Using Embedded Python

Version 2024.1
2024-05-02

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

Using Embedded Python
InterSystems IRIS Data Platform Version 2024.1 2024-05-02

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.

On Microsoft Windows, the InterSystems IRIS installation kit installs the correct version of Python (currently 3.9.5) for use with Embedded Python only. If you are on a development machine and want to use Python for general purposes, InterSystems recommends downloading and installing this same version from https://www.python.org/downloads/.

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

On some operating systems, you can override the version of Python recommended for Embedded Python by using the Flexible Python Runtime feature. This is useful if you are writing code or using a package that depends on a version of Python other than the one recommended. 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.

The configuration setting `PythonRuntimeLibrary` specifies the location of the Python runtime library to use when running Embedded Python. This version of the library overrides the default version of the library used when launching Embedded Python.

To specify a Python runtime library:

1. In the Management Portal, go to **System Administration** > **Configuration** > **Additional Settings** > **Advanced Memory**.
2. In the **PythonRuntimeLibrary** row, click **Edit**.
3. Enter the location of the Python runtime library you want to use.

   For example: `/usr/lib/x86_64-linux-gnu/libpython3.11.so.1`

   This location will vary based on your operating system, Python version, and other factors. The example shown here is for Python 3.11 on Ubuntu 22.04 on the x86 architecture.
4. Click **Save**.

For more information, see PythonRuntimeLibrary.

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.

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`.

For more details on how to configure this feature, 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 library 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

On Windows, use the built-in **irispip** command from the <installdir>/bin directory: `irispip install --target <installdir>\mgr\python <package>`.

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

```
C:\InterSystems\IRIS\bin>irispip 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:

ObjectScript

```
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, including considerations of line indentation.

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.cls('%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.cls('%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.cls('%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.cls('%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.cls('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 library
- 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 Library

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 example, the ReportLab Toolkit is an open source library for generating PDFs and graphics. The following command uses the package installer `irispip` to install ReportLab on a Windows system:

```
C:\InterSystems\IRIS\bin>irispip install --target C:\InterSystems\IRIS\mgr\python reportlab
```

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

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

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

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

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:

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` library, which can be installed from the command line on Windows as follows:

```
C:\InterSystems\IRIS\bin>irispip install --target C:\InterSystems\IRIS\mgr\python usaddress-scourgify
```

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

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

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

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

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 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

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.

5.1 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.

5.2 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__`. 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.cls('User.Person')._New()
```

5.3 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:

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

If the method **mymethod()** were in a module called mymodule.py, 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.

5.4 Passing 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"
}
```

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.cls('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.

5.5 Passing 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
```

5.6 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, if you have an InterSystems IRIS class called User.ArrayTest that has a method **WriteContents()** that writes the contents of an array, you can call it as follows:

```
>>> iris.cls('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.

5.7 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'>
```

5.8 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.cls('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.

5.9 Changing 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 %SYSTEM.SYS 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.cls('%SYSTEM.SYS').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>
```

5.10 Running 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
```

5.11 Exception Handling

The InterSystems IRIS exception handler can handle Python exceptions and pass them seamlessly to ObjectScript. Building on the earlier Python library 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).

5.12 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.

The following example turns an InterSystems IRIS string to a Python object of type bytes:

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

USER>zwrite b
b=8@%SYS.Python  ; b'Hello Bytes!'  ; <OREF>

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

5.13 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")
```

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.

6

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. 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 libraries 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` Python library to send that string to a phone in a text message via the Amazon Simple Notification Service (SNS). The scope of this AWS library 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.

One exception to this rule is that you can implement a callback method in Python if it does not use the input from the adapter.

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. In this example the OnProcessInput() callback method can be implemented in Python because it does not make use of the pInput argument in the method's signature.

```
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 = python ]
{
    import iris
    import random
    fruits = ["apple", "banana", "cherry"]
    fruit = random.choice(fruits)
    request = iris.cls('Ens.StringRequest')._New()
    request.StringValue = fruit + ' ' + iris.cls('Debug.Service.Poller').GetSomeText()
    return self.SendRequestAsync(self.Target,request)
}

ClassMethod GetSomeText() As %String
{
    Quit "is something to eat"
}

}
```

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

7

Use the Flexible Python Runtime Feature

7.1 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, there are 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 use these versions of Python 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.

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.

   For example:`/usr/lib/x86_64-linux-gnu/libpython3.11.so.1`

   This location will vary based on your operating system, Python version, and other factors. The example shown here is for Python 3.11 on Ubuntu 22.04 on the x86 architecture.

   See PythonRuntimeLibrary for more information.
3. Ensure that the `sys.path` variable in Embedded Python includes the correct directories needed to import Python packages.

   See Embedded Python and sys.path.

7.1.1 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, `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

  For example: /usr/local/lib/python3.10/dist-packages.

  This location will vary based on your operating system, Python version, and other factors. The example shown here is for Python 3.10 on Ubuntu 22.04.

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: If `sys.path` contains a directory in that is within your home directory, such as /home/<user>/.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 like Anaconda, 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.

For Ubuntu 22.04, when using Anaconda, edit your iris_site.py file to look something like the following:

```
import sys
from site import getsitepackages as __sitegetsitepackages

# modify EmbeddedPython to get site-packages from Anaconda python distribution.

def set_site_path(platform_name):
    sys.path = sys.path + __sitegetsitepackages(['/opt/anaconda3/'])
```

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
['/opt/anaconda3/lib/python311.zip', '/opt/anaconda3/lib/python3.11',
'/opt/anaconda3/lib/python3.11/lib-dynload',
'/InterSystems/IRIS/lib/python', '/InterSystems/IRIS/mgr/python',
'/opt/anaconda3/lib/python3.11/site-packages']
```

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` configuration setting or iris_site.py take effect on starting a new session. Restarting InterSystems IRIS is not required.

7.1.2 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 20.04 on the x86 architecture, shows that the Python runtime library being used is `/usr/lib/x86_64-linux-gnu/libpython3.10.so.1`, the running version of Embedded Python is `3.10.13`, and the system version is `3.8.10`.

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

USER>zw info
info("CPF_PythonPath")=""
info("CPF_PythonRuntimeLibrary")="/usr/lib/x86_64-linux-gnu/libpython3.10.so.1"
info("RunningLibrary")="/usr/lib/x86_64-linux-gnu/libpython3.10.so.1"
info("RunningVersion")="3.10.13 (main, Aug 25 2023, 13:20:03) [GCC 9.4.0]"
info("SystemPath")="/usr/lib/python3.8/config-3.8-x86_64-linux-gnu"
info("SystemVersion")="3.8.10 (default, Nov 14 2022, 12:59:47) [GCC 9.4.0]"
info("SystemVersionShort")="3.8.10"
```

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

7.2 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. Set the `PythonRuntimeLibrary` configuration setting to `/usr/lib/x86_64-linux-gnu/libpython3.11.so.1`.

   See PythonRuntimeLibrary for more information.

4. 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/lib/python',
   '/InterSystems/IRIS/mgr/python', '/usr/local/lib/python3.11/dist-packages',
   '/usr/lib/python3/dist-packages',
   '/usr/lib/python3.11/dist-packages']
   >>>
   ```

5. 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"
info("RunningLibrary")="/usr/lib/x86_64-linux-gnu/libpython3.11.so.1"
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")="3.10.6 (main, Nov 14 2022, 16:10:14) [GCC 11.3.0]"
info("SystemVersionShort")="3.10.6"
```

   This example shows that the Python runtime library being used is `/usr/lib/x86_64-linux-gnu/libpython3.11.so.1`, the running version of Embedded Python is `3.11.0rc1`, and the system version is `3.10.6`.

7.3 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.

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 from https://www.anaconda.com/download.

   The download will be a shell script with a name similar to Anaconda3-2024.02-1-Linux-x86_64.sh.

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

   ```
   $ sudo sh Anaconda3-2024.02-1-Linux-x86_64.sh -b -p /opt/anaconda3
   ```

3. Set the `PythonRuntimeLibrary` configuration setting to `/opt/anaconda3/lib/libpython3.11.so`.

   See PythonRuntimeLibrary for more information.

4. Edit the iris_site.py file in the directory <installdir>/lib/python to add the Anaconda package repository to `sys.path`:

```
import sys
from site import getsitepackages as __sitegetsitepackages

# modify EmbeddedPython to get site-packages from Anaconda python distribution.

def set_site_path(platform_name):
    sys.path = sys.path + __sitegetsitepackages(['/opt/anaconda3/'])
```

5. If you launch Embedded Python and see the error `<class 'ModuleNotFoundError'>: No module named 'math' - iris loader failed`, edit your `PATH` environment variable to add the directory `/opt/anaconda3/bin` to the front:

```
$ export PATH=/opt/anaconda3/bin:$PATH
$ env |grep PATH
PATH=/opt/anaconda3/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin:/snap/bin
```

6. 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
['/opt/anaconda3/lib/python311.zip', '/opt/anaconda3/lib/python3.11',
'/opt/anaconda3/lib/python3.11/lib-dynload',
'/InterSystems/IRIS/lib/python', '/InterSystems/IRIS/mgr/python',
'/opt/anaconda3/lib/python3.11/site-packages']
```

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

```
USER>zw info
info("AllowNonSystemPythonForIntegratedML")=0
info("CPF_PythonPath")=""
info("CPF_PythonRuntimeLibrary")="/opt/anaconda3/lib/libpython3.11.so"
info("RunningLibrary")="/opt/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")="3.10.6 (main, Nov 14 2022, 16:10:14) [GCC 11.3.0]"
info("SystemVersionShort")="3.10.6"
```

This example shows that the Python runtime library being used is `/opt/anaconda3/lib/libpython3.11.so`, the running version of Embedded Python is `3.11.7`, and the system version is `3.10.6`.

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(), 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 global.

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.cls('User.ArrayTest').Modify(a)
>>> iris.cls('User.ArrayTest').WriteContents(a)
myArray(2,3)=4
myArray("new")=123
myArray("x","y","z")="xyz"
>>> iris.cls('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.cls('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
```

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.cls('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 used in ObjectScript, see Routines.

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"
```