generating PDFs and graphics. On a UNIX-based system, use a command like the following to install it:

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

Important: If you do not install the package into the correct target directory, Embedded Python may not be able to import it. For example, if you install a package without the `--target` attribute (and without using **sudo**), Python will install it into the local package repository within your home directory. If any other user tries to import the package, it will fail.

Although InterSystems recommends using the `--target <installdir>/mgr/python` option, installing packages using **sudo** and omitting the `--target` attribute installs the packages into the global package repository. These packages can also be imported by any user.

2.1.2 Install Python Packages on AIX

On AIX, install packages from the AIX Toolbox for Open Source Software, if they are available.

Before installing the package, make sure the package is in the AIX Toolbox with the command `sudo dnf list | grep <package>`. Then install the package with the command `sudo dnf install <package>`.

Note: If it is not installed already, first install the package `python3.9-pip` from the AIX Toolbox.

For example, confirm that the package `psutil` is the AIX Toolbox:

```
$ sudo dnf list | grep psutil
python3-psutil.ppc                              5.9.0-2          AIX_Toolbox
python3-psutil-tests.ppc                        5.9.0-2          AIX_Toolbox
python3.9-psutil.ppc                            5.9.0-2          AIX_Toolbox
python3.9-psutil-tests.ppc                      5.9.0-2          AIX_Toolbox
```

Then install the package:

```
$ sudo dnf install python3-psutil.ppc
```

Only if a package is not in the AIX Toolbox, install it with the following command:

```
$ python3 -m pip install <package>
```

2.1.3 Install Python Packages on Windows

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 ReportLab package on a Windows machine as follows:

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

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

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

2.1.4 Install Python Packages in a Container

If you are running InterSystems IRIS in a container without using the durable %SYS feature, use the command `python3 -m pip install --target /usr/irissys/mgr/python <package>`.

For example, you can install the ReportLab package in the container as follows:

```
$ python3 -m pip install --target /usr/irissys/mgr/python reportlab
```

If you are running InterSystems IRIS in a container using the durable %SYS feature, use the command `python3 -m pip install --target <durable>/mgr/python <package>`, where *<durable>* is the path defined in the environment variable `ISC_DATA_DIRECTORY` when running the container.

For example, if `ISC_DATA_DIRECTORY=/durable/iris`, you can install the ReportLab package in the container as follows:

```
$ python3 -m pip install --target /durable/iris/mgr/python reportlab
```

Note: Note: If you are using a Dockerfile to create a custom Docker image for InterSystems IRIS, install Python packages in /usr/irissys/mgr/python. Both /usr/irissys/mgr/python and <durable>/mgr/python are included in `sys.path` by default so that the packages can be found whether or not you are using the durable %SYS feature.

For more information on creating and running containers, see Running InterSystems Products in Containers.

2.2 Import Python Packages

After installing a package, you need to import it before you can use it from InterSystems IRIS. This loads the package into memory so that it is available for use.

2.2.1 Import Python Packages from ObjectScript

To import a Python package from ObjectScript, use the **Import()** method of the %SYS.Python class. For example:

```
set pymath = ##class(%SYS.Python).Import("math")
set canvaslib = ##class(%SYS.Python).Import("reportlab.pdfgen.canvas")
```

The first line, above, imports the built-in Python math module into ObjectScript. The second line imports just the canvas.py file from the pdfgen subpackage of ReportLab.

2.2.2 Import Python Packages from a Method Written in Python

You can import packages in an InterSystems IRIS method written in Python, just as you would in any other Python code, for example:

```
ClassMethod Example() [ Language = python ]
{
    import math
    import iris
    import reportlab.pdfgen.canvas as canvaslib

    # Your Python code here
}
```

2.2.3 Import Python Packages via an XData Block

You can also import a list of packages using an XData block in a class, as in the following example:

```
XData %import [ MimeType = application/python ]
{
import math
import iris
import reportlab.pdfgen.canvas as canvaslib
}
```

Important: The name of the XData block must be %import. The MIME type can be application/python or text/x-python. Make sure to use correct Python syntax, but do not indent your code or a complier error will occur.

These packages can then be used in any method within the class that is written in Python, without needing to import them again.

```
ClassMethod Test() [ Language = python ]
{
   # Packages imported in XData block

    print('\nValue of pi from the math module:')
    print(math.pi)
    print('\nList of classes in this namespace from the iris module:')
    iris._SYSTEM.OBJ.ShowClasses()
}
```

For background information on XData blocks, see Defining and Using XData Blocks

3

Run Embedded Python

This page details several ways to run Embedded Python.

3.1 From the Python Shell

You can start the Python shell from an InterSystems Terminal session or from the command line.

3.1.1 Start the Python Shell from Terminal

Start the Python shell from an InterSystems Terminal session by calling the **Shell()** method of the %SYS.Python class. This launches the Python interpreter in interactive mode. The user and namespace from the Terminal session are passed to the Python shell.

Exit the Python shell by typing the command `quit()`.

The following example launches the Python shell from the USER namespace in a Terminal session. It prints the first few numbers in the Fibonacci sequence and then uses the InterSystems IRIS **%SYSTEM.OBJ.ShowClasses()** method to print a list of classes in the current namespace.

```
USER>do ##class(%SYS.Python).Shell()

Python 3.9.5 (default, Jul  6 2021, 13:03:56) [MSC v.1927 64 bit (AMD64)] on win32
Type quit() or Ctrl-D to exit this shell.
>>> a, b = 0, 1
>>> while a < 10:
...     print(a, end = ' ')
...     a, b = b, a + b
...
0 1 1 2 3 5 8 >>>
>>> status = iris._SYSTEM.OBJ.ShowClasses()
User.Company
User.Person
>>> print(status)
1
>>> quit()

USER>
```

The method **%SYSTEM.OBJ.ShowClasses()** returns an InterSystems IRIS %Status value. In this case, a 1 means that no errors were detected.

3.1.2 Start the Python Shell from the Command Line

Start the Python shell from the command line by using the **irispython** command. This works much the same as starting the shell from Terminal, but you must pass in the InterSystems IRIS username, password, and namespace.

The following example launches the Python shell from the Windows command line:

```
C:\InterSystems\IRIS\bin>set IRISUSERNAME = <username>

C:\InterSystems\IRIS\bin>set IRISPASSWORD = <password>

C:\InterSystems\IRIS\bin>set IRISNAMESPACE = USER

C:\InterSystems\IRIS\bin>irispython
Python 3.9.5 (default, Jul  6 2021, 13:03:56) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

On UNIX-based systems, use **export** instead of **set**.

```
/InterSystems/IRIS/bin$ export IRISUSERNAME=<username>
/InterSystems/IRIS/bin$ export IRISPASSWORD=<password>
/InterSystems/IRIS/bin$ export IRISNAMESPACE=USER
/InterSystems/IRIS/bin$ ./irispython
Python 3.9.5 (default, Jul 22 2021, 23:12:58)
[GCC 9.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Note: If you see a message saying `IRIS_ACCESSDENIED`, enable `%Service_Callin`. In the Management Portal, go to **System Administration** > **Security** > **Services**, select **%Service_CallIn**, and check the **Service Enabled** box.

3.2 In a Python Script File (.py)

You can also use the **irispython** command to execute a Python script. Note that in this case, you have to include a step (`import iris`) that provides access to InterSystems IRIS.

Consider a file C:\python\test.py, on a Windows system, containing the following code:

```
# print the members of the Fibonacci series that are less than 10
print('Fibonacci series:')
a, b = 0, 1
while a < 10:
    print(a, end = ' ')
    a, b = b, a + b

# import the iris module and show the classes in this namespace
import iris
print('\nInterSystems IRIS classes in this namespace:')
status = iris._SYSTEM.OBJ.ShowClasses()
print(status)
```

You could run test.py from the command line, as follows:

```
C:\InterSystems\IRIS\bin>set IRISUSERNAME = <username>

C:\InterSystems\IRIS\bin>set IRISPASSWORD = <password>

C:\InterSystems\IRIS\bin>set IRISNAMESPACE = USER

C:\InterSystems\IRIS\bin>irispython \python\test.py
Fibonacci series:
0 1 1 2 3 5 8
InterSystems IRIS classes in this namespace:
User.Company
User.Person
1
```

On UNIX-based systems, use **export** instead of **set**.

```
/InterSystems/IRIS/bin$ export IRISUSERNAME=<username>
/InterSystems/IRIS/bin$ export IRISPASSWORD=<password>
/InterSystems/IRIS/bin$ export IRISNAMESPACE=USER
/InterSystems/IRIS/bin$ ./irispython /python/test.py
Fibonacci series:
0 1 1 2 3 5 8
InterSystems IRIS classes in this namespace:
User.Company
User.Person
1
```

Note: If you try to run `import iris` and see a message saying `IRIS_ACCESSDENIED`, enable `%Service_Callin`. In the Management Portal, go to **System Administration** > **Security** > **Services**, select **%Service_CallIn**, and check the **Service Enabled** box.

3.3 In a Method in an InterSystems IRIS Class

You can write Python methods in an InterSystems IRIS class by using the `Language` keyword. You can then call the method as you would call a method written in ObjectScript.

For example, take the following class with a class method written in Python:

```
Class User.EmbeddedPython
{

/// Description
ClassMethod Test() As %Status [ Language = python ]
{
    # print the members of the Fibonacci series that are less than 10
    print('Fibonacci series:')
    a, b = 0, 1
    while a < 10:
        print(a, end = ' ')
        a, b = b, a + b

    # import the iris module and show the classes in this namespace
    import iris
    print('\nInterSystems IRIS classes in this namespace:')
    status = iris._SYSTEM.OBJ.ShowClasses()
    return status
}

}
```

You can call this method from ObjectScript:

```
USER>set status = ##class(User.EmbeddedPython).Test()
Fibonacci series:
0 1 1 2 3 5 8
InterSystems IRIS classes in this namespace:
User.Company
User.EmbeddedPython
User.Person

USER>write status
1
```

Or from Python:

```
>>> import iris
>>> status = iris.User.EmbeddedPython.Test()
Fibonacci series:
0 1 1 2 3 5 8
InterSystems IRIS classes in this namespace:
User.Company
User.EmbeddedPython
User.Person
>>> print(status)
1
```

3.4 In SQL Functions and Stored Procedures

You can also write a SQL function or stored procedure using Embedded Python by specifying the argument LANGUAGE PYTHON in the **CREATE** statement, as is shown below:

```
CREATE FUNCTION tzconvert(dt TIMESTAMP, tzfrom VARCHAR, tzto VARCHAR)
    RETURNS TIMESTAMP
    LANGUAGE PYTHON
{
    from datetime import datetime
    from dateutil import parser, tz
    d = parser.parse(dt)
    if (tzfrom is not None):
        tzf = tz.gettz(tzfrom)
        d = d.replace(tzinfo = tzf)
    return d.astimezone(tz.gettz(tzto)).strftime("%Y-%m-%d %H:%M:%S")
}
```

The code uses functions from the Python datetime and dateutil modules.

Note: On some platforms, the datetime and dateutil modules may not be installed by default. If you run this example and get a ModuleNotFoundError, install the missing modules as described in Install Python Packages.

The following **SELECT** statement calls the SQL function, converting the current date/time from Eastern time to Coordinated Universal Time (UTC).

```
SELECT tzconvert(now(), 'US/Eastern', 'UTC')
```

The function returns something like:

```
2021-10-19 15:10:05
```

4

Call Embedded Python Code from ObjectScript

The section details several ways to call Embedded Python code from ObjectScript:

- Use a Python package

- Call a method in an InterSystems IRIS class written in Python

- Run an SQL function or stored procedure written in Python

- Run an arbitrary Python command

In some cases, you can call the Python code much the same way as you would call ObjectScript code, while sometimes you need to use the %SYS.Python class to bridge the gap between the two languages. For more information, see Bridge the Gap Between ObjectScript and Embedded Python.

4.1 Use a Python Package

Embedded Python gives you easy access to thousands of useful libraries. Commonly called "packages," these need to be installed from the Python Package Index (PyPI) into the <installdir>/mgr/python directory before they can be used. (For information on how to install a package, see Install Python Packages.)

For example, the ReportLab Toolkit is an open source library for generating PDFs and graphics.

After installing a package, you can use the **Import()** method of the %SYS.Python class to use it in your ObjectScript code.

Given a file location, the following ObjectScript method, **CreateSamplePDF()**, creates a sample PDF file and saves it to that location.

```
Class Demo.PDF
{

ClassMethod CreateSamplePDF(fileloc As %String) As %Status
{
    set canvaslib = ##class(%SYS.Python).Import("reportlab.pdfgen.canvas")
    set canvas = canvaslib.Canvas(fileloc)
    do canvas.drawImage("C:\Sample\isc.png", 150, 600)
    do canvas.drawImage("C:\Sample\python.png", 150, 200)
    do canvas.setFont("Helvetica-Bold", 24)
    do canvas.drawString(25, 450, "InterSystems IRIS & Python. Perfect Together.")
    do canvas.save()
}

}
```

The first line of the method imports the canvas.py file from the `pdfgen` subpackage of ReportLab. The second line of code instantiates a Canvas object and then proceeds to call its methods, much the way it would call the methods of any InterSystems IRIS object.

You can then call the method in the usual way:

```
do ##class(Demo.PDF).CreateSamplePDF("C:\Sample\hello.pdf")
```

The following PDF is generated and saved at the specified location:

If you have written your own Python packages or modules, you can put them in <installdir>/mgr/python and import them from ObjectScript in the same way.

4.2 Call a Method of an InterSystems IRIS Class Written in Python

You can write a method in an InterSystems IRIS class using Embedded Python and then call it from ObjectScript in the same way you would call a method written in ObjectScript.

The next example uses the `usaddress-scourgify` package. (For information on how to install a package, see Install Python Packages.)

The demo class below contains properties for the parts of a U.S. address and a method, written in Python, that uses `usaddress-scourgify` to normalize an address according to the U.S. Postal Service standard.

```
Class Demo.Address Extends %Library.Persistent
{

Property AddressLine1 As %String;

Property AddressLine2 As %String;

Property City As %String;

Property State As %String;

Property PostalCode As %String;

Method Normalize(addr As %String) [ Language = python ]
{

    from scourgify import normalize_address_record
    normalized = normalize_address_record(addr)

    self.AddressLine1 = normalized['address_line_1']
    self.AddressLine2 = normalized['address_line_2']
    self.City = normalized['city']
    self.State = normalized['state']
    self.PostalCode = normalized['postal_code']
}

}
```

Given a address string as input, the **Normalize()** instance method of the class normalizes the address and stores each part in the various properties of a Demo.Address object.

You can call the method as follows:

```
USER>set a = ##class(Demo.Address).%New()

USER>do a.Normalize("One Memorial Drive, 8th Floor, Cambridge, Massachusetts 02142")

USER>zwrite a
a=3@Demo.Address  <OREF>
+---------------- general information ---------------
|       oref value: 3
|       class name: Demo.Address
| reference count: 2
+---------------- attribute values -----------------
|       %Concurrency = 1  <Set>
|       AddressLine1 = "ONE MEMORIAL DR"
|       AddressLine2 = "FL 8TH"
|              City = "CAMBRIDGE"
|         PostalCode = "02142"
|             State = "MA"
+---------------------------------------------------
```

4.3 Run an SQL Function or Stored Procedure Written in Python

When you create a SQL function or stored procedure using Embedded Python, InterSystems IRIS projects a class with a method that can be called from ObjectScript as you would any other method.

For example, the SQL function **tzconvert()** from the example earlier in this document generates a class User.functzconvert, which has a **tzconvert()** method. Call it from ObjectScript as follows:

```
USER>zwrite ##class(User.functzconvert).tzconvert($zdatetime($h,3),"US/Eastern","UTC")
"2021-10-20 15:09:26"
```

Here, **$zdatetime($h,3)** is used to convert the current date and time from **$HOROLOG** format to ODBC date format.

4.4 Run an Arbitrary Python Command

Sometimes, when you are developing or testing Embedded Python code, it can be helpful to run an arbitrary Python command from ObjectScript. You can do this with the **Run()** method of the %SYS.Python class.

Perhaps you want to test the **normalize_address_record()** function from the usaddress_scourgify package used earlier in this document, and you don't remember how it works. You can use the **%SYS.Python.Run()** method to output the help for the function from the Terminal as follows:

```
USER>set rslt = ##class(%SYS.Python).Run("from scourgify import normalize_address_record")

USER>set rslt = ##class(%SYS.Python).Run("help(normalize_address_record)")
Help on function normalize_address_record in module scourgify.normalize:
normalize_address_record(address, addr_map=None, addtl_funcs=None, strict=True)
    Normalize an address according to USPS pub. 28 standards.

    Takes an address string, or a dict-like with standard address fields
    (address_line_1, address_line_2, city, state, postal_code), removes
    unacceptable special characters, extra spaces, predictable abnormal
    character sub-strings and phrases, abbreviates directional indicators
    and street types.  If applicable, line 2 address elements (ie: Apt, Unit)
    are separated from line 1 inputs.
.
.
.
```

The **%SYS.Python.Run()** method returns 0 on success or -1 on failure.

5

Call InterSystems IRIS from Embedded Python

The key to calling InterSystems IRIS from Embedded Python is the `iris` Python module. The `iris` module provides a number of methods that unlock the functionality of InterSystems IRIS from Embedded Python and also allows you to access any InterSystems IRIS class as if it were a Python class.

This section provides a few basic examples of how to use the capabilities of InterSystems IRIS from Embedded Python:

- Use the iris Module

- Use an InterSystems IRIS Class

- Use InterSystems SQL

- Run an Arbitrary ObjectScript Command

For additional examples, see Call the InterSystems IRIS APIs from Python.

See InterSystems IRIS Python Module Reference for detailed descriptions of the most important APIs exposed by the `iris` module.

5.1 Use the iris Module

The `iris` Python module enables you to interact with InterSystems IRIS to use its transaction processing functionality, access globals (the underlying data structure for all storage in InterSystems IRIS), access InterSystems IRIS APIs and classes, or call utility methods that help Embedded Python operate smoothly with ObjectScript.

Use the `iris` module from Embedded Python just as you would any Python module, by using the **import** command:

```
import iris
```

Note: You do not need to import the `iris` module explicitly when running the Python shell using the Shell() method of the %SYS.Python class. In this context only, the import is done for you behind the scenes.

After you import the `iris` module, call its methods just as you would with any other Python module.

For example, the **tstart()** method is used to indicate the start of a transaction in InterSystems IRIS:

```
iris.tstart()
```

5.2 Use an InterSystems IRIS Class

You can use the `iris` module to return a reference to an InterSystems IRIS class. This gives you easy access to both built-in system classes and any custom classes you or someone on your team may have written.

If a system class has a name starting with a percent sign (%), it means that you can access the class from any namespace. If a system class does not begin with a percent sign, you must access the class from the `%SYS` namespace.

For instance, the class %Regex.Matcher creates an object that does pattern matching using regular expressions. The following example finds all of the area codes in a string and replaces them with the area code 212.

Using the method **%New()** is the standard way for creating a new instance of an InterSystems IRIS class.

Remember, class names and method names in Python cannot use the `%` character, so you must substitute an underscore (_), instead.

```
>>> import iris
>>> matcher = iris._Regex.Matcher
>>> m = matcher._New('\(((\d{3})\)')
>>> m.Text = '(617) 555-1212, (202) 555-1313, (415) 555-1414'
>>> print(m.ReplaceAll('(212)'))
(212) 555-1212, (212) 555-1313, (212) 555-1414
```

You can find out more about this and other system classes by looking in the InterSystems Class Reference.

Custom classes work much the same way. They can create a logical grouping for a number of related methods, and often they extend the class %Persistent (short for %Library.Persistent), which enables you to store objects of the class in the InterSystems database.

You can find sample classes (including sample data) to learn from in 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.

Following is an excerpt from the class Sample.Person:

```
Class Sample.Person Extends (%Persistent, %Populate, %XML.Adaptor)
{
/// Person's name.
Property Name As %String(POPSPEC = "Name()") [ Required ];

/// Person's Social Security number. This is validated using pattern match.
Property SSN As %String(PATTERN = "3N1""-""2N1""-""4N") [ Required ];

/// Person's Date of Birth.
Property DOB As %Date(POPSPEC = "Date()");

/// A collection of strings representing the person's favorite colors.
Property FavoriteColors As list Of %String(JAVATYPE = "java.util.List",
POPSPEC = "ValueList("",Red,Orange,Yellow,Green,Blue,Purple,Black,White""):2");

/// Person's age.
/// This is a calculated field whose value is derived from DOB.
Property Age As %Integer [ Calculated, SqlComputeCode = { Set
{Age}=##class(Sample.Person).CurrentAge({DOB})
}, SqlComputed, SqlComputeOnChange = DOB ];

/// This class method calculates a current age given a date of birth date.
/// This method is used by the Age calculated field.
ClassMethod CurrentAge(date As %Date = "") As %Integer [ CodeMode = expression ]
{
$Select(date="":"",1:($ZD($H,8)-$ZD(date,8)\10000))
}
}
```

You can see that a person object has several properties, including `Name`, `SSN`, `DOB`, `FavoriteColors`, and `Age`. The `DOB` property is the person's birth date in $HOROLOG format. The `FavoriteColors` property is a list of strings. The `Age` property is a computed field that uses a method **CurrentAge()** to calculate it from the person's date of birth.

The following example changes the current namespace to the `SAMPLES` namespace, creates a new instance of Sample.Person, sets its properties, and saves it to the database:

```
>>> import iris
>>> iris.system.Process.SetNamespace('SAMPLES')
'SAMPLES'
>>> p = iris.Sample.Person._New()
>>> p.Name = 'Doe,John'
>>> p.SSN = '000-00-0000'
>>> p.DOB = iris._Library.Date.DisplayToLogical('04/24/1999')
>>> print(p.Age)
25
>>> p.FavoriteColors.Insert('Red')
1
>>> p.FavoriteColors.Insert('Blue')
1
>>> p._Save()
1
>>> print(p._Id())
201
```

As you saw earlier, the **%New()** method creates an instance of Sample.Person, while the **%Save()** method saves it to the database. The **%Library.Date.DisplayToLogical()** method takes a date string and converts it to **$HOROLOG** format for storage. Finally, the **Insert()** method of a list property inserts a new element into a list.

Notice that the **Insert()** and **%Save()** methods in this example all return 1. These are examples of InterSystems IRIS status codes, and a 1 indicates that no error occurred during the execution of a method. Often, you will want to check this status code using iris.check_status() to handle any possible error cases.

InterSystems IRIS automatically assigns an ID to the object when it is stored, in this case, 201.

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.

5.3 Use InterSystems SQL

Persistent classes in InterSystems IRIS are projected to SQL, allowing you to access the data using a query using InterSystems SQL. The class Sample.Person, described above, projects to a table of the same name.

The example below selects the ID, name, and age of all persons having red as one of their favorite colors:

```
>>> sql = iris.sql.prepare('SELECT ID, Name, Age FROM Sample.Person WHERE FOR SOME
%ELEMENT(FavoriteColors) (%Value = ?)')
>>> rs = sql.execute('Red')
>>> for idx, row in enumerate(rs):
...     print(f'{idx} = {row}')
...
0 = ['14', 'Roentgen,Sally S.', 78]
1 = ['30', 'Vanzetti,Jane A.', 73]
2 = ['38', 'Vanzetti,Heloisa W.', 8]
3 = ['61', 'Gibbs,Zoe J.', 81]
4 = ['72', 'Klingman,Pat E.', 55]
5 = ['89', 'Beatty,Mario Q.', 91]
6 = ['91', 'Ott,Jeff Z.', 54]
7 = ['97', 'Van De Griek,Kirsten B.', 49]
8 = ['101', 'Newton,Barbara A.', 51]
9 = ['105', 'Hertz,Yan M.', 90]
10 = ['129', 'Leiberman,Mo U.', 54]
11 = ['133', 'Zubik,George X.', 95]
12 = ['137', 'Jung,Lawrence T.', 83]
13 = ['147', 'Ubertini,Nataliya D.', 71]
14 = ['150', 'Ravazzolo,Sally S.', 84]
15 = ['187', 'Nichols,Terry F.', 64]
16 = ['196', 'Houseman,David D.', 97]
17 = ['200', 'Ng,James T.', 36]
18 = ['201', 'Doe,John', 25]
```

You can see that 19 records are returned, including that of John Doe, whom we entered earlier.

5.4 Run an Arbitrary ObjectScript Command

Sometimes it can be handy to be able to run an arbitrary command in ObjectScript. You may want to test an ObjectScript command from Embedded Python, or you may want to access an ObjectScript function or "special variable."

If you ever need to contact customer support, sometimes they will ask you for the InterSystems IRIS version string, **$ZVERSION** (or **$ZV** for short). This special variable contains a string that has the version of InterSystems IRIS you are using and additional details, like build number.

The following example shows how to write the value of **$ZVERSION** from Embedded Python:

```
>>> iris.execute('write $zversion, !')
IRIS for Windows (x86-64) 2024.1 (Build 267_2U) Tue Apr 30 2024 16:35:10 EDT
```

Or you can return the value of **$ZVERSION** and put its value in a Python variable:

```
>>> zv = iris.execute('return $zversion')
>>> print(zv)
IRIS for Windows (x86-64) 2024.1 (Build 267_2U) Tue Apr 30 2024 16:35:10 EDT
```

In this case, there is an equivalent way to get **$ZVERSION** using an API, but this will not always be true for other special variables and functions.

```
>>> print(iris.system.Version.GetVersion())
IRIS for Windows (x86-64) 2024.1 (Build 267_2U) Tue Apr 30 2024 16:35:10 EDT
```

6

Bridge the Gap Between ObjectScript and Embedded Python

Because of the differences between the ObjectScript and Python languages, you will need to know a few pieces of information that will help you bridge the gap between the languages.

From the ObjectScript side, the %SYS.Python class allows you to use Python from ObjectScript. See the InterSystems IRIS class reference for more information.

From the Python side, the iris module allows you to use ObjectScript from Python. From Python, type help(iris) for a list of its methods and functions, or see InterSystems IRIS Python Module Reference for more details.

6.1 Access an InterSystems IRIS Class

The iris module gives you access to any InterSystems IRIS class from Embedded Python.

There are two ways of referring to an InterSystems IRIS class. The recommended syntax is iris.<classname>. You can also use iris.cls('<classname>'), and this syntax is required in InterSystems IRIS 2021.1.x through 2024.1.x.

The following examples are equivalent:

```
>>> matcher = iris._Regex.Matcher          #Recomended syntax
>>> matcher = iris.cls('%Regex.Matcher') #Older syntax (required in 2021.1.x through 2024.1.x)
```

If an InterSystems IRIS class name contains a percent sign (as in %Regex.Matcher), substitute an underscore in Embedded Python (as in _Regex.Matcher). In the case of iris.cls('%Regex.Matcher'), the name of the class is contained in a string argument, so a substitution is not needed.

The following examples are equivalent:

```
>>> p = iris.Sample.Person._New()          #Recomended syntax
>>> p = iris.cls('Sample.Person')._New() #Older syntax (required in 2021.1.x through 2024.1.x)
```

If an InterSystems IRIS method name contains a percent sign (as in **%New()**), substitute an underscore in Embedded Python (as in **_New()**).

6.2 Refer to the Current InterSystems IRIS Class

When you want to refer to class members of the current InterSystems IRIS class in Embedded Python, you can often use `self`, just as you would with a Python class. This is the case for accessing properties or instance methods, for example. However, accessing other class members (such as class parameters or class methods) require a different syntax. For an overview of these and other class members, see Basics Contents of a Class Definition.

The following very simple InterSystems IRIS class definition, User.RedSoxFan can be used to describe Red Sox fans:

```
Class User.RedSoxFan Extends %Library.Persistent
{

Property Name As %String;

Parameter FAVORITETEAM = "Boston Red Sox";

ClassMethod FunFact() [ Language = python ]
{
print('Did you know baseball opening day is in three weeks!')
}

Method Greeting() [ Language = python ]
{
import iris
# option 1
print('Hello, my name is ' + self.Name + '. My favorite team is the ' +
iris.User.RedSoxFan._GetParameter('FAVORITETEAM') + '.')
# option 2
# print('Hello, my name is ' + self.Name + '. My favorite team is the ' +
iris.cls(__name__)._GetParameter('FAVORITETEAM') + '.')
}

Method Conversation() [ Language = python ]
{
import iris
self.Greeting()
# option 1
iris.User.RedSoxFan.FunFact()
# option 2
# iris.cls(__name__).FunFact()
}
}
```

Red Sox fans have a `Name` (a property, or value that is unique to each instance of the class) and a `FAVORITETEAM` (a class parameter, or value that is constant across all instances of the class). The class also has an instance method, **Greeting()**, that allows a member to print a greeting message to introduce themselves.

Looking at the code, you can see that the **Greeting()** method uses `self.Name` to access the `Name` property of the current fan. But to access the `FAVORITETEAM` parameter, you need to call the built-in method **%GetParameter()**. And instead of using `self` to reference the current class, you need to use `iris.User.RedSoxFan` or `iris.cls(__name__)`. In the second option, `__name__` is a built-in variable in Python that evaluates to the name of the current class. It uses the iris.cls() method, which takes a class name and returns a reference to the class.

In this example, **Greeting()** is an instance method, while **FunFact()** is a class method. You can think of an instance method as belonging to an instance of the class, while a class method belongs to the class. And in Embedded Python, the syntax for calling an instance method and a class method differ when you call them from within the class. Looking at the **Conversation()** method, you can see that `self` can be used to call the instance method, while `iris.User.RedSoxFan` or `iris.cls(__name__)` are used to call the class method.

The following example shows the class in action:

```
>>> f = iris.User.RedSoxFan._New()
>>> f.Name = "Jane Doe"
>>> f.Conversation()
Hello, my name is Jane Doe. My favorite team is the Boston Red Sox.
Did you know baseball opening day is in three weeks!
```

Of course, if you call a method from outside its class, you need to specify the class name:

```
>>> iris.User.RedSoxFan.FunFact()
Did you know baseball opening day is in three weeks!
>>> iris.User.RedSoxFan.Conversation(f)
Hello, my name is Jane Doe. My favorite team is the Boston Red Sox.
Did you know baseball opening day is in three weeks!
```

6.3 Use ObjectScript and Python Identifier Names

The rules for naming identifiers are different between ObjectScript and Python. For example, the underscore (_) is allowed in Python method names, and in fact is widely used for the so-called "dunder" methods and attributes ("dunder" is short for "double underscore"), such as __getitem__ or __class__. Dunder methods enable instances of a class to interact with Python's built-in functions and operators. To use such identifiers from ObjectScript, enclose them in double quotes:

```
USER>set mylist = builtins.list()

USER>zwrite mylist."__class__"
2@%SYS.Python  ; <class list>  ; <OREF>
```

Conversely, InterSystems IRIS methods often begin with a percent sign (%). such as **%New()** or **%Save()**. To use such identifiers from Python, replace the percent sign with an underscore. If you have a persistent class User.Person, the following line of Python code creates a new Person object.

```
>>> import iris
>>> p = iris.User.Person._New()
```

6.4 Use Python Builtin Functions

The builtins package is loaded automatically when the Python interpreter starts, and it contains all of the language's built-in identifiers, such as the base object class and all of the built-in datatype classes, exceptions classes, functions, and constants.

You can import this package into ObjectScript to gain access to all of these identifiers as follows:

```
set builtins = ##class(%SYS.Python).Import("builtins")
```

The Python **print()** function is actually a method of the builtins module, so you can now use this function from ObjectScript:

```
USER>do builtins.print("hello world!")
hello world!
```

You can then use the **zwrite** command to examine the builtins object, and since it is a Python object, it uses the **str()** method of the builtins package to get a string representation of that object. For example:

```
USER>zwrite builtins
builtins=5@%SYS.Python  ; <module 'builtins' (built-in)>  ; <OREF>
```

By the same token, you can create a Python list using the method **builtins.list()**. The example below creates an empty list:

```
USER>set list = builtins.list()

USER>zwrite list
list=5@%SYS.Python  ; []  ; <OREF>
```

You can use the **builtins.type()** method to see what Python type the variable `list` is:

```
USER>zwrite builtins.type(list)
3@%SYS.Python  ; <class 'list'>  ; <OREF>
```

Interestingly, the **list()** method actually returns an instance of Python's class object that represents a list. You can see what methods the list class has by using the **dir()** method on the list object:

```
USER>zwrite builtins.dir(list)
3@%SYS.Python  ; ['__add__', '__class__', '__class_getitem__', '__contains__', '__delattr__',
'__delitem__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__',
'__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__',

'__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__','__repr__', '__reversed__',
'__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append','clear',

'copy', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']  ; <OREF>
```

Likewise, you can use the **help()** method to get help on the list object.

```
USER>do builtins.help(list)
Help on list object:
class list(object)
 |  list(iterable=(), /)
 |
 |  Built-in mutable sequence.
 |
 |  If no argument is given, the constructor creates a new empty list.
 |  The argument must be an iterable if specified.
 |
 |  Methods defined here:
 |
 |  __add__(self, value, /)
 |      Return self+value.
 |
 |  __contains__(self, key, /)
 |      Return key in self.
 |
 |  __delitem__(self, key, /)
 |      Delete self[key].
.
.
.
```

Note: Instead of importing the `builtins` module into ObjectScript, you can call the **Builtins()** method of the %SYS.Python class.

6.5 Use Keyword or Named Arguments

A common practice in Python is to use keyword arguments (also called "named arguments") when defining a method. This makes it easy to drop arguments when not needed or to specify arguments according to their names, not their positions. As an example, take the following simple Python method:

```
def mymethod(foo=1, bar=2, baz="three"):
    print(f"foo={foo}, bar={bar}, baz={baz}")
```

Since InterSystems IRIS does not have the concept of keyword arguments, you need to create a dynamic object to hold the keyword/value pairs, for example:

```
USER>set args = { "bar": 123, "foo": "foo"}
```

If the method **mymethod()** were in a module called mymodule.py in the directory <installdir>/mgr/python, you could import it into ObjectScript and then call it, as follows:

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

USER>set args = {"bar": 123, "foo": "foo"}

USER>do obj.mymethod(args...)
foo=foo, bar=123, baz=three
```

Since baz was not passed in to the method, it is assigned the value of "three" by default.

6.6 Pass Arguments By Reference

Arguments in methods written in ObjectScript can be passed by value or by reference. In the method below, the ByRef keyword in front of the second and third arguments in the signature indicates that they are intended to be passed by reference.

```
ClassMethod SandwichSwitch(bread As %String, ByRef filling1 As %String, ByRef filling2 As %String)
{
    set bread = "whole wheat"
    set filling1 = "almond butter"
    set filling2 = "cherry preserves"
}
```

Assume this method is contained in a class called User.EmbeddedPython. When calling the method from ObjectScript, place a period before an argument to pass it by reference, as shown below:

```
USER>set arg1 = "white bread"

USER>set arg2 = "peanut butter"

USER>set arg3 = "grape jelly"

USER>do ##class(User.EmbeddedPython).SandwichSwitch(arg1, .arg2, .arg3)

USER>write arg1
white bread
USER>write arg2
almond butter
USER>write arg3
cherry preserves
```

From the output, you can see that the value of the variable arg1 remains the same after calling **SandwichSwitch**(), while the values of the variables arg2 and arg3 have changed.

Since Python does not support call by reference natively, you need to use the **iris.ref()** method to create a reference to pass to the method for each argument to be passed by reference:

```
>>> import iris
>>> arg1 = 'white bread'
>>> arg2 = iris.ref('peanut butter')
>>> arg3 = iris.ref('grape jelly')
>>> iris.User.EmbeddedPython.SandwichSwitch(arg1, arg2, arg3)
>>> arg1
'white bread'
>>> arg2.value
'almond butter'
>>> arg3.value
'cherry preserves'
```

You can use the value property to access the values of arg2 and arg3 and see that they have changed following the call to the method.

ObjectScript also has a keyword Output, which indicates that an argument is passed by reference and it is expected that this argument is to be used as an output, without any incoming value. From Python, use the **iris.ref**() method to pass the argument the same way as you would for a ByRef argument.

Note: While passing arguments by reference is a feature of ObjectScript methods, there is no equivalent way to pass arguments by reference to a method written in Python. The `ByRef` and `Output` keywords in the signature of an ObjectScript method are conventions used to indicate to the user that the method expects that an argument is to be passed by reference. In fact, `ByRef` and `Output` have no actual function and are ignored by the compiler. Adding `ByRef` or `Output` to the signature of a method written in Python results in a compiler error.

6.7 Pass Values for True, False, and None

The **%SYS.Python** class has the methods **True()**, **False()**, and **None()**, which represent the Python identifiers `True`, `False`, and `None`, respectively.

For example:

```
USER>zwrite ##class(%SYS.Python).True()
2@%SYS.Python  ; True  ; <OREF>
```

These methods are useful if you need to pass `True`, `False`, and `None` to a Python method. The following example uses the method shown in Keyword or Named Arguments.

```
USER>do obj.mymethod(##class(%SYS.Python).True(), ##class(%SYS.Python).False(),
##class(%SYS.Python).None())
foo=True, bar=False, baz=None
```

If you pass unnamed arguments to a Python method that expects keyword arguments, Python handles them in the order they are passed in.

Note that you do not need to use the methods **True()**, **False()**, and **None()** when examining the values returned by a Python method to ObjectScript.

Say the Python module `mymodule` also has a method **isgreaterthan()**, which is defined as follows:

```
def isgreaterthan(a, b):
    return a > b
```

When run in Python, you can see that the method returns `True` if the argument `a` is greater than `b`, and `False` otherwise:

```
>>> mymodule.isgreaterthan(5, 4)
True
```

However, when called from ObjectScript, the returned value is `1`, not the Python identifier `True`:

```
USER>zwrite obj.isgreaterthan(5, 4)
1
```

6.8 Use Dictionaries

In Python, dictionaries are commonly used to store data in key-value pairs, for example:

```
>>> mycar = {
...     'make': 'Toyota',
...     'model': 'RAV4',
...     'color': 'blue'
... }
>>> print(mycar)
{'make': 'Toyota', 'model': 'RAV4', 'color': 'blue'}
>>> print(mycar['color'])
blue
```

You can use the method **iris.arrayref()** to place the contents of the dictionary `mycar` into an ObjectScript array and return a reference to that array:

```
>>> a = iris.arrayref(mycar)
>>> print(a.value)
{'color': 'blue', 'make': 'Toyota', 'model': 'RAV4'}
>>> print(a.value['color'])
blue
```

You can then pass the array to an ObjectScript method.

For example, assume you have an InterSystems IRIS class called User.ArrayTest that has a method **WriteContents()** that writes the contents of an array:

```
ClassMethod WriteContents(myArray) [ Language = objectscript ]
{
    zwrite myArray
}
```

Then, you can call **WriteContents()** as follows:

```
>>> iris.User.ArrayTest.WriteContents(a)
myArray("color")="blue"
myArray("make")="Toyota"
myArray("model")="RAV4"
```

For more information, see iris.arrayref().

On the ObjectScript side, you can manipulate Python dictionaries using the **dict()** method of the Python `builtins` module:

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

USER>do mycar.setdefault("make", "Toyota")

USER>do mycar.setdefault("model", "RAV4")

USER>do mycar.setdefault("color", "blue")

USER>zwrite mycar
mycar=2@%SYS.Python  ; {'make': 'Toyota', 'model': 'RAV4', 'color': 'blue'}  ; <OREF>

USER>write mycar."__getitem__"("color")
blue
```

The example above uses the dictionary method **setdefault()** to set the value of a key and **__getitem__()** to get the value of a key.

6.9 Use Lists

In Python, lists store collections of values, but without keys. Items in a list are accessed by their index.

```
>>> fruits = ['apple', 'banana', 'cherry']
>>> print(fruits)
['apple', 'banana', 'cherry']
>>> print(fruits[0])
apple
```

In ObjectScript, you can work with Python lists using the **list()** method of the Python `builtins` module:

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

USER>do l.append("apple")

USER>do l.append("banana")

USER>do l.append("cherry")

USER>zwrite l
l=13@%SYS.Python  ; ['apple', 'banana', 'cherry']  ; <OREF>

USER>write l."__getitem__"(0)
apple
```

The example above uses the list method **append()** to append an item to the list and **__getitem__()** to get the value at a given index. (Python lists are zero based.)

If you want to convert an ObjectScript list to a Python list, you can use the **ToList()** and **ToListTyped()** methods in %SYS.Python. Given an ObjectScript list, **ToList()** returns a Python list that contains the same data. Given an ObjectScript list containing data and a second ObjectScript list containing integer ODBC data type codes, **ToListTyped()** returns a Python list that contains the same data as the first list, with each item having the data types specified in the second list.

Note: For a table of ODBC data types, see Integer Codes for Data Types.

Some ODBC data types may translate to the same Python data type.

Some data types require the Python package `numpy` to be installed.

In the example below, a Python method **Loop()** in the class **User.Lists** iterates over the items in a list and prints their value and data type.

```
ClassMethod Loop(pyList) [ Language = python ]
{
    for x in pyList:
        print(x, type(x))
}
```

You can then use **ToList()** and **ToListTyped()** as follows:

```
USER>set clist = $listbuild(123, 456.789, "hello world")

USER>set plist = ##class(%SYS.Python).ToList(clist)

USER>do ##class(User.Lists).Loop(plist)
123 <class 'int'>
456.789 <class 'float'>
hello world <class 'str'>

USER>set clist = $listbuild(42, 42, 42, 42)

USER>set tlist = $listbuild(-7, 2, 3, 4)

USER>set plist = ##class(%SYS.Python).ToListTyped(clist, tlist)

USER>do ##class(User.Lists).Loop(plist)
True <class 'bool'>
42.0 <class 'float'>
42 <class 'decimal.Decimal'>
42 <class 'int'>
```

6.10 Use Globals

Most of the time, you will probably access data stored in InterSystems IRIS either by using SQL or by using persistent classes and their properties and methods. However, there may be times when you want to directly access the underlying

native persistent data structures, called globals. This is particularly true if you are accessing legacy data or if you are storing schema-less data that doesn't lend itself to SQL tables or persistent classes.

Though it is an oversimplification, you can think of a global as a dictionary of key-value pairs. (See Introduction to Globals for a more accurate description.)

Consider the following class, which has two class methods written in Python:

```
Class User.Globals
{
ClassMethod SetSquares(x) [ Language = python ]
{
    import iris
    square = iris.gref('^square')
    for key in range(1, x):
        value = key * key
        square.set([key], value)
}
ClassMethod PrintSquares() [ Language = python ]
{
    import iris
    square = iris.gref('^square')
    key = ''
    while True:
        key = square.order([key])
        if key == None:
            break
        print('The square of ' + str(key) + ' is ' + str(square.get([key])))
}
}
```

The method **SetSquares()** loops over a range of keys, storing the square of each key at each node of the global ^square. The method **PrintSquares()** traverses the global and prints each key and the value stored at the key.

Let's launch the Python shell, instantiate the class, and run the code to see how it works.

```
USER>do ##class(%SYS.Python).Shell()

Python 3.9.5 (default, May 31 2022, 12:35:47) [MSC v.1927 64 bit (AMD64)] on win32
Type quit() or Ctrl-D to exit this shell.
>>> g = iris.User.Globals
>>> g.SetSquares(6)
>>> g.PrintSquares()
The square of 1 is 1
The square of 2 is 4
The square of 3 is 9
The square of 4 is 16
The square of 5 is 25
```

Now, let's look at how some of the methods of the built-in `iris` module allow us to access globals.

In method **SetSquares()**, the statement `square = iris.gref('^square')` returns a reference to the global ^square, also known as a *gref*:

```
>>> square = iris.gref('^square')
```

The statement `square.set([key], value)` sets the node of ^square with key `key` to the value `value`, for example you can set node 12 of ^square to the value 144:

```
>>> square.set([12], 144)
```

You can also set the node of a global with the following shorter syntax:

```
>>> square[13] = 169
```

In method **PrintSquares()**, the statement `key = square.order([key])` takes a key as input and returns the next key in the global, similar to the **$ORDER** function in ObjectScript. A common technique for a traversing a global is to continue

using **order()** until it returns None, indicating that no more keys remain. Keys do not need to be consecutive, so **order()** returns the next key even if there are gaps between keys:

```
>>> key = 5
>>> key = square.order([key])
>>> print(key)
12
```

Then, `square.get([key])` takes a key as input and returns the value at that key in the global:

```
>>> print(square.get([key]))
144
```

Again, you can use the following shorter syntax:

```
>>> print(square[13])
169
```

Note that nodes in a global don't have to have a key. The following statement stores a string at the root node of `^square`:

```
>>> square[None] = 'Table of squares'
```

To show that these Python commands did in fact store values in the global, exit the Python shell and then use the **zwrite** command in ObjectScript to print the contents of `^square`:

```
>>> quit()

USER>zwrite ^square
^square="Table of squares"
^square(1)=1
^square(2)=4
^square(3)=9
^square(4)=16
^square(5)=25
^square(12)=144
^square(13)=169
```

See Global Reference API for more details on how to access and manipulate globals from Python.

6.11 Change Namespaces

InterSystems IRIS has the concept of namespaces, each of which has its own databases for storing code and data. This makes it easy to keep the code and data of one namespace separate from the code and data of another namespace. For example, if one namespace has a global with a certain name, another namespace can use a global with the same name without the danger of conflicting with the other global.

If you have two namespaces, NSONE and NSTWO, you could create a global called `^myFavorite` in NSONE, using ObjectScript in Terminal, as shown below. Then you could set the `$namespace` special variable to change to NSTWO and create a separate global called `^myFavorite` in that namespace. (To replicate this example, you can configure these two namespaces on your InterSystems IRIS instance or use two namespaces you already have.)

```
NSONE>set ^myFavorite("fruit") = "apple"

NSONE>set $namespace = "NSTWO"

NSTWO>set ^myFavorite("fruit") = "orange"
```

Here, `^myFavorite("fruit")` has the value `"apple"` in NSONE and the value `"orange"` in NSTWO.

When you call Embedded Python, it inherits the current namespace. We can test this by calling the **NameSpace()** method of the iris.system.Process class from Python, which displays the name of the current namespace, and by confirming that `^myFavorite("fruit") = "orange"`.

```
NSTWO>do ##class(%SYS.Python).Shell()

Python 3.9.5 (default, Jun  2 2023, 14:12:21) [MSC v.1927 64 bit (AMD64)] on win32
Type quit() or Ctrl-D to exit this shell.
>>> iris.system.Process.NameSpace()
'NSTWO'
>>> myfav = iris.gref('^myFavorite')
>>> print(myfav['fruit'])
orange
```

You've seen how to use `$namespace` to change namespaces in ObjectScript. In Embedded Python, you use the **SetNamespace()** method of the iris.system.Process class. For example, you can change to the namespace `NSONE` and confirm that `^myFavorite("fruit") = "apple"`.

```
>>> iris.system.Process.SetNamespace('NSONE')
'NSONE'
>>> myfav = iris.gref('^myFavorite')
>>> print(myfav['fruit'])
apple
```

Finally, when you exit from the Python shell, you remain in namespace `NSONE`.

```
>>> quit()

NSONE>
```

6.12 Run an ObjectScript Routine from Embedded Python

You may encounter older ObjectScript code that uses routines instead of classes and methods and want to call a routine from Embedded Python. In such cases, you can use the method **iris.routine()** from Python.

The following example, when run in the `%SYS` namespace, calls the routine **^SECURITY**:

```
>>> iris.routine('^SECURITY')


1) User setup
2) Role setup
3) Service setup
4) Resource setup
.
.
.
```

If you have a routine **^Math** that has a function **Sum()** that adds two numbers, the following example adds 4 and 3:

```
>>> sum = iris.routine('Sum^Math',4,3)
>>> sum
7
```

6.13 Handle Exceptions

The InterSystems IRIS exception handler can handle Python exceptions and pass them seamlessly to ObjectScript. Building on the earlier Python package example, the following example shows what happens if you try to call **canvas.drawImage()** using a non-existent file. Here, ObjectScript catches the exception in the special variable $zerror:

```
USER>try { do canvas.drawImage("C:\Sample\bad.png", 150, 600) } catch { write "Error: ", $zerror, ! }
Error: <THROW> *%Exception.PythonException <THROW> 230 ^^0^DO canvas.drawImage("W:\Sample\isc.png",
150, 600)
<class 'OSError'>: Cannot open resource "W:\Sample\isc.png" -
```

In this case, `<class 'OSError'>: Cannot open resource "W:\Sample\isc.png"` is the exception passed back from Python.

For more information on $zerror, see $ZERROR (ObjectScript).

For information on raising an ObjectScript status error as a Python exception, see check_status(status).

6.14 Bytes and Strings

Python draws a clear distinction between objects of the "bytes" data type, which are simply sequences of 8-bit bytes, and strings, which are sequences of UTF-8 bytes that represent a string. In Python, bytes objects are never converted in any way, but strings might be converted depending on the character set in use by the host operating system, for example, Latin-1.

InterSystems IRIS makes no distinction between bytes and strings. While InterSystems IRIS supports Unicode strings (UCS-2/UTF-16), any string that contains values of less than 256 could either be a string or bytes. For this reason, the following rules apply when passing strings and bytes to and from Python:

- InterSystems IRIS strings are assumed to be strings and are converted to UTF-8 when passed from ObjectScript to Python.

- Python strings are converted from UTF-8 to InterSystems IRIS strings when passed back to ObjectScript, which may result in wide characters.

- Python bytes objects are returned to ObjectScript as 8-bit strings. If the length of the bytes object exceeds the maximum string length, then a Python bytes object is returned.

- To pass bytes objects to Python from ObjectScript, use the **##class(%SYS.Python).Bytes()** method, which does not convert the underlying InterSystems IRIS string to UTF-8.

- To turn the bytes object back into a string, use the builtins method **builtins.bytes()**.

The following example turns an InterSystems IRIS string to a Python object of type bytes and then converts that object back into a string:

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

USER>set b = ##class(%SYS.Python).Bytes("Hello Bytes!")

USER>zwrite b
b=3@%SYS.Python  ; b'Hello Bytes!'  ; <OREF>

USER>zwrite builtins.type(b)
4@%SYS.Python  ; <class 'bytes'>  ; <OREF>

USER>set s = builtins.bytes(b)

USER>zwrite s
s="Hello Bytes!"

USER>zwrite builtins.type(s)
4@%SYS.Python  ; <class 'str'>  ; <OREF>
```

To construct Python bytes objects bigger than the 3.8MB maximum string length in InterSystems IRIS, you can use a bytearray object and append smaller chunks of bytes using the **extend()** method. Finally, pass the bytearray object into the builtins **bytes()** method to get a bytes representation:

```
USER>set ba = builtins.bytearray()

USER>do ba.extend(##class(%SYS.Python).Bytes("chunk 1"))

USER>do ba.extend(##class(%SYS.Python).Bytes("chunk 2"))

USER>zwrite builtins.bytes(ba)
"chunk 1chunk 2"
```

6.15 Standard Output and Standard Error Mappings

When using Embedded Python, standard output is mapped to the InterSystems IRIS console, which means that the output of any print() statements is sent to the Terminal. Standard error is mapped to the InterSystems IRIS messages.log file, located in the directory <install-dir>/mgr.

As an example, consider this Python method:

```
def divide(a, b):
    try:
        print(a/b)
    except ZeroDivisionError:
        print('Cannot divide by zero')
    except TypeError:
        import sys
        print('Bad argument type', file=sys.stderr)
    except:
        print('Something else went wrong')
```

Assume the method is contained in a module called mymodule.py in the directory <installdir>/mgr/python. Then, if you test this method in Terminal, you might see the following:

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

USER>do obj.divide(5, 0)
Cannot divide by zero

USER>do obj.divide(5, "hello")
```

If you try to divide by zero, the error message is directed to the Terminal, but if you try to divide by a string, the message is sent to messages.log:

```
11/19/21-15:49:33:248 (28804) 0 [Python] Bad argument type
```

Only important messages should be sent to messages.log, to avoid cluttering the file.

7

Use Embedded Python in Interoperability Productions

If you are writing custom business host classes or adapter classes for interoperability productions in InterSystems IRIS, any callback methods must be written in ObjectScript because their signatures make use of `ByRef` and `Output` keywords. A callback method is an inherited method that does nothing by default, but is designed to be implemented by the user. The ObjectScript code in a callback method can, however, make use of Python packages or call other methods implemented in Python.

The following example shows a business operation that takes the string value from an incoming message and uses the Amazon Web Services (AWS) `boto3` SDK for Python to send that string to a phone in a text message via the Amazon Simple Notification Service (SNS). The details of this package is out of scope for this discussion, but you can see in the example that the **OnInit()** and **OnMessage()** callback methods are written in ObjectScript, while the methods **PyInit()** and **SendSMS()** are written in Python.

```
/// Send SMS via AWS SNS
Class dc.opcua.SMS Extends Ens.BusinessOperation
{

Parameter INVOCATION = "Queue";

/// AWS boto3 client
Property client As %SYS.Python;

/// json.dumps reference
Property tojson As %SYS.Python;

/// Phone number to send SMS to
Property phone As %String [ Required ];

Parameter SETTINGS = "phone:SMS";

Method OnMessage(request As Ens.StringContainer, Output response As Ens.StringContainer) As %Status
{
    #dim sc As %Status = $$$OK
    try {
        set response = ##class(Ens.StringContainer).%New(..SendSMS(request.StringValue))
        set code = +{}.%FromJSON(response.StringValue).ResponseMetadata.HTTPStatusCode
        set:(code'=200) sc = $$$ERROR($$$GeneralError, $$$FormatText("Error sending SMS,
            code: %1 (expected 200), text: %2", code, response.StringValue))
    } catch ex {
        set sc   = ex.AsStatus()
    }

    return sc
}

Method SendSMS(msg As %String) [ Language = python ]
{
    response = self.client.publish(PhoneNumber=self.phone, Message=msg)
    return self.tojson(response)
}
```

```
Method OnInit() As %Status
{
    #dim sc As %Status = $$$OK
    try {
        do ..PyInit()
    } catch ex {
        set sc = ex.AsStatus()
    }
    quit sc
}

/// Connect to AWS
Method PyInit() [ Language = python ]
{
    import boto3
    from json import dumps
    self.client = boto3.client("sns")
    self.tojson = dumps
}

}
```

Note: The code in the **OnMessage()** method, above, contains an extra line break for better formatting when printing this document.

The following business service example is known as a poller. In this case, the business service can be set to run at intervals and generates a request (in this case containing a random string value) that is sent to a business process for handling.

```
Class Debug.Service.Poller Extends Ens.BusinessService
{

Property Target As Ens.DataType.ConfigName;

Parameter SETTINGS = "Target:Basic";

Parameter ADAPTER = "Ens.InboundAdapter";

Method OnProcessInput(pInput As %RegisteredObject, Output pOutput As %RegisteredObject,
    ByRef pHint As %String) As %Status [ Language = objectscript ]
{
    set request = ##class(Ens.StringRequest).%New()
    set request.StringValue = ..RandomMessage()
    return ..SendRequestSync(..Target, request, .pOutput)
}

ClassMethod RandomMessage() As %String [ Language = python ]
{
    import iris
    import random
    fruits = ["apple", "banana", "cherry"]
    fruit = random.choice(fruits)
    return fruit + ' ' + iris.Debug.Service.Poller.GetSomeText()
}

ClassMethod GetSomeText() As %String [ Language = objectscript ]
{
    quit "is something to eat"
}

}
```

For more information on programming for interoperability productions, see Programming Business Services, Processes and Operations.

8

Use the Flexible Python Runtime Feature

8.1 Overview of the Flexible Python Runtime Feature

When you run Embedded Python, InterSystems IRIS expects that you are using the default version of Python for your operating system. However, Microsoft Windows does not come with Python by default, and as of version 2024.2, the InterSystems IRIS installer on Windows does not install Python for you. Also, there may be times when you might want to upgrade to a later version of Python or switch to an alternate distribution like Anaconda.

The Flexible Python Runtime feature enables you to choose what version of Python you want to use with Embedded Python in InterSystems IRIS.

Note: The Flexible Python Runtime feature is not supported on all operating systems. See Other Supported Features for a complete list of platforms that support the feature.

Important: There are known incompatibilities between InterSystems IRIS and Python 3.13, so you are advised to avoid Python 3.13 at this time. This issue will be addressed in an upcoming version of InterSystems IRIS.

Preparing to use the Flexible Python Runtime feature involves three basic steps:

1. Install the version of Python you want to use.

2. Set the `PythonRuntimeLibrary` configuration setting to specify the location of the Python runtime library to use when running Embedded Python.

   This location will vary based on your operating system, Python version, and other factors.

   Windows example: `C:\Program Files\Python311\python3.dll` (Python 3.11 on Windows)

   Linux example: `/usr/lib/x86_64-linux-gnu/libpython3.11.so.1.0` (Python 3.11 on Ubuntu 22.04 on the x86 architecture)

   See PythonRuntimeLibrary for more information.

3. Set the `PythonRuntimeLibraryVersion` configuration setting to specify the version number of the Python runtime library to use when running Embedded Python.

   For example: `3.11`.

   See PythonRuntimeLibraryVersion for more information.

4. Ensure that the `sys.path` variable in Embedded Python includes the correct directories needed to import Python packages.

   See Embedded Python and sys.path.

See the following sections for step-by-step instructions for some common Flexible Python Runtime scenarios:

- Example: Python 3.11 on Windows

- Example: Python 3.11 on Ubuntu 22.04

- Example: Anaconda on Windows

- Example: Anaconda on Ubuntu 22.04

8.1.1 Embedded Python and sys.path

After you launch Embedded Python, it uses the directories contained in the `sys.path` variable to locate any Python packages you want to import.

When you use Embedded Python with the default version of Python for your operating system, or on Microsoft Windows, `sys.path` already includes the correct directories, for example.

- <installdir>/lib/python (Python packages reserved for InterSystems IRIS)

- <installdir>/mgr/python (Python packages installed by the user)

- Global package repositories for the default Python version

  This location will vary based on your operating system, Python version, and other factors.

  Windows example: C:\Program Files\Python311\Lib\site-packages (Python 3.11 on Windows)

  Linux example: /usr/local/lib/python3.10/dist-packages (Python 3.10 on Ubuntu 22.04)

On Microsoft Windows with Python 3.11, `sys.path` in Embedded Python might look something like this:

```
USER>do ##class(%SYS.Python).Shell()

Python 3.11.9 (tags/v3.11.9:de54cf5, Apr  2 2024, 10:12:12) [MSC v.1938 64 bit (AMD64)] on win32
Type quit() or Ctrl-D to exit this shell.
>>> import sys
>>> sys.path
['C:\\Program Files\\Python311\\python311.zip', 'C:\\Program Files\\Python311\\DLLs',
'C:\\Program Files\\Python311\\Lib', 'c:\\intersystems\\IRIS\\bin',
'c:\\intersystems\\IRIS\\mgr\\python', 'c:\\intersystems\\IRIS\\lib\\python',
'C:\\Program Files\\Python311', 'C:\\Program Files\\Python311\\Lib\\site-packages']
```

Important: On Microsoft Windows, if `sys.path` contains a directory that is within your user directory, such as C:\Users\<username>\AppData\Local\Programs\Python\Python311, it indicates that you have installed Python for the current user only. InterSystems recommends installing Python for all users to avoid unexpected results.

On Ubuntu 22.04, with the default version of Python (3.10), `sys.path` in Embedded Python might look something like this:

```
USER>do ##class(%SYS.Python).Shell()

Python 3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11.4.0] on linux
Type quit() or Ctrl-D to exit this shell.
>>> import sys
>>> sys.path
['/usr/lib/python310.zip', '/usr/lib/python3.10', '/usr/lib/python3.10/lib-dynload',
'/InterSystems/IRIS/lib/python',
'/InterSystems/IRIS/mgr/python', '/usr/local/lib/python3.10/dist-packages',
'/usr/lib/python3/dist-packages',
'/usr/lib/python3.10/dist-packages']
```

Important: On Linux, if `sys.path` contains a directory that is within your home directory, such as /home/<username>/.local/lib/python3.10/site-packages, it could indicate that you have installed packages in your local package repository. For example, if you install a package without the `--target` attribute (and without using **sudo**), Python will install it into the local package repository within your home directory. If any other user tries to import the package, it will fail.

Although InterSystems recommends using the `--target <installdir>/mgr/python` option, installing packages using **sudo** and omitting the `--target` attribute installs the packages into the global package repository. These packages can also be imported by any user.

If you switch to an alternate distribution of Python, InterSystems IRIS may not know where its package repositories are located. InterSystems IRIS provides you with a tool that can help you tailor your `sys.path` to include the correct directories, namely the iris_site.py file in the directory <installdir>/lib/python.

The iris_site.py file covers the most common use cases, but if you need to customize the file, copy it to the <installdir>/mgr/python directory. This will ensure that your configuration is not wiped out when you upgrade to a later version of InterSystems IRIS. Read the comments in the file for helpful tips.

If you are using Anaconda, InterSystems IRIS 2024.2 comes with examples you can edit to your needs. For example, if you are using Anaconda on Linux, copy the file iris_site_anaconda_ubuntu_linux_example.py from the <installdir>/lib/python directory to the <installdir>/mgr/python directory and rename it iris_site.py. Then edit the path names in the file to match your installation.

For Anaconda on Ubuntu 22.04, for example, edit the last few lines of your iris_site.py file to substitute your actual username:

```
sys.path = sys.path + __sitegetsitepackages(['/home/<username>/anaconda3'])
sys.path = sys.path + ["/home/<username>/anaconda3/lib/python3.11/lib-dynload"]
sys.path = ["/home/<username>/anaconda3/lib/python3.11"] + sys.path
sys.path = ["/home/<username>/anaconda3/lib/python311.zip"] + sys.path
```

With the iris_site.py above, `sys.path` in Embedded Python might look something like this:

```
USER>do ##class(%SYS.Python).Shell()

Python 3.11.7 (main, Dec 15 2023, 18:24:52) [GCC 11.2.0] on linux
Type quit() or Ctrl-D to exit this shell.
>>> import sys
>>> sys.path
['/home/<username>/anaconda3/lib/python311.zip', '/home/<username>/anaconda3/lib/python3.11',
'/home/<username>/anaconda3/lib/python311.zip', '/home/<username>/anaconda3/lib/python3.11',
'/home/<username>/anaconda3/lib/python3.11/lib-dynload', '/InterSystems/IRIS/mgr/python',
'/InterSystems/IRIS/lib/python', '/home/<username>/anaconda3/lib/python3.11/site-packages',
'/home/<username>/anaconda3/lib/python3.11/lib-dynload']
```

Again, the `<username>` in the above example will reflect your actual username.

It may take you a few iterations to get your `sys.path` correct for Flexible Python Runtime. It may be helpful to launch Python outside of InterSystems IRIS and compare its `sys.path` with the `sys.path` inside Embedded Python to make sure you have all of the expected directories.

Note: Changes to the `PythonRuntimeLibrary` and `PythonRuntimeLibraryVersion` configuration settings or iris_site.py take effect on starting a new session. Restarting InterSystems IRIS is not required.

8.1.2 Check Python Version Information

If you are using the Flexible Python Runtime feature, it can be useful to check the version of Python that Embedded Python is using versus the default version that your system is using. An easy way to do this is by calling the **GetPythonInfo()** method of the %SYS.Python class.

The following example, on Ubuntu 22.04 on the x86 architecture, shows that the Python runtime library being used is /usr/lib/x86_64-linux-gnu/libpython3.11.so.1.0, the running version of Embedded Python is 3.11.0rc1, and the system version is 3.10.12.

```
USER>do ##class(%SYS.Python).GetPythonInfo(.info)

USER>zw info
info("AllowNonSystemPythonForIntegratedML")=0
info("CPF_PythonPath")=""
info("CPF_PythonRuntimeLibrary")="/usr/lib/x86_64-linux-gnu/libpython3.11.so.1.0"
info("CPF_PythonRuntimeLibraryVersion")=3.11
info("IRISInsidePython")=0
info("PythonInsideIRIS")=2
info("RunningLibrary")="/usr/lib/x86_64-linux-gnu/libpython3.11.so.1.0"
info("RunningVersion")="3.11.0rc1 (main, Aug 12 2022, 10:02:14) [GCC 11.2.0]"
info("SystemPath")="/usr/lib/python3.10/config-3.10-x86_64-linux-gnu"
info("SystemVersion")="3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11.4.0]"
info("SystemVersionShort")="3.10.12"
info("iris_site.py_platform")="lnxubuntu2204x64"
```

This information will vary based on your operating system, Python version, and other factors.

8.2 Flexible Python Runtime Example: Python 3.11 on Windows

Microsoft Windows does not come with Python, and as of InterSystems IRIS 2024.2, the InterSystems IRIS installer for Windows does not install Python. You must download and install Python yourself.

Important: The installer for InterSystems IRIS 2024.1 on the Windows platform previously installed OpenSSL 3.1.4 along with Python 3.9.5. However, installing Python 3.9 from Python.org installs OpenSSL 1.1.1n. If you need to use OpenSSL 3 with Embedded Python on InterSystems IRIS 2024.2, install Python 3.11 or higher. This installs whatever version of OpenSSL 3 is included with the Python installer. (The version of OpenSSL used by Embedded Python is independent from any other versions used by the Windows operating system or by InterSystems IRIS itself.)

The example below shows how to determine the version of OpenSSL used by Embedded Python:

```
USER>do ##class(%SYS.Python).Shell()

Python 3.11.9 (tags/v3.11.9:de54cf5, Apr  2 2024, 10:12:12) [MSC v.1938 64 bit (AMD64)]
 on win32
Type quit() or Ctrl-D to exit this shell.
>>> import ssl
>>> ssl.OPENSSL_VERSION
'OpenSSL 3.0.13 30 Jan 2024'
```

Important: On InterSystems IRIS 2024.2 for Windows, the AutoML feature of IntegratedML requires using Python 3.11.

The following example shows how to use install and use Python 3.11 with InterSystems IRIS 2024.2 on Windows:

1. Download Python 3.11 from https://www.python.org/downloads/.

2. Launch the Python installer.

3. Click **Customize Installation**.

4. Click **Next** until you reach the **Advanced Options** screen.

5. Select the option to **Install Python for All Users**.

6. Click **Install**.

7. In the InterSystems IRIS Management Portal, go to **System Administration** > **Configuration** > **Additional Settings** > **Advanced Memory**.

8. On the **Advanced Memory Settings** page, in the **PythonRuntimeLibrary** row, click **Edit**.

9. Enter C:\Program Files\Python311\python3.dll.

> **Important:**   If python3.dll is located in a folder that is within your user directory, such as
> C:\Users\<username>\AppData\Local\Programs\Python\Python311, it indicates that you have installed
> Python for the current user only. InterSystems recommends installing Python for all users to avoid
> unexpected results.

10. Click **Save**.

11. On the **Advanced Memory Settings** page, in the **PythonRuntimeLibraryVersion** row, click **Edit**.

12. Enter 3.11.

13. Click **Save**.

14. From Terminal, launch Embedded Python and verify that sys.path now includes the Python 3.11 package directories.

```
USER>do ##class(%SYS.Python).Shell()

Python 3.11.9 (tags/v3.11.9:de54cf5, Apr  2 2024, 10:12:12) [MSC v.1938 64 bit (AMD64)] on win32
Type quit() or Ctrl-D to exit this shell.
>>> import sys
>>> sys.path
['C:\\Program Files\\Python311\\python311.zip', 'C:\\Program Files\\Python311\\DLLs',
'C:\\Program Files\\Python311\\Lib', 'c:\\intersystems\\IRIS\\bin',
'c:\\intersystems\\IRIS\\mgr\\python', 'c:\\intersystems\\IRIS\\lib\\python',
'C:\\Program Files\\Python311', 'C:\\Program Files\\Python311\\Lib\\site-packages']
```

15. From Terminal, use the **GetPythonInfo()** method of the %SYS.Python class to view the Python version information.

```
USER>do ##class(%SYS.Python).GetPythonInfo(.info)

USER>zw info
info("AllowNonSystemPythonForIntegratedML")=0
info("CPF_PythonPath")=""
info("CPF_PythonRuntimeLibrary")="C:\Program Files\Python311\python3.dll"
info("CPF_PythonRuntimeLibraryVersion")=3.11
info("IRISInsidePython")=0
info("PythonInsideIRIS")=16
info("RunningLibrary")="C:\Program Files\Python311\python3.dll"
info("RunningVersion")="3.11.9 (tags/v3.11.9:de54cf5, Apr  2 2024, 10:12:12) [MSC v.1938 64 bit
(AMD64)]"
info("SystemPath")="F:\AllPythons\python.3.9.13\tools\"
info("SystemVersion")="3.9.13 (tags/v3.9.13:6de2ca5, May 17 2022, 16:36:42) [MSC v.1929 64 bit
(AMD64)]"
info("SystemVersionShort")="3.9.13"
info("iris_site.py_platform")="winamd64"
```

This example shows that the Python runtime library being used is C:\Program Files\Python311\python3.dll
and the running version of Embedded Python is 3.11.9. The SystemVersion is not relevant on Windows.

8.3 Flexible Python Runtime Example: Python 3.11 on Ubuntu 22.04

Python 3.10 is the default version of Python on Ubuntu 22.04. This example shows how to use Python 3.11 with Embedded Python.

Note: This example is for Ubuntu 22.04 on the x86 architecture. File and directory names may vary if you are on the ARM architecture.

1. Install Python 3.11 from the command line.

   ```
   $ sudo apt install python3.11-full
   ```

2. Install the Python 3.11 libpython.so shared library.

   ```
   $ sudo apt install libpython3.11
   ```

3. In the InterSystems IRIS Management Portal, go to **System Administration** > **Configuration** > **Additional Settings** > **Advanced Memory**.

4. On the **Advanced Memory Settings** page, in the **PythonRuntimeLibrary** row, click **Edit**.

5. Enter `/usr/lib/x86_64-linux-gnu/libpython3.11.so.1.0`.

6. Click **Save**.

7. On the **Advanced Memory Settings** page, in the **PythonRuntimeLibraryVersion** row, click **Edit**.

8. Enter `3.11`.

9. Click **Save**.

10. From Terminal, launch Embedded Python and verify that `sys.path` now includes the Python 3.11 package directories.

    ```
    USER>do ##class(%SYS.Python).Shell()

    Python 3.11.0rc1 (main, Aug 12 2022, 10:02:14) [GCC 11.2.0] on linux
    Type quit() or Ctrl-D to exit this shell.
    >>> import sys
    >>> sys.path
    ['/usr/lib/python311.zip', '/usr/lib/python3.11', '/usr/lib/python3.11/lib-dynload',
    '/InterSystems/IRIS/mgr/python',
    '/InterSystems/IRIS/lib/python', '/usr/local/lib/python3.11/dist-packages',
    '/usr/lib/python3/dist-packages',
    '/usr/lib/python3.11/dist-packages']
    ```

11. From Terminal, use the **GetPythonInfo()** method of the %SYS.Python class to view the Python version information.

    ```
    USER>do ##class(%SYS.Python).GetPythonInfo(.info)

    USER>zw info
    info("AllowNonSystemPythonForIntegratedML")=0
    info("CPF_PythonPath")=""
    info("CPF_PythonRuntimeLibrary")="/usr/lib/x86_64-linux-gnu/libpython3.11.so.1.0"
    info("CPF_PythonRuntimeLibraryVersion")=3.11
    info("IRISInsidePython")=0
    info("PythonInsideIRIS")=2
    info("RunningLibrary")="/usr/lib/x86_64-linux-gnu/libpython3.11.so.1.0"
    info("RunningVersion")="3.11.0rc1 (main, Aug 12 2022, 10:02:14) [GCC 11.2.0]"
    info("SystemPath")="/usr/lib/python3.10/config-3.10-x86_64-linux-gnu"
    info("SystemVersion")="3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11.4.0]"
    info("SystemVersionShort")="3.10.12"
    info("iris_site.py_platform")="lnxubuntu2204x64"
    ```

    This example shows that the Python runtime library being used is `/usr/lib/x86_64-linux-gnu/libpython3.11.so.1.0`, the running version of Embedded Python is `3.11.0rc1`, and the system version is `3.10.12`.

8.4 Flexible Python Runtime Example: Anaconda on Windows

Anaconda is a Python-based platform commonly used for data science and artificial intelligence applications. This example shows how to use Anaconda with Windows. InterSystems recommends installing Anaconda for all users.

1. Download Anaconda from https://anaconda.com/download.

2. Launch the Anaconda installer, and on the **Welcome** screen, click **Next**.

3. Approve the license agreement.

4. Under **Select Installation Type**, install for **All Users**.

5. Under **Choose Install Location**, for **Destination Folder**, leave the default: `C:\ProgramData\anaconda3`.

6. Under **Advanced Installation Options**, deselect **Register Anaconda3 as the System Python**.

7. Click **Install** to begin the installation process.

8. In the InterSystems IRIS Management Portal, go to **System Administration** > **Configuration** > **Additional Settings** > **Advanced Memory**.

9. On the **Advanced Memory Settings** page, in the **PythonRuntimeLibrary** row, click **Edit**.

10. Enter `C:\ProgramData\anaconda3\python312.dll`.

11. Click **Save**.

12. On the **Advanced Memory Settings** page, in the **PythonRuntimeLibraryVersion** row, click **Edit**.

13. Enter `3.12`.

14. Click **Save**.

15. Copy the file iris_site_anaconda_windows_example.py from the <installdir>/lib/python directory to the <installdir>/mgr/python directory and rename it iris_site.py.

    If you installed Anaconda in the default location, you should not need to modify the file.

16. From Terminal, launch Embedded Python and verify that `sys.path` now includes the Anaconda package repositories.

```
USER>do ##class(%SYS.Python).Shell()

Python 3.12.4 | packaged by Anaconda, Inc. | (main, Jun 18 2024, 15:03:56) [MSC v.1929 64 bit
(AMD64)] on win32
Type quit() or Ctrl-D to exit this shell.
>>> import sys
>>> sys.path
['C:\\ProgramData\\anaconda3\\python312.zip', 'C:\\ProgramData\\anaconda3\\DLLs',
'C:\\ProgramData\\anaconda3\\Lib', \'c:\\intersystems\\IRIS\\bin',
'c:\\intersystems\\IRIS\\mgr\\python', 'c:\\intersystems\\IRIS\\lib\\python',
'C:\\ProgramData\\anaconda3', 'C:\\ProgramData\\anaconda3\\Lib\\site-packages',
'C:\\ProgramData\\anaconda3\\Lib\\site-packages\\win32',
'C:\\ProgramData\\anaconda3\\Lib\\site-packages\\win32\\lib',
'C:\\ProgramData\\anaconda3\\Lib\\site-packages\\Pythonwin']
```

17. From Terminal, use the **GetPythonInfo()** method of the %SYS.Python class to view the Python version information.

```
USER>do ##class(%SYS.Python).GetPythonInfo(.info)

USER>zw info
info("AllowNonSystemPythonForIntegratedML")=0
info("CPF_PythonPath")=""
info("CPF_PythonRuntimeLibrary")="C:\ProgramData\anaconda3\python312.dll"
info("CPF_PythonRuntimeLibraryVersion")=3.12
info("IRISInsidePython")=0
info("PythonInsideIRIS")=1
info("RunningLibrary")="C:\ProgramData\anaconda3\python312.dll"
info("RunningVersion")="3.12.4 | packaged by Anaconda, Inc. | (main, Jun 18 2024, 15:03:56) [MSC
v.1929 64 bit (AMD64)]"
info("SystemPath")="F:\AllPythons\python.3.9.13\tools\"
info("SystemVersion")="3.9.13 (tags/v3.9.13:6de2ca5, May 17 2022, 16:36:42) [MSC v.1929 64 bit
(AMD64)]"
info("SystemVersionShort")="3.9.13"
info("iris_site.py_platform")="winamd64"
```

This example shows that the Python runtime library being used is `C:\ProgramData\anaconda3\python312.dll` and the running version of Embedded Python is `3.12.4`. The `SystemVersion` is not relevant on Windows.

8.5 Flexible Python Runtime Example: Anaconda on Ubuntu 22.04

Anaconda is a Python-based platform commonly used for data science and artificial intelligence applications. This example shows how to use Anaconda with Ubuntu 22.04. InterSystems recommends doing an installation for your user.

Note: This example is for Ubuntu 22.04 on the x86 architecture. File and directory names may vary if you are on the ARM architecture.

1. Download Anaconda.

   `curl -O https://repo.anaconda.com/archive/Anaconda3-2024.02-1-Linux-x86_64.sh`

2. Run the Anaconda installation script from the command line, for example:

   `$ sh Anaconda3-2024.02-1-Linux-x86_64.sh`

3. Accept the license terms.

4. When prompted for an installation directory, enter `/home/<username>/anaconda3`, where `<username>` is your Ubuntu username.

5. You will be asked if you want to activate conda on startup. For this example, enter `no`.

6. This will end the Anaconda installation process.

   You will see the message `Thank you for installing Anaconda3!`

7. Activate Anaconda.

   `$ source /home/<username>/anaconda3/bin/activate`

   Substitute your actual username.

8. Initialize Anaconda.

   `$ conda init`

9. In the InterSystems IRIS Management Portal, go to **System Administration** > **Configuration** > **Additional Settings** > **Advanced Memory**.

10. On the **Advanced Memory Settings** page, in the **PythonRuntimeLibrary** row, click **Edit**.

11. Enter `/home/<username>/anaconda3/lib/libpython3.11.so.1.0`.

    Substitute your actual username.

12. Click **Save**.

13. On the **Advanced Memory Settings** page, in the **PythonRuntimeLibraryVersion** row, click **Edit**.

14. Enter `3.11`.

15. Click **Save**.

16. Copy the file iris_site_anaconda_ubuntu_linux_example.py from the <installdir>/lib/python directory to the <installdir>/mgr/python directory and rename it iris_site.py.

17. Edit the iris_site.py file to substitute your actual username in the last few lines of the file::

    ```
    sys.path = sys.path + __sitegetsitepackages(['/home/<username>/anaconda3'])
    sys.path = sys.path + ["/home/<username>/anaconda3/lib/python3.11/lib-dynload"]
    sys.path = ["/home/<username>/anaconda3/lib/python3.11"] + sys.path
    sys.path = ["/home/<username>/anaconda3/lib/python311.zip"] + sys.path
    ```

18. From Terminal, launch Embedded Python and verify that `sys.path` now includes the Anaconda package repositories.

    ```
    USER>do ##class(%SYS.Python).Shell()

    Python 3.11.7 (main, Dec 15 2023, 18:24:52) [GCC 11.2.0] on linux
    Type quit() or Ctrl-D to exit this shell.
    >>> import sys
    >>> sys.path
    ['/home/<username>/anaconda3/lib/python311.zip', '/home/<username>/anaconda3/lib/python3.11',
    '/home/<username>/anaconda3/lib/python311.zip', '/home/<username>/anaconda3/lib/python3.11',
    '/home/<username>/anaconda3/lib/python3.11/lib-dynload', '/InterSystems/IRIS/mgr/python',
    '/InterSystems/IRIS/lib/python', '/home/<username>/anaconda3/lib/python3.11/site-packages',
    '/home/<username>/anaconda3/lib/python3.11/lib-dynload']
    ```

    The <username> in the above example will reflect your actual username.

19. From Terminal, use the **GetPythonInfo()** method of the %SYS.Python class to view the Python version information.

    ```
    USER>do ##class(%SYS.Python).GetPythonInfo(.info)

    USER>zw info
    info("AllowNonSystemPythonForIntegratedML")=0
    info("CPF_PythonPath")=""
    info("CPF_PythonRuntimeLibrary")="/home/<username>/anaconda3/lib/libpython3.11.so"
    info("CPF_PythonRuntimeLibraryVersion")=3.11
    info("IRISInsidePython")=0
    info("PythonInsideIRIS")=3
    info("RunningLibrary")="/home/<username>/anaconda3/lib/libpython3.11.so"
    info("RunningVersion")="3.11.7 (main, Dec 15 2023, 18:24:52) [GCC 11.2.0]"
    info("SystemPath")="/usr/lib/python3.10/config-3.10-x86_64-linux-gnu"
    info("SystemVersion")="3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11.4.0]"
    info("SystemVersionShort")="3.10.12"
    info("iris_site.py_platform")="lnxubuntu2204x64"info("PythonInsideIRIS")=2
    info("RunningLibrary")="/usr/lib/x86_64-linux-gnu/libpython3.11.so.1.0"
    info("RunningVersion")="3.11.0rc1 (main, Aug 12 2022, 10:02:14) [GCC 11.2.0]"
    info("SystemPath")="/usr/lib/python3.10/config-3.10-x86_64-linux-gnu"
    info("SystemVersion")="3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11.4.0]"
    info("SystemVersionShort")="3.10.12"
    info("iris_site.py_platform")="lnxubuntu2204x64"
    ```

    Again, the <username> in the above example will reflect your actual username.

    This example shows that the Python runtime library being used is `/home/<username>/anaconda3/lib/libpython3.11.so`, the running version of Embedded Python is `3.11.7`, and the system version is `3.10.12`.

InterSystems IRIS Python Module Reference

If you are using Embedded Python and need to interact with InterSystems IRIS, you can use the `iris` module to enable InterSystems IRIS functionality.

InterSystems IRIS Python Module Core API

This section provides API documentation for the core functions of the InterSystems IRIS Python Module. These functions allow you to access InterSystems IRIS classes and methods, use the transaction processing capabilities of InterSystems IRIS, and perform other core InterSystem IRIS tasks.

Summary

The following table summaries the core functions of the iris module. To use this module from Embedded Python, use `import iris`.

Group	Functions
Code Execution	check_status(), execute(), routine()
Locking and Concurrency Control	lock(), unlock()
Reference Creation	arrayref(), cls(), gref(), ref(),
Transaction Processing[*]	tcommit(), tlevel(), trollback(), trollbackone(), tstart(),

[*]See Transaction Processing for information on using transactions to maintain the logical integrity of your InterSystems IRIS database.

arrayref(dictionary)

Creates an ObjectScript array from a Python dictionary and returns a reference to the array.

Assume you have an InterSystems IRIS class called User.ArrayTest that has the following ObjectScript methods that expect an array as an argument:

```
ClassMethod WriteContents(myArray) [ Language = objectscript ]
{
    zwrite myArray
}

ClassMethod Modify(myArray) [ Language = objectscript ]
{
    set myArray("new") = 123
    set myArray("x","y","z") = "xyz"
}

ClassMethod StoreGlobal(myArray)  [ Language = objectscript ]
{
    kill ^MyGlobal
    if '$data(myArray) return "no data"
    merge ^MyGlobal = myArray
    return "ok"
}
```

The method **WriteContents()** writes the contents of the array, **Modify()** modifies the contents of the array, and **StoreGlobal()** takes the contents of the array and stores it in the global ^MyGlobal.

From Python, you can create a dictionary `mydict` and use **iris.arrayref()** to place its contents in an ObjectScript array and return a reference to that array. Then you can pass that reference to the three methods in User.ArrayTest.

```
>>> mydict = {2:{3:4}}
>>> mydict
{2: {3: 4}}
>>> a = iris.arrayref(mydict)
>>> a.value
{2: {3: 4}}
>>> iris.User.ArrayTest.Modify(a)
>>> iris.User.ArrayTest.WriteContents(a)
myArray(2,3)=4
myArray("new")=123
myArray("x","y","z")="xyz"
>>> iris.User.ArrayTest.StoreGlobal(a)
'ok'
```

Then, from ObjectScript, you can verify that global `^MyGlobal` now contains the same data as the array did:

```
USER>zwrite ^MyGlobal
^MyGlobal(2,3)=4
^MyGlobal("new")=123
^MyGlobal("x","y","z")="xyz"
```

For information on ObjectScript arrays, see Multidimensional Arrays.

check_status(status)

Raises an exception if *status* contains an error. Returns None if no error condition occurs.

If you have an InterSystems IRIS class Sample.Company that has a Name property that is required, trying to save an instance of that class without a Name property results in an error status. The following example uses **iris.check_status()** to check the status returned by the **_Save()** method and throws an exception if it contains an error.

```
>>> mycompany = iris.Sample.Company._New()
>>> mycompany.TaxID = '123456789'
>>> try:
...     status = mycompany._Save()
...     iris.check_status(status)
... except Exception as ex:
...     print(ex)
...
ERROR #5659: Property 'Sample.Company::Name(4@Sample.Company,ID=)' required
```

cls(class_name)

Returns a reference to an InterSystems IRIS class. This allows you access the properties and methods of that class in the same way you would a with a Python class. You can use **iris.cls()** to access both built-in InterSystems IRIS classes or custom InterSystems IRIS classes you write yourself.

The following example uses **iris.cls()** to return a reference to the built-in InterSystems IRIS class %SYS.System. It then calls its **GetInstanceName()** method.

```
>>> system = iris.cls('%SYS.System')
>>> print(system.GetInstanceName())
IRIS2023
```

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.

execute(statements)

Executes an ObjectScript statement and optionally returns a value.

The following example uses **iris.execute()** to write the value of **$HOROLOG**, an ObjectScript special value that contains the current local date and time in the internal InterSystems IRIS format.

```
>> iris.execute('write $horolog,!')
66682,45274
```

The following example returns the value of **$HOROLOG**, converted to a human-readable string by the ObjectScript **$ZDATE** function.

```
>>> today = iris.execute('return $zdate($horolog)')
>>> print(today)
07/27/2023
```

gref(global_name)

Returns a reference to an InterSystems IRIS global. The global may or may not already exist.

The following example uses **iris.gref()** to set variable day to a reference to global ^day.

```
>>> day = iris.gref('^day')
```

The next example prints the value stored at ^day(1, "name"), and since no value is currently stored for those keys, it prints None. Next it stores the value "Sunday" at that location and retrieves and prints the stored value.

```
>>> print(day[1, 'name'])
None
>>> day[1, 'name'] = 'Sunday'
>>> print(day[1, 'name'])
Sunday
```

For information on the methods that can be used on an InterSystems IRIS global reference, see Global Reference API.

For background information on globals, see Introduction to Globals.

lock(lock_list, timeout_value, locktype)

Sets locks, given a list of lock names, an optional timeout value (in seconds), and an optional lock type. If *locktype* is "S", this indicates a shared lock.

In InterSystems IRIS, a lock is used to prevent more than one user or process from accessing or modifying the same resource (usually a global) at the same time. For example, a process that writes to a resource should request an exclusive lock (the default) so that another process does not attempt to read or write to that resource simultaneously. A process that reads a resource can request a shared lock so that other processes can read that resource at the same time, but not write to that resource. A process can specify a timeout value, so that it does not wait forever waiting for a resource to become available.

The following example uses **iris.lock()** to request exclusive locks on locks named ^one and ^two. If the request is successful, the call returns True.

```
>>> iris.lock(['^one','^two'])
True
```

If another process then requests a shared lock on ^one, and the first process does not release the lock within 30 seconds, the call below returns False.

```
>>> iris.lock(['^one'],30,'S')
False
```

A process should use unlock() to relinquish locks when the resources they protect are no longer being used.

For more information on how locks are used in InterSystems IRIS, see Locking and Concurrency Control.

ref(value)

Creates an iris.ref object with a specified value. This is useful for situations when you need to pass an argument to an ObjectScript method by reference.

The following example uses **iris.ref()** to create an iris.ref object with the value 2000.

```
>>> calories = iris.ref(2000)
>>> calories.value
2000
```

Assume an InterSystems IRIS class User.Diet has a method called **Eat()** that takes as arguments the name of a food you're about to consume and your current calorie count for the day, and that *calories* is passed in by reference and is updated with your new calorie count. The following example shows that after the call to **Eat()**, the value of the variable calories has been updated from 2000 to 2250.

```
>>> iris.User.Diet.Eat('hamburger', calories)
>>> calories.value
2250
```

For information on passing arguments by reference in ObjectScript, see Indicating How Arguments Are to Be Passed.

routine(routine, args)

Invokes an InterSystems IRIS routine, optionally at a given tag. Any arguments that need to be passed in the call are comma-delimited, following the name of the routine.

The following example, when run in the `%SYS` namespace, uses **iris.routine()** to call the routine **^SECURITY**:

```
>>> iris.routine('^SECURITY')


1) User setup
2) Role setup
3) Service setup
4) Resource setup
.
.
.
```

If you have a routine **^Math** that has a function **Sum()** that adds two numbers, the following example adds 4 and 3:

```
>>> sum = iris.routine('Sum^Math',4,3)
>>> sum
7
```

For more information on how routines are called in ObjectScript, see Invoking Code and Passing Arguments.

tcommit()

Marks the successful end of an InterSystems IRIS transaction.

Use **iris.tcommit()** to mark the successful end of a transaction and decrement the nesting level by 1:

```
>>> iris.tcommit()
```

To ensure that transactions nest properly, every **iris.tstart()** should be paired with an **iris.tcommit()**.

If **iris.tcommit()** is called when not in a transaction, an exception occurs, with the value <COMMAND>.

See also tstart(), tlevel(), trollback(), and trollbackone().

tlevel()

Detects whether a transaction is currently in progress and returns the nesting level. A call to **iris.tstart()** increments the nesting level, and a call to **iris.tcommit()** decrements the nesting level. A value of zero means not in a transaction.

The following example shows the value returned by **iris.level()** at different transaction nesting levels.

```
>>> iris.tlevel()
0
>>> iris.tstart()
>>> iris.tstart()
>>> iris.tlevel()
2
>>> iris.tcommit()
>>> iris.tlevel()
1
```

See also tstart(), tcommit(), trollback(), and trollbackone().

trollback()

Rolls back all current transactions in progress and restores all journaled database values to their values at the start of the initial transaction. It also resets the transaction nesting level to 0.

This simple example initializes the global `^a(1)` to the value "hello." It then starts a transaction and sets `^a(1)` to the value "goodbye." But before the transaction is committed, it calls **iris.trollback()**. This resets the transaction nesting level to 0 and restores `^a(1)` to the value it had before the start of the transaction.

```
>>> a = iris.gref('^a')
>>> a[1] = 'hello'
>>> iris.tstart()
>>> iris.tlevel()
1
>>> a[1] = 'goodbye'
>>> iris.trollback()
>>> iris.tlevel()
0
>>> a[1]
'hello'
```

See also tstart(), tcommit(), tlevel(), and trollbackone().

trollbackone()

Rolls back the current level of nested transactions, that is, the one initiated by the most recent **iris.tstart()**. It also decrements the transaction nesting level by 1.

This example initializes the global `^a(1)` to the value 4 and `^b(1)` to the value "lemon." It then starts a transaction and sets `^a(1)` to 9. Next, it starts a nested transaction and sets `^b(1)` to "lime." It then calls **iris.trollbackone()** to roll back the inner transaction and calls **iris.commit()** to commit the outer transaction. When all is said and done, `^a(1)` retains its new value, while `^b(1)` is rolled back to its original value.

```
>>> a = iris.gref('^a')
>>> b = iris.gref('^b')
>>> a[1] = 4
>>> b[1] = 'lemon'
>>> iris.tstart()
>>> iris.tlevel()
1
>>> a[1] = 9
>>> iris.tstart()
>>> iris.tlevel()
2
>>> b[1] = 'lime'
>>> iris.trollbackone()
>>> iris.tlevel()
1
>>> iris.tcommit()
>>> iris.tlevel()
0
>>> a[1]
```

```
9
>>> b[1]
'lemon'
```

See also tstart(), tcommit(), tlevel(), and trollback().

tstart()

Marks the start of an InterSystems IRIS transaction.

A transaction is a group of commands that must all complete in order for the transaction to be considered successful. For example, if you have a transaction that transfers a sum of money from one bank account to another, the transaction is only successful if withdrawing the money from the first account and depositing it into the second account are both successful. If the transaction fails, the database can be rolled back to the state it was in before the start of the transaction.

Use **iris.start()** to mark the start of a transaction and increment the transaction nesting level by 1:

```
>>> iris.tstart()
```

See also tcommit(), tlevel(), trollback(), and trollbackone().

For more information on how transaction processing works in InterSystems IRIS, see Transaction Processing.

unlock(lock_list, timeout_value, locktype)

Removes locks, given a list of lock names, an optional timeout value (in seconds), and an optional lock type.

If your code sets locks to control access to resources, it should unlock them when it is done using those resources.

The following example uses **iris.unlock()** to unlock the locks named ^one and ^two.

```
>>> iris.unlock(['^one','^two'])
True
```

See also lock().

Global Reference API

This section provides API documentation for the methods of the gref class of the InterSystems IRIS Python Module. These methods allow you to access and manipulate InterSystems IRIS globals.

Summary

The following table summaries the methods of the gref class of the InterSystems IRIS Python Module. To use this class from Embedded Python, first do import iris, and then use the **iris.gref()** function to obtain a reference to a global. (See iris.gref().)

Group	Settings
Work on a Node of a Global	data(), get(), getAsBytes(), kill(), set()
Traverse a Global	keys(), order(), orderiter(), query()

For background information on globals, see Introduction to Globals.

data(key)

Checks if a node of a global contains data and/or has descendants. The *key* of the node is passed as a list. Passing a key with the value None (or an empty list) indicates the root node of the global.

You can use **data()** to inspect a node to see if it contains data before attempting to access that data and possibly encountering an error. The method returns 0 if the node is undefined (contains no data), 1 if the node is defined (contains data), 10 if the node is undefined but has descendants, or 11 if the node is defined and has descendants.

Assume you have a global ^a with the following contents:

```
^a(2) = "two"
^a(3,1) = "three one"
^a(4) = "four"
^a(4,1) = "four one"
```

Then you can use **data()** to test the various nodes of the global as in these examples:

```
>>> a = iris.gref('^a')
>>> a.data([1])
0
>>> a.data([2])
1
>>> a.data([3])
10
>>> a.data([4])
11
>>> a.data([None])
10
>>> a.data([3,1])
1
```

You can use modulo 2 arithmetic to check whether a node contains data, regardless of whether it has descendants or not.

```
>>> a.data([3]) % 2
0
>>> a.data([4]) % 2
1
```

get(key)

Gets the value stored at a node of a global. The *key* of the node is passed as a list. Passing a key with the value None (or an empty list) indicates the root node of the global.

Assume you have a global `^a` with the following contents:

```
^a(2) = "two"
^a(3,1) = "three one"
^a(4) = "four"
^a(4,1) = "four one"
```

Then you can use **get()** to retrieve data from the various nodes of the global as in these examples:

```
>>> a = iris.gref('^a')
>>> a.get([2])
'two'
>>> a.get([3,1])
'three one'
```

Alternatively, you can get the value of a node directly, as you would for a Python dictionary, or you can use the dunder method **__getitem__()**.

```
>>> a[3,1]
'three one'
>>> a.__getitem__([3,1])
'three one'
```

Using **get()** to get data from a node that is undefined results in an error.

```
>>> a.get([5])
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 'Global Undefined'
```

You can use the data() method to test whether a node contains data before trying to retrieve it.

Getting data from an undefined node directly or by using **__getitem__()** returns None, instead of causing an error.

```
>>> print(a[5])
None
>>> print(a.__getitem__([5]))
None
```

See also getAsBytes().

getAsBytes(key)

Gets a string value stored at a node of a global and converts it to the Python bytes data type. The *key* of the node is passed as a list. Passing a key with the value None (or an empty list) indicates the root node of the global.

Assume you have a global `^a` with the following contents:

```
^a(2) = "two"
^a(3,1) = "three one"
^a(4) = "four"
^a(4,1) = "four one"
```

Then you can use **getAsBytes()** to retrieve data from the various nodes of the global as in these examples:

```
>>> a = iris.gref('^a')
>>> a.getAsBytes([2])
b'two'
>>> a.getAsBytes([3,1])
b'three one'
```

Using **getAsBytes()** to get data from a node that is undefined results in an error.

```
>>> a.getAsBytes([5])
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 'Global Undefined'
```

You can use the data() method to test whether a node contains data before trying to retrieve it.

See also get().

keys(key)

Returns the keys of a global, starting from a given key. The starting *key* is passed as a list. Passing an empty list indicates the root node of the global.

Assume you have a global `^mlb` with the following contents:

```
^mlb = "Major League Baseball"
^mlb("AL") = "American League"
^mlb("AL","Central") = "AL Central"
^mlb("AL","East") = "AL East"
^mlb("AL","East",1) = "Baltimore"
^mlb("AL","East",2) = "Boston"
^mlb("AL","East",3) = "NY Yankees"
^mlb("AL","East",4) = "Tampa Bay"
^mlb("AL","East",5) = "Toronto"
^mlb("AL","West") = "AL West"
^mlb("AL","West",1) = "Houston"
^mlb("AL","West",2) = "LA Angels"
^mlb("AL","West",3) = "Oakland"
^mlb("AL","West",4) = "Seattle"
^mlb("AL","West",5) = "Texas"
^mlb("NL") = "National League"
```

You can use **keys()** to get the keys of the global and print out their values, as follows:

```
>>> m = iris.gref('^mlb')
>>> for key in m.keys([]):
...     value = m[key]
...     print(f'{key} = {value}')
...
['AL'] = American League
['AL', 'Central'] = AL Central
['AL', 'East'] = AL East
['AL', 'East', '1'] = Baltimore
['AL', 'East', '2'] = Boston
['AL', 'East', '3'] = NY Yankees
['AL', 'East', '4'] = Tampa Bay
['AL', 'East', '5'] = Toronto
['AL', 'West'] = AL West
['AL', 'West', '1'] = Houston
['AL', 'West', '2'] = LA Angels
['AL', 'West', '3'] = Oakland
['AL', 'West', '4'] = Seattle
['AL', 'West', '5'] = Texas
['NL'] = National League
```

Note that the starting key does not have to exist as a node in the global. Since a global is stored in sorted order, **keys()** begins with the key of the next node according to the sort order, for example:

```
>>> m = iris.gref('^mlb')
>>> for key in m.keys(['AL','North']):
...     value = m[key]
...     print(f'{key} = {value}')
...
['AL', 'West'] = AL West
['AL', 'West', '1'] = Houston
['AL', 'West', '2'] = LA Angels
['AL', 'West', '3'] = Oakland
['AL', 'West', '4'] = Seattle
['AL', 'West', '5'] = Texas
['NL'] = National League
```

You can also use get() to retrieve the value of each node, but you need to test each node first by using data() to make sure it is contains data.

Use order() to traverse the nodes at one level of a global.

kill(key)

Deletes the node of a global, if it exists. The *key* of the node is passed as a list. This also deletes any descendants of the node. Passing a key with the value None (or an empty list) indicates the root node of the global.

Assume you have a global ^a with the following contents:

```
^a(2) = "two"
^a(3,1) = "three one"
^a(4) = "four"
^a(4,1) = "four one"
```

Then you can use **kill()** to kill a node of the global, and its descendants, as in this example:

```
>>> a = iris.gref('^a')
>>> a.kill([4])
```

Now the global has the following contents:

```
^a(2) = "two"
^a(3,1) = "three one"
```

Passing None for the key kills the entire global.

```
>>> a.kill([None])
```

order(key)

Returns the next key in that level of the global, starting from a given key. The starting *key* is passed as a list. If no key follows the starting key, **order()** returns None.

Assume you have a global ^mlb with the following contents:

```
^mlb = "Major League Baseball"
^mlb("AL") = "American League"
^mlb("AL","Central") = "AL Central"
^mlb("AL","East") = "AL East"
^mlb("AL","East",1) = "Baltimore"
^mlb("AL","East",2) = "Boston"
^mlb("AL","East",3) = "NY Yankees"
^mlb("AL","East",4) = "Tampa Bay"
^mlb("AL","East",5) = "Toronto"
^mlb("AL","West") = "AL West"
^mlb("AL","West",1) = "Houston"
^mlb("AL","West",2) = "LA Angels"
^mlb("AL","West",3) = "Oakland"
^mlb("AL","West",4) = "Seattle"
^mlb("AL","West",5) = "Texas"
^mlb("NL") = "National League"
```

You can use **order()** to get the next key from a given key, as in the following examples:

```
>>> m = iris.gref('^mlb')
>>> m.order(['AL','Central'])
'East'
>>> m.order(['AL','East'])
'West'
>>> m.order(['AL','East',1])
'2'
>>> m.order(['AL','East',2])
'3'
```

Note that the starting key does not have to exist as a node in the global. Since a global is stored in sorted order, **order()** returns the key of the next node according to the sort order, for example:

```
>>> m.order(['AL','West',3.5])
'4'
```

Use a **while** loop to traverse the nodes of a global at a certain level, breaking when the return value is None. Setting the starting key to the empty string means "start at the beginning of that level."

The following example traverses the top-level nodes of a global:

```
>>> m = iris.gref('^mlb')
>>> key = ''
>>> while True:
...     key = m.order([key])
...     if key == None:
...         break
...     print(m[key])
...
American League
National League
```

The following example traverses the third-level nodes of a global:

```
>>> m = iris.gref('^mlb')
>>> key = ''
>>> while True:
...     key = m.order(['AL','East',key])
...     if key == None:
...         break
...     print(m['AL','East',key])
...
Baltimore
Boston
NY Yankees
Tampa Bay
Toronto
```

You can nest **while** loops as necessary, or use keys() or query() to traverse an entire global.

orderiter(key)

Returns the keys and values of a global, starting from a given key, down to the next leaf node. The starting *key* is passed as a list. Passing an empty list indicates the root node of the global.

Assume you have a global ^mlb with the following contents:

```
^mlb = "Major League Baseball"
^mlb("AL") = "American League"
^mlb("AL","Central") = "AL Central"
^mlb("AL","East") = "AL East"
^mlb("AL","East",1) = "Baltimore"
^mlb("AL","East",2) = "Boston"
^mlb("AL","East",3) = "NY Yankees"
^mlb("AL","East",4) = "Tampa Bay"
^mlb("AL","East",5) = "Toronto"
^mlb("AL","West") = "AL West"
^mlb("AL","West",1) = "Houston"
^mlb("AL","West",2) = "LA Angels"
^mlb("AL","West",3) = "Oakland"
^mlb("AL","West",4) = "Seattle"
^mlb("AL","West",5) = "Texas"
^mlb("NL") = "National League"
```

The following example uses **orderiter()** to traverse the global down to the next leaf node starting from the root:

```
>>> m = iris.gref('^mlb')
>>> for (key, value) in m.orderiter([]):
...     print(f'{key} = {value}')
...
['AL'] = American League
['AL', 'Central'] = AL Central
```

Note that the starting key does not have to exist as a node in the global. Since a global is stored in sorted order, **orderiter()** finds the next node according to the sort order, for example:

```
>>> m = iris.gref('^mlb')
>>> for (key, value) in m.orderiter(['AL','North']):
...     print(f'{key} = {value}')
...
['AL', 'West'] = AL West
['AL', 'West', '1'] = Houston
```

query(key)

Traverses a global starting at the specified key, returning each key and value. The starting *key* is passed as a list. Passing an empty list indicates the root node of the global.

Assume you have a global ^mlb with the following contents:

```
^mlb = "Major League Baseball"
^mlb("AL") = "American League"
^mlb("AL","Central") = "AL Central"
^mlb("AL","East") = "AL East"
^mlb("AL","East",1) = "Baltimore"
^mlb("AL","East",2) = "Boston"
^mlb("AL","East",3) = "NY Yankees"
^mlb("AL","East",4) = "Tampa Bay"
^mlb("AL","East",5) = "Toronto"
^mlb("AL","West") = "AL West"
^mlb("AL","West",1) = "Houston"
^mlb("AL","West",2) = "LA Angels"
^mlb("AL","West",3) = "Oakland"
^mlb("AL","West",4) = "Seattle"
^mlb("AL","West",5) = "Texas"
^mlb("NL") = "National League"
```

The following example uses **query()** to traverse the global starting from the root:

```
>>> m = iris.gref('^mlb')
>>> for (key, value) in m.query([]):
...     print(f'{key} = {value}')
...
['AL'] = American League
['AL', 'Central'] = AL Central
['AL', 'East'] = AL East
['AL', 'East', '1'] = Baltimore
['AL', 'East', '2'] = Boston
['AL', 'East', '3'] = NY Yankees
['AL', 'East', '4'] = Tampa Bay
['AL', 'East', '5'] = Toronto
['AL', 'West'] = AL West
['AL', 'West', '1'] = Houston
['AL', 'West', '2'] = LA Angels
['AL', 'West', '3'] = Oakland
['AL', 'West', '4'] = Seattle
['AL', 'West', '5'] = Texas
['NL'] = National League
```

Note that the starting key does not have to exist as a node in the global. Since a global is stored in sorted order, **query()** finds the next node according to the sort order, for example:

```
>>> m = iris.gref('^mlb')
>>> for (key, value) in m.query(['AL','North']):
...     print(f'{key} = {value}')
...
['AL', 'West'] = AL West
['AL', 'West', '1'] = Houston
['AL', 'West', '2'] = LA Angels
['AL', 'West', '3'] = Oakland
['AL', 'West', '4'] = Seattle
['AL', 'West', '5'] = Texas
['NL'] = National League
```

set(key, value)

Sets a node in a global to a given value. The *key* of the node is passed as a list, and *value* is the value to be stored. Passing a key with the value None (or an empty list) indicates the root node of the global.

The following example obtains a reference to global ^messages and uses **set()** to set the value of some nodes in the global:

```
>>> msg = iris.gref('^messages')
>>> msg.set([None], 'list of messages')
>>> msg.set(['greeting',1], 'hello')
>>> msg.set(['greeting',2], 'goodbye')
```

If the global ^messages did not already exist, it would look as follows:

```
^messages = "list of messages"
^messages("greeting",1) = "hello"
^messages("greeting",2) = "goodbye"
```

If ^messages already existed, the new values would be added to the global, possibly overwriting existing data at those nodes. You can use the data() method to test whether a node already contains data before trying to set it.

You can also set a global node directly, as you would for a Python dictionary, as in the following example:

```
>>> msg = iris.gref('^messages')
>>> msg['greeting',3] = 'aloha'
```

Now the global ^messages looks like this:

```
^messages = "list of messages"
^messages("greeting",1) = "hello"
^messages("greeting",2) = "goodbye"
^messages("greeting",3)="aloha"
```

IRIS System API

This section provides documentation for the classes in the system package of the InterSystems IRIS Python Module. These classes allow you to access some common System API classes in the InterSystems IRIS %SYSTEM package.

Summary

The following table summaries the classes of the system package of the InterSystems IRIS Python Module. To access these classes from Embedded Python, use iris.system.<classname>.

Class
DocDB, Encryption, Error, Event, Monitor, Process, SQL, SYS, Security, Semaphore, Status, Util, Version

This summary of the classes in the iris.system package does not attempt to document all of the methods available in each class, some of which number in the dozens. For information how to find complete documentation on InterSystems IRIS classes, see Locating and Exploring Class Reference Documentation.

Note: You can access %SYSTEM classes not included in iris.system by using the InterSystems IRIS class directly. For example, list all InterSystems IRIS classes in the current namespace by using the command iris._SYSTEM.OBJ.ShowClasses().

DocDB

The %SYSTEM.DocDB class provides an interface for managing Document Databases.

The following example uses the **Exists()** method of the iris.system.DocDB class to check for the existence of a document database called People. Not finding the database, it then uses the **CreateDatabase()** method to create it:

```
>>> iris.system.DocDB.Exists('People')
0
>>> db = iris.system.DocDB.CreateDatabase('People')
>>> iris.system.DocDB.Exists('People')
1
```

For more information, see Introducing InterSystems IRIS Document Database.

Encryption

The %SYSTEM.Encryption class provides class functions to perform data encryption, Base64 encoding, hashing, and generation of message authentication codes.

The following example uses the **ListEncryptionKeys()** method of the iris.system.Encryption class to list the key IDs of all active encryption keys, which can be used for data element encryption for applications.

```
>>> iris.system.Encryption.ListEncryptionKeys()
'1E5C9E0D-1257-4707-8864-3428E17A6FCE'
```

The following example uses the **CreateEncryptionKey()** method to create a 32-byte (256–bit) encryption key. The method returns the key ID of the generated encryption key. Creating and activating keys need to be done in the %SYS namespace.

```
>>> iris.system.Process.SetNamespace('%SYS')
'%SYS'
>>> st = iris.ref('')
>>> iris.system.Encryption.CreateEncryptionKey('c:\\temp\\mykeyfile.txt','<username>','<password>',32,st)
'C861F668-435A-4AE8-A302-FBD4ED528432'
>>> st.value
1
```

The **CreateEncryptionKey()** method passes back a %Status by reference in the last argument, so the example above uses **iris.ref()** to create a reference to the variable st. Access the value property of st to retrieve the actual status.

The following example uses the **ActivateEncryptionKey()** method to activate the encryption key created in the previous example. Then it calls **ListEncryptionKeys()** again to show it is now active.

```
>>> status =
iris.system.Encryption.ActivateEncryptionKey('c:\\temp\mykeyfile.txt','<username>','<password>')
>>> status
1
>>> iris.system.Encryption.ListEncryptionKeys()
'1E5C9E0D-1257-4707-8864-3428E17A6FCE,C861F668-435A-4AE8-A302-FBD4ED528432'
```

For more information on using encryption in InterSystems IRIS, see Encryption.

Error

The %SYSTEM.Error class is a generic error container used to return error information reported from various sources, in various different forms. The class has properties that can store a %Status, a $ZERROR code, a SQLCODE, and/or a free text message. It can also contains methods that convert a %SYSTEM.Error to an InterSystems IRIS exception and vice-versa.

The following example shows how to create a new generic error object from a %Status, after trying to create and save an object of an InterSystems IRIS class called Sample.Company:

```
>>> my_company = iris.Sample.Company._New()
>>> my_company.TaxID = '123456789'
>>> status = myCompany._Save()
>>> e = iris.system.Error._New(status)
>>> print(e.Message)
ERROR #5659: Property 'Sample.Company::Name(8@Sample.Company,ID=)' required
```

For more information, see iris.system.Status, Working with %Status Values, $ZERROR (ObjectScript), and SQL Error Messages.

Event

The %SYSTEM.Event class provides an interface to the Event API. It allows processes to go to sleep waiting for a particular resource, which can be a named resource or a process ID. The **Wait()** method of the iris.system.Event class causes a process to wait on a resource, while the **Signal()** method wakes up a process waiting for a resource.

A named resource is a string representation of a valid identifier, much the same as that used for locks. Named resources are explicitly created and deleted, using the **Create()** and **Delete()** methods respectively.

A process ID is automatically created when a process is created and automatically deleted when a process exits. A process may wait only on its own process identifier.

In the following example, a process creates a named resource called MyResource by using the **Create()** method of iris.system.Event:

```
>>> iris.system.Event.Create('MyResource')
1
```

A second process can then use the **Wait()** method of iris.system.Event to wait on this resource before continuing, as shown in this Python module, waiter.py:

```
import iris

def run():
    print('Waiting for signal')
    ret = iris.system.Event.Wait('MyResource',30)
    if ret == 1:
        print('Signal received')
    elif ret == -1:
        print('Resource deleted')
    elif ret == 0:
        print('Wait timed out')
```

The **run()** function of the waiter module prints "Waiting for signal," and then calls the **Wait()** method of iris.system.Event. It then waits up to 30 seconds for `MyResource` to become available before continuing along one of three paths, depending on the return value of the **Wait()** method.

If the first process then uses the **Signal()** method of the iris.system.Event class, it wakes up the second process:

```
>>> iris.system.Event.Signal('MyResource')
1
```

If no process is waiting for the resource, the wakeup is queued, and when another process waits for the resource, it is awakened immediately.

Executing the **run()** function of the waiter module might look something like this:

```
>>> import waiter
>>> waiter.run()
Waiting for signal
<process sleeps for up to 30 seconds>
Signal received
```

If the first process uses the **Delete()** method of the iris.system.Event class instead, the **run()** function prints "Resource deleted."

If no response is received after 30 seconds, the output is "Wait timed out."

For more information, see %SYSTEM.Event and iris.lock().

Monitor

The %SYSTEM.Monitor class provides an interface for accessing the System Monitor, allowing you to perform such functions as getting the number of alerts posted to messages.log, reading the messages in alerts.log, or checking the system state.

The following example uses the **State()** method of the iris.system.Monitor.State class to return the current system state as an integer:

```
>>> iris.system.Monitor.State()
2
```

System states are determined by the number of system alerts posted to messages.log during or following startup. In this case, a system state of 2 means Alert (or RED). For further details on system states, see System Monitor Health State.

Use the **Alerts()** method to return the current number of system alerts:

```
>>> iris.system.Monitor.Alerts()
10
```

The following example (for InterSystems IRIS 2023.2 or later) calls the **GetAlerts()** method of the iris.system.Monitor class, which returns the number of alerts in alerts.log, the text of the alerts as an array of messages, and a string containing the text of the most recent alert:

```
>>> alerts = iris.ref(0)
>>> messages = iris.arrayref({})
>>> last_alert = iris.ref('')
>>> status = iris.system.Monitor.GetAlerts(alerts, messages, last_alert)
>>> for x in range(1, alerts.value + 1):
...     print(messages[x])
...
12/02/24-14:31:55:470 (68232) 2 [Generic.Event] Failed to allocate 1592MB shared memory using large
pages.  Switching to small pages.
12/02/24-14:31:57:762 (56676) 2 [Utility.Event] Preserving journal files
C:\InterSystems\IRIS\mgr\journal\20241128.002 and later for journal recovery and transaction rollback
12/03/24-06:29:54:435 (66624) 2 [Utility.Event] LMF Error:  Could not send startup message to license
 server
12/03/24-19:48:32:895 (66624) 2 [Utility.Event] LMF Error:  Could not send startup message to license
 server
12/04/24-11:18:43:264 (66624) 2 [Utility.Event] LMF Error:  Could not send startup message to license
 server
12/04/24-12:41:12:897 (66624) 2 [Utility.Event] LMF Error:  Could not send startup message to license
 server
12/07/24-07:35:44:424 (66624) 2 [Utility.Event] LMF Error:  Could not send startup message to license
 server
12/10/24-00:52:38:787 (45112) 2 [Utility.Event] Preserving journal files
c:\intersystems\iris\mgr\journal\20241207.002 and later for journal recovery and transaction rollback
12/10/24-05:42:01:245 (66624) 2 [Utility.Event] LMF error:  Could not connect to license server
(127.0.0.1,4002).
12/10/24-15:29:54:916 (42260) 2 [Generic.Event] Process terminated abnormally (pid 13056, jobid
0x0002001b)
>>> last_alert.value
'12/10/24-15:29:54:916 (42260) 2 [Generic.Event] Process terminated abnormally (pid 13056, jobid
0x0002001b)'
```

The **GetAlerts()** method's three arguments are all passed by reference, so it is necessary to use iris.ref() and iris.arrayref() to pass in references to the Python variables that will hold the returned results.

For more information, see Using System Monitor and Monitoring InterSystems IRIS Logs.

Process

The %SYSTEM.Process class allows you to monitor and manipulate a process.

Some of the methods in iris.system.Process require a process ID (pid) to be passed as an argument, such as **State()**, which returns the current state of the process:

```
>>> iris.system.Process.State(8608)
'READ'
```

Some of its methods act upon the current process, such as **SetNamespace()**, which sets the namespace of the current process:

```
>>> iris.system.Process.SetNamespace('USER')
'USER'
```

Other methods work on either the current process or another process, such as **NameSpace()**, which returns the current namespace for a given process:

```
>>> iris.system.Process.NameSpace()
'USER'
>>> iris.system.Process.NameSpace(44828)
'%SYS'
```

For more information, see %SYSTEM.Process.

SQL

The %SYSTEM.SQL class provides a mechanism for preparing and executing SQL queries. The key methods of iris.system.SQL are **Prepare()** and **Execute()**.

The following examples use the Sample.Person class described in Use an InterSystems IRIS Class.

The example below executes a SQL query that requests the name, age, and date of birth of the first 10 rows in the table Sample.Person and places the result set in the variable `rs1`:

```
>>> rs1 = iris.system.SQL.Execute('SELECT TOP 10 Name,Age,DOB FROM Sample.Person','DISPLAY')
```

The second argument in the call to **Execute()** is the SelectMode, which in this case tells the method to print the Display values of each field, rather than the Logical values. For example, a human-readable format for the DOB field will be used, rather than the internal **$HOROLOG** format.

The result set is an instance of the class %SQL.StatementResult, so you can manipulate the result set using the methods in that class, such as **%Display()** to print the contents of the result set:

```
>>> rs1._Display()
Name      Age     DOB
Avery,Zelda O.   40      05/13/1984
Quincy,Debby I. 48      06/11/1976
Schaefer,Barb G.         20      03/08/2005
Newton,Angela L.         46      05/30/1978
Thompson,Emily Y.        23      11/11/2001
O'Brien,James D.         71      10/12/1953
Baker,Valery F. 73      09/22/1951
Kratzmann,Frances T.     68      09/07/1956
Quixote,Janice I.        32      05/25/1992
Tsatsulin,Danielle P.    42      08/04/1982

10 Rows(s) Affected
```

The next example stores a similar query in a variable `q`:

```
q = iris.system.SQL.Prepare('SELECT TOP ? Name,Age,DOB FROM Sample.Person','DISPLAY')
```

The question mark in the query allows you to pass in an argument when you execute the query, in this case, specifying how many rows to place in the result set.

The query is an instance of the class %SQL.Statement, so you can manipulate the query using the methods in that class, such as **%Execute()** to run the query. This example requests the top 5 rows in the table and places the result set in the variable `rs2`.

```
>>> rs2 = q._Execute('5')
```

You can use the %Next() method of the of the class %SQL.StatementResult to iterate through the result set:

```
>>> while rs2._Next():
...     print(f'{rs2.Name} is {rs2.Age} years old and was born on {rs2.DOB}')
...
Avery,Zelda O. is 40 years old and was born on 05/13/1984
Quincy,Debby I. is 48 years old and was born on 06/11/1976
Schaefer,Barb G. is 20 years old and was born on 03/08/2005
Newton,Angela L. is 46 years old and was born on 05/30/1978
Thompson,Emily Y. is 23 years old and was born on 11/11/2001
```

For more information, see Using Dynamic SQL.

Use the iris.sql class to return a result set that can be iterated in a more Python-like way. See Use InterSystems SQL for more information.

SYS

The %SYSTEM.SYS class provides a language-independent way to access selected system variables, also known as "special variables."

The **Horolog()** method of the iris.system.SYS class gets the value of system variable **$HOROLOG**, which contains the local date and time in internal format. The first number is the current day, where day 1 is January 1, 1841, and the second number is the number of seconds since midnight of the current day. See $HOROLOG for details.

```
>>> iris.system.SYS.Horolog()
'67292,57639'
```

The **TimeStamp()** method gets the UTC date and time in internal format. See $ZTIMESTAMP for details.

```
>>> iris.system.SYS.TimeStamp()
'67292,72041.4628646'
```

Subtracting the number of seconds since midnight in the above examples indicates that the current time is 4 hours behind UTC.

The **TimeZone()** method retrieves the offset from the Greenwich meridian in minutes. See $ZTIMEZONE for details.

```
>>> iris.system.SYS.TimeZone()
300
```

This shows that the local time in this example is 5 time zones from the Greenwich meridian. This apparent discrepancy arises due to the fact that UTC is unaffected by daylight saving time or other time variants that may affect local time.

For more information, see %SYSTEM.SYS.

Security

The %SYSTEM.Security class provides an interface for performing certain tasks related to user permissions.

For example, role escalation can be used when a user occasionally needs to perform certain functions that require a higher level of permissions than the user would normally have. This requires that the user already have the `%DB_IRISSYS:W` and `%Service_EscalateLogin:U` privileges.

You can check to see if a user has a particular privilege by using the **CheckUserPermission()** method of the iris.system.Security class, for example:

```
>>> iris.system.Security.CheckUserPermission('userone','%DB_IRISSYS')
'READ,WRITE'
>>> iris.system.Security.CheckUserPermission('userone','%Service_EscalateLogin')
'USE'
```

or

```
>>> iris.system.Security.CheckUserPermission('userone','%DB_IRISSYS','WRITE')
1
>>> iris.system.Security.CheckUserPermission('userone','%Service_EscalateLogin','USE')
1
```

The user also needs to have the role with the higher level of privilege defined as an escalation role. This can be configured in the Management Portal by going to **System** > **Security Management** > **Users** and selecting the user you wish to have an escalation role. Then click the **EscalationRoles** tab, where you can assign roles that the user can escalate to.

Then, to escalate the user to a new role, use the **EscalateLogin()** method. The following example escalates the current user to the `%Manager` role:

```
>>> iris.system.Security.EscalateLogin('%Manager')

Escalated Login for User: userone (Role: %Manager)
Password: *********

USER (%Manager)#
```

This drops you back to the ObjectScript shell with `%Manager` role. You can then execute commands in ObjectScript or re-enter the Python shell with the new level of permissions.

For more information, see Roles.

Semaphore

The %SYSTEM.Semaphore class provides an interface for managing semaphores. Whereas a lock can be used to ensure that only one process can use a given resource, a semaphore can be used when multiple copies of a resource can exist. For example, you might want to allow a certain number of concurrent connections to a database, forcing an additional process to wait for a resource to become available.

The following sample module, sem.py, shows how semaphores can be used to regulate a process that generates a resource and a process that consumes a resource.

```python
import time
import random
import iris

# Create a semaphore 'resource_sem' with a starting number of resources
def create(num_resources):
    my_sem = iris.system.Semaphore._New()
    my_sem.Create('resource_sem', num_resources)
    print(f'Created semaphore with value {my_sem.GetValue()}.')

# Return current value of semaphore 'resource_sem'
def value():
    my_sem = iris.system.Semaphore._New()
    my_sem.Open('resource_sem')
    return(my_sem.GetValue())

# Produce a resource and increment semaphore 'resource_sem'
def produce():
    my_sem = iris.system.Semaphore._New()
    my_sem.Open('resource_sem')
    time.sleep(random.uniform(4, 8))  # Insert some randomness
    my_sem.Increment(1)  # Increment semaphore
    print('Done producing.')

# Consume a resource and decrement semaphore 'resource_sem'
def consume():
    my_sem = iris.system.Semaphore._New()
    my_sem.Open('resource_sem')
    time.sleep(random.uniform(5, 10))  # Insert some randomness
    result = my_sem.Decrement(1, 10)  # Decrement semaphore with a 10-sec timeout
    if result == 0:
        print('Timed out waiting for semaphore.')
    else:
        print('Done consuming.')

# Delete semaphore 'resource_sem'
def delete():
    my_sem = iris.system.Semaphore._New()
    my_sem.Open('resource_sem')
    print(f'Deleting semaphore with value {my_sem.GetValue()}.')
    my_sem.Delete()
```

The function **create()** instantiates the iris.system.Semaphore class, uses the **Create()** method of that class to create a semaphore with the name resource_sem and set its initial value to num_resources. This semaphore can now be accessed by any process.

The **value()** function uses the **Open()** method of iris.system.Semaphore to open the semaphore with the name resource_sem and then uses the **GetValue()** method to retrieve the current value of the semaphore. This value represents the current number of resources that are waiting to be consumed.

The **produce()** function represents the production of a resource for later consumption and uses the **Increment()** method of iris.system.Semaphore to increment the value of the semaphore.

The **consume()** function represents the consumption of a resource and uses the **Decrement()** method of iris.system.Semaphore to decrement the value of the semaphore and indicate that a resource was consumed. If the method finds that the semaphore's value is 0 at the time **Decrement()** is called, the method will wait up to 10 seconds for a resource to be produced.

Finally, the **delete()** function uses the **Delete()** method of iris.system.Semaphore to delete the semaphore.

When run in Terminal, the following example creates a the `num_resources` semaphore and assigns it the initial value 0, which means no resources are yet created. It then produces a resource and shows that the value of the semaphore is now 1, meaning that a resources is ready to be consumed.

```
>>> import sem
>>> sem.create(0)
Created semaphore with value 0.
>>> sem.produce()
Done producing.
>>> sem.value()
1
```

When run in a second Terminal session, the following example shows that you can retrieve the value of the semaphore. Since the value of the semaphore is 1, the process can consume the resource immediately and decrement the semaphore value.

```
>>> import sem
>>> sem.value()
1
>>> sem.consume()
Done consuming.
```

Now, the value of the semaphore is 0, and if a process tries to consume a resource, it will wait up to 10 seconds for a process to produce a new resource. If this happens, it will proceed and consume the resource. Otherwise, the call to **Decrement()** returns 0 to indicate it timed out.

```
>>> sem.value()
0
>>> sem.consume()
Timed out waiting for semaphore.
```

Since semaphores are shared objects, they do not go away unless you delete them explicitly by calling the **Delete()** method of iris.system.Semaphore.

```
>>> sem.delete()
Deleting semaphore with value 0.
```

The following sample module, threads.py, exercises the sample semaphore code in sem.py by spinning up threads for an equal number of producers and consumers.

```
# Driver takes starting number of resources
# And number of producer/consumer threads to run
def main(num_resources, num_threads):

    # Create semaphore with specified number of resources
    sem.create(num_resources)

    # Thread to produce a resource
    def start_producer(thread_id):
        print(f'Producer {thread_id} started. Semaphore value is {sem.value()}.')
        sem.produce()

    # Thread to consume a resource
    def start_consumer(thread_id):
        print(f'Consumer {thread_id} started. Semaphore value is {sem.value()}.')
        sem.consume()

  # Spawn multiple threads simulating concurrent producer/consumer requests
    threads = []
    for i in range(num_threads):
        t = threading.Thread(target=start_producer, args=(i,))
        time.sleep(2)
        threads.append(t)
        t.start()
        time.sleep(2)
        t = threading.Thread(target=start_consumer, args=(i,))
        threads.append(t)
        t.start()

    # Wait for all threads to finish
    for t in threads:
        t.join()
```

```
    # Clean up
    print(f"Exercise complete. Final semaphore value is {sem.value()}.")
    sem.delete()
```

The following example runs threads.py, with a starting semaphore value of 0 and spins up 5 producer threads and 5 consumer threads.

```
>>> import threads
>>> threads.main(0,5)
Created semaphore with value 0.
Producer 0 started. Semaphore value is 0.
Consumer 0 started. Semaphore value is 0.
Producer 1 started. Semaphore value is 0.
Done producing.
Consumer 1 started. Semaphore value is 1.
Producer 2 started. Semaphore value is 1.
Done producing.
Consumer 2 started. Semaphore value is 2.
Done consuming.
Producer 3 started. Semaphore value is 1.
Done consuming.
Consumer 3 started. Semaphore value is 0.
Done producing.
Producer 4 started. Semaphore value is 1.
Done producing.
Consumer 4 started. Semaphore value is 2.
Done consuming.
Done consuming.
Done producing.
Done consuming.
Exercise complete. Final semaphore value is 0.
Deleting semaphore with value 0.
```

Since the number of producers and consumers is equal and no timeouts occurred, the final value of the semaphore is 0.

To read about other semaphore scenarios, see Semaphores in InterSystems Products or %SYSTEM.Semaphore.

Status

The %SYSTEM.Status class provides methods that can display or construct InterSystems IRIS status values, that is, objects of type %Library.Status (or %Status for short). Status values are commonly returned by methods in InterSystems IRIS. If a status has the value 1, it indicates that the method was a success. Otherwise, the status contains one or more error codes and messages to indicate any error conditions that occurred while running the method.

If you call an InterSystems IRIS method that returns a status, **iris.system.Status.IsOK()** returns 1 if the status is not an error, while **iris.system.Status.IsError()** returns 1 if the status is contains one or more errors. The **DisplayError()** method displays the contents of a status, including its error codes and messages.

If you write a method and you want to return a status, **iris.system.Status.OK()** returns a non-error status, while **iris.system.Status.Error()** lets you create and return an error status.

For example, say you are working with a simple persistent class that defines a pet object, which contains two required properties, Name and Species:

```
Class User.Pet Extends %Library.Persistent
{
Property Name As %String [ Required ];

Property Species As %String [ Required ];
}
```

Creating a new instance of a pet and trying to save it returns a status value, which you can check to see if the save succeeded, as well as display any errors that occur. If you try to save a pet without giving the Species property a value, you will see error #5659, indicating a required property is missing.

```
>>> status = iris.system.Status
>>> p = iris.User.Pet._New()
>>> p.Name = 'Fluffy'
>>> st = p._Save()
>>> if status.IsOK(st):
...      print('Pet saved')
... elif status.IsError(st):
...      status.DisplayError(st)
...

ERROR #5659: Property 'User.Pet::Species(8@User.Pet,ID=)' required
```

See InterSystems Error Reference for a list of possible error codes and messages.

You can also **iris.check_status()** to check the status returned by the **_Save()** method and throw a Python exception if it contains an error. This allows you to handle the error in a standard Python Try...Except. See iris.check_status() for more information.

Also, you might find situations where you are writing a method in Python and you need to return a status value, for instance, to some ObjectScript code. For example, you are writing a module souffle.py that contains a **bake()** function that needs to return a status that tells the caller if the souffle came out successfully or not.

```
def bake():
    import iris
    import random
    result = random.randint(1,10)
    if result > 6:
        st = iris.system.Status.Error(5001,'The souffle fell.')
    else:
        st = iris.system.Status.OK()
    return st
```

From looking at the code, you can see that there is a 40 percent chance that the souffle will fall. In such cases, the code will create a custom error, using the **iris.system.Status.Error()** method, and return it to the caller. InterSystems IRIS reserves the error codes 83 and 5001 for custom errors. If the souffle comes out successfully, it returns a non-error status, using the **OK()** method.

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

USER>set st = souffle.bake()

USER>do ##class(%SYSTEM.Status).DisplayError(st)

ERROR #5001: The souffle fell.
```

For more information, see Working with %Status Values.

Util

The %SYSTEM.Util class provides an assortment of utility methods that can be useful for a variety of purposes. Just a few of these methods are explained here, to give you an idea of the breadth of this class.

Some of the utility methods in the iris.system.Util class can be used to monitor or administer the current state of InterSystems IRIS.

The method **IsDST()** determines whether the timestamp representing the current date and time is in daylight saving time. A return value of 1 indicates daylight saving time is currently in effect. For more information on the InterSystems IRIS date format, see $HOROLOG.

```
>>> iris.system.Util.IsDST()
1
```

The **SetSwitch()** method allows you to set a switch, controlling the instance-level functioning of InterSystems IRIS. For example, switch 12 inhibits new users or processes from logging in to the instance. See Using Switches for more information.

```
>>> iris.system.Util.SetSwitch(12,1)
0
```

Here, **SetSwitch()** is being used to turn on switch 12. The return value of 0 indicates the previous state of the switch 12, which was off. With switch 12 turned on, a user trying to log in to InterSystems IRIS sees the message:

```
<InterSystems IRIS Startup Error: Sign-on and JOB inhibited: Switch 12 is set (0)>
```

Other methods in iris.system.Util can be used to check certain attributes at the operating system level. The following examples are for Microsoft Windows.

The **GetEnviron()** method retrieves the value of an environment variable:

```
>>> iris.system.Util.GetEnviron('PATHEXT')
'.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC;.PY;.PYW'
```

The **InstallDirectory()** method retrieves the directory where the instance of InterSystems IRIS is installed:

```
>>> iris.system.Util.InstallDirectory()
'c:\\intersystems\\iris\\'
```

Still other methods in iris.system.Util perform simple tasks that are useful for manipulating data.

The **HexToDecimal()** and **DecimalToHex()** methods convert hex data to decimal and vice versa:

```
>>> iris.system.Util.HexToDecimal('BEAD')
'48813'
>>> iris.system.Util.DecimalToHex(1023)
'3FF'
```

The methods **Compress()** and **Decompress()** can be used to compress and decompress data, for example, with the zlib algorithm:

```
>>> compressed_string = iris.system.Util.Compress('The %SYSTEM.Util class contains useful utility
methods','zlib')
>>> compressed_string
'6x\x9c\x0bÉHUP\r\x8e\x0c\x0eqõÕ\x0b-ÉÌQHÎI,.VHÎÏ+IÌÌ+V(-NM+ÍQ(\x05Je\x96T*ä¦\x96dä§\x14\x03\x00\x02
\x13É\x01'
>>> decompressed_string = iris.system.Util.Decompress(compressed_string)
>>> decompressed_string
'The %SYSTEM.Util class contains useful utility methods'
```

For more information, see %SYSTEM.Util.

Version

The %SYSTEM.Version class provides methods for retrieving product version information.

For example, you can retrieve the entire version string (often referred to as the **$zv** string) or a piece of it:

```
>>> iris.system.Version.GetVersion()
'IRIS for Windows (x86-64) 2024.2 (Build 230U) Mon Jul 1 2024 16:40:47 EDT'
>>> iris.system.Version.GetOS()
'Windows'
```

You can also set the system mode using the **SystemMode()** method, instead of having to use the Management Portal:

```
>>> iris.system.Version.SystemMode('TEST')
'DEVELOPMENT'
```

The example above causes the Management Portal to display "Test System" at the top, to let you quickly identify what environment you are in. The return value of the method is the previous system mode of the InterSystems IRIS instance.

For more information, see %SYSTEM.Version.