A Python extension for InterSystems M/Caché/IRIS and YottaDB

mg_python

M/Gateway Developments Ltd.
Chris Munt

Revision History:
4 December 2019
17 January 2020

Contents

1 Introduction

mg_python is an Open Source Python extension providing direct access to InterSystems **Caché**, **IRIS** and the **YottaDB** database. It will also work with other M-like databases.

Although this document refers to ***InterSystems Caché*** throughout, it can be assumed that the same applies to ***InterSystems IRIS***.

2 Pre-requisites

It is assumed that you have the following three components already installed:

- Python http://www.python.org

Either:

- InterSystems Caché or IRIS http://www.intersystems.com

Or:

- YottaDB https://www.yottadb.com
- A Web Server (if Python is to be used for developing web applications).

3 Installing mg_python

There are three parts to **mg_python** installation and configuration.

- The Python extension (**mg_python.pyd**).
- The database (or server) side code: **zmgsi.**
- A network configuration to bind the former two elements together.

3.1 InterSystems Caché or IRIS

Log in to the Manager UCI and install the **zmgsi** routines held in either **/m/zmgsi_cache.xml** or **/m/zmgsi_iris.xml** as appropriate.

```
do $system.OBJ.Load("/m/zmgsi_cache.xml","ck")
```

Change to your development UCI and check the installation:

```
do ^%zmgsi

M/Gateway Developments Ltd - Service Integration Gateway
Version: 3.0; Revision 1 (13 June 2019)
```

3.2 YottaDB

The instructions given here assume a standard 'out of the box' installation of **YottaDB** deployed in the following location:

```
/usr/local/lib/yottadb/r122
```

The primary default location for routines:

```
/root/.yottadb/r1.22_x86_64/r
```

Copy all the routines (i.e. all files with an 'm' extension) held in the GitHub **/yottadb** directory to:

```
/root/.yottadb/r1.22_x86_64/r
```

Change directory to the following location and start a **YottaDB** command shell:

```
cd /usr/local/lib/yottadb/r122
./ydb
```

Check the installation:

```
do ^%zmgsi

M/Gateway Developments Ltd - Service Integration Gateway
Version: 3.0; Revision 1 (13 June 2019)
```

Note that the version of **zmgsi** is successfully displayed.

3.3 Setting up the network service (if required)

The network setup described here is only required if TCP based connectivity is to be used to connect your Python code to the database, as opposed to the API based approach described in section 5.1.

The default TCP server port for **zmgsi** is **7041**. If you wish to use an alternative port then modify the following instructions accordingly. Python code using the **mg_python** functions will, by default, expect the database server to be listening on port **7041** of the local server (localhost). However, **mg_python** provides the functionality to modify these default settings at run-time. It is not necessary for the web server/Python installation to reside on the same host as the database server.

3.3.1 InterSystems Caché or IRIS

Start the M-hosted concurrent TCP service in the Manager UCI:

```
do start^%zmgsi(0)
```

To use a server TCP port other than 7041, specify it in the start-up command (as opposed to using zero to indicate the default port of 7041).

3.3.2 YottaDB

Network connectivity to **YottaDB** is managed via the **xinetd** service. First create the following launch script (called **zmgsi_ydb** here):

```
/usr/local/lib/yottadb/r122/zmgsi_ydb
```

Content:

```
#!/bin/bash
cd /usr/local/lib/yottadb/r122
export ydb_dir=/root/.yottadb
export ydb_dist=/usr/local/lib/yottadb/r122
export
ydb_routines="/root/.yottadb/r1.22_x86_64/o*(/root/.yottadb/r1.22_x86_6
4/r /root/.yottadb/r) /usr/local/lib/yottadb/r122/libyottadbutil.so"
export ydb_gbldir="/root/.yottadb/r1.22_x86_64/g/yottadb.gld"
$ydb_dist/ydb -r xinetd^%zmgsis
```

Create the **xinetd** script (called **zmgsi_xinetd** here):

```
/etc/xinetd.d/zmgsi_xinetd
```

Content:

```
service zmgsi_xinetd
{
     disable         = no
     type            = UNLISTED
     port            = 7041
     socket_type     = stream
     wait            = no
     user            = root
     server          = /usr/local/lib/yottadb/r122/zmgsi_ydb
}
```

- Note: sample copies of **zmgsi_xinetd** and **zmgsi_ydb** are included in the /unix directory.

Edit the services file:

```
/etc/services
```

Add the following line to this file:

```
zmgsi_xinetd          7041/tcp                        # ZMGSI
```

Finally restart the **xinetd** service:

```
/etc/init.d/xinetd restart
```

3.4 Installing and using the mg_python component

Either build the Python component (mg_python.pyd) from source (/c/mg_python.c) or use a pre-build module from the distribution (if available for your platform).

Windows:

Copy this module to the 'dlls' directory which should be directly below the Python root directory. For example:

```
C:\Python27\dlls\
```

UNIX:

Copy this module to the 'site-packages' directory which should be below the Python lib/root directory. For example:

```
/usr/local/lib/python2.7/site-packages
```

Having done this, Python programs may refer to, and load, the **mg_python** module using the following directive at the top of the script.

```
import mg_python
```

Having added this line, all methods listed provided by the module can be invoked using the following syntax.

```
mg_python.<method>
```

Alternatively, an alias can be assigned to the module name. For example:

```
import mg_python as <alias>
```

Then methods can be invoked as:

```
<alias>.<method>
```

For web development, the distribution also contains the following XMLHTTP script file:

```
/usr/mgwsi/java/mg_client.js
```

Copy the XMLHTTP file into your applications root directory. For example:

```
C:\Inetpub\wwwroot\
```

4 Python Lists and M/Caché Globals

The M/Caché database is made up of any number of *global variables*. Each global variable (or *global*) roughly equates to a table in a relational database. The data within each global is organized as a B-Tree structure. The key to each *global node* (or *record*) can be divided up into any number of individual sub-key items (or *subscripts*). The ability to divide-up the key to a global node in this manner gives the Caché database its *multidimensional* characteristics.

This section will describe the conventions used within **mg_python** to map Caché globals on to simple numerically indexed Python arrays or *List*s as they are known within Python. Two data constructs are used to represent Caché data.

1. The key (or subscripts) to a global node are held in simple numerically indexed Python Lists. By convention, position zero in the key list contains the number of subscripts.

2. Sets of global nodes (or records) are held in Python Lists. With Lists, the *len* function can be used to determine the number of records held. Within individual List records, the Caché global subscripts and data are concatenated together (in that order) to create a compound data record containing all the individual parts. The **mg_python** module provides methods to create and manage these data constructs. The *m_local* suite of methods are used to maintain locally held sets of M/Caché records.

5 Method Reference for mg_python

The methods provided by the core **mg_python** component allow you to directly manipulate the M/Caché database from within the Python scripting environment. Methods are also supplied to allow you to directly call M/Caché extrinsic functions (and methods) and M/Caché procedures that are capable of generating sections of HTML form data.

All **mg_python** methods are implemented according to the following patterns.

- The first argument to most methods is a page context or server **'handle'** variable. In most cases this can be specified as zero. It is only necessary to create a customized context if alternative parameters are required for your calls (e.g. Values for M/Caché server and/or NameSpace and/or storage method which are different to those defined in the default profile).

- Methods working to local data (ma_local_* methods): The second argument is always the name of a records List. Further arguments represent subscripts and data for a record.

- Methods working directly to M/Caché: The second argument is always the name of either a M/Caché global or a function/procedure depending on the nature of the operation. Further arguments are either passed as subscripts to a global or arguments to a Caché function (or method), depending on the context.

5.1 Connecting to the database

By default, and assuming TCP based connectivity is used, the **mg_python** methods will address M/Caché host '**localhost'** listening on TCP port **7041**. The default handle is always zero. The following methods allow you to redefine the default host, and to address hosts other than the default.

5.1.1 Modifying the default M/Caché Host

```
mg_python.m_set_host(handle, netname, username, password)
```

In addition to addressing the default host associated with the default handle (zero), other M/Caché servers can be specified on a per-call basis as shown in this document's many examples. You can specify the M/Caché host to be used within an instance of the **mg_python** module using the '**m_set_host'** method.

For example:

```
handle = mg_python.m_allocate_server_handle()
mg_python.m_set_host(handle, "MyCachéServer", 7041, "", "")

# Process calls on MyCachéServer

mg_python.m_release_server_handle(handle)
```

All **mg_python** method calls in the page will now target M/Caché server: `MyCachéServer`.

Scope

The name of the host is scoped in accordance with the instance of the **mg_python** module and server handle in use.

5.1.2 Modifying the default M/Caché NameSpace

```
mg_python.m_set_uci(handle, uci)
```

This method will change the Namespace associated with the current server handle.

For example:

```
handle = mg_python.m_set_uci(handle, "USER")
```

5.1.3 Non-network based access to the database via its API

As an alternative to connecting to the database using TCP based connectivity, **mg_python** provides the option of high-performance embedded access to a local installation of the database via its API.

5.1.3.1 InterSystems Caché or IRIS.

Use the following functions to bind to the database API.

```
mg_python.m_set_uci(handle, namespace)
mg_python.m_bind_server_api(handle, dbtype, path, username, password,
                            envvars, params)
```

Where:

```
handle:     Current server handle.
namespace:  Namespace.
```

```
dbtype:     Database type ('Cache' or 'IRIS').
path:       Path to database manager directory.
username:   Database username.
password:   Database password.
envvars:    List of required environment variables.
params:     Reserved for future use.
```

Example:

```
mg_python.m_set_uci(0, "USER")
result = mg_python.m_bind_server_api(0, "IRIS", "/usr/iris20191/mgr",
                                     "_SYSTEM", "SYS", "", "")
```

The bind function will return '1' for success and '0' for failure.

Before leaving your Python application, it is good practice to gracefully release the binding to the database:

```
mg_python.m_release_server_api(handle)
```

Example:

```
mg_python.m_release_server_api(0)
```

5.1.3.2 YottaDB

Use the following function to bind to the database API.

```
mg_python.m_bind_server_api(handle, dbtype, path, username, password,
                            envvars, params)
```

Where:

```
handle:     Current server handle.
dbtype:     Database type ('YottaDB').
path:       Path to the YottaDB installation/library.
username:   Database username.
password:   Database password.
envvars:    List of required environment variables.
params:     Reserved for future use.
```

Example:

This assumes that the YottaDB installation is in: `/usr/local/lib/yottadb/r122`
This is where the **libyottadb.so** library is found.
Also, in this directory, as indicated in the environment variables, the YottaDB routine interface file resides (zmgsi.ci in this example). The interface file must contain the following line:

```
ifc_zmgsis: ydb_char_t * ifc^%zmgsis(I:ydb_char_t*, I:ydb_char_t *, I:ydb_char_t*)
```

Moving on to the Python code for binding to the YottaDB database. Modify the values of these environment variables in accordance with your own YottaDB installation. Note that each line is terminated with a linefeed character, with a double linefeed at the end of the list.

```
envvars = "";
envvars = envvars + "ydb_dir=/root/.yottadb\n"
envvars = envvars + "ydb_rel=r1.22_x86_64\n"
envvars = envvars + "ydb_gbldir=/root/.yottadb/r1.22_x86_64/g/yottadb.gld\n"
envvars =envvars +
"ydb_routines=/root/.yottadb/r1.22_x86_64/o*(/root/.yottadb/r1.22_x86_64/r
/root/.yottadb/r) /usr/local/lib/yottadb/r122/libyottadbutil.so\n"
envvars = envvars + "ydb_ci=/usr/local/lib/yottadb/r122/zmgsi.ci\n"
envvars = envvars + "\n"

result = mg_python.m_bind_server_api(0, "YottaDB",
                                     "/usr/local/lib/yottadb/r122",
                                     "", "", envvars, "")
```

The bind function will return '1' for success and '0' for failure.

Before leaving your Python application, it is good practice to gracefully release the binding to the database:

```
mg_python.m_release_server_api(handle)
```

Example:

```
mg_python.m_release_server_api(0)
```

5.2 Direct access to the M/Caché database

The **mg_python** methods in this section give you direct access to the M/Caché commands for manipulating global data. A M/Caché global is essentially a multi-dimensional associative array that's held in permanent storage.

There are two forms for each of these methods: those that express a global reference as a variable number of input arguments are prefixed by 'm_' and those for which the keys are expressed as a Python list are prefixed by 'ma_'.

5.2.1 Set a global node.

```
result = mg_python.m_set(<handle>, <global>, <keys…>, <data>)
result = mg_python.ma_set(<handle>, <global>, <keylist>, <data>)
```

Types:

```
handle            (int)
```

```
result            (String)
global            (String)
keys              (Any)
keylist           (List)
data              (String)
```

By convention, the last argument is always the global node's data record.

Example 1:

M/Caché command:

```
Set ^Customer(1234)="Chris Munt"
```

Equivalent **mg_python** methods:

```
mg_python.m_set(0, "^Customer", 1234, "Chris Munt")

Or:

key = [1, "1234"]
mg_python.ma_set(0, "^Customer", key, "Chris Munt")
```

5.2.2 Retrieve the data from a global node.

```
data = mg_python.m_get(<handle>, <global>, <keys…>)
data = mg_python.ma_get(<handle>, <global>, <keylist>)
```

Types:

```
handle            (int)
data              (String)
global            (String)
keys              (Any)
keylist           (List)
```

Example 1:

M/Caché command:

```
Set data=$Get(^Customer(1234))
```

Equivalent **mg_python** methods:

```
data = mg_python.m_get(0, "^Customer", 1234)

Or:

key = [1, "1234"]
data = mg_python.ma_get(0, "^Customer", key)
```

5.2.3 Check that a global node exists.

```
defined = mg_python.m_defined(<handle>, <global>, <keys…>)
defined = mg_python.ma_defined(<handle>, <global>, <keylist>)
```

Types:

```
handle            (int)
defined           (String)
global            (String)
keys              (Any)
keylist           (List)
```

Example 1:

M/Caché command:

```
Set defined=$Data(^Customer(1234))
```

Equivalent **mg_python** methods:

```
defined = mg_python.m_defined(0, "^Customer", 1234)
```

Or:

```
key = [1, "1234"]
defined = mg_python.ma_defined(0, "^Customer", key)
```

5.2.4 Delete a global node.

```
result = mg_python.m_delete(<handle>, <global>, <keys…>)
result = mg_python.ma_delete(<handle>, <global>, <keylist>)
```

Types:

```
handle            (int)
result            (String)
global            (String)
keys              (Any)
keylist           (List)
```

Example 1:

M/Caché command:

```
Kill ^Customer(1234))
```

Equivalent **mg_python** method:

```
mg_python.m_delete(0, "^Customer", 1234)
```

Or:

```
key = [1, "1234"]
mg_python.ma_delete(0, "^Customer", key)
```

5.2.5 Get the next key value for a global node.

```
next = mg_python.m_order(<handle>, <global>, <keys…>)
next = mg_python.ma_order(<handle>, <global>, <keylist>)
```

Types:

```
handle            (int)
next              (String)
global            (String)
keys              (Any)
keylist           (List)
```

Example 1:

M/Caché command:

```
Set nextID=$Order(^Customer(1234))
```

Equivalent **mg_python** methods:

```
nextID = mg_python.m_order(0, "^Customer", 1234)
```

Or:

```
key = [1, "1234"]
nextID = mg_python.ma_order(0, "^Customer", key)
```

Example 2 (Parse a global in order):

M/Caché command:

```
Set nextID=""
For  Set nextID=$Order(^Customer(1234)) Quit:nextID=""  Do
. Write "<br>", nextID," = ",$Get(^Customer(nextID))
. Quit
```

Equivalent **mg_python** method:

```
key = mg_python.m_order(0, "^Customer", "")
while (key != ""):
   print(key, " = ", mg_python.m_get(0, "^Customer", key))
```

```
   key = mg_python.m_order(0, "^Customer", key)
```

Or:

```
key = [1, “”]
while (mg_python.ma_order(0, "^Customer", key) <> “”):
   print(key[1], " = ", mg_python.ma_get(0, "^Customer", key))
```

5.2.6 Get the previous key value for a global node.

```
prev = mg_python.m_previous(<handle>, <global>, <keys…>)
prev = mg_python.ma_previous(<handle>, <global>, <keylist>)
```

Types:

```
handle            (int)
prev              (String)
global            (String)
keys              (Any)
keylist           (List)
```

Example 1:

M/Caché command:

```
Set prevID=$Order(^Customer(1234),-1)
```

Equivalent **mg_python** methods:

```
prevID = mg_python.ma_previous(0, “^Customer”, 1234)
```

Or:

```
key = [1, "1234"]
prevID = mg_python.ma_previous(0, “^Customer”, key)
```

Example 2 (Parse a global in reverse order):

M/Caché command:

```
Set nextID=””
For  Set nextID=$Order(^Customer(1234), -1) Quit:nextID=””  Do
. Write "<br>", nextID," = ",$Get(^Customer(nextID))
. Quit
```

Equivalent **mg_python** method:

```
key = mg_python.m_previous(0, "^Customer", "")
while (key != ""):
```

```
   print(key, " = ", mg_python.m_get(0, "^Customer", key))
   key = mg_python.m_previous(0, "^Customer", key)
```

Or:

```
key = [1, ""]
while (mg_python.ma_previous(0, "^Customer", key) <> ""):
   print(key[1], " = ", mg_python.ma_get(0, "^Customer", key))
}
```

5.3 Manipulate the local representation of M/Caché data in Python

The **mg_python** methods in this section create and manipulate the local Python data structures that represent M/Caché data in the Python environment. These (ma_local) methods will be used extensively in the examples shown in subsequent sections.

5.3.1 Set a local node.

```
result = mg_python.ma_local_set(<handle>, <records>, <index>,
                                <keylist>,
                                <data>)
```

Types:

```
handle            (int)
result            (String)
records           (List)
index             (int)
keylist           (List)
data              (String)
```

The index argument can take one of the following values:

```
Positive integer:
      The index number in the records List (if known).
-1:
      Instruct the method to assign the next available index number
      or overwrite the existing key value.
-2:
      Instruct the method to assign the next available index number
      at the end of the record set regardless of whether or
      not the key value exists.
```

It should be noted that this method will not automatically insert records into the set in M/Caché-order. Use the 'ma_local_sort' method to order the contents of a records List.

Example 1:

M/Caché command:

```
Set Customer(1234)=”Chris Munt”
```

Equivalent **mg_python** method:

```
key = [1, "1234"]
mg_python.ma_local_set(0, Customer, -1, key, “Chris Munt”)
```

5.3.2 Retrieve the data from a local node.

```
data = mg_python.ma_local_get(<handle>, <records>, <index>,
                              <keylist>)
```

Types:

```
handle            (int)
data              (String)
records           (List)
index             (int)
keylist           (List)
```

The index argument can take one of the following values:

```
Positive integer:
      The index number in the records List (if known).
-1:
      Instruct the method to locate the record required based
      on the key value supplied.
```

Example 1:

M/Caché command:

```
Set data=$Get(Customer(1234))
```

Equivalent **mg_python** method:

```
key = [1, "1234"]
data = mg_python.ma_local_get(0, Customer, -1, key)
```

5.3.3 Check that a local node exists.

```
defined = mg_python.ma_local_data(<handle>, <records>, <index>,
                                  <keylist>)
```

Types:

```
handle            (int)
defined           (String)
records           (List)
index             (int)
keylist           (List)
```

The index argument can take one of the following values:

```
Positive integer:
      The index number in the records List (if known).
```

```
-1:
      Instruct the method to locate the record required based
      on the key value supplied.
```

Example 1:

M/Caché command:

```
Set defined=$Data(Customer(1234))
```

Equivalent **mg_python** method:

```
key = [1, "1234"]
defined = mg_python.ma_local_data(0, Customer, -1, key)
```

5.3.4 Delete a local node.

```
result = mg_python.ma_local_kill(<handle>, <records>, <index>,
                                 <keylist>)
```

Types:

```
handle            (int)
result            (String)
records           (List)
index             (int)
keylist           (List)
```

The index argument can take one of the following values:

```
Positive integer:
      The index number in the records List (if known).
-1:
      Instruct the method to locate the record required based
      on the key value supplied.
```

Example 1:

M/Caché command:

```
Kill Customer(1234))
```

Equivalent **mg_python** method:

```
key = [1, "1234"]
result = mg_python.ma_local_kill(0, Customer, -1, key)
```

5.3.5 Get the next key value for a local node.

```
next = mg_python.ma_local_order(<handle>, <records>, <index>,
                                <keylist>)
```

Types:

```
handle            (int)
next              (String)
records           (List)
index             (int)
keylist           (List)
```

The index argument can take one of the following values:

```
Positive integer:
      The current index number in the records List (if known).
-1:
      Instruct the method to locate the next record based
      on the current key value supplied.
```

Example 1:

M/Caché command:

```
Set nextID=$Order(Customer(1234))
```

Equivalent **mg_python** method :

```
key = [1, "1234"]
nextID = mg_python.ma_local_order(0, Customer, -1, key)
```

Example 2 (Parse a local record set in order):

M/Caché command:

```
Set nextID=""
For  Set nextID=$Order(Customer(1234)) Quit:nextID=""  Do
. Write "<br>", nextID
. Quit
```

Equivalent **mg_python** method:

```
key = [1, ""]
while (mg_python.ma_local_order(0, Customer, -1, key) <> -1):
   print '<br>', key[1]
```

5.3.6 Get the previous key value for a local node.

```
prev = mg_python.ma_local_get(<handle>, <records>, <index>,
                              <keylist>)
```

Types:

```
handle            (int)
prev              (String)
records           (List: Use List or Vector)
index             (int)
keylist           (List)
```

The index argument can take one of the following values:

```
Positive integer:
      The current index number in the records List (if known).
-1:
      Instruct the method to locate the next record based
      on the current key value supplied.
```

Example 1:

M/Caché command:

```
Set prevID=$Order(Customer(1234), -1)
```

Equivalent **mg_python** method:

```
key = [1, "1234"]
prevID = mg_python.ma_local_previous(0, Customer, -1, ref key)
```

Example 2 (Parse a local record set in reverse order):

M/Caché command:

```
Set prevID=""
For  Set prevID=$Order(Customer(1234), -1) Quit:prevID=""  Do
. Write "<br>", prevID
. Quit
```

Equivalent **mg_python** method:

```
key = [1, ""]
while (mg_python.ma_local_previous(0, Customer, -1, key) <> -1):
   print '<br>', key[1]
```

5.3.7 Sort the contents of a local records set.

```
data = mg_python.ma_local_sort(<handle>, <records>)
```

Types:

```
handle            (int)
data              (String)
records           (List)
```

This method will sort the contents of a records List into M/Caché order. The default M/Caché server will be used to perform the sort operation.

Example 1:

mg_python method:

```
result = mg_python.ma_local_sort(0, records)
```

5.3.8 Merge a Python array to a global.

```
result = mg_python.ma_merge_to_db(<handle>, <global>, <keylist>,
                                  <records>, <options>)
```

Types:

```
handle            (int)
result            (String)
global            (String)
keylist           (List)
records           (List)
options           (String)
```

The 'options' argument can currently take the following value:

```
ks
```

This means '**K**ill at **S**erver'. If this option is selected, the M/Caché global will be deleted at the level specified within the merge function before the actual merge is performed.

Example 1:

M/Caché commands:

```
Set custList(1234)="Chris Munt"
Set custList(1235)="Rob Tweed"
```

```
Merge ^Customer=custList
```

Equivalent **mg_python** method:

```
custList = [] # Clear records List
key = [1, "1234"]
mg_python.ma_local_set(0, custList, -1, key, "Chris Munt")
key = [1, "1235"]
mg_python.ma_local_set(0, custList, -1, key, "Rob Tweed")

key = [0] # No fixed key for merge global
result = mg_python.ma_merge_to_db(0, "^Customer", key, custList, "")
```

In both cases the following records will be created in M/Caché:

```
^Customer(1234)="Chris Munt"
^Customer(1235)="Rob Tweed"
```

Example 2:

M/Caché commands:

```
Set custInvoice(1)="1.2.2001"
Set custInvoice(2)="5.3.2001"
Merge ^CustomerInvoice(1234)=custInvoice
```

Equivalent **mg_python** method:

```
custInvoice = [] # Clear records List
key = [1, "1"]
mg_python.ma_local_set(0, custInvoice, -1, key, "1.2.2001")
key = [1, "2"]
mg_python.ma_local_set(0, custInvoice, -1, key, "5.3.2001")

key = [1, "1234"] # Fixed key for merge global
result = mg_python.ma_merge_to_db(0, "^CustomerInvoice", key,
                                  custInvoice, "");
```

In both cases, the following records will be created in M/Caché:

```
^CustomerInvoice(1234,1)="1.2.2001"
^CustomerInvoice(1234,2)="5.3.2001"
```

Example 3 (Using the 'ks' option):

Existing M/Caché database:

```
^CustomerInvoice(1234,1)="1.2.2001"
^CustomerInvoice(1234,2)="5.3.2001”
^CustomerInvoice(1234,3)="7.9.2001”
```

M/Caché commands:

```
Set custInvoice(3)=”8.9.2001”
Set custInvoice(4)=”12.11.2001”
Kill ^CustomerInvoice(1234)

Merge ^CustomerInvoice(1234)=custInvoice
```

Equivalent **mg_python** method:

```
custInvoice = [] # Clear records List
key = [1, "3"]
mg_python.ma_local_set(0, custInvoice, -1, key, "8.9.2001")
key = [1, "4"]
mg_python.ma_local_set(0, custInvoice, -1, key, "12.11.2001")

key = [1, "1234"] # Fixed key for merge global
result = mg_python.ma_merge_to_db(0, "^CustomerInvoice", key,
                                  custInvoice, "ks");
```

In both cases, after this operation, M/Caché will hold the following records:

```
^CustomerInvoice(1234,3)="8.9.2001"
^CustomerInvoice(1234,4)="12.11.2001”
```

Example 4 (Dealing with multi-dimensional arrays):

M/Caché commands:

```
Set custInvoice(1234,1)=”1.2.2001”
Set custInvoice(1234,2)=”5.3.2001”
Set custInvoice(1235,7)=”7.6.2002”
Set custInvoice(1235,8)=”1.12.2002”
Merge ^CustomerInvoice=custInvoice
```

Equivalent **mg_python** method:

```
custInvoice = [] # Clear records List
key = [2, "1234", “1”]
mg_python.ma_local_set(0, custInvoice, -1, key, "1.2.2001")
key = [2, "1234", “2”]
mg_python.ma_local_set(0, custInvoice, -1, key, "5.3.2001")
key = [2, "1235", “7”]
mg_python.ma_local_set(0, custInvoice, -1, key, "7.6.2002")
key = [2, "1235", “8”]
```

```
mg_python.ma_local_set(0, custInvoice, -1, key, "1.12.2002")

key = [0] # No fixed key for merge global
result = mg_python.ma_merge_to_db(0, "^CustomerInvoice", key,
                                  custInvoice, "");
```

In both cases, after this operation, M/Caché will hold the following records:

```
^CustomerInvoice(1234,1)=”1.2.2001”
^CustomerInvoice(1234,2)=”5.3.2001”
^CustomerInvoice(1235,7)=”7.6.2002”
^CustomerInvoice(1235,8)=”1.12.2002”
```

5.3.9 Merge a global to a Python array.

```
result = mg_python.ma_merge_from_db(<handle>, <global>, <keylist>,
                                    <records>, <options>)
```

Types:

```
handle            (int)
result            (String)
global            (String)
keylist           (List)
records           (List)
options           (String)
```

The last ‘options’ argument is reserved for future use.

Example 1:

M/Caché database:

```
^Customer("1234")="Chris Munt"
^Customer("1235")="Rob Tweed”
```

M/Caché command:

```
Merge custList=^Customer
```

Equivalent **mg_python** method:

```
custList = [] # Clear records List

key = [0] # No fixed key for merge global
result = mg_python.ma_merge_from_db(0, "^Customer", key, custList, “”)
```

In both cases, the following records will be created in Python:

```
custList[1] = “1234” + “Chris Munt”
custList[2] = “1235” + “Rob Tweed”
```

To get the number of records:

```
custList.count
```

Example 2:

M/Caché database:

```
^CustomerInvoice(1234,1)="1.2.2001"
^CustomerInvoice(1234,2)="5.3.2001”
```

M/Caché commands:

```
Merge custInvoice=^CustomerInvoice(1234)
```

Equivalent **mg_python** method:

```
custInvoice = [] # Clear records List

key = [1, "1234"] # Fixed key for merge global
result = mg_python.ma_merge_from_db(0, "^Customer", key,
                                    custInvoice, “”)
```

In both cases, the following records will be created in Python:

```
custInvoice[1] = "1" + "1.2.2001"
custInvoice[2] = "2" + "5.3.2001”
```

Example 3 (Dealing with multi-dimensional arrays):

M/Caché database:

```
^CustomerInvoice(1234,1)=”1.2.2001”
^CustomerInvoice(1234,2)=”5.3.2001”
^CustomerInvoice(1235,7)=”7.6.2002”
^CustomerInvoice(1235,8)=”1.12.2002”
```

M/Caché commands:

```
Merge custInvoice=^CustomerInvoice
```

Equivalent **mg_python** method:

```
custInvoice = [] # Clear records List

key = [0] # No fixed key for merge global
result = mg_python.ma_merge_from_db(0, "^CustomerInvoice", key,
                                    custInvoice, “”)
```

In both cases, the following records will be created in Python:

```
custInvoice[1] = “1234” + “1” + ”1.2.2001”
custInvoice[2] = “1234” + “2” + ”5.3.2001”
custInvoice[3] = “1235” + “7” + ”7.6.2002”
custInvoice[4] = “1235” + ”8” + ”1.12.2002”
```

5.4 Direct access to M/Caché functions and procedures

M/Caché provides a rich scripting language (formally known as MUMPS or M) for developing function and procedures. The following methods are supplied by the mg_python module for the purpose of directly accessing M/Caché functions and procedures within the Python environment.

A note on terminology: M/Caché procedures and functions are contained within blocks of code known as routines. A function or procedure is referred to using the following syntax:

```
<FunctionName>^<Routine>
```

For example:

```
MyFunction^MyRoutine
```

A routine name of 'MyRoutine' will be used throughout the following examples.

Input arguments to M/Caché functions can be expressed as a variable number of input arguments for which the Python method is prefixed by 'm_'. Input arguments to M/Caché functions can also be supplied as Python lists in which case the Python method is prefixed by 'ma_'. The latter form must be used if arguments are to be passed by reference (i.e. M/Caché will be modifying their values).

5.4.1 Call a M/Caché extrinsic function.

```
result = mg_python.m_function(<handle>, <function>, <arguments…>)

result = mg_python.ma_function(<handle>, <function>, <arguments>,
                               <no_arguments>)
```

Types:

```
handle            (int)
result            (String)
function          (String)
arguments         (Any)
argumentlist      (List)
no_arguments      (int)
```

The following methods are used to create arguments to a function call.

Add an item to the arguments list:

```
mg_python.ma_arg_set(<handle> <arguments>, <arg_no>, <item>, <by_ref))
```

Where:

```
arguments   (List)
      Arguments array.  Holds physical data and associated properties.
arg_no      (int)
      Argument number.
item  (List or String)
      Argument
by_ref      (int)
      By reference flag - set to 1 for pass-by-reference; 0 for
      pass-by-value.
```

Example 1 (A simple function call):

M/Caché procedure:

```
GetTime()   ; Get the current date and time in M/Caché internal format
            Set result=$Horolog
            Quit result
            ;
```

This function returns the date and time in M/Caché's internal format.

Equivalent **mg_python** method:

```
time = mg_python.m_function(0, "GetTime^MyRoutine")
```

Or:

```
time = mg_python.ma_function(0, "GetTime^MyRoutine", arguments, 0)
```

Example 2 (Passing arguments by reference):

M/Caché procedure:

```
GetDateDecoded(dateDisp)      ; Get the current date in decoded form
                              Set result=+$Horolog
                              Set dateDisp=$ZD(result,2);
                              Quit result
                              ;
```

This function returns the date in M/Caché's internal format. In addition, it will return the date in a human-readable format (i.e. decoded).

Equivalent **mg_python** method:

```
mg_python.ma_arg_set(0, arguments, 1, "", 1);
date = mg_python.ma_function(0, "GetDateDecoded^MyRoutine",
                                arguments, 1)
dateDisp = arguments[1]
```

Notice that the 'by reference' flag is set in the 'ma_arg_set' method.

Example 3 (Another simple function call):

M/Caché procedure:

```
GetCust(custID)   ; Return the customer name
                  Set cust=$Get(^Customer(custID))
                  Quit cust
                  ;
```

Equivalent **mg_python** method:

```
cust = mg_python.m_function(0, "GetCust^MyRoutine", "1234")
```

Or:

```
mg_python.ma_arg_set(0, arguments, 1, "1234", 0);
cust = mg_python.ma_function(0, "GetCust^MyRoutine", arguments, 1)
```

Example 4 (Passing an array from Python to M/Caché):

M/Caché procedure:

```
ProcCustList(custList)  ; Return the number of active customers
                  Set custID="",activeCust=0
                  For  Set CustID=$Order(^CustList(custID)) Do
                  . If $Data(^CustOrderStatus(custID))="Active" Do
                  .. Set activeCust=activeCust+1
                  Quit activeCust
                  ;
```

Equivalent **mg_python** method:

```
custList = []
key = [1, "1234"]
mg_python.ma_local_set(0, custList, -1, key, "")
key = [1, "1235"]
mg_python.ma_local_set(0, custList, -1, key, "")
key = [1, "1236"]
mg_python.ma_local_set(0, custList, -1, key, "")

mg_python.ma_arg_set(0, arguments, 1, custList, 0);
activeCust = mg_python.ma_function(0, "ProcCustList^MyRoutine",
                                   arguments, 1)
```

Example 5 (Passing an array from M/Caché to Python):

M/Caché procedure:

```
ActCList(custList)      ; Return a list of active customers
                  Set custID="",activeCust=0
```

```
                  For  Set CustID=$Order(^CustOrderStatus(custID))
Quit:custID=””  Do
                  . If $Data(^CustOrderStatus(custID))=”Active” Do
                  .. activeCust=activeCust+1
                  .. Set custList(custID)=$Get(^Customer(custID))
                  Quit activeCust
                  ;
```

Equivalent **mg_python** method:

```
custList = []
mg_python.ma_arg_set(0, arguments, 1, custList, 1);
activeCust = mg_python.ma_function(0, “ActCList^MyRoutine”,
                                   arguments, 1)
```

Example 6 (Using a M/Caché function to modify a Python array):

M/Caché procedure:

```
GetCustNames(custList)  ; Return the names for a list of customers
                  Set custID=””,result=””
                  For  Set CustID=$Order(^CustList(custID)) Do
                  . Set custList(custID)=$Get(^Customer(custID))
                  Quit result
                  ;
```

Equivalent **mg_python** method:

```
custList = []
key = [1, "1234"]
mg_python.ma_local_set(0, custList, -1, key, "")
key = [1, "1235"]
mg_python.ma_local_set(0, custList, -1, key, "")
key = [1, "1236"]
mg_python.ma_local_set(0, custList, -1, key, "")

mg_python.ma_arg_set(0, arguments, 1, custList, 1);
activeCust = mg_python.ma_function(0, “GetCustNames^MyRoutine”,
                                   arguments, 1)
```

5.4.2 Return a block of HTML from a M/Caché function.

```
connection_handle = mg_python.ma_html(<handle>, <function>,
                                      <arguments>,
                                      <no_arguments>)

buffer = mg_python.ma_get_stream_data(<handle>, <connection_handle>)
```

Types:

```
handle            (int)
connection_handle (int)
function          (String)
function          (String)
arguments         (List)
no_arguments      (int)
```

Usage:

```
connection_handle = mg_python.ma_html_ex(<handle>, <function>,
                                         <arguments>,
                                         <no_arguments>)
buffer = mg_python.ma_get_stream_data(<handle>, <connection_handle>)
while (buffer <> ""):
   print buffer
   buffer = mg_python.ma_get_stream_data(<handle>, <connection_handle>)
```

Example 1:

M/Caché procedure:

```
MyHtml      ; Return some HTML to Python
            Write "<p>This text was returned from M/Caché"
            Write "<br>You can return as text much as you like …"
            Quit
            ;
```

Equivalent **mg_python** method:

```
connection_handle = mg_python.ma_html_ex(0, "MyHtml^MyRoutine",
                                         arguments,
                                         0);
buffer = mg_python.ma_get_stream_data(0, connection_handle)
while (buffer <> ""):
   print buffer
   buffer = mg_python.ma_get_stream_data(0, connection_handle)
```

Example 2:

M/Caché procedure:

```
GetHTML(custID)   ; Return some HTML to Python
                  Write "<p>This text was returned from M/Caché"
                  Write "<br>You can return as text much as you like …"
                  Write "<br>A single argument '",custID,"' was passed
from Python"
                  Quit
                  ;
```

Equivalent **mg_python** method:

```
mg_python.ma_arg_set(0, arguments, 1, "1234", 0)
connection_handle = mg_python.ma_html_ex(0, "MyHtml^MyRoutine",
                                         arguments,
                                         1)
buffer = mg_python.ma_get_stream_data(0, connection_handle)
while (buffer <> ""):
   print buffer
   buffer = mg_python.ma_get_stream_data(0, connection_handle)
```

Example 3 (Passing an array from Python to M/Caché):

M/Caché procedure:

```
GetCTable(custList)     ; Return a table of customers to Python
                  Write "<table>"
                  Set custID=""
                  For  Set CustID=$Order(^CustList(custID))
Quit:custID=""  Do
                  . Set custName=$Get(^Customer(custID))
                  . Set custList(custID)=custName ; Return name to
Python
                  . Write "<tr><td>",custID,"</td>"
                  . Write "<td>",custName,"</td></tr>"
                  Write "</table>"
                  Quit
                  ;
```

Equivalent **mg_python** method:

```
custList = []
key[0] = "1";
key[1] = "1234";
mg_python.ma_local_set(0, custList, -1, key, "")
```

```
key[1] = "1235";
mg_python.ma_local_set(0, custList, -1, key, "")
key[1] = "1236";
mg_python.ma_local_set(0, custList, -1, key, "")

mg_python.ma_arg_set(0, arguments, 1, custList, 0)
connection_handle = mg_python.ma_html_ex(0, "GetCTable^MyRoutine",
                                         arguments,
                                         1)
buffer = mg_python.ma_get_stream_data(0, connection_handle)
while (buffer <> ""):
   print buffer
   buffer = mg_python.ma_get_stream_data(0, connection_handle)
```

5.5 Direct access to Caché methods

Caché provides an Object Oriented development environment (Caché Objects). The following methods are supplied by the mg_python module for the purpose of directly accessing Caché class methods from within the Python environment.

This section will show the same examples used in the previous section (Caché functions and procedures), but implemented as methods of an object class.

The methods described here will belong to a class called 'MyUtilities.MyClass'. This translates to a Caché package name of 'MyUtilities' and a class name of 'MyClass'. Of course, in a real application the methods described would be contained within a class more appropriate to the functionality they implement.

Input arguments to M/Caché methods can be expressed as a variable number of input arguments for which the Python method is prefixed by 'm_'. Input arguments to M/Caché methods can also be supplied as Python lists in which case the Python method is prefixed by 'ma_'. The latter form must be used if arguments are to be passed by reference (i.e. M/Caché will be modifying their values).

5.5.1 Call a Caché ClassMethod.

```
result = mg_python.m_classmethod(<handle>, <class_name>,
                                 <method_name>,
                                 <arguments…>)

result = mg_python.ma_classmethod(<handle>, <class_name>,
                                  <method_name>,
                                  <argumentlist>, <no_arguments>)
```

Types:

```
handle            (int)
result            (String)
class_name        (String)
method_name       (String)
arguments         (Any)
argumentlist      (List)
no_arguments      (int)
```

Example 1 (A simple method):

M/Caché ClassMethod:

```
Class MyUtilities.MyClass Extends etc …
```

```
ClassMethod GetTime()
{
      ; Get the current date and time in Caché's internal format
      Set result=$Horolog
      Quit result
}
```

This method returns the date and time in Caché's internal format.

Equivalent **mg_python** method:

```
date = mg_python.m_classmethod(0, "MyUtilities.MyClass", "GetTime")

Or

date = mg_python.ma_classmethod(0, "MyUtilities.MyClass", "GetTime",
                                   arguments, 0)
```

Example 2 (Passing arguments by reference):

Caché ClassMethod:

```
Class MyUtilities.MyClass Extends etc …

ClassMethod GetDateDecoded(dateDisp)
{
      ; Get the current date in decoded form
      Set result=+$Horolog
      Set dateDisp=$ZD(result,2);
      Quit result
}
```

This method returns the date in Caché's internal format. In addition, it will return the date in a human-readable format (i.e. decoded).

Equivalent **mg_python** method:

```
mg_python.ma_arg_set(0, arguments, 1, "", 1)
date = mg_python.ma_classmethod(0, "MyUtilities.MyClass",
                              "GetDateDecoded",
                               arguments, 1)
dateDisp = arguments[1]
```

Example 3 (Another simple method):

Caché ClassMethod:

```
Class MyUtilities.MyClass Extends etc …

ClassMethod GetCust(custID)
{
      ; Return the customer name
      Set cust=$Get(^Customer(custID))
      Quit cust
}
```

Equivalent **mg_python** method:

```
cust = mg_python.m_classmethod(0, "MyUtilities.MyClass", "GetCust",
                                  "1234")
```

Or

```
mg_python.ma_arg_set(0, arguments, 1, "1234", 0)
cust = mg_python.ma_classmethod(0, "MyUtilities.MyClass", "GetCust",
                              arguments, 1)
```

Example 4 (Passing an array from Python to Caché):

Caché ClassMethod:

```
Class MyUtilities.MyClass Extends etc …

ClassMethod ProcCustList(custList)
{
      ; Return the number of active customers
      Set custID="",activeCust=0
      For  Set CustID=$Order(^CustList(custID)) Do
      . If $Data(^CustOrderStatus(custID))="Active" Do
      .. Set activeCust=activeCust+1
      Quit activeCust
}
```

Equivalent **mg_python** method:

```
custList = []
key = [1, "1234"]
mg_python.ma_local_set(0, custList, -1, key, "")
key = [1, "1235"]
mg_python.ma_local_set(0, custList, -1, key, "")
key = [1, "1236"]
mg_python.ma_local_set(0, custList, -1, key, "")

mg_python.ma_arg_set(0, arguments, 1, custList, 0)
activeCust = mg_python.ma_classmethod(0, "MyUtilities.MyClass",
                                  "ProcCustList",
                                  arguments, 1)
```

Example 5 (Passing an array from Caché to Python):

Caché ClassMethod:

```
Class MyUtilities.MyClass Extends etc …

ClassMethod ActCList(custList)
{
      ; Return a list of active customers
      Set custID="",activeCust=0
      For  Set CustID=$Order(^CustOrderStatus(custID)) Quit:custID=""
Do
      . If $Data(^CustOrderStatus(custID))="Active" Do
      .. activeCust=activeCust+1
      .. Set custList(custID)=$Get(^Customer(custID))
      Quit activeCust
}
```

Equivalent **mg_python** method:

```
custList = []
mg_python.ma_arg_set(0, arguments, 1, custList, 1)
activeCust = mg_python.ma_classmethod(0, "MyUtilities.MyClass",
                                      "ActCList",
                                       arguments, 1)
```

Example 6 (Using a Caché method to modify a Python array):

Caché ClassMethod:

```
Class MyUtilities.MyClass Extends etc …

ClassMethod GetCustNames(custList)
{
      ; Return the names for a list of customers
      Set custID="",result=""
      For  Set CustID=$Order(^CustList(custID)) Do
      . Set custList(custID)=$Get(^Customer(custID))
      Quit result
}
```

Equivalent **mg_python** method:

```
custList = []
key = [1, "1234"]
mg_python.ma_local_set(custList, -1, key, "")
key = [1, "1235"]
mg_python.ma_local_set(custList, -1, key, "")
key = [1, "1236"]
```

```
mg_python.ma_local_set(custList, -1, key, "")

mg_python.ma_arg_set(0, arguments, 1, custList, 0)
activeCust = mg_python.ma_classmethod(0, "MyUtilities.MyClass",
                                      "GetCustNames",
                                       arguments, 1)
```

5.5.2 Return a block of HTML from a Caché ClassMethod.

```
connection_handle = mg_python.ma_html_classmethod_ex(<handle>,
                                                     <class_name>,
                                                     <method_name>,
                                                     <arguments>,
                                                     <no_arguments>)

buffer = mg_python.ma_get_stream_data(<handle>, <connection_handle>)
```

Types:

```
handle            (int)
connection_handle (int)
class_name        (String)
method_name       (String)
arguments         (List)
no_arguments      (int)
```

Usage:

```
connection_handle = mg_python.ma_html_classmethod_ex(<handle>,
                                                     <class_name>,
                                                     <method_name>,
                                                     <arguments>,
                                                     <no_arguments>)

buffer = mg_python.ma_get_stream_data(0, connection_handle)
while (buffer <> ""):
   print buffer
   buffer = mg_python.ma_get_stream_data(0, connection_handle)
```

Example 1:

Caché ClassMethod:

```
Class MyUtilities.MyClass Extends etc …

ClassMethod MyHtml()
{
      ; Return some HTML to Python
      Write "<p>This text was returned from Caché"
      Write "<br>You can return as text much as you like …"
```

```
      Quit
}
```

Equivalent **mg_python** method:

```
connection_handle = mg_python.ma_html_classmethod_ex(0,
                                                     "MyUtilities.MyClass",
                                                     "MyHtml",
                                                     arguments,
                                                     0)
buffer = mg_python.ma_get_stream_data(0, connection_handle)
while (buffer <> ""):
   print buffer
   buffer = mg_python.ma_get_stream_data(0, connection_handle)
```

Example 2:

Caché ClassMethod:

```
Class MyUtilities.MyClass Extends etc …

ClassMethod GetHTML(custID)
{
      ; Return some HTML to Python
      Write "<p>This text was returned from Caché"
      Write "<br>You can return as text much as you like …"
      Write "<br>A single argument '",custID,"' was passed from Python"
      Quit
}
```

Equivalent **mg_python** method:

```
mg_python.ma_arg_set(0, arguments, 1, "1234", 0)
connection_handle = mg_python.ma_html_classmethod_ex(0,
                                                     "MyUtilities.MyClass",
                                                     "GetHTML",
                                                     arguments,
                                                     1)

buffer = mg_python.ma_get_stream_data(0, connection_handle)
while (buffer <> ""):
   print buffer
   buffer = mg_python.ma_get_stream_data(0, connection_handle)
```

Example 3 (Passing an array from Python to Caché):

Caché ClassMethod:

```
Class MyUtilities.MyClass Extends etc …

ClassMethod GetCTable(custList)
{
```

```
      ; Return a table of customers to Python
      Write "<table>"
      Set custID=""
      For  Set CustID=$Order(^CustList(custID)) Quit:custID=""  Do
      . Set custName=$Get(^Customer(custID))
      . Set custList(custID)=custName ; Return name to Python
      . Write "<tr><td>",custID,"</td>"
      . Write "<td>",custName,"</td></tr>"
      Write "</table>"
      Quit
}
```

Equivalent **mg_python** method:

```
custList = []
key = [0, "1234"]
mg_python.ma_local_set(0, custList, -1, key, "")
key = [0, "1235"]
mg_python.ma_local_set(0, custList, -1, key, "")
key = [0, "1236"]
mg_python.ma_local_set(0, custList, -1, key, "")

mg_python.ma_arg_set(0, arguments, 1, custList, 0)
connection_handle = mg_python.ma_html_classmethod_ex(0,
                                                     "MyUtilities.MyClass",
                                                     "GetCTable",
                                                     arguments,
                                                     1)
buffer = mg_python.ma_get_stream_data(0, connection_handle)
while (buffer <> ""):
   print buffer
   buffer = mg_python.ma_get_stream_data(0, connection_handle)
```

5.6 Direct access to Python and M/Caché functions from the browser

There are many situations in web application programming where it is desirable to directly access server-side functionality from the context of the browser environment. For example, such a facility could be use for performing individual field validation and simple table lookups. For trivial operations of this sort it is rather expensive to have to submit the whole form to the server in order to communicate with the database. Browser components are supplied with mg_python to provide the capability of accessing Python and, subsequently, M/Caché functionality directly through browser-based scripting (for example, JavaScript).

Using the XMLHTTP script (mg_client.js)

The XMLHTTP script file (JavaScript) is included in the mg_python distribution:

```
/js/mg_client.js
```

The procedure for installing this file in the hosting web server environment is the same as for the Java Applet. The examples shown below are based on the XMLHTTP script being installed in the web server's documents root directory.

This represents the functionality of the mg_client browser component implemented in standard JavaScript. The new XMLHTTP functionality contained within the latest browsers is used to implement the connectivity between the browser environment and M/Caché (via Python /mg_python).

The rationale for providing this implementation is because of ongoing problems (both technical and commercial) with Java technology embedded in the newer browsers. It is therefore felt that customers should be offered a non-Java-based equivalent in order to secure their applications for the future.

Using the functions contained within 'mg_client.js' is straightforward. Indeed the interface is identical to that provided by the functionally equivalent Java Applet.

The following procedure should be used:

1. Include the JavaScript file in the hosting page instead of the Java Applet:

```
<script language="JavaScript" src="/mg_client.js"></script>
```

Example:

```
<HTML>
<HEAD><TITLE>My Form</TITLE>
<script language="JavaScript" src="/mg_client.js"></script>
```

```
</HEAD>
<BODY>

etc ...
```

2. Having initialized the JavaScript environment for 'mg_client.js' as described in the previous step, the internal functions can be used in exactly the same way as they would have been used with the Java Applet.

The function names (being plain JavaScript functions) are not qualified with a name. Therefore, what would have been ...

```
result = mg_client.server_proc(<URL>);
```

... for 'mg_client.class' (or mg_client.jar) is now ...

```
result = server_proc(<URL>);
```

... for 'mg_client.js'

The example shown (and described) in the previous section is shown below, recoded to use the XMLHTTP script instead of the Java Applet. Again, Python is used to script the form.

```
#
#   mg_python Test Page
#
#      Copyright (c) 2008 M/Gateway Developments Ltd.
#      All rights reserved.
#


import sys
import os
import cgi
import mg_python


key = []
keys = []
nvp = []
vars = {}

print 'Status: 200 OK'
print 'Content-Type: text/html'
print

content = sys.stdin.read(int(os.environ['CONTENT_LENGTH']))

keys = content.rsplit("&")
n = 0
while (n < len(keys)):
   nvp = keys[n].rsplit("=")
   if (len(nvp) > 1):
      vars[nvp[0]] = nvp[1]
   n = n + 1

if vars.has_key("m_fun"):
   m_fun = vars['m_fun']
   m_arg = vars['m_arg']
   if (m_fun != ""):
      if (m_fun == "NameLookup"):
         key[0] = 1
         key[1] = m_arg
         print mg_python.ma_return_to_client(0, mg_python.ma_get(0,
"^MGWCust", key))
         exit()

print '<html>'
print '<head>'

print '<script language="JavaScript" src="/mg_client.js"></script>'

print '<script LANGUAGE = "JavaScript">'

print 'function NameLookup(FormObject, value) {'
print '   FormObject.value =
server_proc("/GetName.rb?m_fun=NameLookup&m_arg=" + value);'
print '   return;'
print '}'
```

```
print '</script>'

print '<TITLE>Python to M - applet demo</TITLE>'

print '</head>'
print '<body>'
print '<form>'

print '<h1>Python to M - applet demo</h1>'

print 'Customer No <INPUT TYPE=TEXT NAME=id SIZE=30
ONCHANGE="NameLookup(form.name, this.value)">'
print 'Name <INPUT TYPE=TEXT NAME=name SIZE=30>'

print '<p>'

print 'Setup database when we load form ...'


key = [1, "1"]
mg_python.ma_set(0, "^MGWCust", key, "Chris Munt")
key[1] = "2"
mg_python.ma_set(0, "^MGWCust", key, "Rob Tweed")
key[1] = "3"
mg_python.ma_set(0, "^MGWCust", key, "John Smith")

print '<hr>'

print '</form>'
print '</body>'
print '</html>'
```

5.7 Handling error conditions

```
error = mg_python.m_get_last_error(handle)
```

Occasionally it is necessary for an mg_python method to return an error condition after failing to complete the prescribed task. For example, a target M/Caché server may be unavailable or there may be a problem with the supporting network.

Use the above method to return the last error message. The internal error message variable will only be reset as a result of a call to this method. Therefore, this method can be placed at the end of a series of mg_python method calls in order to see whether there were any error conditions encountered in processing the form.

On error, the mg_python methods will return an error code. The value of the error code is minus 1 by default (-1). This will be returned as either a string ("-1") or number (-1) depending on context. The corresponding error message can be obtained using the 'ma_get_last_error' method. For example:

```
key = [0]
error_code = mg_python.ma_delete(0, "^MGWCust", key)
if (error_code == "-1"):
   error_message = mg_python.m_get_last_error(0)
   print "<br>ERROR: ",  error_code, " ", error_message
```

Of course, in some cases the error code of '-1' may clash with legitimate return values. If this is anticipated to be the case, you can specify your own error code using the following method:

```
mg_python.m_set_error_code(handle, error_code)
```

For example:

```
handle = mg_python.m_allocate_page_handle()

mg_python.m_set_error_code(handle, "-7") # Reset error code

key = [0]
error_code = mg_python.ma_delete(handle, "^MGWCust", key)
if (error_code == "-7"):
   error_message = mg_python.m_get_last_error(handle)
   print "<br>ERROR: ",  error_code, " ", error_message

mg_python.m_release_page_handle(handle)
```

Scope

The public global variables mentioned in this section are scoped in accordance with the instance of the mg_python module and page handle in use.

5.8 Handling Python strings that exceed the maximum size allowed under M/Caché

M/Caché, in common with other M-based systems imposes a hard limit on the length of string that can be assigned to global nodes and variables in programs. In M/Caché the maximum string length is 32767 Bytes.

Python does not impose any such limit and the following convention can be used to trade 'oversize' strings between M/Caché and Python. Oversize strings are broken up into individual sections in M/Caché as follows:

```
variable = <First Section>
variable(extra, <Section Number>) = <Subsequent Section>
```

For example, take a string that exceeds the maximum length allowed in M/Caché by a factor of three:

```
variable = <First Section>
variable(extra, 1) = <Second Section>
variable(extra, 2) = <Third Section>
```

The same convention applies to arrays:

```
array("key") = <First Section>
array("key", extra, 1) = <Second Section>
array("key", extra, 2) = <Third Section>
```

The special variable '**`extra`**' is set by the mg_python engine. Its default value is **`ASCII 1`**. There is a possibility that this special data marker will clash with your data particularly where arrays are concerned. For example, you may have data keyed by the default value of ASCII 1. If this is the case, you can change the value of extra by editing its value in mg_python core routine: `VARS^%ZMGWSIS`

```
VARS  ; Public system variables
      Set extra=$C(1)
```

It should be noted however that this variable can only be reassigned on a per-installation basis in the above procedure. It should not be dynamically changed in other code.

mg_python will handle the transformation of oversize values to and from this form between Python and Caché and vice versa. The Python scripts are unaffected by these transformations.

Example 1 (Pass an oversize Python variable to M/Caché):

M/Caché procedure:

```
MyFun(arg)  ; Accept an oversize variable
            Set s1=$Get(arg)
            Set s1=$Get(arg(extra,1))
            Set s1=$Get(arg(extra,2))
            Quit 1
            ;
```

Equivalent Python method:

```
var = "large value …….."
mg_python.ma_arg_set(0, arguments, 1, var, 0)
result = mg_python.ma_proc(0, "MyFun^MyRoutine", arguments, 1)
```

Example 2 (Pass an oversize M/Caché variable to Python – by reference):

M/Caché procedure:

```
MyFun(arg)  ; Accept an oversize variable
            Set arg="first section …….."
            Set arg(extra,1)="second section …….."
            Set arg(extra,2))="third section …….."
            Quit 1
            ;
```

Equivalent Python method:

```
var = "large value …….."
mg_python.ma_arg_set(0, arguments, 1, var, 1)
result = mg_python.ma_proc(0, "MyFun^MyRoutine", arguments, 1)
var = arguments[1]
```

Example 3 (Pass an oversize M/Caché variable to Python – by return value):

Note the use of the '**`oversize`**' flag to instruct the mg_python engine to expect additional sections of an oversize return value to be held in global node **`^WORK($Job,0,`** where **`$Job`** is the M/Caché process ID returned by the M/Caché environment.

M/Caché procedure:

```
MyFun()     ; Return an oversize variable
            Set result="first section …….."
            Set ^WORKJ($Job,0,extra,1)="second section …….."
            Set ^WORKJ($Job,0,extra,2))="third section …….."
            Set oversize=1
            Quit result
            ;
```

Equivalent Python method:

```
result = mg_python.ma_proc(0, "MyFun^MyRoutine", arguments, 0)
```

Example 4 (Pass an oversize Python array node to M/Caché):

M/Caché procedure:

```
MyFun(array) ; Accept an oversize array node
             Set s1=$Get(array("key to long string")
             Set s2=$Get(array("key to long string",extra,1)
             Set s3=$Get(array("key to long string",extra,2)
             Quit 1
             ;
```

Equivalent Python method:

```
var = "large value …….."
key[0] = 1
key[1] = "key to long string"
mg_python.ma_local_set(0, records, -1, key, " large value ……..")
mg_python.ma_arg_set(0, arguments, 1, records, 0)
result = mg_python.ma_proc(0, "MyFun^MyRoutine", arguments, 1)
```

Example 5 (Pass an oversize M/Caché array node to Python):

M/Caché procedure:

```
MyFun(array) ; Accept an oversize array node
             Set array("key to long string")="Section 1"
             Set array("key to long string",extra,1"="Section 2"
             Set array("key to long string",extra,2) ")="Section 3"
             Quit 1
             ;
```

Equivalent Python method:

```
var = "large value …….”
key[0] = 1
key[1] = "key to long string"
mg_python.ma_local_set(0, records, -1, key, " large value …….")
mg_python.ma_arg_set(0, arguments, 1, records, 1)
result = mg_python.ma_proc(0, "MyFun^MyRoutine”, arguments, 1)
```

5.9 Handling large Python arrays/lists in M/Caché

```
mg_python.m_set_storage_mode(handle, mode)
```

In addition to imposing limits on the maximum length of string that can be used, M/Caché also limits the amount of memory (or *partition* space) that each process can use. Python, on the other hand, does not impose such limits but must, of course, work within the system limits imposed by the hosting computer. If large Python arrays are sent to M/Caché, it may be necessary to specify that these arrays be held in a permanent storage within the M/Caché environment (i.e. a workfile) rather than in memory. The amount of memory that M/Caché allows for each process will depend on individual configurations.

The 'storage_mode' facility gives mg_python software some control over how array data is projected to the M/Caché environment.

Mode 0

```
mg_python.m_set_storage_mode(handle, 0)
```

This is the default. Python arrays are projected into (and out of) the M/Caché environment as simple memory-based arrays.

Mode 1

```
mg_python.m_set_storage_mode(handle, 1)
```

Python arrays are projected into (and out of) the M/Caché environment as equivalently structured global arrays (in permanent storage). These globals are structured as follows:

```
^WORKJ($Job,argn,
```

Where:

$Job – The M/Caché process ID (supplied by the M/Caché environment).
argn - The argument number in the function call.

Example 1 (Pass a Python array into M/Caché global storage):

M/Caché procedure:

```
MyFun(array) ; Process an array held in a global
             Set key=""
             For  Set key=$Order(^WORKJ($Job,1,key)) Quit:key=""  Do
             . Set data=$Get(^WORKJ($Job,1,key))
             . Quit
```

```
            Quit 1
            ;
```

Equivalent Python method:

```
handle = mg_python.m_allocate_page_handle()

mg_python.m_set_storage_mode(handle, 1)

records = [] # Clear records array
key[0] = 1
key[1] = "key 1"
mg_python.ma_local_set(handle, records, -1, key, "value 1")
key(1) = "key 2"
mg_python.ma_local_set(handle, records, -1, key, "value 2")
mg_python.ma_arg_se(handle, arguments, 1, records, 0)
result = mg_python.ma_proc(handle, "MyFun^MyRoutine", arguments, 1)

mg_python.m_release_page_handle(handle)
```

Example 2 (Pass a M/Caché array held in global storage to Python):

M/Caché procedure:

```
MyFun(array) ; Process an array held in a global
            Set ^WORKJ($Job,1,"key 1")="Value 1"
            Set ^WORKJ($Job,1,"key 2")="Value 2"
            Set ^WORKJ($Job,1,"key to long string")="Section 1"
            Set ^WORKJ($Job,1,"key to long string",extra,1)="Section 2"
            Set ^WORKJ($Job,1,"key to long string",extra,2)="Section 3"
            Quit 1
            ;
```

Equivalent Python method:

```
handle = mg_python.m_allocate_page_handle()

mg_python.m_set_storage_mode(handle, 1)

records = [] # Clear records array
mg_python.ma_arg_set(handle, arguments, 1, records, 1)
result = mg_python.ma_proc(handle, "MyFun^MyRoutine", arguments, 1)

mg_python.m_release_page_handle(handle)
```

Scope

The value of 'storage_mode' is scoped in accordance with the instance of the mg_python module and page handle in use.

6 License

http://www.mgateway.com
Email: cmunt@mgateway.com