組み込み Python の使用法

Version 2024.1
2024-06-03

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

組み込み Python の使用法
InterSystems IRIS Data Platform　Version 2024.1　2024-06-03

InterSystems Worldwide Response Center (WRC)
Tel:　+1-617-621-0700
Tel:　+44 (0) 844 854 2917
Email:　support@InterSystems.com

目次

1

概要と前提条件

組み込み Python により、InterSystems IRIS アプリケーションをプログラミングするネイティブ・オプションとして Python を使用できるようになります。組み込み Python を使用するのが初めてである場合は、まず "組み込み Python の概要" を参照し、その後組み込み Python のより詳細な説明として、このドキュメントをお読みください。

このドキュメントは組み込み Python について学習する人すべてに役立ちますが、理解するうえで ObjectScript についてのある程度の知識を持っていることが望ましいでしょう。InterSystems IRIS および ObjectScript に不慣れな Python 開発者の場合、"サーバ側プログラミングの入門ガイド" も参照してください。

1.1 推奨される Python バージョン

組み込み Python の使用時に推奨される Python のバージョンは、実行しているプラットフォームによって異なります。ほとんどの場合は、お使いのオペレーティング・システムの Python の既定バージョンです。オペレーティング・システムと対応する Python のサポート対象バージョンの完全なリストは、"その他のサポート対象機能" を参照してください。

注釈　一部のオペレーティング・システムでは、フレキシブル Python ランタイム機能を使用して、推奨される Python バージョンをオーバーライドできます。

Microsoft Windows では、InterSystems IRIS インストール・キットによって、組み込み Python 専用の正しいバージョンの Python (現在は 3.9.5) がインストールされます。開発マシンを使用していて、Python を一般用途に使用したい場合は、この同じバージョンを https://www.python.org/downloads/ からダウンロードしてインストールすることをお勧めします。

多くの UNIX ベースのオペレーティング・システムでは、既に Python がインストールされています。インストールする必要がある場合は、パッケージ・マネージャによりご使用のオペレーティング・システムに推奨されるバージョンを使用してください。以下に例を示します。

- Ubuntu : `apt install python3`
- Red Hat Enterprise Linux または Oracle Linux : `yum install python3`
- SUSE : `zypper install python3`
- macOS : Homebrew を使用して Python 3.11 をインストールします。

  ```
  brew install python@3.11
  ```

  また、OpenSSL の最新バージョンを使用していることも確認してください。

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

- AIX：AIX Toolbox for Open Source Software から dnf (dandified yum) を使用して Python 3.9.18 以降をインストールします。

“Python をロードできませんでした” というエラーが表示された場合は、システムに Python がインストールされていないか、Python の予期しないバージョンが検出されたかのどちらかです。″その他のサポート対象機能″ を調べて、必要なバージョンの Python がインストールされていることを確認し、必要に応じて、上記のいずれの方法で Python をインストールまたは再インストールしてください。または、フレキシブル Python ランタイム機能を使用して、推奨される Python バージョンをオーバーライドします。(すべてのプラットフォームで利用できるわけではありません。)

フレキシブル Python ランタイム機能をサポートしていないプラットフォームを使用していて、コンピュータに複数のバージョンの Python がインストールされている場合、コマンド行から組み込み Python の実行を試行すると、irispython は最初に検出された python3 実行可能ファイル (PATH 環境変数よって決まる) を実行します。必要なバージョンの実行可能ファイルが最初に検出されるように、パスのフォルダが適切に設定されていることを確認してください。irispython コマンドの使用の詳細は、″コマンド行からの Python シェルの開始″ を参照してください。

1.2 必要なサービス

組み込み Python の実行時に `IRIS_ACCESSDENIED` エラーが発生しないようにするには、`%Service_Callin` を有効にします。管理ポータルで、**[システム管理]→[セキュリティ]→[サービス]** に移動し、[%Service_CallIn] を選択し、**[サービス有効]** ボックスにチェックを付けます。

1.3 フレキシブル Python ランタイム機能

一部のオペレーティング・システムでは、フレキシブル Python ランタイム機能を使用して、組み込み Python に推奨される Python のバージョンをオーバーライドできます。これは、コードを記述する場合や、推奨されるバージョン以外の Python に依存するパッケージを使用している場合に便利です。オペレーティング・システムの既定のバージョン以降の Python バージョンを使用する必要があります。例えば、Red Hat Enterprise Linux 9 には Python 3.9 が付属しているため、このオペレーティング・システムではバージョン 3.9 以降を使用する必要があります。

構成設定 `PythonRuntimeLibrary` は、組み込み Python を実行する際に使用する Python ランタイム・ライブラリの場所を指定します。このバージョンのライブラリは、組み込み Python の起動時に使用される既定のバージョンのライブラリをオーバーライドします。

Python ランタイム・ライブラリを指定するには、以下の手順に従います。

1. 管理ポータルで、**[システム管理]→[構成]→[追加の設定]→[メモリ詳細]** に移動します。
2. [PythonRuntimeLibrary] の行で、**[編集]** をクリックします。
3. 使用する Python ランタイム・ライブラリの場所を入力します。

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

   この場所は、オペレーティング・システム、Python バージョン、およびその他の要素によって異なります。ここに示す例は、x86 アーキテクチャでの Ubuntu 22.04 上の Python 3.11 の場合です。
4. **[保存]** をクリックします。

詳細は、″PythonRuntimeLibrary″ を参照してください。

注釈　フレキシブル Python ランタイム機能は、すべてのオペレーティング・システムでサポートされるわけではありません。この機能をサポートするプラットフォームの完全なリストは、"その他のサポート対象機能" を参照してください。

Python の新しいバージョンをインストールしたが、Python ランタイム・ライブラリが見つからない場合、別個にインストールしなければならない可能性があります。例えば、Ubuntu 22.04 に Python 3.11 ランタイム・ライブラリをインストールする場合は、`apt install libpython3.11` を実行します。

この機能の構成方法の詳細は、"フレキシブル Python ランタイム機能の使用" を参照してください。

2

Python パッケージのインストールおよびインポート

組み込み Python では、何千もの便利なライブラリに簡単にアクセスできます。一般に "パッケージ" と呼ばれますが、これらは使用する前に InterSystems IRIS のファイル・システムにインストールする必要があります。その後、コードで使用するために、インポートしてメモリにロードする必要があります。これを行うには、組み込み Python の使用方法によって、さまざまな方法があります。

2.1 Python パッケージのインストール

組み込み Python で使用する前に、コマンド行から Python パッケージをインストールします。使用するコマンドは、InterSystems IRIS を UNIX ベースのシステム (AIX を除く)、AIX、Windows、またはコンテナのいずれで使用するかによって異なります。

2.1.1 UNIX ベースのシステム (AIX を除く) への Python パッケージのインストール

UNIX ベースのシステムでは、`python3 -m pip install --target <installdir>/mgr/python <package>` コマンドを使用します。

注釈　まだインストールしていない場合は、まずシステムのパッケージ・マネージャでパッケージ `python3-pip` をインストールしてください。

例えば、ReportLab Toolkit は、PDF およびグラフィックを生成するためのオープン・ソース・ライブラリです。UNIX ベースのシステムでは、以下のようなコマンドを使用してインストールします。

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

重要　パッケージを正しいターゲット・ディレクトリにインストールしなければ、組み込み Python はパッケージをインポートできません。例えば、`--target` 属性を指定せずに (および sudo を使用せずに) パッケージをインストールする場合、Python はホーム・ディレクトリ内のローカル・パッケージ・リポジトリにパッケージをインストールします。他のユーザがパッケージをインポートしようとすると、失敗します。

インターシステムズでは `--target <installdir>/mgr/python` オプションを使用することを推奨していますが、sudo を使用して `--target` 属性を省略してパッケージをインストールすると、パッケージはグローバル・パッケージ・リポジトリにインストールされます。これらのパッケージは、どのユーザでもインポートできます。

2.1.2 AIX への Python パッケージのインストール

AIX で、AIX Toolbox for Open Source Software (利用できる場合) からパッケージをインストールします。

パッケージをインストールする前に、コマンド `sudo dnf list | grep <package>` を使用して、パッケージが AIX Toolbox にあることを確認します。次に、コマンド `sudo dnf install <package>` を使用してパッケージをインストールします。

注釈　まだインストールしていない場合は、まず AIX Toolbox からパッケージ `python3.9-pip` をインストールしてください。

例えば、パッケージ `psutil` が AIX Toolbox であることを確認します。

```
$ 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
```

次に、パッケージをインストールします。

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

パッケージが AIX Toolbox にない場合のみ、以下のコマンドを使用してインストールします。

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

2.1.3 Windows への Python パッケージのインストール

Windows では、<installdir>/bin ディレクトリから組み込みの irispip コマンドを使用します：`irispip install --target <installdir>\mgr\python <package>`。

例えば、以下のようにして Windows マシンに ReportLab パッケージをインストールできます。

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

2.1.4 コンテナへの Python パッケージのインストール

永続的な %SYS 機能を使用せずに、コンテナで InterSystems IRIS を実行している場合は、コマンド `python3 -m pip install --target /usr/irissys/mgr/python <package>` を使用します。

例えば、以下のようにしてコンテナに ReportLab パッケージをインストールできます。

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

永続的な %SYS 機能を使用して、コンテナで InterSystems IRIS を実行している場合は、コマンド `python3 -m pip install --target <durable>/mgr/python <package>` を使用します。<durable> は、コンテナの実行時に環境変数 `ISC_DATA_DIRECTORY` で定義されたパスです。

例えば、`ISC_DATA_DIRECTORY=/durable/iris` の場合、以下のようにしてコンテナに ReportLab パッケージをインストールできます。

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

注釈　注意：Dockerfile を使用して InterSystems IRIS のカスタム Docker イメージを作成する場合は、/usr/irissys/mgr/python に Python パッケージをインストールします。/usr/irissys/mgr/python と <durable>/mgr/python の両方が既定で sys.path に含まれているため、永続的な %SYS 機能を使用しているかどうかに関係なくパッケージを見つけることができます。

コンテナの作成と実行の詳細は、"コンテナ内でのインターシステムズ製品の実行" を参照してください。

2.2 Python パッケージのインポート

パッケージをインストールした後、InterSystems IRIS から使用するにはこれをインポートする必要があります。これにより、パッケージがメモリにロードされ、使用できるようになります。

2.2.1 ObjectScript からの Python パッケージのインポート

ObjectScript から Python パッケージをインポートするには、%SYS.Python クラスの Import() メソッドを使用します。以下に例を示します。

ObjectScript

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

上記の最初の行は、組み込みの Python math モジュールを ObjectScript にインポートします。2 番目の行は、ReportLab の pdfgen サブパッケージから canvas.py ファイルのみをインポートします。

2.2.2 Python で記述されたメソッドからの Python パッケージのインポート

Python で記述された InterSystems IRIS メソッドで、他の Python コードの場合と同様にパッケージをインポートできます。以下に例を示します。

```
ClassMethod
Example() [ Language
= python ]
{



import math



import iris



import reportlab.pdfgen.canvas as canvaslib









# Your Python code here
}
```

2.2.3 XData ブロックを使用した Python パッケージのインポート

以下の例のように、クラス内の XData ブロックを使用してパッケージのリストをインポートすることもできます。

```
XData
%import [ MimeType
= application/python ]
{



import math



import iris



import reportlab.pdfgen.canvas as canvaslib
}
```

重要　　XData ブロックの名前は `%import` にする必要があります。MIME タイプは、`application/python` または `text/x-python` にできます。行のインデントの考慮を含め、正しい Python 構文を使用してください。

その後、これらのパッケージは、Python で記述されたクラス内の任意のメソッドで使用できます。再度インポートする必要はありません。

```
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()
}
```

XData ブロックの背景情報は、"XData ブロックの定義と使用" を参照してください。

3

組み込み Python の実行

このページでは、組み込み Python を実行するいくつかの方法を説明します。

3.1 Python シェルから実行

インターシステムズのターミナル・セッションまたはコマンド行から Python シェルを開始できます。

3.1.1 ターミナルからの Python シェルの開始

インターシステムズのターミナル・セッションから Python シェルを開始するには、**%SYS.Python** クラスの Shell() メソッドを呼び出します。これにより、Python インタプリタがインタラクティブ・モードで起動します。ターミナル・セッションからユーザとネームスペースが Python シェルに渡されます。

コマンド **quit()** を入力して、Python シェルを終了します。

以下の例は、ターミナル・セッションの **USER** ネームスペースから Python シェルを起動します。これは、フィボナッチ数列の最初の数個の数字を出力し、その後 InterSystems IRIS の %SYSTEM.OBJ.ShowClasses() メソッドを使用して現在のネームスペース内のクラスのリストを出力します。

```
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>
```

メソッド %SYSTEM.OBJ.ShowClasses() は、InterSystems IRIS の **%Status** の値を返します。この場合、1 はエラーが検出されなかったことを意味します。

3.1.2 コマンド行からの Python シェルの開始

irispython コマンドを使用して、コマンド行から Python シェルを開始します。これは、ターミナルからのシェルの開始とほぼ同様に機能しますが、InterSystems IRIS のユーザ名、パスワード、およびネームスペースを渡す必要があります。

以下の例は、Windows コマンド行から Python シェルを起動します。

```
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.
>>>
```

UNIX ベースのシステムでは、set ではなく、export を使用します。

```
/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.
>>>
```

注釈　`IRIS_ACCESSDENIED` というメッセージが表示された場合は、`%Service_Callin` を有効にしてください。管理ポータルで、**[システム管理]→[セキュリティ]→[サービス]** に移動し、[%Service_CallIn] を選択し、**[サービス有効]** ボックスにチェックを付けます。

3.2 Python スクリプト・ファイル (.py) 内で実行

irispython コマンドを発行して、Python スクリプトを実行することもできます。この場合は、InterSystems IRIS へのアクセスを提供する手順 (`import iris`) を含める必要があります。

Windows システムで、以下のコードを持つ C:¥python¥test.py ファイルを見てみましょう。

```
# 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)
```

test.py は、コマンド行から以下のように実行できます。

```
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
```

UNIX ベースのシステムでは、set ではなく、export を使用します。

```
/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
```

注釈　`import iris` を実行しようとして、`IRIS_ACCESSDENIED` というメッセージが表示された場合は、`%Service_Callin` を有効にしてください。管理ポータルで、[**システム管理**]→[**セキュリティ**]→[**サービス**] に移動し、[%Service_CallIn] を選択し、[**サービス有効**] ボックスにチェックを付けます。

3.3 InterSystems IRIS クラスのメソッド内で実行

`Language` キーワードを使用して、InterSystems IRIS クラス内で Python メソッドを記述できます。その後、ObjectScript で記述されたメソッドを呼び出す場合と同じように、メソッドを呼び出せます。

例えば、Python で記述されたクラス・メソッドを持つ以下のクラスを取り上げます。

```
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
}

}
```

このメソッドは、ObjectScript から以下のように呼び出せます。

```
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
```

または Python から以下のように呼び出せます。

```
>>> 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 SQL 関数およびストアド・プロシージャ内で実行

組み込み Python を使用して SQL 関数またはストアド・プロシージャを記述することもできます。これには、以下のように CREATE 文で引数 `LANGUAGE PYTHON` を指定します。

```
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")
}
```

コードは Python の `datetime` および `dateutil` モジュールから関数を使用します。

注釈　プラットフォームによっては、`datetime` および `dateutil` モジュールが既定でインストールされていないことがあります。この例を実行して `ModuleNotFoundError` が発生した場合は、"Python パッケージのインストール" の説明に従って欠落しているモジュールをインストールします。

以下の SELECT 文は、SQL 関数を呼び出し、現在の日付/時刻を東部標準時から協定世界時 (UTC) に変換します。

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

この関数は以下のような結果を返します。

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

4

ObjectScript からの組み込み Python コードの呼び出し

このセクションでは、ObjectScript から組み込み Python コードを呼び出すいくつかの方法を説明します。

- Python ライブラリの使用
- Python で記述された InterSystems IRIS クラス内でのメソッドの呼び出し
- Python で記述された SQL 関数またはストアド・プロシージャの実行
- 任意の Python コマンドの実行

ObjectScript コードを呼び出すのとほとんど同じ方法で Python コードを呼び出せる場合もありますが、**%SYS.Python** クラスを使用して 2 つの言語間のギャップを埋める必要がある場合もあります。詳細は、"ObjectScript と組み込み Python 間のギャップを埋める" を参照してください。

4.1 Python ライブラリの使用

組み込み Python では、何千もの便利なライブラリに簡単にアクセスできます。一般に "パッケージ" と呼ばれますが、これらは使用する前に Python Package Index (PyPI) から **<installdir>/mgr/python** ディレクトリにインストールする必要があります。

例えば、ReportLab Toolkit は、PDF およびグラフィックを生成するためのオープン・ソース・ライブラリです。以下のコマンドでは、パッケージ・インストーラ **irispip** を使用して Windows システムに ReportLab をインストールします。

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

UNIX ベースのシステムでは (AIX を除く)、以下を使用します。

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

InterSystems IRIS をコンテナで実行している場合は、"コンテナへの Python パッケージのインストール" を参照してください。

InterSystems IRIS を AIX で実行している場合は、"AIX への Python パッケージのインストール" を参照してください。

パッケージをインストールしたら、**%SYS.Python** クラスの Import() メソッドにより、それを ObjectScript コードで使用できます。

ファイルの場所を指定すると、以下の ObjectScript メソッド CreateSamplePDF() は、サンプルの PDF ファイルを作成し、その場所に保存します。

```
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()
}

}
```

このメソッドの最初の行は、ReportLab の `pdfgen` サブパッケージから `canvas.py` ファイルをインポートします。コードの 2 番目の行は Canvas オブジェクトをインスタンス化し、その後続行して、InterSystems IRIS オブジェクトのメソッドを呼び出すのとほぼ同じ方法でそのメソッドを呼び出します。

その後、以下に示すように通常の方法でメソッドを呼び出せます。

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

以下の PDF が生成され、指定された場所に保存されます。

4.2 Python で記述された InterSystems IRIS クラス内でのメソッドの呼び出し

組み込み Python を使用して InterSystems IRIS クラスでメソッドを記述し、その後 ObjectScript で記述されたメソッドを呼び出すのと同じ方法で、ObjectScript からそのメソッドを呼び出すことができます。

次の例は `usaddress-scourgify` ライブラリを使用しています。このライブラリは Windows のコマンド行から以下のようにインストールできます。

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

UNIX ベースのシステムでは (AIX を除く)、以下を使用します。

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

InterSystems IRIS をコンテナで実行している場合は、"コンテナへの Python パッケージのインストール" を参照してください。

InterSystems IRIS を AIX で実行している場合は、"AIX への Python パッケージのインストール" を参照してください。

以下のデモ・クラスには、米国の住所の各部分のプロパティと、Python で記述されたメソッドが含まれます。これは、`usaddress-scourgify` を使用して住所をアメリカ合衆国郵便公社の標準に従って正規化します。

```
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']
}

}
```

住所の文字列を入力として指定すると、クラスの Normalize() インスタンス・メソッドは、住所を正規化し、各部分を Demo.Address オブジェクトのさまざまなプロパティに格納します。

このメソッドは以下のように呼び出せます。

```
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 Python で記述された SQL 関数またはストアド・プロシージャの実行

組み込み Python を使用して SQL 関数またはストアド・プロシージャを作成する際、InterSystems IRIS は、他のメソッドを呼び出すのと同じように ObjectScript から呼び出せるメソッドを持つクラスを投影します。

例えば、このドキュメントで前述した例に示す SQL 関数は、tzconvert() メソッドを持つクラス User.functzconvert を生成します。これを ObjectScript から呼び出すには、以下のようにします。

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

ここでは、$zdatetime($h,3) を使用して現在の日付と時刻が $HOROLOG 形式から ODBC 日付形式に変換されます。

4.4 任意の Python コマンドの実行

組み込み Python コードを開発またはテストしている際、任意の Python コマンドを ObjectScript から実行すると都合がよい場合があります。これは、%SYS.Python クラスの Run() メソッドによって行えます。

例えば、このドキュメントの前半で使用した `usaddress_scourgify` パッケージからの normalize_address_record() 関数をテストしたいのに、その動作を思い出せない場合があるかもしれません。%SYS.Python.Run() メソッドを使用して、以下のようにターミナルから関数のヘルプを出力できます。

```
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.
.
.
.
```

%SYS.Python.Run() メソッドは、成功した場合に 0 を返し、失敗した場合に -1 を返します。

5

ObjectScript と組み込み Python 間のギャップを埋める

ObjectScript と Python の言語には違いがあるため、その言語間のギャップを埋めるのに知っておく必要のある情報があります。

ObjectScript 側では、**%SYS.Python** クラスにより、ObjectScript から Python を使用できるようになります。詳細は、InterSystems IRIS のクラス・リファレンスを参照してください。

Python 側では、`iris` モジュールにより、Python から ObjectScript を使用できるようになります。Python から、モジュールと関数のリストを表示するには「`help(iris)`」と入力します。詳細は、"InterSystems IRIS Python モジュール・リファレンス" を参照してください。

5.1 Python の組み込み関数の使用

`builtins` パッケージは Python インタプリタの起動時に自動的にロードされます。このパッケージには、ベース・オブジェクト・クラスや、すべての組み込みのデータ型クラス、例外クラス、関数、および定数など、この言語に組み込まれたすべての識別子が含まれます。

これらの識別子すべてにアクセスできるようにするには、以下のようにこのパッケージを ObjectScript にインポートします。

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

Python の print() 関数は、実際には `builtins` モジュールのメソッドであるため、この関数を以下のように ObjectScript から使用できるようになります。

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

次に zwrite コマンドを使用して `builtins` オブジェクトを調べることができます。これは Python オブジェクトであるため、`builtins` パッケージの str() メソッドを使用して、そのオブジェクトの文字列表現を取得します。以下に例を示します。

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

同じトークンによって、builtins.list() メソッドを使用して Python リストを作成できます。以下の例は、空のリストを作成します。

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

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

builtins.type() メソッドを使用して、変数 `list` がどの Python の型であるかを調べることができます。

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

興味深いことに、list() メソッドは、実際にはリストを表す Python のクラス・オブジェクトのインスタンスを返します。以下のようにリスト・オブジェクトで dir() メソッドを使用することで、list クラスが持つメソッドを確認できます。

```
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>
```

同様に、help() メソッドを使用して、リスト・オブジェクトについてのヘルプを取得できます。

```
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].
.
.
.
```

注釈　ObjectScript に `builtins` モジュールをインポートする代わりに、%SYS.Python クラスの Builtins() メソッドを呼び出すことができます。

5.2 識別子名

識別子の命名規則は、ObjectScript と Python で異なります。例えば、アンダースコア (_) は Python のメソッド名で許可されており、実のところいわゆる “ダンダー (dunder)” メソッドおよび属性 (“dunder” は “double underscore” の省略形)

用に広く使用されています (__getitem__ や __class__ など)。ObjectScript からそのような識別子を使用するには、以下のようにそれらを二重引用符で囲みます。

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

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

逆に、InterSystems IRIS メソッドは多くの場合パーセント記号 (%) で始まります。例えば、%New() や %Save() のようになります。Python からそのような識別子を使用するには、パーセント記号をアンダースコアで置き換えます。永続クラス User.Person がある場合、Python コードの以下の行は、新しい Person オブジェクトを作成します。

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

5.3 キーワードまたは名前付き引数

Python では、メソッドを定義する際にキーワード引数 (または "名前付き引数" とも呼ばれる) を使用するのが一般的な方法です。これにより、必要でない場合に引数を削除したり、場所ではなく名前に応じて引数を指定したりすることが容易になります。例として、以下の簡単な Python メソッドを見てみましょう。

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

InterSystems IRIS にはキーワード引数の概念がないため、ダイナミック・オブジェクトを作成して、キーワード/値のペアを保持する必要があります。例えば、以下のようになります。

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

メソッド mymethod() が mymodule.py というモジュール内にある場合、それを ObjectScript にインポートして、以下のように呼び出すことができます。

```
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
```

baz はメソッドに渡されていないため、既定で "three" の値が割り当てられます。

5.4 引数の参照渡し

ObjectScript で記述されたメソッドの引数は、値または参照によって渡すことができます。以下のメソッドでは、シグニチャの 2 つ目と 3 つ目の引数の前にある ByRef キーワードは、それらを参照渡ししようとしていることを示しています。

```
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"
}
```

ObjectScript からメソッドを呼び出す際は、以下のように引数の前にピリオドを配置して、それを参照渡しします。

```
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
```

出力から、変数 arg1 の値が、SandwichSwitch() を呼び出した後も同じであるのに対し、変数 arg2 および arg3 の値は変化したことがわかります。

Python は参照による呼び出しをネイティブでサポートしないため、iris.ref() メソッドを使用して、参照渡しする各引数についてメソッドに渡す参照を作成する必要があります。

```
>>> 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'
```

value プロパティを使用して arg2 と arg3 の値にアクセスし、それらがメソッドへの呼び出しの後に変化したことを確認できます。

ObjectScript には、引数が参照によって渡され、この引数が値を受け取らずに出力として使用されることが想定されることを示すキーワード Output もあります。Python から iris.ref() メソッドを使用して、ByRef 引数の場合と同じ方法で引数を渡します。

注釈　引数の参照渡しは ObjectScript メソッドの機能ですが、Python で記述されたメソッドに引数を参照渡しする同等の方法はありません。ObjectScript メソッドのシグニチャの ByRef および Output キーワードは、引数が参照渡しされることをメソッドが期待していることをユーザに示すための規則です。実際、ByRef と Output には実際の機能はなく、コンパイラからは無視されます。ByRef または Output を Python で記述されたメソッドのシグニチャに追加すると、コンパイラ・エラーが発生します。

5.5 True、False、None の値を渡す方法

%SYS.Python クラスには True()、False()、および None() のメソッドがあり、これらはそれぞれ Python の True、False、および None の識別子を表します。

以下に例を示します。

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

これらのメソッドは、Python メソッドに True、False、および None を渡す必要がある場合に役立ちます。以下の例は、"キーワードまたは名前付き引数" で示したメソッドを使用しています。

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

キーワード引数を期待している Python メソッドに名前のない引数を渡すと、Python はそれらを渡された順序で処理します。

Python メソッドによって ObjectScript に返される値を調べる際に True()、False()、および None() のメソッドを使用する必要がないことに注意してください。

Python モジュール mymodule に、isgreaterthan() メソッドもあるとします。これは、以下のように定義されます。

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

Python で実行すると、このメソッドは引数 a が b よりも大きい場合に True を返し、そうでない場合に False を返すことがわかります。

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

しかし、ObjectScript から呼び出した場合、返される値は 1 であり、Python 識別子の True ではありません。

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

5.6 ディクショナリ

Python では、ディクショナリは一般的にキーと値のペアの形式でデータを格納するために使用されます。例えば以下のようになります。

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

メソッド iris.arrayref() を使用してディクショナリ mycar の内容を ObjectScript 配列に配置して、その配列への参照を返すことができます。

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

その後、その配列を ObjectScript メソッドに渡すことができます。例えば、配列の内容を書き込むメソッド WriteContents() を持つ、User.ArrayTest という InterSystems IRIS クラスがある場合、以下のように呼び出すことができます。

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

詳細は、"iris.arrayref()" を参照してください。

ObjectScript 側では、Python の builtins モジュールの dict() メソッドを使用して、Python ディクショナリを操作できます。

```
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
```

上記の例では、ディクショナリ・メソッド setdefault() を使用してキーの値を設定し、_getitem_() によってキーの値を取得しています。

5.7 リスト

Python では、リストは値のコレクションを格納しますが、キーを使用しません。リスト内の項目には、インデックスによってアクセスします。

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

ObjectScript では、Python の builtins モジュールの list() メソッドを使用して、Python リストを操作できます。

```
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
```

上記の例では、リスト・メソッド append() を使用してリストに項目を追加し、_getitem_() によって指定したインデックスの値を取得しています(Python リストはゼロベースです)。

ObjectScript リストを Python リストに変換する場合は、%SYS.Python の ToList() および ToListTyped() メソッドを使用できます。ObjectScript リストを指定すると、ToList() は同じデータを含む Python リストを返します。データを含む ObjectScript リストと、整数の ODBC データ型コードを含む 2 番目の ObjectScript リストを指定すると、ToListTyped() は各項目が 2 番目のリストで指定したデータ型を持つ、最初のリストと同じデータを含む Python リストを返します。

注釈　ODBC データ型のテーブルは、"データ型の整数コード" を参照してください。

一部の ODBC データ型は、同じ Python データ型に変換できます。

一部のデータ型は、Python パッケージ numpy をインストールする必要があります。

以下の例では、クラス User.Lists の Python メソッド Loop() がリスト内の項目を反復処理し、その値とデータ型を出力します。

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

その後、ToList() と ToListTyped() を以下のように使用できます。

```
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 グローバル

ほとんどの場合、InterSystems IRIS に保存されているデータには、SQL を使用するか、永続クラスとそのプロパティおよびメソッドを使用してアクセスできます。しかし、グローバルと呼ばれる、基盤となるネイティブの永続データ構造に直接アクセスしたいこともあります。従来のデータにアクセスする場合、または SQL テーブルや永続クラスには適していないスキーマレスのデータを保存している場合は特にそうです。

単純化しすぎではありますが、グローバルは、キーと値のペアのディクショナリと考えることができます。(正確な説明は、"グローバルの概要" を参照してください。)

Python で記述された 2 つのクラス・メソッドを持つ、以下のクラスを考えてみます。

```
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])))
}
}
```

メソッド SetSquares() はキーの範囲をループして、グローバル ^square の各ノードで各キーの二乗を格納します。メソッド PrintSquares() はグローバルを走査し、キーに格納されている各キーと値を出力します。

Python シェルを起動して、クラスをインスタンス化し、コードを実行してどのように動作するかを確認してみましょう。

```
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
```

今度は、組み込みの iris モジュールのいくつかのメソッドでグローバルにアクセスする方法を見てみましょう。

メソッド SetSquares() では、文 square = iris.gref("^square") はグローバル ^square への参照 (gref とも呼ばれます) を返します。

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

文 square.set([key], value) は、キー key を持つ ^square のノードを値 value に設定します。例えば、^square のノード 12 を値 144 に設定できます。

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

グローバルのノードを、次の短い構文で設定することもできます。

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

メソッド PrintSquares() では、文 key = square.order([key]) は ObjectScript の $ORDER 関数のように、キーを入力として取り、グローバルの次のキーを返します。グローバルを走査するには、通常、キーがもう残っていないことを示す None が返されるまで order() を使用し続けます。キーが連続している必要はないため、キーの間にギャップがある場合でも、order() は次のキーを返します。

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

次に、square.get([key]) はキーを入力として取り、グローバルのそのキーの値を返します。

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

再び、以下の短い構文を使用できます。

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

グローバルのノードがキーを持つ必要はありません。以下の文は、^square のルート・ノードに文字列を格納します。

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

これらの Python コマンドが実際にグローバルに値を格納したことを示すために、Python シェルを終了し、ObjectScript で zwrite コマンドを使用して ^square の内容を出力します。

```
>>> 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
```

Python からグローバルにアクセスして操作する方法の詳細は、"グローバル参照 API" を参照してください。

5.9 ネームスペースの変更

InterSystems IRIS にはネームスペースという概念があり、各ネームスペースには、コードとデータを格納するための独自のデータベースがあります。これにより、あるネームスペースのコードおよびデータを、別のネームスペースのコードおよびデータと容易に分離できます。例えば、あるネームスペースに特定の名前のグローバルがある場合、別のネームスペースは同じ名前のグローバルを他のグローバルと競合する恐れなしに使用できます。

NSONE と NSTWO という 2 つのネームスペースがある場合、以下に示すように、ターミナルで ObjectScript を使用して、NSONE に ^myFavorite というグローバルを作成できます。その後、$namespace 特殊変数を設定して NSTWO に切り替え、このネームスペースに ^myFavorite という別個のグローバルを作成できます。(この例を複製するには、InterSystems IRIS インスタンス上にこれら 2 つのネームスペースを構成するか、既に存在する 2 つのネームスペースを使用できます。)

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

NSONE>set $namespace = "NSTWO"

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

ここで、^myFavorite("fruit") は NSONE に "apple" の値、NSTWO に "orange" の値を持ちます。

組み込み Python を呼び出すと、これは現在のネームスペースを継承します。これをテストするには、Python から現在のネームスペースの名前を表示する %SYSTEM.SYS クラスの NameSpace() メソッドを呼び出し、^myFavorite("fruit") = "orange" であることを確認します。

```
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
```

$namespace を使用して ObjectScript でネームスペースを変更する方法を確認しました。組み込み Python では、iris.system.Process クラスの SetNamespace() メソッドを使用します。例えば、ネームスペース NSONE に切り替えて、^myFavorite("fruit") = "apple" であることを確認できます。

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

最後に、Python シェルを終了すると、ネームスペース NSONE にとどまります。

```
>>> quit()
NSONE>
```

5.10 組み込み Python からの ObjectScript ルーチンの実行

クラスやメソッドの代わりにルーチンを使用する古い ObjectScript コードが使用されていて、組み込み Python からルーチンを呼び出したい場合があります。このような場合、Python から iris.routine() メソッドを使用できます。

以下の例では、%SYS ネームスペースでの実行時にルーチン ^SECURITY を呼び出します。

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

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

2 つの数値を加算する関数 Sum() を持つルーチン ^Math がある場合、以下の例では 4 と 3 を加算します。

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

5.11 例外処理

InterSystems IRIS 例外ハンドラは、Python の例外を処理し、それらをシームレスに ObjectScript に渡します。以下の例は、前述の Python ライブラリの例を基にして、存在しないファイルを使用して canvas.drawImage() を呼び出そうとするとどうなるかを示しています。ここで、ObjectScript は特殊変数 $zerror で例外を検出しています。

```
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" -
```

この場合、<class 'OSError'>: Cannot open resource "W:\Sample\isc.png" は Python から戻された例外です。

$Zerror の詳細は、"$ZERROR (ObjectScript)" を参照してください。

ObjectScript ステータス・エラーを Python 例外として生成する方法は、"check_status(status)" を参照してください。

5.12 Bytes 型と String 型

Python では、“bytes” データ型 (単に 8 ビット・バイトのシーケンス) のオブジェクトと、string (文字列を表す UTF-8 バイトのシーケンス) の間に明確な区別があります。Python では、bytes オブジェクトは決して変換されませんが、string はホスト・オペレーティング・システムによって使用される文字セットに応じて変換される場合があります (Latin-1 など)。

InterSystems IRIS では、bytes と string を区別しません。InterSystems IRIS では Unicode 文字列 (UCS-2/UTF-16) をサポートしていますが、256 より小さい値を含む文字列は、string と bytes のどちらにもなり得ます。このため、Python との間で string および bytes をやり取りする際は、以下のルールが適用されます。

- InterSystems IRIS の string は文字列と見なされ、ObjectScript から Python に渡される際に UTF-8 に変換されます。
- Python の string は、ObjectScript に戻されるときに、UTF-8 から InterSystems IRIS 文字列に変換され、その結果ワイド文字になります。
- Python の bytes オブジェクトは、8 ビット文字列として ObjectScript に返されます。bytes オブジェクトの長さが最大文字列長を超えると、Python の bytes オブジェクトが返されます。
- ObjectScript から Python に bytes オブジェクトを渡すには、##class(%SYS.Python).Bytes() メソッドを使用します。このメソッドは、基礎となる InterSystems IRIS 文字列を UTF-8 に変換しません。

以下の例は、InterSystems IRIS 文字列を bytes 型の Python オブジェクトに変換します。

```
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>
```

InterSystems IRIS の最大文字列長である 3.8MB より大きい Python bytes オブジェクトを構築するには、bytearray オブジェクトを使用して、extend() メソッドにより bytes のより小さなチャンクを追加できます。最後に、bytearray オブジェクトを builtins の bytes() メソッドに渡して、bytes 表現を取得します。

```
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 標準出力と標準エラーのマッピング

組み込み Python を使用する際、標準出力は InterSystems IRIS コンソールにマッピングされます。つまり、print() 文の出力はすべてターミナルに送信されます。標準エラーは、ディレクトリ <install-dir>/mgr にある InterSystems IRIS messages.log ファイルにマッピングされます。

例として、この Python メソッドについて考えてみましょう。

```
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")
```

ターミナルでこのメソッドをテストする場合、以下のような画面が表示されます。

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

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

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

0 による除算を試行すると、エラー・メッセージがターミナルに転送されますが、文字列による除算を試行すると、メッセージは messages.log に送信されます。

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

ファイルが乱雑になることを防ぐため、重要なメッセージのみが messages.log に送信されるようにする必要があります。

6

相互運用プロダクション内での組み込み Python の使用法

InterSystems IRIS 内の相互運用プロダクション向けにカスタムのビジネス・ホスト・クラスやアダプタ・クラスを記述している場合、コールバック・メソッドはすべて ObjectScript で記述する必要があります。コールバック・メソッドは既定では何もしない継承メソッドですが、ユーザにより実装されるよう設計されています。ただし、コールバック・メソッドの ObjectScript コードは、Python ライブラリを活用したり、Python で実装されたその他のメソッドを呼び出すことができます。

以下の例は、受信メッセージから文字列値を取得し、Amazon Web Services (AWS) `boto3` Python ライブラリを使用して、その文字列を Amazon Simple Notification Service (SNS) を介してテキスト・メッセージとして電話に送信するビジネス・オペレーションを示しています。この AWS ライブラリはここでの説明の範囲外ですが、この例では、OnInit() および OnMessage() コールバック・メソッドは ObjectScript で記述されているのに対し、PyInit() および SendSMS() のメソッドは Python で記述されていることがわかります。

```
/// 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
}

}
```

注釈　上記の OnMessage() メソッドのコードには、このドキュメントを印刷する際により適した書式設定とするために余分の改行が含まれています。

このルールの 1 つの例外は、アダプタからの入力を使用していない場合に、Python でコールバック・メソッドを実装できることです。

以下のビジネス・サービスの例は poller として知られています。この場合、ビジネス・サービスは間を置いて実行するよう設定でき、処理のためにビジネス・プロセスに送信される要求 (この場合はランダムな文字列値を含む) を生成します。この例では、OnProcessInput() コールバック・メソッドを Python に実装できます。これはこのメソッドのシグニチャで pInput 引数を利用していないためです。

```
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"
}

}
```

相互運用プロダクションのプログラミングに関する詳細は、"ビジネス・サービス、ビジネス・プロセス、およびビジネス・オペレーションのプログラミング" を参照してください。

7

フレキシブル Python ランタイム機能の使用

7.1 フレキシブル Python ランタイム機能の概要

組み込み Python を実行する際、InterSystems IRIS はユーザがオペレーティング・システムの既定バージョンの Python を使用していると想定します。しかし、Python の新しいバージョンにアップグレードしたり、Anaconda のような代替ディストリビューションに切り替えたい場合もあります。フレキシブル Python ランタイム機能により、InterSystems IRIS の組み込み Python でこれらのバージョンの Python を使用することができます。

注釈　フレキシブル Python ランタイム機能は、すべてのオペレーティング・システムでサポートされるわけではありません。この機能をサポートするプラットフォームの完全なリストは、"その他のサポート対象機能" を参照してください。

フレキシブル Python ランタイム機能を使用するための準備は、次の 3 つの基本ステップで構成されます。

1. 使用するバージョンの Python をインストールします。
2. `PythonRuntimeLibrary` 構成設定を設定して、組み込み Python を実行する際に使用する Python ランタイム・ライブラリの場所を指定します。

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

   この場所は、オペレーティング・システム、Python バージョン、およびその他の要素によって異なります。ここに示す例は、x86 アーキテクチャでの Ubuntu 22.04 上の Python 3.11 の場合です。

   詳細は、"PythonRuntimeLibrary" を参照してください。
3. 組み込み Python の `sys.path` 変数に、Python パッケージのインポートに必要な正しいディレクトリが含まれていることを確認します。

   "組み込み Python と sys.path" を参照してください。

7.1.1 組み込み Python と sys.path

組み込み Python を起動すると、これは `sys.path` 変数に含まれるディレクトリを使用して、インポートする Python パッケージを見つけます。

オペレーティング・システムの既定バージョンの Python で組み込み Python を使用する場合、`sys.path` には既に、以下のような正しいディレクトリが含まれています。

- <installdir>/lib/python (InterSystems IRIS 用に予約された Python パッケージ)
- <installdir>/mgr/python (ユーザがインストールした Python パッケージ)

- 既定の Python バージョンのグローバル・パッケージ・リポジトリ

  例：/usr/local/lib/python3.10/dist-packages

  この場所は、オペレーティング・システム、Python バージョン、およびその他の要素によって異なります。ここに示す例は、Ubuntu 22.04 での Python 3.10 の場合です。

Ubuntu 22.04 上の既定バージョンの Python (3.10) では、組み込み Python の sys.path は以下のようになります。

```
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']
```

重要　sys.path に、/home/<user>/.local/lib/python3.10/site-packages などのホーム・ディレクトリ内のディレクトリが含まれる場合、ローカル・パッケージ・リポジトリにパッケージがインストールされていることを示している可能性があります。例えば、--target 属性を指定せずに (および sudo を使用せずに) パッケージをインストールする場合、Python はホーム・ディレクトリ内のローカル・パッケージ・リポジトリにパッケージをインストールします。他のユーザがパッケージをインポートしようとすると、失敗します。

インターシステムズでは --target <installdir>/mgr/python オプションを使用することを推奨していますが、sudo を使用して --target 属性を省略してパッケージをインストールすると、パッケージはグローバル・パッケージ・リポジトリにインストールされます。これらのパッケージは、どのユーザでもインポートできます。

Anaconda のような代替ディストリビューションに切り替える場合、InterSystems IRIS はそのパッケージ・リポジトリの場所を認識できない可能性があります。InterSystems IRIS は、正しいディレクトリ、すなわちディレクトリ <installdir>/lib/python の iris_site.py ファイルを含むように sys.path を調整するのに役立つツールを提供します。

Ubuntu 22.04 で Anaconda を使用している場合は、iris_site.py ファイルを以下のように編集します。

```
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/'])
```

上記の iris_site.py では、組み込み Python の sys.path は以下のようになります。

```
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']
```

フレキシブル Python ランタイムに適した sys.path にするのに、何回かの反復処理が必要になることがあります。InterSystems IRIS 外で Python を起動し、その sys.path を組み込み Python 内の sys.path と比較して、想定されるディレクトリがすべて含まれていることを確認するとよいでしょう。

注釈　PythonRuntimeLibrary 構成設定または iris_site.py に対する変更は、新しいセッションの開始時に有効になります。InterSystems IRIS を再起動する必要はありません。

7.1.2 Python バージョン情報の確認

フレキシブル Python ランタイム機能を使用している場合、組み込み Python が使用している Python のバージョンと、システムで使用されている既定のバージョンを確認すると便利なことがあります。そのためには、%SYS.Python クラスの GetPythonInfo() メソッドを呼び出すと簡単です。

以下の例は、x86 アーキテクチャの Ubuntu 22.04 で、使用されている Python ランタイム・ライブラリが `/usr/lib/x86_64-linux-gnu/libpython3.10.so.1`、実行されている組み込み Python のバージョンが `3.10.13`、システム・バージョンが `3.8.10` であることを示しています。

```
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"
```

この情報は、オペレーティング・システム、Python バージョン、およびその他の要素によって異なります。

7.2 フレキシブル Python ランタイムの例 : Ubuntu 22.04 の Python 3.11

Python 3.10 は、Ubuntu 22.04 の Python の既定バージョンです。この例では、組み込み Python で Python 3.11 を使用する方法を示します。

注釈　この例は、x86 アーキテクチャの Ubuntu 22.04 の場合です。ARM アーキテクチャの場合、ファイル名とディレクトリ名は異なることがあります。

1. コマンド行から Python 3.11 をインストールします。

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

2. Python 3.11 libpython.so 共有ライブラリをインストールします。

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

3. `PythonRuntimeLibrary` 構成設定を `/usr/lib/x86_64-linux-gnu/libpython3.11.so.1` に設定します。

   詳細は、"PythonRuntimeLibrary" を参照してください。

4. ターミナルから組み込み Python を起動し、`sys.path` に Python 3.11 パッケージ・ディレクトリが含まれていることを確認します。

   ```
   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. ターミナルから %SYS.Python クラスの GetPythonInfo() メソッドを使用して、Python のバージョン情報を表示します。

```
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"
```

この例は、使用されている Python ランタイム・ライブラリが `/usr/lib/x86_64-linux-gnu/libpython3.11.so.1`、実行されている組み込み Python のバージョンが `3.11.0rc1`、システム・バージョンが `3.10.6` であることを示しています。

7.3 フレキシブル Python ランタイムの例 : Ubuntu 22.04 の Anaconda

Anaconda は、一般的にデータ・サイエンスや人工知能アプリケーションに使用される Python ベースのプラットフォームです。この例では、Ubuntu 22.04 で Anaconda を使用する方法を示します。

注釈　この例は、x86 アーキテクチャの Ubuntu 22.04 の場合です。ARM アーキテクチャの場合、ファイル名とディレクトリ名は異なることがあります。

1. https://www.anaconda.com/download から Anaconda をダウンロードします。

   ダウンロードは、Anaconda3-2024.02-1-Linux-x86_64.sh. のような名前のシェル・スクリプトです。

2. コマンド行から、以下のように Anaconda インストール・スクリプトを実行します。

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

3. `PythonRuntimeLibrary` 構成設定を `/opt/anaconda3/lib/libpython3.11.so` に設定します。

   詳細は、"PythonRuntimeLibrary" を参照してください。

4. ディレクトリ <installdir>/lib/python の iris_site.py ファイルを編集して、Anaconda パッケージ・リポジトリを `sys.path` に追加します。

```
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. 組み込み Python を起動してエラー `<class 'ModuleNotFoundError'>: No module named 'math' - iris loader failed` が表示された場合は、PATH 環境変数を編集してディレクトリ `/opt/anaconda3/bin` を先頭に追加します。

```
$ 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. ターミナルから組み込み Python を起動し、sys.path に Anaconda パッケージ・リポジトリが含まれていることを確認します。

```
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. ターミナルから %SYS.Python クラスの GetPythonInfo() メソッドを使用して、Python のバージョン情報を表示します。

```
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"
```

この例は、使用されている Python ランタイム・ライブラリが /opt/anaconda3/lib/libpython3.11.so、実行されている組み込み Python のバージョンが 3.11.7、システム・バージョンが 3.10.6 であることを示しています。

InterSystems IRIS Python モジュール・リファレンス

組み込み Python を使用していて、InterSystems IRIS とやり取りする必要がある場合、`iris` モジュールを使用して InterSystems IRIS 機能を有効にできます。

InterSystems IRIS Python モジュール·コア API

このセクションでは、InterSystems IRIS Python モジュールのコア関数の API ドキュメントを提供します。これらの関数を使用して、InterSystems IRIS のクラスやメソッドにアクセスし、InterSystems IRIS のトランザクション処理機能を使用したり、その他の InterSystems IRIS のコア・タスクを実行することができます。

概要

以下のテーブルに、iris モジュールのコア関数の概要を示します。組み込み Python からこのモジュールを利用するには、`import iris` を使用します。

グループ	関数
コードの実行	check_status()、routine()
ロックと並行処理の制御	lock()、unlock()
参照の作成	arrayref()、cls()、gref()、ref()
トランザクション処理 *	tcommit()、tlevel()、trollback()、trollbackone()、tstart()

*トランザクションを使用して InterSystems IRIS データベースの論理的な整合性を維持する方法は、"トランザクション処理" を参照してください。

arrayref(dictionary)

Python ディクショナリから ObjectScript 配列を作成し、その配列への参照を返します。

配列を引数として期待する以下の ObjectScript メソッドを持つ、User.ArrayTest という InterSystems IRIS クラスがあるとします。

```
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"
}
```

メソッド WriteContents() は配列の内容を書き込み、Modify() は配列の内容を変更し、StoreGlobal() は配列の内容を取得してグローバルに格納します。

Python からディクショナリ myDict を作成し、iris.arrayref() を使用してその内容を ObjectScript 配列に配置し、その配列への参照を返します。その後、その参照を User.ArrayTest の 3 つのメソッドに渡すことができます。

```
>>> 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'
```

さらに、ObjectScript から、グローバル ^MyGlobal に配列と同じデータが含まれていることを確認できます。

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

ObjectScript 配列の詳細は、"多次元配列" を参照してください。

check_status(status)

status にエラーが含まれる場合に例外を生成します。エラー状態が発生していない場合は、None を返します。

必須の Name プロパティを持つ InterSystems IRIS クラス Sample.Company がある場合、Name プロパティなしでこのクラスのインスタンスを保存しようとすると、エラー・ステータスが生じます。以下の例では、iris.check_status() を使用して _Save() メソッドから返されたステータスを確認し、エラーが含まれる場合は例外をスローします。

```
>>> 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)

InterSystems IRIS クラスへの参照を返します。これにより、Python クラスと同じ方法でそのクラスのプロパティとメソッドにアクセスできます。iris.cls() を使用して、組み込み InterSystems IRIS クラスまたは自分で記述したカスタム InterSystems IRIS クラスの両方にアクセスできます。

以下の例では、iris.cls() を使用して、組み込み InterSystems IRIS クラス %SYS.System への参照を返します。その後、その GetInstanceName() メソッドを呼び出します。

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

gref(global_name)

InterSystems IRIS グローバルへの参照を返します。グローバルは、既に存在する場合も存在しない場合もあります。

以下の例は、iris.gref() を使用して、変数 day をグローバル ^day への参照に設定します。

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

次の例は、^day(1, "name") に格納された値を出力し、それらのキーに現在値が格納されていないため、None を出力します。次に、値 "Sunday" をその場所に格納し、格納された値を取得して出力します。

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

InterSystems IRIS グローバル参照で使用できるメソッドの詳細は、"グローバル参照 API" を参照してください。

グローバルに関する背景情報は、"グローバルの概要" を参照してください。

lock(lock_list, timeout_value, locktype)

ロック名のリスト、オプションのタイムアウト値 (秒)、オプションのロック・タイプを指定して、ロックを設定します。locktype が "S" の場合、共有ロックであることを示します。

InterSystems IRIS では、ロックは、複数のユーザまたはプロセスが同じリソース (通常はグローバル) に同時にアクセスしたり、変更を加えるのを防ぐために使用されます。例えば、リソースに書き込むプロセスは排他ロック (既定) を要求して、別のプロセスがそのリソースに同時に読み取りや書き込みを行わないようにする必要があります。リソースを読み取るプロセスは共有ロックを要求して、別のプロセスがそのリソースを同時に読み取ることはできるが、リソースに書き込むことはできないようにすることができます。プロセスはタイムアウト値を指定して、リソースが利用可能になるまで永久に待機しないようにすることができます。

以下の例では、iris.lock() を使用して、^one および ^two というロックに対して排他ロックを要求します。要求が成功した場合、この呼び出しは True を返します。

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

その後、別のプロセスが ^one に対して共有ロックを要求し、最初のプロセスが 30 秒以内にロックを解放しなかった場合、以下の呼び出しは False を返します。

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

保護するリソースが使用されなくなったら、プロセスは unlock() を使用してロックを放棄する必要があります。

InterSystems IRIS でのロックの使用方法の詳細は、"ロックと並行処理の制御" を参照してください。

ref(value)

指定された値を使用して iris.ref オブジェクトを作成します。これは、ObjectScript メソッドに引数を参照渡しする必要がある場合に便利です。

以下の例は、iris.ref() を使用して、値 2000 の iris.ref オブジェクトを作成します。

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

InterSystems IRIS クラス User.Diet に、これから摂取する食品の名前とその日の現在のカロリー摂取量を引数として取る Eat() いうメソッドがあり、その calories が参照によって渡され、新しいカロリー摂取量で更新されるとします。以下の例は、Eat() の呼び出し後に、変数 calories の値が 2000 から 2250 に更新されたことを示しています。

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

ObjectScript での引数の参照渡しの詳細は、"引数の渡し方の指示" を参照してください。

routine(routine, args)

オプションで指定されたタグで、InterSystems IRIS ルーチンを呼び出します。呼び出しで渡す必要のある引数は、ルーチン名の後にコンマで区切って指定します。

以下の例では、%SYS ネームスペースでの実行時に、iris.routine() を使用してルーチン ^SECURITY を呼び出します。

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


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

2 つの数値を加算する関数 Sum() を持つルーチン ^Math がある場合、以下の例では 4 と 3 を加算します。

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

ObjectScript でルーチンを使用する方法の詳細は、"ルーチン" を参照してください。

tcommit()

InterSystems IRIS トランザクションの正常終了を表します。

iris.tcommit() を使用してトランザクションの正常終了を表し、入れ子レベルを 1 デクリメントします。

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

トランザクションを適切に入れ子にするには、すべての iris.tstart() を iris.tcommit() とペアにする必要があります。

トランザクション中でないときに iris.tcommit() を呼び出すと、値 <COMMAND> で例外が発生します。

"tstart()"、"tlevel()"、"trollback()"、"trollbackone()" も参照してください。

tlevel()

トランザクションが現在進行中かどうかを検出し、入れ子レベルを返します。iris.tstart() を呼び出すと入れ子レベルがインクリメントされ、iris.tcommit() を呼び出すと入れ子レベルがデクリメントされます。値がゼロの場合、トランザクション中でないことを示します。

以下の例は、トランザクションのさまざまな入れ子レベルで iris.level() が返す値を示しています。

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

"tstart()"、"tcommit()"、"trollback()"、"trollbackone()" も参照してください。

trollback()

現在進行中のトランザクションをすべてロール・バックし、ジャーナリングされたすべてのデータベース値を最初のトランザクションの開始時の値にリストアします。また、トランザクションの入れ子レベルを 0 にリセットします。

この簡単な例では、グローバル ^a(1) を値 “hello” に初期化します。その後、トランザクションを開始して、^a(1) を値 “goodbye” に設定します。ただし、トランザクションがコミットされる前に、iris.trollback() を呼び出します。これにより、トランザクションの入れ子レベルが 0 にリセットされ、^a(1) がトランザクション開始前に保持していた値にリストアされます。

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

"tstart()"、"tcommit()"、"tlevel()"、"trollbackone()" も参照してください。

trollbackone()

入れ子になったトランザクションの現在のレベル、つまり、直近の iris.tstart() よって開始されたトランザクションをロール・バックします。また、トランザクションの入れ子レベルを 1 デクリメントします。

この例では、グローバル ^a(1) を値 4、^b(1) を値 “lemon” に初期化します。その後、トランザクションを開始して、^a(1) を 9 に設定します。次に、入れ子になったトランザクションを開始して、^b(1) を “lime” に設定します。その後、iris.trollbackone() を呼び出して内側のトランザクションをロール・バックし、iris.commit() を呼び出して外側のトランザクションをコミットします。つまり、^a(1) は新しい値を保持しますが、^b(1) は元の値にロール・バックされます。

```
>>> 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'
```

"tstart()"、"tcommit()"、"tlevel()"、"trollback()" も参照してください。

tstart()

InterSystems IRIS トランザクションの開始を表します。

トランザクションはコマンドの集まりで、トランザクションが成功したと見なすにはそれらすべてのコマンドを完了する必要があります。例えば、ある銀行口座から別の銀行口座に送金するトランザクションがある場合、最初の口座からの引き出しと 2 番目の口座への預け入れの両方が成功した場合にのみ、トランザクションは成功したと言えます。トランザクションが失敗した場合、データベースはトランザクション開始前の状態にロール・バックすることができます。

iris.start() を使用してトランザクションの開始を表し、トランザクションの入れ子レベルを 1 インクリメントします。

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

"tcommit()"、"tlevel()"、"trollback()"、"trollbackone()" も参照してください。

InterSystems IRIS のトランザクション処理の仕組みの詳細は、"トランザクション処理" を参照してください。

unlock(lock_list, timeout_value, locktype)

ロック名のリスト、オプションのタイムアウト値 (秒)、オプションのロック・タイプを指定して、ロックを削除します。

コードによってロックを設定し、リソースへのアクセスを制御している場合は、それらのリソースの使用が終了したら、アンロックする必要があります。

以下の例では、iris.unlock() を使用して、^one および ^two というロックを解除します。

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

"lock()" も参照してください。

グローバル参照 API

このセクションでは、InterSystems IRIS Python モジュールの **gref** クラスのメソッドの API ドキュメントを提供します。これらのメソッドにより、InterSystems IRIS グローバルにアクセスし、操作することができます。

概要

以下のテーブルに、InterSystems IRIS Python モジュールの **gref** クラスのメソッドの概要を示します。組み込み Python からこのクラスを使用するには、まず `import iris` を実行し、次に iris.gref() 関数を使用してグローバルへの参照を取得します。("iris.gref()" を参照してください。)

グループ	設定
グローバルのノードでの操作	data()、get()、getAsBytes()、kill()、set()
グローバルの検索	keys()、order()、orderiter()、query()

グローバルに関する背景情報は、"グローバルの概要" を参照してください。

data(key)

グローバルのノードにデータが含まれるかどうか、および下位ノードがあるかどうかを確認します。ノードの key は、リストとして渡されます。値が None (または空のリスト) のキーを渡すことは、グローバルのルート・ノードを意味します。

データへのアクセスを試行してエラーが発生する前に、data() を使用して、データが含まれているかどうかノードを調べることができます。ノードが定義されていない (データが含まれていない) 場合、メソッドは 0 を返し、ノードが定義されている (データが含まれている) 場合は 1、ノードは定義されていないが下位ノードがある場合は 10、ノードが定義されていて下位ノードがある場合は 11 を返します。

以下の内容のグローバル `^a` があるとします。

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

data() を使用して、以下の例のように、グローバルのさまざまなノードをテストできます。

```
>>> 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
```

モジュロ 2 演算を使用すると、下位ノードがあるかどうかに関係なく、ノードにデータが含まれるかどうかを確認できます。

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

get(key)

グローバルのノードに格納されている値を取得します。ノードの key は、リストとして渡されます。値が None (または空のリスト) のキーを渡すことは、グローバルのルート・ノードを意味します。

以下の内容のグローバル ^a があるとします。

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

get() を使用して、以下の例のように、グローバルのさまざまなノードからデータを取得できます。

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

また、Python ディクショナリの場合と同様に、ノードの値を直接取得することもできます。dunder メソッド __getitem__() を使用することもできます。

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

get() を使用して定義されていないノードからデータを取得すると、エラーが発生します。

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

データの取得を試行する前に、data() メソッドを使用して、ノードにデータが含まれているかどうかをテストできます。

定義されていないノードから直接、または __getitem__() を使用してデータを取得すると、エラーが発生する代わりに None が返されます。

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

"getAsBytes()" も参照してください。

getAsBytes(key)

グローバルのノードに格納されている文字列値を取得して、Python の bytes データ型に変換します。ノードの key は、リストとして渡されます。値が None (または空のリスト) のキーを渡すことは、グローバルのルート・ノードを意味します。

以下の内容のグローバル ^a があるとします。

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

getAsBytes() を使用して、以下の例のように、グローバルのさまざまなノードからデータを取得できます。

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

getAsBytes() を使用して定義されていないノードからデータを取得すると、エラーが発生します。

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

データの取得を試行する前に、data() メソッドを使用して、ノードにデータが含まれているかどうかをテストできます。

"get()" も参照してください。

keys(key)

指定されたキーから始めて、グローバルのキーを返します。開始 key はリストとして渡されます。空のリストを渡すことは、グローバルのルート・ノードを意味します。

以下の内容のグローバル ^mlb があるとします。

```
^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"
```

以下のように、keys() を使用してグローバルのキーを取得し、その値を出力できます。

```
>>> 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
```

開始キーがグローバルのノードとして存在する必要はありません。グローバルはソートされた順序で格納されるため、keys() はソート順序に従って次のノードのキーから始まります。以下に例を示します。

```
>>> 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
```

get() を使用して各ノードの値を取得することもできますが、まず data() を使用して各ノードをテストし、データが含まれていることを確認する必要があります。

order() を使用すると、グローバルの 1 つのレベルでノードが検索されます。

kill(key)

グローバルのノードを削除します (存在する場合)。ノードの key は、リストとして渡されます。この操作により、ノードの下位ノードも削除されます。値が None (または空のリスト) のキーを渡すことは、グローバルのルート・ノードを意味します。

以下の内容のグローバル ^a があるとします。

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

kill() を使用して、以下の例のように、グローバルのノードとその下位ノードを削除できます。

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

これで、グローバルの内容は以下のようになります。

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

キーに None を渡すと、グローバル全体が削除されます。

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

order(key)

指定されたキーから始めて、グローバルのそのレベルの次のキーを返します。開始 key はリストとして渡されます。開始キーに続くキーがない場合、order() は None を返します。

以下の内容のグローバル ^mlb があるとします。

```
^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"
```

order() を使用して、以下の例に示すように、指定されたキーの次のキーを取得できます。

```
>>> 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'
```

開始キーがグローバルのノードとして存在する必要はありません。グローバルはソートされた順序で格納されるため、order() はソート順序に従って次のノードのキーを返します。以下に例を示します。

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

while ループを使用して、特定レベルでグローバルのノードを検索し、戻り値が None になると終了します。開始キーを空の文字列に設定することは、“そのレベルの最初から開始する” ことを意味します。

以下の例では、グローバルの最上位のノードを検索します。

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

以下の例では、グローバルの 3 番目のレベルのノードを検索します。

```
>>> 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
```

必要に応じて while ループを入れ子にしたり、keys() または query() を使用してグローバル全体を検索できます。

orderiter(key)

指定されたキーから始めて次のリーフ・ノードまで、グローバルのキーと値を返します。開始 key はリストとして渡されます。空のリストを渡すことは、グローバルのルート・ノードを意味します。

以下の内容のグローバル ^mlb があるとします。

```
^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"
```

以下の例は、orderiter() を使用し、ルートから始めて次のリーフ・ノードまで、グローバルを検索します。

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

開始キーがグローバルのノードとして存在する必要はありません。グローバルはソートされた順序で格納されるため、orderiter() はソート順序に従って次のノードを探します。以下に例を示します。

```
>>> 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)

指定されたキーから始めてグローバルを検索し、各キーと値を返します。開始 key はリストとして渡されます。空のリストを渡すことは、グローバルのルート・ノードを意味します。

以下の内容のグローバル ^mlb があるとします。

```
^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"
```

以下の例は、query() を使用し、ルートから始めてグローバルを検索します。

```
>>> 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
```

開始キーがグローバルのノードとして存在する必要はありません。グローバルはソートされた順序で格納されるため、query() はソート順序に従って次のノードを探します。以下に例を示します。

```
>>> 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)

グローバルのノードを指定された値に設定します。ノードの key はリストとして渡され、value は格納される値です。値が None (または空のリスト) のキーを渡すことは、グローバルのルート・ノードを意味します。

以下の例は、グローバル ^messages への参照を取得し、set() を使用してグローバルのいくつかのノードの値を設定します。

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

グローバル ^messages がまだ存在しない場合、以下のようになります。

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

^messages が既に存在する場合、グローバルに新しい値が追加され、それらのノードの既存のデータは上書きされる可能性があります。設定を試行する前に、data() メソッドを使用して、ノードにデータが既に含まれているかどうかをテストできます。

以下の例のように、Python ディクショナリの場合と同様に、グローバル・ノードを直接設定することもできます。

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

これで、グローバル ^messages は以下のようになります。

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