Rocket UniVerse

U2 Python User Guide

Version 11.3.1

October 2016
UNV-1131-PYRG-1

Notices

Edition

Publication date: October 2016
Book number: UNV-1131-PYRG-1
Product version: Version 11.3.1

Copyright

Trademarks

Examples

This information might contain examples of data and reports. The examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

License agreement

Note: This product may contain encryption technology. Many countries prohibit or restrict the use, import, or export of encryption technologies, and current use, import, and export regulations should be followed when exporting this product.

Corporate information

Rocket Software, Inc. develops enterprise infrastructure products in four key areas: storage, networks, and compliance; database servers and tools; business information and analytics; and application development, integration, and modernization.

Website: www.rocketsoftware.com

Rocket Global Headquarters
77 4th Avenue, Suite 100
Waltham, MA 02451-1468
USA

To contact Rocket Software by telephone for any reason, including obtaining pre-sales information and technical support, use one of the following telephone numbers.

Country	Toll-free telephone number
United States	1-855-577-4323
Australia	1-800-823-405
Belgium	0800-266-65
Canada	1-855-577-4323
China	800-720-1170
France	08-05-08-05-62
Germany	0800-180-0882
Italy	800-878-295
Japan	0800-170-5464
Netherlands	0-800-022-2961
New Zealand	0800-003210
South Africa	0-800-980-818
United Kingdom	0800-520-0439

Contacting Technical Support

The Rocket Customer Portal is the primary method of obtaining support. If you have current support and maintenance agreements with Rocket Software, you can access the Rocket Customer Portal and report a problem, download an update, or read answers to FAQs. To log in to the Rocket Customer Portal or to request a Rocket Customer Portal account, go to www.rocketsoftware.com/support.

In addition to using the Rocket Customer Portal to obtain support, you can use one of the telephone numbers that are listed above or send an email to support@rocketsoftware.com.

Contents

Chapter 1: U2 Python

Rocket UniVerse and UniData (referred to in this manual as U2) have extended the database language capabilities to include the use of Python, a dynamic and more modern object-oriented programming language. The integration of Python and U2 allows you to program backend database logic with high extensibility, in a language that supports the development of new applications based on U2.

Note: This document assumes you know about the Python programming language. For more information about how to use the Python language itself, see https://www.python.org.

The choice of Python among many other modern programming language candidates was made on several factors, including how open the language platform is (meaning there is formal support for extending and embedding the language), the availability of third-party packages and the size of its development communities, how steep the learning curve is, and so on. The integration between Python and U2 is one that allows Python to stand on its own as a U2 application development language in addition to being a complimentary one to U2 BASIC.

Throughout this document, the integration of Python and U2 is referred to as U2 Python.

Note: A supplementary video is available to help you learn about U2 Python. To watch the overview video, click here. To learn how to use the `RUNPY` and `PYTHON` commands, click here.

About U2 Python

U2 Python has the capability to access U2 resources such as data, subroutines, query tools, and so on. U2 applications can invoke Python code, be it third party or locally made, to perform various tasks that are difficult or even impossible to do in U2 BASIC.

This approach requires two tasks be accomplished by C function calls so that there is no need for any interprocess-communication:

- The U2 server and the Python interpreter run in the same process, and
- The interaction between the U2 server and the Python interpreter

At UniVerse 11.3.1, only strings, numbers, and Python object references can be passed between U2 and Python.

The Python integration

U2 Python integrates into ECL/TCL and BASIC. In ECL/TCL, two new commands have been added: `RUNPY` and `PYTHON`. In BASIC, a set of functions and @variables have been updated. The u2py extension module is also available as a well-defined C API for writing new built-in modules to Python.

Since Python does not reload any module already imported, in order to always call the latest Python code each time the `PYTHON`, `RUNPY`, or `RUN` commands are used, the Python environment is reinitialized after the commands are finished.

Warning: Although rare, if the Python module does not support reinitialization, such as the Python imaging library's image module, when the module is imported the second time, a warning or error message, unexpected exit, or corruption can occur in the UniVerse session. Instead, use the `SH` or `DOS` command to enter the operating system prompt, then start another UniVerse session to perform the command.

ECL/TCL integration

Two new commands, `PYTHON` and `RUNPY`, have been added at the ECL/TCL level.

- `PYTHON`: At the ECL/TCL prompt, run this command to launch into Python's interactive shell and execute Python commands.
- `RUNPY`: Run a Python program from ECL/TCL. U2 files can be used to store Python programs as well as BASIC programs.

U2 BASIC integration

A set of functions and @variables allow you to run a Python program from within BASIC, passing data back and forth with the Python program. For more information about the API, see U2 BASIC Python API, on page 13.

The u2py extension module

Python provides a well-defined C API for writing new built-in modules to Python. Such extension modules can do two things that cannot be done directly in Python: they can implement new built-in object types, and they can call C library functions and system calls.

`u2py` is one such extension module for Python that makes use of the modified UniObjects API to provide access to various aspects of the U2 server, such as direct U2 file access, transaction control, BASIC subroutine calls, and so on. The original UniObjects API is used for client/server/middleware products of U2. For U2 Python, the UniObjects API has been refactored to make it usable by C programs on the server side.

For a detailed description of the functionalities of `u2py` module, including functions, classes, global definitions, enter `help(u2py)` at a Python prompt, as described in u2py functionality, on page 24.

Chapter 2: Installing U2 Python

U2 Python version 3.4.1 is a default installation version bundled with UniVerse 11.3.1 release. This version of Python was used in Rocket development and testing environments, and had passed minimum testing requirements. Other versions of Python have not been tested and should you attempt to use a later version of Python with your installation, Rocket support will only be able to provide assistance with issues encountered if they are reproduced with the default U2 Python installation.

Separate Python installations are also not supported, though should you use a separate installation of Python, we recommend that any separate Python version installed should be at the same release as the version supplied with U2 (for example Python version 3.4.1 with UniVerse 11.3.1). Other versions might not be able to utilize U2 functionality due to changes in the Python APIs and interfaces.

Note: All examples in this document are included with the XDEMO account version 3.1.0 or later.

By default, U2 Python is installed together with the U2 server under the `python` directory in `$UDTHOME` or `$UVHOME`.

For specific steps about installing UniVerse or UniData, see the *Installation Guide*.

U2 Python licensing

The Python add-on must be licensed in UniVerse in order for it to be used.

To license U2 Python on UniVerse, perform one of the following actions:

- On Windows platforms, select the **Python** check box during installation.
- In XAdmin, from the Admin Tasks pane, double-click **License**. From the **Update** tab, click the **PYTHON** check box, or use the wizard and select **PY**.
- On Linux platforms, add `PY:1` to the Linux license screen on the package line.
- On either Windows or Linux platforms, using `uvregen`, enter `bin/uvregen -p PY:1` in $UVHOME.

Whenever Python is used in a U2 process, the Python library is loaded into memory and UniVerse checks whether a license is already consumed for that U2 process. A non-phantom process must have already acquired a license, so the check succeeds and the loading of the Python library is allowed to continue. If a phantom process is already charged for an I-phantom license, then the check succeeds as well and the loading of the Python library is allowed to proceed. For a phantom process that has taken no license, UniVerse tries to acquire an I-phantom license for it and if successful, then the process is allowed to continue; if unsuccessful, an error occurs and prints to the associated _PH_ file and the process terminates.

Operating system requirements

For U2 Python to work on your operating system, make sure the location of any `.pth` configuration files and the `.pyconfig` file are placed in the correct directories.

.pth configuration files

For the `u2py` extension module to be loadable in Python, the `u2.pth` file is placed in specific directories under the bundled Python installation directory:

- On Windows for UniData, `$UDTHOME\python`
- On Windows for UniVerse, `$UVHOME\python`
- On Linux for UniData, `$UDTHOME/python/python#.#/site-packages`
- On Linux for UniVerse, `$UVHOME/python/python#.#/site-packages`

The full path of `$UDTBIN` or `$UVBIN` is listed in the `u2.pth` file so that Python will load it into its search path when starting up. The XDEMO's PP directory path is included in the `u2.pth` file. Do not add any custom paths to the `u2.pth` file because this file is overwritten on upgrades.

To add your directories containing Python modules to the module search path, place a unique path configuration file with the `.pth` extension in the appropriate path for your database and platform, listed above. Each path needed is included on a separate line in the custom `.pth` files. For example, on UniVerse and Windows in the `$UVHOME/python` directory with a file called `myprograms.pth`, the paths will look like:

```
c:\U2\Accounts\SALES\PP
c:\U2\Accounts\RENTALS\PP
```

The custom `.pth` files are not modified on upgrades.

Remember: Directory paths and the `.pth` extensions are case-sensitive on Linux.

.pyconfig file

For the U2 BASIC Python API to find and load the bundled Python library, a `.pyconfig` file is created under the `$UDTHOME` or `$UVHOME` directory.

The following two lines are examples of the added code to the `.pyconfig` file for UniVerse:

```
PYHOME=C:\U2\uv113\python
PYLIB=C:\U2\uv113\python\python34.dll
```

This file is used by U2 servers when it needs to dynamically load the Python library and set it up during run time. If you want to try a different version of Python other than the bundled one, you can modify this file to point to its installation directory and the Python library.

Note: You can attempt to use a later version of Python with your installation, but Rocket support does not provide assistance to any issues that you may encounter. Rocket may require reproduction of the error on the version of Python that is shipped with U2.

Applicable platforms

You can install U2 Python on the following Windows and Linux platforms.

- 64-bit Windows 7 and 8.1
- Linux

If you have installed more than one instance of Python on the same Linux or Windows machine, in order to avoid the U2 process loading two different Python shared libraries, verify that the Python library in the LD_LIBRARY_PATH (Linux platforms) or PATH (Windows platforms) environment variable is the same as the one specified in $UDTHOME (UniData) or $UVHOME/.pyconfig (UniVerse).

Changing the location of the Python installation (Windows)

The installation places the UniVerse bin and Python home directories in the Windows PATH environment variable. You need to restart your computer so that the directories are recognized. Perform the following steps if you need to change the location of the installation.

Note: Rocket will support use of the Python version supplied with UniVerse. If an issue arises from use of an alternate Python version, Rocket may request that the issue be reproduced with the default Python version in order to provide assistance.

1. Install the 64-bit Python version.

   Usually, the appropriate source for this is https://www.python.org/.
2. Locate the `.pyconfig` configuration file in the UniVerse home directory. This location should not be overwritten or changed. Save the existing file to another file (for example, `.pyconfig_original`).
3. Modify the `.pyconfig` file from its default values to point to the alternate location.

   For example, the original `.pyconfig` file looks similar to the following on Windows for UniVerse:

   ```
   PYHOME=C:\U2\UV\python
   PYLIB=C:\U2\UV\python\python34.dll
   ```

   The modified `.pyconfig` file might look like the following:

   ```
   PYHOME=C:\Program Files\Python\Python35
   PYLIB=C:\Program Files\Python\Python35\python35.dll
   ```

Modifying the $PATH variable (Linux)

On some UNIX systems, you might have a standard Python installation preinstalled. Commonly, even on Red Hat 6, this is a Python 2.x version. If you want the Python version that is installed with UniVerse to appear when typing `python3`, you need to modify the $PATH variable.

1. Enter `which python` to see whether your Python version points to `/usr/bin` or another system directory.
2. If the Python version is not the one included with UniVerse 11.3.1, add the following code to the front of your $PATH variable. You will need to change the path to match what is in `/.uvhome` or `/.udthome`.

   `/usr/uv/python/bin`

Using other versions of Python with U2

Although U2 is bundled with Python 3.4.1, it can work with other Python 3.4.x feature releases.

If you are already using Python 3.4.x and want to keep it as the Python interpreter for U2, you can modify the `.pyconfig` file to specify an existing Python installation that you want U2 to use, as shown in the following example:

```
PYHOME=C:\python34
PYLIB=C:\U2\UV\python\python34.dll
```

Note: Other Python versions such as 3.5 are not allowed to import the u2py module in the Python environment. If attempted, the following error message appears:
"ImportError: Module use of python34.dll conflicts with this version of Python."

Additionally, you need to copy any `*.pth` files from the bundled Python 3.4.1 installation directory to the existing Python home directory or site-packages directory.

After you complete the installation, `LOGTO` the XDEMO account, start the U2 database, and then follow the steps in the next example to make sure U2 and Python have been installed correctly and can actually work together.

```
>PYTHON
python> import u2py
python> u2py.run("COUNT VOC")

857 records counted.
python>
```

Finding what version of Python is installed

If you do not know what version of Python you are using, execute the `sys.version_info` command.

```
python>import sys
python>sys.version_info
sys.version_info'3.4.1 (v3.4.1:c0e311e010fc, May 18 2014, 10:45:13)
[MSC v.1600 64 bit (AMD64)]'
```

Chapter 3: Accessing Python from U2

You can start using Python from a U2 shell in a few different ways. After you are in a Python prompt, you can start developing Python programs and calling Python functions.

At the TCL prompt, you can use the `PYTHON` and `RUNPY` commands to access Python. `PYTHON` invokes the Python interactive shell, and `RUNPY` allows you to run a Python program. For more information, see The PYTHON command, on page 12 and The RUNPY command, on page 12.

You can also utilize the new U2 BASIC API to call Python functions and speed up your development efforts. For more information, see U2 BASIC Python API, on page 13 and Calling Python functions from a U2 BASIC Program, on page 16.

The PYTHON command

The Python interactive shell is a convenient environment for exploring existing modules and getting help with any part of Python functionality. It is also useful for quickly testing your code. On Windows, enter `PYTHON` to access the `python>` prompt. On UNIX, enter `python3` to access the `>>>` prompt.

After entering the Python interactive shell, you can import the Python extension module and query its capabilities using the `help(u2py)` and `dir(u2py)` commands, and then drilling down to the classes, functions, and properties. The directory where you run the Python interactive shell must be a valid U2 account in order to import and use the `u2py` module. For more information, see:

- Importing u2py, on page 24
- Accessing u2py help, on page 24
- Accessing u2py object attributes, on page 25

The following example shows a UniVerse session using the `PYTHON` command to start the `python>` prompt, and then running commands from the `python>` prompt.

```
[root@pythonhost XDEMO]# uv
UniVerse Command Language 11.3.1
Copyright Rocket Software, Inc. or its affiliates, All Rights Reserved 1985-2016
XDEMO logged on: Wed Mar 18 15:34:24 2015

Welcome to the XDEMO Account
Version 2.03 - March 18, 2015
>PYTHON
python> import sys
python> sys.version
'3.4.1 (default, Feb 18 2015, 13:36:51) \n[GCC 4.4.7 20120313 (red Hat 4.4.7-3)]'
python> quit
>COUNT VOC

862 records counted
>
```

The RUNPY command

If you are at the U2 prompt and need to quickly test a Python program, you do not have to leave the U2 environment to do so. The `RUNPY` command, which has a similar syntax to the `RUN` command, allows you to call Python functions directly from ECL and TCL.

Syntax

RUNPY [*filename*] *Python_program*

The following UniVerse example displays a new file that was created in the XDEMO account, PP, that contains a Python print program called Greeting.py.

```
>CT PP Greeting.py

     Greeting.py
0001 # Copyright (C) Rocket Software 1993-2015
0002 hello_name = input ('Type your name: ')
0003 print("Hello", hello_name)
```

You can run this program at ECL or TCL by executing the RUNPY command and viewing the results:

```
>RUNPY PP Greeting.py
Type your name: Charlie
Hello Charlie
>
```

U2 BASIC Python API

This section describes the APIs added for Python at UniVerse 11.3.1.

PYOBJECT

At UniVerse 11.3.1, a new BASIC variable type, PYOBJECT is available.

PYOBJECT is used internally to receive a Python object. A PYOBJECT variable cannot be printed or otherwise manipulated in U2 BASIC, but it can be passed to another U2 BASIC Python API function.

Note: U2 BASIC does not have a way to tell the type of a variable.

@variables

The following table describes the BASIC @variables added at UniVerse 11.3.1.

Note: These @variables capture the details of a Python exception when it is thrown in the Python code called from BASIC program.

@Variable	Description
@PYEXCEPTIONMSG	A string that stores the detailed exception message; if no exception is thrown, its value is an empty string.
@PYEXCEPTIONTRACEBACK	A string that stores the traceback of the exception; if no exception is thrown, its value is an empty string.
@PYEXCEPTIONTYPE	A string that stores the exception type; if no exception is thrown, its value is an empty string.

PyCall function

The `PyCall` function calls a Python callable object.

Syntax

```
pyresult = PyCall(PyCallableObject[,arg1, arg2, ...])
```

Parameters

The following table describes the parameters for this function.

Parameter	Description
pyresult	A standard U2 BASIC variable or a PYOBJECT variable.
pycallableobject	A PYOBJECT variable pointing to a Python object that is callable, such as a function object, class object, or method object.
arg1,arg2,…	The arguments to the callable Python object that can be evaluated to a string, a number, or a PYOBJECT.

PyCallFunction function

The `PyCallFunction` function calls a Python function on a Python module.

Syntax

```
pyresult = PyCallFunction(moduleName, functionName[, arg1, arg2, ...])
```

Parameters

The following table describes the parameters for this function.

Parameter	Description
pyresult	A standard U2 BASIC variable or a PYOBJECT variable.
moduleName	The name of the module where the function is defined.
functionName	The name of the function to be called.
arg1,arg2	The arguments to the function object that can be evaluated to a string, a number, or a PYOBJECT.

PyCallMethod function

The `PyCallMethod` function calls a method on a Python object.

Syntax

```
pyresult = PyCallMethod(pyobject, methodName [,arg1, arg2, ...]
```

Parameters

The following table describes the parameters for this function.

Parameter	Description
pyresult	A standard U2 BASIC variable or a PYOBJECT variable.
pyobject	A PYOBJECT variable pointing to a Python object
methodName	The name of the method to be called. Must be defined on the class of the object.
arg1,arg2	The arguments to the method that can be evaluated to a string, a number, or a PYOBJECT.

PyGetAttr function

The `PyGetAttr` function gets the value of an attribute of a Python object.

Syntax

```
pyresult = PyGetAttr(pyobject, attrName)
```

Parameters

The following table describes the parameters for this function.

Parameter	Description
pyresult	A standard U2 BASIC variable or a PYOBJECT variable.
pyobject	A PYOBJECT variable pointing to a Python object.
attrName	The name of the attribute whose value is to be retrieved.

PyImport function

The `PyImport` function imports a Python module.

Syntax

```
pyresult = PyImport(moduleName)
```

Parameters

The following table describes the parameters for this function.

Parameter	Description
pyresult	A PYOBJECT variable pointing to the Python module object.
moduleName	The name of the module to be imported.

PySetAttr function

The `PySetAttr` function sets the value of an attribute of a Python object.

Syntax

```
pyresult = PySetAttr(pyobject, attrName, value)
```

Parameters

The following table describes the parameters for this function.

Parameter	Description
pyresult	An integer value, -1: failure.
pyobject	A PYOBJECT variable pointing to a Python object.
attrName	The name of the attribute whose value to be set.
value	A value expression that can be evaluated to a string, a number, or a PYOBJECT.

Passing number and string variables to Python functions

U2 BASIC allows variables to be both a number and a string simultaneously while Python does not. To specify which value type BASIC Python API should use when passing the variable to a Python function, you must specify an expression.

If a variable should be used as a number, instead of passing the variable directly into the function, use the following expression:
`variable_name + 0`

If a variable should be used as a string, use the following expression:
`variable_name : ""`

For example:

```
mypythonmodule.py:
    def mypythonfunc(a_string, a_number):
        print("this is a string" + a_string)
        print(a_number * 10)
```

In BASIC:

```
anumber = 123456
astring = "123456"
result = PyCallFunction("mypythonmodule", "mypythonfunc", anumber:"", astring + 0)
```

Calling Python functions from a U2 BASIC Program

Calling Python functions from U2 programs can speed up your development efforts by bringing the wealth of available Python modules and other community and third party code to your U2 applications.

Note: When searching online for a suitable piece of Python code, make sure it is usable in Python version 3, as some code only works in previous versions of Python. The notable change is that `print()` is a function in Python 3 and not a statement. Refer to https://docs.python.org/3.0/whatsnew/3.0.html for more information.

To include custom Python modules in Python's search path, additional `.pth` files are required. For more information, see .pth configuration files, on page 9.

The following examples provide a more detailed look into calling Python functions from U2 programs.

Example 1: FINDWAREHOUSE_PYFUNC

A U2 BASIC program FINDWAREHOUSE_PYFUNC retrieves the address of a movie club member from the MEMBERS file in the XDEMO account, then retrieves the addresses of all movie warehouses from the LOCATIONS file. It then determines which warehouse is closest to the member with the help of two Python functions, `decodeAddressToGeocode`() and `distanceBetweenPoints`(). These two Python functions demonstrate the ease with which you can call external web services in Python, parse JSON, or perform calculations with decimal numbers.

U2 BASIC Program - FINDWAREHOUSE_PYFUNC:

```
*************************************************************************
*
*   Program:    FINDWAREHOUSE_PYFUNC
*   Desc:   This sample program reads a record from MEMBERS file in
*   the xdemo account (used in Rocket demos) and determines which movie
*   warehouse (contained in LOCATIONS file) is closest to the
*   member's address.
*   It demonstrates the use of the new function in U2 Basic,
*   PyCallFunction(), by calling two Python functions:
*   decodeAddressToGeocode() and distanceBetweenPoints().
*
*   Python function decodeAddressToGeocode() takes an address
*   as an input string and returns the corresponding latitude and
*   longitude using Google MAPS API.
*   FINDWAREHOUSE_PYFUNC calls decodeAddressToGeocode() with
*   the member's address and addresses of all warehouses.
*
*   Python function distanceBetweenPoints() takes latitudes and
*   longitudes of two locations and calculates the distance between
*   them in miles.
*
*************************************************************************

ModuleName = "AddressToCoordObj"
FuncName = "decodeAddressToGeocode"

*   read a record from MEMBERS file
OPEN 'MEMBERS' TO MEMBERS.FILE ELSE STOP "CAN'T OPEN MEMBERS"
MEMBERS.ID = '0277'
READ REC FROM MEMBERS.FILE, MEMBERS.ID ELSE
  STOP "cannot read record 0277"
END
CRT "MEMBER NAME: ":REC<2>:" ":REC<1>
map_address = REC<5>:", ":REC<6>:", ":REC<7>:", ":REC<8>
CRT "CUSTOMER ADDRESS IS: ":map_address
*   call first Python function to get member's location coordinates
*   it returns an object of class Point which has two attributes,
*   lat and lng, and method str
```

```
member_location = PyCallFunction(ModuleName, FuncName, map_address)

*   check the outcome and print exception and traceback information
*   if an exception was raised
IF @PYEXCEPTIONTYPE NE '' THEN
  GOSUB CRT.EXCEPTION.INFO
  STOP
END

*   Get CRTable latitude and longitude from member_location.
*   One way to do it is to get a string in the form "lat, lng"
*   using str method of class Point
member_coords = PyCallMethod(member_location, 'str')
IF @PYEXCEPTIONTYPE NE '' THEN
  GOSUB CRT.EXCEPTION.INFO
  STOP
END
CRT "CUSTOMER LOCATION COORDINATES ARE: " :member_coords

*   Import the second module
ModuleName2 = "DistanceBtwnPoints"
FuncName2 = "distanceBetweenPoints"

*   retrieve all Warehouse locations and
*   calculate the distance from this customer address to each warehouse
OPEN 'LOCATIONS' TO LOCATIONS.FILE ELSE STOP "CAN'T OPEN LOCATIONS"
SELECT LOCATIONS.FILE
DONE = 0
MINDISTANCE = -1
LOOP
  READNEXT LOCATIONS.ID ELSE DONE = 1
WHILE NOT(DONE) DO
  READ LOC.REC FROM LOCATIONS.FILE,LOCATIONS.ID ELSE
    CRT "No LOCATIONS RECORD: ":LOCATIONS.ID
    EXIT
  END
  CRT "WAREHOUSE : ":LOC.REC<1>
  wh_address = LOC.REC<5>:", ":LOC.REC<6>:", ":LOC.REC<7>:", ":LOC.REC<8>
  CRT "WAREHOUSE ADDRESS IS: ":wh_address
  * call first Python function to get this warehouse location coords
  wh_location = PyCallFunction(ModuleName, FuncName, wh_address)
  IF @PYEXCEPTIONTYPE NE '' THEN
    GOSUB CRT.EXCEPTION.INFO
    STOP
  END
  * Get printable latitude and longitude from warehouse location
  * An alternative way to do it is to get lat and lng attribute
  * values of object wh_location of class Point
  wh_lat = PyGetAttr(wh_location, 'lat')
  IF @PYEXCEPTIONTYPE NE '' THEN
    GOSUB CRT.EXCEPTION.INFO
    STOP
  END
  wh_lng = PyGetAttr(wh_location, 'lng')
  IF @PYEXCEPTIONTYPE NE '' THEN
    GOSUB CRT.EXCEPTION.INFO
    STOP
  END
  CRT "WAREHOUSE LOCATION COORDINATES ARE: " :wh_lat:", ":wh_lng

  * call second Python function to calculate the distance
  pydist = PyCallFunction(ModuleName2, FuncName2, member_location, wh_location)
```

```
  IF @PYEXCEPTIONTYPE NE '' THEN
    GOSUB CRT.EXCEPTION.INFO
    STOP
  END

  CRT "DISTANCE IS " :OCONV(pydist,"MD1P"):" MILES"
  IF MINDISTANCE = -1 THEN
     MINDISTANCE = pydist
     WAREHOUSE = LOC.REC<1>
  END ELSE
     IF pydist < MINDISTANCE THEN
        MINDISTANCE = pydist
        WAREHOUSE = LOC.REC<1>
     END
  END
REPEAT

CRT "MIN DISTANCE IS: ":OCONV(MINDISTANCE,"MD1P")
CRT "WAREHOUSE IS: ":WAREHOUSE

STOP

CRT.EXCEPTION.INFO:
  CRT "EXCEPTION TYPE IS " :@PYEXCEPTIONTYPE
  CRT "EXCEPTION MESSAGE IS " :@PYEXCEPTIONMSG
  CRT "EXCEPTIONTRACEBACK IS " :@PYEXCEPTIONTRACEBACK
RETURN
END
*********************************************************************
```

Python Module – AddressToCoordObj.py:

```
######################################################################
# Geocoding example: calculate coordinates of a street address
# using Google MAPS API and response in JSON format
# Input is a "postal address" in the form of a human-readable address string and
# an indication whether the request comes from a device with a location sensor
# Response in JSON format contains an array of geocoded address information
# and geometry information. We are specifically interested in "lat" and "lng"
# values of the geometry location result.

import io as StringIO
from urllib.request import urlopen
from urllib.parse import urlencode
import json

class Point:
    def __init__ (self, lat=0, lng=0):
        self.lat = lat
        self.lng = lng

    def str (self):
        return str(self.lat) + ', ' + str(self.lng)


def decodeAddressToGeocode( address ):
    urlParams = {
        'address': address,
        'sensor': 'false',
    }
    url = 'http://maps.google.com/maps/api/geocode/json?' + urlencode(urlParams)
    response = urlopen( url )
```

```
    responseBody = str(response.read(), encoding='UTF-8')
    body = StringIO.StringIO( responseBody )
    result = json.load( body )

    if 'status' not in result or result['status'] != 'OK':
        return None
    else:
        #latitude,longitude
        coordinates = Point(result['results'][0]['geometry']['location']['lat'],\
        result['results'][0]['geometry']['location']['lng'])
        return coordinates
#####################################################
```

Python Module – DistanceBtwnPoints.py:

```
#################################################################
# Calculate the distance between two points on Earth
# assuming it is a perfect sphere
# using Haversine formula
# code borrowed from Wayne Dyck at
# http://www.platoscave.net/blog/2009/oct/5/calculate-distance-latitude-
    longitude-python/

import math
from decimal import *
def distanceBetweenPoints(point1, point2):
    lat1 = Decimal(point1.lat)
    lat2 = Decimal(point2.lat)
    long1 = Decimal(point1.lng)
    long2 = Decimal(point2.lng)

    # Convert latitude and longitude to
    # spherical coordinates in radians.
    radius = 3960 # miles

    dlat = math.radians(lat2-lat1)
    dlon = math.radians(long2-long1)
    a = math.sin(dlat/2) * math.sin(dlat/2) + math.cos(math.radians(lat1)) \
        * math.cos(math.radians(lat2)) * math.sin(dlon/2) * math.sin(dlon/2)
    c = 2 * math.atan2(math.sqrt(a), math.sqrt(1-a))
    d = radius * c

    return d
#################################################################
```

Result:

```
>RUN PBP FINDWAREHOUSE_PYFUNC
MEMBER NAME: Raul Rajon
CUSTOMER ADDRESS IS: 1187 Fleet Street, Bergenfield, NV, 23451
CUSTOMER LOCATION COORDINATES ARE: 36.8643566, -75.9985693
WAREHOUSE : Main warehouse
WAREHOUSE ADDRESS IS: 80204, Chris Wingman, 3737968345, 3737961123
WAREHOUSE LOCATION COORDINATES ARE: 39.738, -105.0265
DISTANCE IS 1579.9 MILES
WAREHOUSE : Secondary warehouse
WAREHOUSE ADDRESS IS: 60613, Mark Allen, 6509848989, 6509843334
WAREHOUSE LOCATION COORDINATES ARE: 41.6476, -85.9052
DISTANCES IS 624.2 MILES
WAREHOUSE : Tertiary Warehouse
WAREHOUSE ADDRESS IS: 90012, Fred Bloggs, 3335551234, 3335551236
WAREHOUSE LOCATION COORDINATES ARE: 34.0653, -118.2439
```

```
DISTANCE IS 2366.6 MILES
MIN DISTANCE I: 624.2
WAREHOUSE IS: Secondary warehouse
>
```

Example 2: SENDALERT_PYFUNC

A U2 BASIC program, SENDALERT_PYFUNC, sends a hardware failure notification to a system administrator with the use of Python's smtplib and Google Gmail SMTP server. The U2 BASIC program prompts the user for the sender's and recipient's email addresses, message subject and body, Gmail user name and password, and then calls the Python function `SendAlert`(). The Python function `SendAlert`() demonstrates the simplest way to send an email message using smtplib. For more interesting cases, such as email attachments and multiple recipients, there are additional helpful modules in Python, such as base64 and email.

Note: To use this example with Gmail, turn on "Access for less secure apps" in your Gmail settings.

U2 BASIC Program - SENDALERT_PYFUNC (NOTE: the `PyCallFunction` line wraps):

```
*************************************************************************
*
*   Program:    SENDALERT_PYFUNC
*   Desc:   This sample program sends a hardware failure notification to
*   a system administrator with the use of Python's smtplib and
*   Google Gmail SMTP server.
*   The U2 Basic program prompts the user for sender's and recipient's
*   email addresses, message subject and body, Gmail username
*   and password.
*   It demonstrates the use of the new function in U2 Basic,
*   PyCallFunction(), by calling the Python function SendAlert().
*
*   Python function SendAlert() uses input parameters received from
*   SENDALERT_PYFUNC to construct the email message text
*   and sends it through GMail server.
*
*************************************************************************
PROMPT ""
ModuleName = "Notification"
FuncName = "SendAlert"

CRT "Enter Sender's Address: ":; INPUT fromaddr
CRT "Enter Recipient's Address: ":; INPUT toaddrs
CRT "Enter Message Subject: ":; INPUT subject
CRT "Enter Message Body: ":; INPUT msgtext
ECHO OFF
CRT "(Authentication) Enter gmail username: ":; INPUT username
ECHO ON
CRT ""
ECHO OFF
CRT "(Authentication) Enter user password: ":; INPUT password
ECHO ON
CRT ""
*   call the Python function
pyresult = PyCallFunction(ModuleName, FuncName, fromaddr, toaddrs, subject,
msgtext, username, password)
*   check the outcome and CRT exception and traceback information
*   if an exception was raised
IF @PYEXCEPTIONTYPE = '' THEN
```

```
   CRT "Successfully sent the message to ":toaddrs
END ELSE
   CRT "EXCEPTION TYPE IS " :@PYEXCEPTIONTYPE
   CRT "EXCEPTION MESSAGE IS " :@PYEXCEPTIONMSG
   CRT "EXCEPTIONTRACEBACK IS " :@PYEXCEPTIONTRACEBACK
END
*********************************************************
```

Python Module – Notification.py:

```
#####################################################
# This is an example of sending an email message using Google
# GMail SMTP server.
# All input parameters: From Address, To Address, Subject,
# Message Text, GMail username and password, are expected to be
# in a string format.

def SendAlert( fromaddr, toaddrs, subject, msgtext, username, password):
    import smtplib

    # Turn the input string of comma-separated recipient addresses
    # into a list as required by sendmail function toaddrs = toaddrs.split()

    # Note: The next line may word-wrap
    toaddrs = toaddrs.split()
    message = """\From: %s\nTo: %s\nSubject: %s\n\n%s """ \
    % (fromaddr, ", ".join(toaddrs), subject, msgtext)

    # End of 'message' line

    server = smtplib.SMTP('smtp.gmail.com:587')
    server.ehlo()
    server.starttls()

    # Provide GMail user and password information
    server.login(username,password)
    server.sendmail(fromaddr, toaddrs, message)
    server.quit()

#####################################################
```

Result:

```
>Run PBP SENDALERT_PYFUNC
Enter Sender's Address: myco@co.com
Enter Recipient's Address: email@address.com
Enter Message Subject: Testing Python in UV
Enter Message Body: Text of Body
(Authentication) Enter gmail username:
(Authentication) Enter user password:
Successfully sent the message to email@address.com
```

Example 3: LARGENUM_TEST

A U2 BASIC program, LARGENUM_TEST, shows that a higher precision can be achieved in Python when working with decimal numbers than in U2 BASIC. It first calculates a square root using U2 BASIC `SQRT` function and then by calling Python function `getsqrt`().

U2 BASIC Program – LARGENUM_TEST:

```
*************************************************************************
*
*   Program:    LARGENUM_TEST
*   Desc:   This sample program calculates square root of a large decimal number,
*   first using U2 SQRT function and then by calling a Python function.
*   Precision is lost when using U2 SQRT due to the use of floating
*   point calculations. The Python function is able to return a better
*   precision through the use of Decimal module and higher precision
*   specification.
*
*   Python function getsqrt() takes a number as an input and
*   returns its square root. Note the use of getcontext().prec = 64
*   which overrides the default precision of 28.
*
*************************************************************************
NMBR = "12345678901234567894.1111111"
CRT "NMBR=" :NMBR
CRT "U2 RESULT: "
CRT "SQRT(NMBR)= " :SQRT(NMBR)

ModuleName = "largenum_sqrt"
FuncName = "getsqrt"

*   call the Python function
pyresult = PyCallFunction(ModuleName, FuncName, NMBR)

*   check the outcome and CRT exception and traceback information
*   if an exception was raised
IF @PYEXCEPTIONTYPE = '' THEN
  CRT "Python RESULT: "
  CRT "SQRT(NMBR)= " :pyresult
END ELSE
  CRT "EXCEPTION TYPE IS " :@PYEXCEPTIONTYPE
  CRT "EXCEPTION MESSAGE IS " :@PYEXCEPTIONMSG
  CRT "EXCEPTIONTRACEBACK IS " :@PYEXCEPTIONTRACEBACK
END
*************************************************************************
```

Python Module – `largenum_sqrt.py`:

```
################################################
from decimal import *
getcontext().prec = 64

def getsqrt( num ):
    num = Decimal(num)
    d = num**Decimal('.5')
    return str(d)
################################################
```

Result:

```
>RUN PBP LARGENUM_TEST
NMBR-12345678901234567894.1111111
U2 RESULT:
SQRT(NMBR)= 3513641828.8201
Python RESULT:
SQRT(NMBR)= 3513641828.820144253678675540738389727207403350606497150292801683
```

Chapter 4: Writing Python programs that access U2

After you are familiar with the Python language, you can write Python programs that access currently existing U2 BASIC programs.

u2py functionality

At the Python prompt, you can perform developer tasks such as calling U2 BASIC cataloged subroutines, run TCL and ECL commands, and more.

- Importing u2py
- Calling U2 BASIC cataloged subroutines
- Running U2 TCL and ECL commands
- Reading and writing U2 files
- Handling U2 dynamic arrays
- Managing U2 SELECT lists
- Controlling U2 transactions

Importing u2py

After entering the Python interactive shell, you can import the Python extension module and query its capabilities.

At the prompt (`python>` for Windows or `>>>` for UNIX), enter `import u2py`.

The u2py extension module is imported and you can start writing Python programs that access U2.

If you need help or information about objects, the next steps are:

- Accessing u2py help, on page 24
- Accessing u2py object attributes, on page 25

Parent topic: u2py functionality

Accessing u2py help

Use the built-in Python help system to see information about a module, function, class, method, keyword, or more.

To view the generated documentation for u2py at the Python prompt, enter `help(u2py)`.

Note: The `u2py` subroutine only passes string-type parameters to the BASIC subroutine. If a parameter is not a string type, `u2py` will get a string representation of it and then pass the string value to the BASIC subroutine.

The built-in help displays, as shown in the following example:

```
>>> help(u2py)
Help on module u2py:

NAME
    u2py

DESCRIPTION
    U2 module for Python, it allows Python developers to
    1) call U2BASIC catalogued subroutines
    2) run U@ ECL/TCL commands
    3) read/write U2 files
    4) handle U2 dynamic arrays
    5) manage U2 SELECT list
    6) control U2 transactions
...
```

You can view the help about any of the Python functionalities by entering the name in the parenthesis, as shown in the following example:

```
python> help(u2py.DynArray)
Help on class DynArray in module u2py:

class DynArray(_u2py._DynArray)
 |  DynArray([arg]) -> new U2PY DynArray object -- to support manipulation of MV
 delimited data, stores data internally as bytes
 |
 |  DynArray() -- an empty DynArray object
 |  DynArray(arg) -- an DynArray object with its value set to arg's bytes/string
 representation
 |
 |  DynArray is iterable. The iterator returns a Python tuple object with two it
ems (v, d) :
 |     v is the dynamic array element extracted;
 |     d is the system delimiter found: 0 End of dynamic array, 1 Item mark, 2 F
ield mark, 3 Value mark, 4 Subvalue mark, 5 Text mark.
 |
 |  Method resolution order:
 |      DynArray
 |      _u2py._DynArray
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __init__(self, *args)
-- More  --
```

Accessing u2py object attributes

View information about the valid attributes for u2py.

To view the directory of all attributes for u2py at the Python prompt, enter `dir(u2py)`.

A list of attributes for u2py displays, as shown in the following example:

```
>>> dir(u2py)
['Command', 'DATA_FILE', 'DICT_FILE', 'DynArray', 'FM', 'File', 'IM', 'LOCK_EXCLUSIVE',
'LOCK_RETAIN', 'LOCK_SHARED', 'LOCK_WAIT', 'List', 'SM', 'Subroutine',
'TM', 'U2Error', 'VM', '__doc__', '__file__', '__loader__', '__name__', '__package__',
'__spec__', 'call', 'clearlocks', 'config', 'run', 'tx_commit', 'tx_getlevel',
```

```
'tx_isactive', 'tx_rollback', 'tx_start']
```

You can view the directory attributes about any of the Python functionalities by entering the name in the parenthesis, as shown in the following example:

```
python> dir(u2py.DynArray)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__form
at__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__iter__
', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduc
e_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
 '__weakref__', '_bytes', '_normalize_list', 'alpha', 'convert', 'count', 'dcoun
t', 'delete', 'extract', 'field', 'fieldstore', 'format', 'iconv', 'insert', 'lo
cate', 'lowerdelim', 'next', 'oconv', 'raisedelim', 'replace', 'to_list']
python>
```

Calling U2 BASIC cataloged subroutines

The following examples show the convenience syntax and the standard syntax for calling a U2 BASIC cataloged subroutine on UNIX.

Subroutine:

```
SUBROUTINE MYSUBROUTINE(p1, p2)
p2 = "Hello " : p1
RETURN
END
```

Convenience syntax:

```
>>> import u2py
>>> u2py.call("MYSUBROUTINE", "Charlie", None)
[<u2py.DynArray value=b'Charlie'>, <u2py.DynArray value=b'Hello Charlie'>]
```

Standard syntax:

```
>>> sub = u2py.Subroutine("MYSUBROUTINE", 2)
>>> sub.args[0]="Charlie"
>>> sub.call()
>>> sub.args
[<u2py.DynArray value=b'Charlie'>, <u2py.DynArray value=b'Hello Charlie'>]
```

Parent topic: u2py functionality

Running U2 TCL and ECL commands

The `u2py.run` and `u2py.Command.run` methods are used for running U2 TCL and ECL commands.

u2py.run

u2py.run(*cmdtext*, capture=False|True)

cmdtext is the TCL or ECL command. Set capture to True to return the output of the command as a string. The default value of capture is False, for example:

```
>>> u2py.run("WHO")
```

```
7 XDEMO From PYSERVER\admin
```

u2py.Command.run

```
u2py.Command.run([capture=False|True])
```

Set capture to True to return the output of the command as a string. The default value of capture is False, for example:

```
>>>
>>> cmd = u2py.Command("WHO")
>>> cmd_output = cmd.run(capture=True)
>>> cmd_output
'1 XDEMO From cbrown\n'
>>>
```

Parent topic: u2py functionality

Reading and writing U2 files

The following example shows the standard syntax for reading and writing U2 files.

```
import u2py

f = u2py.File("CUSTOMER")
r = f.read("7")
r = r.replace(1,"Mike")
f.write( "7", r )
f.write( "new", r)
f.delete( "7" )
```

Parent topic: u2py functionality

Handling U2 dynamic arrays

Use the DynArray class on the u2py module to convert U2-delimited data as a dynamic array.

Syntax

```
u2py.DynArray(arg)
```

For *arg*, enter a value set to the argument's bytes or string representation. The data will be stored internally as bytes. With no argument entered, the object is an empty DynArray object.

The following example shows the standard syntax for handling U2 dynamic arrays.

```
theArray = u2py.DynArray()
theArray.insert(1,0,0,"charlie")
tmp = theArray.extract(1,0.0)
u2py.DynArray("10/14/2014").iconv("d4/")
```

The first line shows the dynamic array being created. The second line shows it being added to, then extracted from at the third line. The fourth line manipulates the dynamic array.

For help about the DynArray class, enter `help(u2py.DynArray)` at the Python prompt. Information about the DynArray class is displayed, including information about converting a dynamic array to a Python list. For example, the `to_list()` method splits the DynArray object into a Python list object:

```
>>> d = u2py.DynArray(b'A' + u2py.FM + b'B' + u2py.FM + b'C1' + u2py.VM + b'C2'
+ u2py.FM + b'D')
>>> d
<u2py.DynArray value=b'A\xfeB\xfeC1\xfdC2\xfeD'>
>>> d.to_list()
['A', 'B', ['C1', 'C2'], 'D']
```

Parent topic: u2py functionality

Managing U2 SELECT lists

The following example shows the standard syntax for handling U2 `SELECT` lists.

```
mcmd = u2py.Command( "SELECT VOC")
mcmd.run()
myList = u2py.List(0)
for x in range(0, 5):
    id = myList.next()
    print( id )

myList.clear()
```

Parent topic: u2py functionality

Controlling U2 transactions

The following table describes the commands you can use to control U2 transactions.

Command	Description
`tx_commit()`	Commit the current transaction.
`tx_getlevel()`	Get the current transaction level (integer).
`tx_isactive()`	Check whether inside a transaction (boolean).
`tx_rollback()`	Rollback the current transaction.
`tx_start()`	Start a transaction.

Parent topic: u2py functionality

Python and NLS

In Python 3.0 and later, string data is stored as Unicode characters. U2 engines (except for UniVerse in NLS mode), on the other hand, store and process strings and characters as bytes in a particular language encoding. This means when string data is passed from Python to U2, it must be converted to the native encoding in order to be correctly handled by U2.

When bytes data is passed from U2 to Python, it will be converted to Unicode strings; when bytes data is passed from Python to U2, no conversion is needed. When Unicode strings are passed from Python to U2, they will be converted to the native language encoding so that U2 can consume them.

U2 native encoding

Since the native encoding which U2 engines uses is not always known, U2 Python assumes that it is the same as the current operating system's input and output encoding. An overriding mechanism is available for users to set the U2 native encoding manually.

Specifically, in the `u2py` module, a property `config.encoding` is added, the value of which is set by default to the current operating system's input and output encoding. In most cases, the property will work as intended, but it is also programmatically changeable. So if the property does not match the native encoding the U2 engine is using and causes issues for the Python applications, the Python application developer can always change it.

Examples

The following sections show some examples of Python programs accessing U2.

Accepting input from the DATA statement

Many programmers have BASIC subroutines and/or programs that accept standard input. One way to automate the call and run these programs is to set the data that will be used in the input as part of the `DATA` statement. For more information, see the section about the `DATA` statement in the *BASIC Commands Reference*.

The following subroutine puts information into the `DATA` statement:

```
>CT BP SET_DATA
     SET_DATA
0001 SUBROUTINE SET_DATA( THE_DATA )
0002 DATA THE_DATA
0003 RETURN
```

Another subroutine is used that accepts input:

```
>CT BP INPUT_TEST
     INPUT_TEST
0001 SUBROUTINE INPUT_TEST( GET_DATA )
0002 INPUT GET_DATA
0003 RETURN
```

The following example calls these routines in Python:

```
>PYTHON
python> import u2py
python> setData = u2py.Subroutine("SET_DATA", 1)
python> setData.args[0] = "test data"
python> setData.call()
python> theTest = u2py.Subroutine("INPUT_TEST", 1)
python> theTest.args[0] = ""
python> theTest.call()
?test data
```

```
python> print(theTest.args[0])
test data
```

Setting up parameters in a U2 process

The `NAMED COMMON` program is a way for U2 users to set up parameters that stay active for the life of the U2 process.

```
CT BP SET_NAMED_COMMON
BP:
SET_NAMED_COMMON
*
COMMON /MYNAME/ ARG1, ARG2, ARG3
ARG1 = "ONE"
ARG2 = "TWO"
ARG3 = "THREE"
END
:CT BP CHECK_NAMED_COMMON
BP:
CHECK_NAMED_COMMON
*
COMMON /MYNAME/ ARG1, ARG2, ARG3
PRINT "ARG1 = ":ARG1
PRINT "ARG2 = ":ARG2
PRINT "ARG3 = ":ARG3
END
```

The following code shows these two programs from ECL/TCL:

```
:SET_NAMED_COMMON
:CHECK_NAMED_COMMON
ARG1 = ONE
ARG2 = TWO
ARG3 = THREE
And from Python:
python> import u2py
python> set_named_common = u2py.Command("SET_NAMED_COMMON")
python> set_named_common.run()
python> check_named_common = u2py.Command("CHECK_NAMED_COMMON")
python> check_named_common.run()
ARG1 = ONE
ARG2 = TWO
ARG3 = THREE
```