InterSystems™
IRIS Data Platform

Using Embedded Python

Version 2023.1
2023-06-15

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

Using Embedded Python
InterSystems IRIS Data Platform Version 2023.1 2023-06-15
Copyright © 2023 InterSystems Corporation
All rights reserved.

InterSystems®, InterSystems IRIS®, IntegratedML®, InterSystems HealthShare®, InterSystems HealthShare Care Community®, HealthShare Unified Care Record®, InterSystems Caché®, and InterSystems Ensemble® are registered trademarks of InterSystems Corporation.InterSystems IRIS for Health™ and HealthShare® CMS Solution Pack™ are trademarks of InterSystems Corporation.

All other brand or product names used herein are trademarks or registered trademarks of their respective companies or organizations.

This document contains trade secret and confidential information which is the property of InterSystems Corporation, One Memorial Drive, Cambridge, MA 02142, or its affiliates, and is furnished for the sole purpose of the operation and maintenance of the products of InterSystems Corporation. No part of this publication is to be used for any other purpose, and this publication is not to be reproduced, copied, disclosed, transmitted, stored in a retrieval system or translated into any human or computer language, in any form, by any means, in whole or in part, without the express prior written consent of InterSystems Corporation.

The copying, use and disposition of this document and the software programs described herein is prohibited except to the limited extent set forth in the standard software license agreement(s) of InterSystems Corporation covering such programs and related documentation. InterSystems Corporation makes no representations and warranties concerning such software programs other than those set forth in such standard software license agreement(s). In addition, the liability of InterSystems Corporation for any losses or damages relating to or arising out of the use of such software programs is limited in the manner set forth in such standard software license agreement(s).

THE FOREGOING IS A GENERAL SUMMARY OF THE RESTRICTIONS AND LIMITATIONS IMPOSED BY INTERSYSTEMS CORPORATION ON THE USE OF, AND LIABILITY ARISING FROM, ITS COMPUTER SOFTWARE. FOR COMPLETE INFORMATION REFERENCE SHOULD BE MADE TO THE STANDARD SOFTWARE LICENSE AGREEMENT(S) OF INTERSYSTEMS CORPORATION, COPIES OF WHICH WILL BE MADE AVAILABLE UPON REQUEST.

InterSystems Corporation disclaims responsibility for errors which may appear in this document, and it reserves the right, in its sole discretion and without notice, to make substitutions and modifications in the products and practices described in this document.

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

Using Embedded Python

Embedded Python allows you to use Python as a native option for programming InterSystems IRIS applications. If you are new to Embedded Python, read Demo: Using 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 Prerequisites

The version of Python required to use 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. Using a different version of Python will result in errors when using Embedded Python.

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:

- macOS: Install Python 3.9 using Homebrew (https://formulae.brew.sh/formula/python@3.9)
- Ubuntu: `apt-get install python3`
- Red Hat Enterprise Linux or Oracle Linux: `yum install python3`
- SUSE: `zypper install python3`

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.

Important: If 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 correct 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.

On a UNIX-based system, you may want to install Python packages with the **pip3** command. If you do not have **pip3** installed already, install the package `python3-pip` with your system’s package manager.

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.

2 Run Embedded Python

The section details several ways to run Embedded Python:

- From the Python Shell
- In a Python Script File (.py)
- In a Method in an InterSystems IRIS Class
- In SQL functions and stored procedures

All of these ways allow you to call InterSystems IRIS APIs by importing the `iris` Python module and using its methods.

2.1 From the Python Shell

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

2.1.1 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.

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. Just go ahead and use the module.

2.1.2 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 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.

2.2 In a Python Script File (.py)

You can also use the **irispython** command to execute a Python script.

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.

2.3 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
```

2.4 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 Use a Python Library.

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
```

3 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 Bridging the Gap Between ObjectScript and Embedded Python.

3.1 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, use:

```
$ pip3 install --target /InterSystems/IRIS/mgr/python reportlab
```

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:

InterSystems IRIS & Python. Perfect Together.

3.2 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, use:

```
$ pip3 install --target /InterSystems/IRIS/mgr/python usaddress-scourgify
```

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"
+-----------------------------------------------------
```

3.3 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.

3.4 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.

4 Bridging 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.

The section includes information on the following topics:

- Use Python Builtin Functions
- Identifier Names

- Keyword or Named Arguments
- Passing Arguments By Reference
- Passing Values for True, False, and None
- Dictionaries
- Lists
- Globals
- Running an ObjectScript Command from Embedded Python
- Exception Handling
- Bytes and Strings
- Signal Handling

4.1 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.

4.2 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()
```

4.3 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.

4.4 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.

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` keyword in the signature of an ObjectScript method is just a convention used to indicate to the user that the method expects that an argument is to be passed by reference. In fact, `ByRef` has no actual function and is ignored by the compiler. Adding `ByRef` to the signature of a method written in Python has no effect.

4.5 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
```

4.6 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
```

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.

4.7 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.)

4.8 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
```

From the Python shell, type `help(iris)` for the complete list of methods that can be used on a global reference.

4.9 Running an ObjectScript Command from Embedded Python

There are times you may want to run an ObjectScript command from Embedded Python, for example, to access a system-provided "special variable", to call a routine written in ObjectScript, or to perform other tasks where there is no available method to call. In such cases, you can use the **iris.execute()** method from Python.

The following example writes the special variable `$zversion`, which contains the InterSystems IRIS version string:

```
>>> iris.execute("write $zversion,!")
IRIS for Windows (x86-64) 2022.3 (Build 602U) Mon Jan 23 2023 14:05:04 EST
```

Sometimes you may want to return a value from **iris.execute()** and assign it to a Python variable. This example assigns the value of the special variable `$horolog`, which contains the local date and time for the current process in InterSystems IRIS internal storage format, to the variable `t`:

```
>>> t = iris.execute("return $horolog")
>>> t
'66499,55283'
```

This is equivalent to `t = iris.cls('%SYSTEM.SYS').Horolog()`, which uses the **Horolog()** method of the class %SYSTEM.SYS.

You may encounter older ObjectScript code that uses routines instead of classes and methods and want to call a routine from Embedded Python. If you have a routine `^Math` that has a function **Sum()** that returns the sum of two numbers, you can add two numbers and assign the return value to the Python variable `sum`, as follows:

```
>>> sum = iris.execute("return $$Sum^Math(4,3)")
>>> sum
>>> 7
```

While the recommended way to access globals from Embedded Python is to use the **iris.gref()** method, you can also set and retrieve values from a global by using **iris.execute()**. The following example sets the global `^motd` to the value `"hello world"` and then retrieves the value from the global.

```
>>> iris.execute("set ^motd = \"hello world\"")
>>> iris.execute("return ^motd")
'hello world'
```

4.10 Exception Handling

The InterSystems IRIS exception handler can handle Python exceptions and pass them seamlessly to ObjectScript. Building on the earlier Python library example, if you try to call **canvas.drawImage()** using a non-existent file, and catch the exception in ObjectScript, you see the following:

```
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" -
```

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

4.11 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"
```

4.12 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.

5 Using 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.