Article
José Pereira · Dec 27, 2021 12m read

Using Python to Implement an IMAP Client in InterSystems IRIS

In the previous articles, we learned the basics of using IMAP protocol to handle messages from mailboxes in an e-mail server. That was cool and interesting, but you could take advantage of implementations created by other ones, available in libraries ready to use.

One of the improvements to the IRIS data platform is the ability to write Python code alongside ObjectScript in the same IRIS process. This new feature is called Embedded Python. Embedded Python lets us bring to our ObjectScript code the power of the huge Python ecosystem ' s libraries

In this article, we ' ll use one of those libraries, called imaplib, to implement an IMAP client and integrate it with the IRIS Email Framework. We ' ll also review a practical example of how to use embedded Python to resolve real-world challenges on the IRIS platform with the help of the Python ecosystem.

You can find all code implemented here in this GitHub repository, in the python directory.

Note that Python code just works in recent IRIS versions. In this example, the used version was 2021.1.0.215.3-zpm. You can follow updates about Embedded Python here.

Using Embedded Python

The key to using embedded Python is the class %SYS.Python. By using this class, we can:

- Import Python libraries: ##class(%SYS.Python).Import(" package-name ")
- Import custom Python modules (*.py files) available into the local system: ##class(%SYS.Python).Import(" module-file.py ")
- Get some Python built-in types to be used in assignments or parameters, for instance:
  - Python None object: ##class(%SYS.Python).None()
  - Python True object: ##class(%SYS.Python).True()
  - Python False object: ##class(%SYS.Python).False()
- Convert ObjectScript strings to Python Bytes objects (8-bit strings): ##class(%SYS.Python).Bytes(" ObjectScript string ")

These methods create Python objects and return an ObjectScript object. We can use the Python object ' s properties and methods directly in our ObjectScript code.

For instance, let ' s see how we can implement this Python recipe for using the secrets library to generate passwords:

```
USER>Set string = ##class(%SYS.Python).Import("string")

USER>Set secrets = ##class(%SYS.Python).Import("secrets")

USER>ZWrite secrets // let's check what this object is...
secrets=1@%SYS.Python  ; <module 'secrets' from '/usr/irissys/lib/python3.7/secrets.p
y'>  ; <OREF>

USER>ZWrite string // same for this one...
```

```
string=2@%SYS.Python  ; <module 'string' from '/usr/irissys/lib/python3.7/string.py'>
  ; <OREF>

USER>Set alphabet = string."ascii_letters" _ string.digits // here we are accessing P
ython properties from string object

USER>Set pwd = ""

USER>For i=1:1:8 { Set pwd = pwd _ secrets.choice(alphabet) }

USER>Write pwd
Qv7HuOPV
```

In this code, we use several properties and methods from Python objects to set ObjectScript variables. We use ObjectScript variables as parameters for Python objects methods.

Another key point to using embedded Python is unique attributes and methods, sometimes called magical methods. Because everything in Python data models are objects, these attributes and methods provide the Python interpreter's interface. For example, here ' s how we retrieve an item from a list by its index, using the__getitem__ special method:

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

USER>Set list = b.list() // creates a Python list

USER>Do list.append(1)

USER>Do list.append(2)

USER>Do list.append(3)

USER>ZWrite list
list=4@%SYS.Python  ; [1, 2, 3]  ; <OREF>

USER>w list."__getitem__"(0) // in Python, indexes are 0-based
1
USER>w list."__getitem__"(2)
3
```

In the same way, we can get the length of the list by using the __len__ special method:

```
USER>Set listLen = list."__len__"()

USER>ZWrite listLen
listLen=3
```

We can combine them to iterate the list using ObjectScript:

```
USER>For i=0:1:(listLen - 1) { Write list."__getitem__"(i), ! }
1
2
3
```

If we need to use constant values like None, True, or False, we can use the following methods from the %SYS.Python class:

```
USER>Set none = ##class(%SYS.Python).None()

USER>Set true = ##class(%SYS.Python).True()

USER>Set false = ##class(%SYS.Python).False()

USER>ZWrite none, true, false
none=5@%SYS.Python  ; None  ; <OREF>
true=6@%SYS.Python  ; True  ; <OREF>
false=7@%SYS.Python  ; False  ; <OREF>
```

Similarly, we can convert an ObjectScript string to a Python Bytes object:

```
USER>Set bytes = ##class(%SYS.Python).Bytes("This is a string")

USER>ZWrite bytes
bytes=8@%SYS.Python  ; b'This is a string'  ; <OREF>
```

Finally, we define our custom Python modules and import them into the ObjectScript context.

You can find more useful resources on how to use embedded Python here. For instance, check out this nice example by Robert Cemper.

Writing an Alternative IMAP Client

To use imaplib to implement our IMAP client, we use the regular ObjectScript. We override its methods with the imaplib methods instead of implementing the IMAP protocol from the beginning.

First, we create a new class named dc.demo.imap.python.IMAPPy. This class uses two properties to store references to Python objects:

```
Class dc.demo.imap.python.IMAPPy Extends dc.demo.imap.IMAP
{

/// Stores the imaplib object reference
Property imaplib As %SYS.Python;

/// Stores the imaplib client instance
Property client As %SYS.Python;

...
```

Next, we import the imaplib library into the ObjectScript context, in the class constructor:

```
Method %OnNew() As %Status [ Private ]
{
    Set ..imaplib = ##class(%SYS.Python).Import("imaplib")
    Return $$$OK
```

```
}
```

Now, we can access all imaplib properties and methods using the imaplib class property. The first method that we override was the Connect method. This method uses the imaplib IMAP4_SSL method to make a connection to the IMAP server. It stores the imaplib client instance as a client property.

The login method of the imaplib client will authenticate login requests, as follows:

```
Method Connect(pServer As %String, pUserName As %String, pPassword As %String) As %St
atus
{
    If ..Connected Return $$$ERROR($$$ConnectedError)
    Set sc = $$$OK
    Try {
        Set ..Server = pServer
        Set ..UserName = pUserName
        Set ..client = ..imaplib."IMAP4_SSL"(..Server)
        Set resp = ..client.login(..UserName, pPassword)
        Set ..Connected = 1
    }
    Catch ex {
        Set sc = ex.AsStatus()
    }
    Return sc
}
```

The next method we override is the Disconnect method. This method now calls the logout method from the imaplib client:

```
Method Disconnect() As %Status
{
    Set sc = $$$OK
    Try {
        If ..Connected {
            Set tuple = ..client.logout()
            Set ..Connected = 0
        }
    }
    Catch ex {
        Set sc=ex.AsStatus()
    }
    Return sc
}
```

The method GetMailBoxStatus was overridden to use the select method from imaplib to specify which mailbox to access.

```
Method GetMailBoxStatus(ByRef NumberOfMessages As %Integer, ByRef NumberOfBytes As %I
nteger) As %Status
{
    Set sc = $$$OK
    Try {
        Do ..CheckConnection()
```

```
        Set resp = ..client.select(..MailboxName)
        Set ackToken = resp."__getitem__"(0)
        Set dataArray = resp."__getitem__"(1)
        Set NumberOfMessages = dataArray."__getitem__"(0)
        Set NumberOfBytes = -1
    }
    Catch ex {
        Set sc=ex.AsStatus()
    }
    Return sc
}
```

Note that this method returns a tuple, so the special method getitem allows us to retrieve information. Also, remember that a tuple can store another tuple, so we can recursively use getitem.

The following method overridden was GetSizeOfMessages. This method now uses the select method to choose the current mailbox and the fetch method to get the size of the message stored in the MessageNumber parameter.

```
Method GetSizeOfMessages(MessageNumber As %String = "", ByRef ListOfSizes As %ArrayOf
DataTypes) As %Status
{
    Set sc = $$$OK
    Try {
        Do ..CheckConnection()
        // select the mailbox
        Set resp = ..client.select(..MailboxName)
        // hack to ensure that MessageNumber is of type %String
        Set MessageNumber = MessageNumber_""
        Set resp = ..client.fetch(MessageNumber, "(RFC822.SIZE)")
        Set ackToken = resp."__getitem__"(0)
        Set dataArray = resp."__getitem__"(1)
        Set:('$ISOBJECT($Get(ListOfSizes))) ListOfSizes = ##class(%ArrayOfDataTypes).
%New()
        Set data = dataArray."__getitem__"(0)
        Set msgIdx = +$Piece(data, " ", 1)
        Set size = +$Piece(data, " ", 3)
        Do ListOfSizes.SetAt(size, msgIdx)
    }
    Catch ex {
        Set sc=ex.AsStatus()    }
    Return sc
}
```

We override the GetMessageUIDArray method in the same way to use the fetch method, but now we use it to get the UID codes:

```
Method GetMessageUIDArray(MessageNumber As %String = "", ByRef ListOfUniqueIDs As %Ar
rayOfDataTypes) As %Status
{
    Set sc = $$$OK
    Try {
        Do ..CheckConnection()
        // select the mailbox
        Set resp = ..client.select(..MailboxName)
        Set mailboxSize = resp."__getitem__"(1)."__getitem__"(0)
```

```
        If (mailboxSize > 0) {
            // hack to ensure that MessageNumber is of type %String
            Set MessageNumber = MessageNumber_""
            // then get the mailbox UIDs
            Set param = $CASE(MessageNumber, "":"1:*", :MessageNumber)
            Set resp = ..client.fetch(param, "UID")
            Set ackToken = resp."__getitem__"(0)
            Set dataArray = resp."__getitem__"(1)
            Set len = dataArray."__len__"()
        } Else {
            Set len = 0
        }

        Set:('$ISOBJECT($Get(ListOfUniqueIDs))) ListOfUniqueIDs = ##class(%ArrayOfDat
aTypes).%New(len)
        For i = 1:1:len {
            Set data = dataArray."__getitem__"(i - 1)
            Set msgIdx = +$Piece(data, " ", 1)
            Set size = +$Piece(data, " ", 3)
            Do ListOfUniqueIDs.SetAt(size, msgIdx)
        }
    }
    Catch ex {
        Set sc=ex.AsStatus()
    }
    Return sc
}
```

Note the use of the getitem and len methods to iterate over the tuples in the dataArray variable:

```
…
        Set len = dataArray."__len__"()
…
        For i = 1:1:len {
            Set data = dataArray."__getitem__"(i - 1)
            Set msgIdx = +$Piece(data, " ", 1)
            Set size = +$Piece(data, " ", 3)
            Do ListOfUniqueIDs.SetAt(size, msgIdx)
        }
```

Next, we override the Fetch method, which we use to retrieve the whole message body:

```
Method Fetch(MessageNumber As %Integer, ByRef Msg As %Net.MailMessage, Delete As %Boo
lean, messageStream As %BinaryStream) As %Status
{
    Set sc = $$$OK
    Try {
        Do ..CheckConnection()
        // select the mailbox
        Set resp = ..client.select(..MailboxName)
        // hack to ensure that MessageNumber is of type %String
        Set MessageNumber = MessageNumber_""
        // get the whole message
        Set resp = ..client.fetch(MessageNumber, "BODY.PEEK[]")
        Set rawMsg = ..TransversePythonArray(resp."__getitem__"(1))
```

```
        ...
    }
    Catch ex {
        Set sc=ex.AsStatus()
    }
    Return sc
}
```

Note the presence of the method TransversePythonArray. Because the message body returned by the fetch method is a composition of collections, we created this method to recursively transverse this collection and flatten it into a single string.

```
ClassMethod TransversePythonArray(pArray As %SYS.Python) As %String
{
    Set acc = ""
    If ($IsObject(pArray)) {
        Set len = pArray."__len__"()
        For i = 1:1:len {
            Set item = pArray."__getitem__"(i - 1)
            If ($IsObject(item)) {
                Set acc = acc_..TransversePythonArray(item)
            } Else {
                Set acc = acc_item
            }
            Set acc = acc_$Char(13, 10)
        }
    } Else {
        Set acc = pArray_$Char(13, 10)
    }
    Return acc
}
```

We also override the Ping method to use the imaplib noop method.

```
Method Ping() As %Status
{
    Set sc = $$$OK
    Try {
        Do ..CheckConnection()
        Set resp = ..client.noop()
    }
    Catch ex {
        Set sc=ex.AsStatus()
    }
    Return sc
}
```

The last method overridden was the CommitMarkedAsDeleted method. It now uses the methods store and expunge to mark messages for deletion and to commit such operations.

```
Method CommitMarkedAsDeleted() As %Status [ Internal, Private ]
{
```

```
    Set sc = $$$OK
    Try {
        Do ..CheckConnection()
        // select the mailbox
        Set resp = ..client.select(..MailboxName)
        // transverse array in inverse order to keep numbers integrity,
        // that is, ensures that when the number is deleted no other
        // message can assume such number
        Set messageNumber = $Order(..MarkedAsDeleted(""), -1)
        While (messageNumber '= "") {
            // hack to ensure that messageNumber is of type %String
            Set messageNumber = messageNumber_""
            Set resp = ..client.store(messageNumber, "+FLAGS", "\Deleted")
            Set messageNumber = $Order(..MarkedAsDeleted(messageNumber), -1)
        }
        Kill ..MarkedAsDeleted

        Set resp = ..client.expunge()
    }
    Catch ex {
        Set sc=ex.AsStatus()
    }
    Return sc
}
```

Conclusion

This method is much easier to implement compared to the original one, where we had to implement each IMAP command manually using IRIS TCP commands. Now that you ' ve seen a good example of how we can use the rich Python library ecosystem for real-world problems, start powering up your ObjectScript applications!

References

- imaplib — IMAP4 protocol client
- Video: Embedded Python in InterSystems IRIS: Sneak Peek
- Embedded Python: Bring the Python Ecosystem to Your ObjectScript App
- Learn Python Network Programming: Python - IMAP
- Python Documentation: Recipes and Best Practices
- Python Documentation: Data Model
- WebSocket Client with Embedded Python
- InterSystems Developer Community: #Python

#Embedded Python #Interoperability #Python #InterSystems IRIS