Python Runtime (rev1) [05/April/2020]

Introduction

The IRIS Python Runtime is a fusion of the "C" Python runtime with the IRIS runtime — this allows for "toll free bridging" between the IRIS and Python worlds. Python objects are 1st class citizens in IRIS and vice-versa.

This is made possible by the fact that Python has been designed to be embedded and extended, much of Python's popularity revolves around the large eco-system which in many ways uses Python as "glue" around C code (for example Pandas, Numpy, OpenCV etc). Python is also embedded in a large number of products and very well suited to this (for example Sublime Text embeds Python as a scripting language).

The "C" Python runtime is also very similar to the IRIS runtime, Python's type system is compatible and values themselves are easily converted between the two runtimes. For example Python strings are UTF-8, whereas IRIS ones are Unicode, numerical types are identical — integers, doubles etc. Python objects also have similar dispatch system that makes it easy for us to "project" Python objects into IRIS and vice-versa.

Installation

Requirements

There are no requirements beyond having an IRIS kit with the Python Runtime present — due to the well known issues with Python versions we intentionally ship our own version of Python 3 (currently 3.7.6) to ensure we have a correctly functioning runtime and also to ensure that we are not shipping code with any known vulnerabilities/CVEs). Upgrading the Python-enabled IRIS instances with later versions will additionally upgrade the bundled Python.

Platform Support

Platform support can be broken down as follows into one of "Full" (includes Integrated ML and Python), "Python Only" and "None":

Platform	Status	Notes
Linux	Full	Incl. containers
macOS	Full	
AIX	Python Only	Limited python package support (e.g. missing xgboost)
Windows	None	We will have full support once we have solved some internal build infrastructure issues

NOTE: It's unlikely that AIX support will move beyond Python only!

Install Footprint

The Python runtime makes some changes to the install footprint, primarily it adds the following:

Item	Description
`bin/libpython*.(so|dll)`	Python runtime engine
`lib/python3.7`	Python runtime support (read only)
`mgr/python`	Python "user" code location

NOTE: The `lib/python3.7` directory is removed and replaced during upgrade so NO USER ADDED CODE should be put here; USER ADDED CODE should be placed in `mgr/python`

Getting Started

The initial version of the Python runtime is “IRIS centric”, that is you are starting in IRIS and “using” Python code. Subsequent versions will allow you to start from the execution of a Python script and “use” IRIS code.

Initially, we need an IRIS version of the Python `import` statement, so we use `$system.Python.Import()` in IRIS for that.

Let’s start with an example:

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

USER>do builtins.print("Hello World!")
Hello World!
```

Here, we import the Python `builtins` package and call the `print()` method. When we import the builtins package, IRIS returns an oref that represents that Python object as a 1st class citizen in the IRIS system, we can pass this value to any IRIS function. IRIS knows how to pass values to and from Python, so in this case, we pass the IRIS string “Hello World!” to Python and convert it to a Python UTF-8 string.

We can use the `ZWRITE` command in IRIS to examine Python objects (internally, if `ZWRITE` detects a Python oref, then it uses the `str()` method from the `builtins` package to return a Python string representation of the object:

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

Since the builtins oref represents a Python module. We can also see this if we created a python list using the builtins `list()` method:

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

USER>zwrite a
a=2@%SYS.Python  ; []  ; <OREF>
```

We can use the `builtins.type()` method to see what Python thinks `a` actually is:

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

Interestingly, the `type()` method actually returns a 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(a)
3@%SYS.Python  ; ['__add__', '__class__', '__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>
```

Working with Python

Installing Packages

The IRIS Python Runtime looks in two places for Python packages, in `$INSTALLDIR/lib/python3.7` (for InterSystems supplied packages) and `$INSTALLDIR/mgr/python` (for user packages). You can install additional packages into `$INSTALLDIR/mgr/python` using the `$system.Python.Install()` command which is essentially equivalent to a `pip3 install` invocation:

```
USER>do $system.Python.Install("packagename")
```

NOTE: You must have the `%Developer` role as well as write permissions to the `$INSTALLDIR/mgr/python` in order for this to succeed.

```
USER>do $system.Python.Install("imapclient")
Collecting imapclient
  Using cached
https://files.pythonhosted.org/packages/dc/39/e1c2c2c6e2356ab6ea81fcfc0a74b044b311d6a91a45300811d9a6077ef7/IMAPClient-2.1.0-
py2.py3-none-any.whl
Collecting six (from imapclient)
  Using cached https://files.pythonhosted.org/packages/65/eb/1f97cb97bfc2390a276969c6fae16075da282f5058082d4cb10c6c5c1dba/six-
1.14.0-py2.py3-none-any.whl
Installing collected packages: six, imapclient
Successfully installed imapclient-2.1.0 six-1.14.0

USER>!ls -al ../python
total 72
drwxrwxrwx   8 root   irisusr    256 Apr  3 11:59 .
drwxrwxr-x  58 root   irisusr   1856 Apr  3 00:30 ..
drwxr-xr-x   7 user   staff      224 Apr  3 11:59 IMAPClient-2.1.0.dist-info
drwxr-xr-x   3 user   staff       96 Apr  3 11:59 __pycache__
drwxr-xr-x  23 user   staff      736 Apr  3 11:59 imapclient
drwxr-xr-x   8 user   staff      256 Apr  3 11:59 six-1.14.0.dist-info
-rw-r--r--   1 user   staff    34074 Apr  3 11:59 six.py
```

At this point, the imapclient package is available for use in IRIS:

```
USER>s imap = $system.Python.Import("imapclient")

USER>zw imap
imap=4@%SYS.Python  ; <module 'imapclient' from '/usr/local/isc/quickml/mgr/python/imapclient/__init__.py'>  ; <OREF>
```

Language Bridging

While Python is a very similar language to IRIS' ObjectScript, there are a few "impedance mismatches" that need some bridging of the two languages and their respective runtimes.

Identifier Names

The rules for identifier names in Python are different from IRIS, for example underbar `_` is allowed in methods names, and in fact is widely used for the so-called "dunder" methods such as `__getitem__` or `__class__`. We can easily work around this as IRIS allows for method and property names that contain "illegal" characters to be simply quoted; for example:

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

USER>zwrite a."__class__"
3@%SYS.Python  ; <class 'list'>  ; <OREF>
```

Sentinel Values

Python makes some use of sentinel values such as `None`, `True` and `False`. IRIS exposes these constants via methods on `$system.Python`:

```
USER>zwrite $system.Python.None()
3@%SYS.Python  ; None  ; <OREF>

USER>zwrite $system.Python.True()
3@%SYS.Python  ; True  ; <OREF>

USER>zwrite $system.Python.False()
3@%SYS.Python  ; False  ; <OREF>
```

Additionally, if you pass “missing" arguments to Python methods then we pass `None` as the value (which means on the Python side you'll either get `None`, the default value, or an argument error.

Keyword Arguments

A common pattern in Python is to have initializer (or class) methods that return objects initialized based on a named keyword arguments, for example:

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

In many cases, such an initializer method may take a dozen or more keyword arguments along with reasonable defaults, for example using the Python requests package to make an HTTP GET request:

```
r = requests.get(url, headers=headers, cookies=jar)
```

As an example, the get method in the requests package takes the following parameters:

Parameter	Description	Default
`url`	Required. The url of the request	
`params`	A dictionary, list of tuples or bytes to send as a query string.	`None`
`allow_redirects`	A Boolean to enable/disable redirection.	`True`
`auth`	A tuple to enable a certain HTTP authentication.	`None`
`cert`	A String or Tuple specifying a cert file or key.	`None`
`cookies`	A dictionary of cookies to send to the specified url.	`None`
`headers`	A dictionary of HTTP headers to send to the specified url.	`None`
`proxies`	A dictionary of the protocol to the proxy url.	`None`
`stream`	A Boolean indication if the response should be immediately downloaded (False) or streamed (True).	`False`
`timeout`	A number, or a tuple, indicating how many seconds to wait for the client to make a connection and/or send a response.	`None`
`verify`	A Boolean or a String indication to verify the servers TLS certificate or not.	`True`

Since IRIS does not allow for named arguments (at least from the callee's point of view), we need a mechanism to be able to aribitrarly provide

name argument pairs. IRIS does in fact already support variadic arguments via the `args...` syntax, but it only worked with an array of posititional values. For keyword arguments, we have extended the `args...` to allow for args being a dynamic object instance (or dynamic object literal), for example:

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

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

USER>do obj.mymethod({ "foo": 42, "baz": "baz" }...)
foo=42, bar=2, baz=baz
```

Exception Handling

The IRIS Python runtime integrates Python exceptions with the IRIS exception handling mechanism, so you can use `Try`/`Catch` to handle Python errors:

```
USER>try { do imap.foo() } catch { write "Error: ",$ze,! }
Error: <OBJECT DISPATCH> *<class 'AttributeError'>: module 'imapclient' has no attribute 'foo' - Undefined method
```

Similarly, IRIS errors that are `throw`'n can be caught on the Python side.

Bytes vs Strings

Python draws a clear distinction between "bytes" which are simply 8-bit bytes and strings which are sequences of UTF-8 bytes that represent a string. In Python, bytes are never converted in anyway, but strings might be converted depending on the character set in use by the host OS (e.g. Latin-1).

In IRIS, we have no distinction between bytes and strings, while we support Unicode strings (USC-2/UTF-16) any string that contains values <256 could either be a string, or could be bytes. For this reason, the following rules apply when passing strings and bytes to and from Python:

1. IRIS Strings are assumed to be strings and are converted to UTF-8 going to Python
2. Python strings are converted back from UTF-8 to IRIS strings and may result in wide characters
3. Python "bytes" are copied back as 8-bit strings, if the length of the bytes exceeds the maximum string length, then a Python bytes object is returned (see below)
4. To pass bytes into Python, use the `$system.Python.Bytes()` value constructor which will NOT convert the underlying IRIS string to UTF-8

```
USER>set b = $system.Python.Bytes("Hello Bytes!")

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

USER>zw builtins.type(b)
5@%SYS.Python  ; <class 'bytes'>  ; <OREF>
```

To construct large bytes objects in Python bigger than the 3.8MB max string length in IRIS, you can use the 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 or call `__bytes__()`

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

USER>do ba.extend($system.Python.Bytes("chunk 1"))

USER>do ba.extend($system.Python.Bytes("chunk 2"))

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

DataFrames

IRIS has some limited support for the construction of Pandas DataFrames from IRIS SQL resultsets — this is currently based on some helper methods designed for the Integrated ML's AutoML provider, although this helpers can be called from any caller:

```
USER>set stmt = ##class(%SQL.Statement).%New()

USER>set status = stmt.%Prepare("SELECT * FROM Sample.Person")

USER>set resultset = stmt.%Execute()

USER>set automl = ##class(%ML.AutoML.Provider).%New()

USER>set status = automl.%OnInit()

USER>set status = automl.%ResultSetToDataFrame(resultset, , .dataframe)
```

Dictionaries

Python makes extensive use of dictionaries, typically Python uses some syntactic sugar to enable this, for example:

```
>>>d = { 'foo': 42 }
>>> d
{'foo': 1}
```

On the IRIS side, the same can be achieved as follows:

```
USER>set d = builtins.dict()

USER>do d.setdefault("foo",42)

USER>zwrite d
d=2@%SYS.Python  ; {'foo': 42}  ; <OREF>
```

And values extracted via the `__getitem__()` "dunder" method:

```
USER>zw d."__getitem__"("foo")
42
```

Lists

Similar to dictionaries, lists can be created via the builtins package list() method and manipulated via various methods:

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

USER>do l.append(1)

USER>do l.append(2)

USER>zwrite l
l=3@%SYS.Python  ; [1, 2]  ; <OREF>
```

And values extracted via the `__getitem__()` "dunder" method:

```
USER>zw l."__getitem__"(0)
1
```

NOTE: Indexes in Python are zero based!

Runtime & Debugging

Fusing the Python and IRIS runtimes means that there are some additional issues to consider.

StdOut / StdErr

We map Python's `StdOut` to the IRIS console, this means any `print()` or `put()` statements will appear on the Terminal. Additionally you can use IRIS's I/O redirection to trap this output as needed (see the `%RunMethodWithCapture()` method in `%ML.Utils`).

NOTE: `StdErr` output is mapped to IRIS's messages.log file so only pertinent error information should be written here to avoid cluttering the file

Signal Handling

Both IRIS and Python establish signal handlers to ensure proper integration with the UNIX operating system, however sometimes these can conflict. Ideally, we would run with the IRIS signal handlers in place while running IRIS code and swap to Python signal handlers while running Python code. Benchmarking has show calls to `sigaction()` have some cost so by default we leave the IRIS signal handlers in place. This means that for a long running Python method, Ctrl-C would not be passed to it.

If you need to have signal handling enabled when calling a Python method, you can set or clear this via the `ChangeSignalState()` method in `$system.Python`:

```
USER>set oldstate = $system.Python.ChangeSignalState(0)  ; Enable signal handling for Python

USER>do obj."slow_python_method"()  ; Ctrl-C will now work and be captured by Python

USER>do $system.Python.ChangeSignalState(oldstate)  ; Restore previous signal state
```

Debugging

Sometimes it's necessary to debug Python code, for example to diagnose a traceback in Python code. It's possible to use the Python debugger *inside* IRIS! However, you need to enable it by calling to prevent the IRIS Python runtime from handling and clearing errors when invoking Python code. This is done by the `Debugging()` method in `$system.Python`:

```
USER>do $system.Debugging(1)  ; don't allow IRIS to consume python errors

USER>set pdb = $system.Python.Import("pdb")  ; import the Python debugging

USER>do obj."erroneous_python_method"()

USER>do pdb.pm()  ; run a postmortem on the most recent traceback
```

Profiling

By the magic of the IRIS/Python runtime, it's also possible to do profiling of Python code using the cProfile and pstats packages:

```
USER>set cp = $system.Python.Import("cProfile")

USER>set pstats = $system.Python.Import("pstats")

USER>set profiler = cp.Profile()

USER>do profiler.enable()

USER>do obj."python_method_to_profile"()
```

```
USER>do profiler.disable()

USER>set ps = pstats.Stats(profiler)

USER>do ps."sort_stats"(pstats.SortKey.CUMULATIVE)

USER>do ps."print_stats"()
```

This will print out a profiling report based on the cumulative time spent in the Python functions.

Examples

IMAP Client

This example is an IRIS routine that attaches to an IMAP server via TLS using the `imapclient` package. This assumes you have already installed the `imapclient` package via `$system.Python.Install("imapclient")`:

```
imaptest(host,user,password) ;
    set ssl = $system.Python.Import("ssl")
    set sslctx = ssl."create_default_context"()
    set sslctx."check_hostname"=0
    set sslctx."verify_mode"=ssl."CERT_NONE"
    set imapclient = $system.Python.Import("imapclient")
    set args = { "ssl": 1, "ssl_context": (sslctx) }
    set conn = imapclient.IMAPClient(host,args...)
    do conn.login(user,password)
    set folder = conn."select_folder"("INBOX")
    set key = $system.Python.Bytes("UNSEEN")
    set count = folder.get(key)."__getitem__"(0)
    quit
```

Note that currently the Python `ssl` package is unrelated to any `SSLConfig` instances defined in IRIS.

References

Links

Description	Link
Special Method Names ("Dunder" methods)	https://diveinto.org/python3/special-method-names.html