Rocket

U2 Python Lab Guide

Education Lab

Version 2.220

May 2016

Preface
Lab overview

Abstract and environment

This lab is intended to give you hands-on training using Python and the u2py module for accessing a U2 database. This lab covers how U2 BASIC can access Python.

The lab environment uses the following:

- UniData 8.2 Beta, UniVerse 11.3 Beta
- Basic Developer Toolkit (from U2 DBTools)
- PyCharm (or the Python editor of your choice – Notepad++ would work, but be careful to use spaces not tabs when indenting)
- Windows
- XDEMO Accounts
- Programs and templates

There are four exercises in this lab:

- Exercise 1: Getting started with U2 Python
- Exercise 2: Learning the u2py module
- Exercise 3: Exceptions raised in u2py module
- Exercise 4: Basic access to Python

Part I: Exercise 1 - Getting Started with U2 Python

This exercise is a basic overview of the U2 Python environment. We will show how to start Python from U2 and execute some simple commands, as well as how to call Python programs from U2.

After this exercise you will be able to:

- Start the Python command prompt from U2 ECL/TCL
- Execute a Python program from U2 ECL/TCL
- Start and/or run Python from the OS command prompt
- Use PyCharm (or the editor of your choice) to edit and run Python code

1. Install the UniData or UniVerse Python beta product. It comes with an XDEMO account installed.
2. Add the extra files and records provided in the lab zip file.
   a. From a UDT or UniVerse Shell (U2 Shell) execute the following statements:

      UniData

      ```
      :LOGTO XDEMO
      :CREATE.FILE DIR PP_SOLUTIONS
      :CREATE.FILE DIR BP_SOLUTIONS
      ```

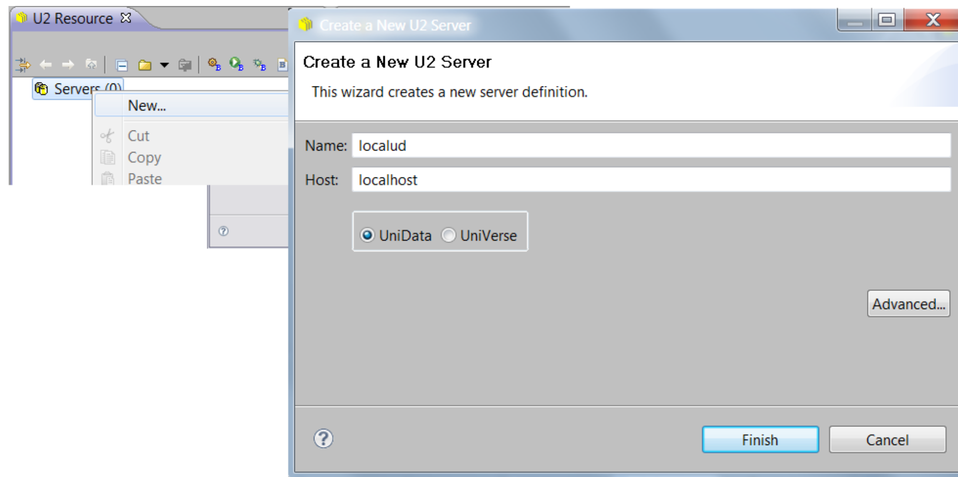
      UniVerse

      ```
      >LOGTO XDEMO
      >CREATE.FILE PP_SOLUTIONS 19
      >CREATE.FILE BP_SOLUTIONS 19
      ```

   b. Unzip the lab file. There should be four folders in the zip file: `BP`, `BP_SOLUTIONS`, `PP`, and `PP_SOLUTIONS`.
   c. Copy all records from each unzipped folder to the appropriate directory in the XDEMO account.
   d. From the U2 Shell, execute the following statements:

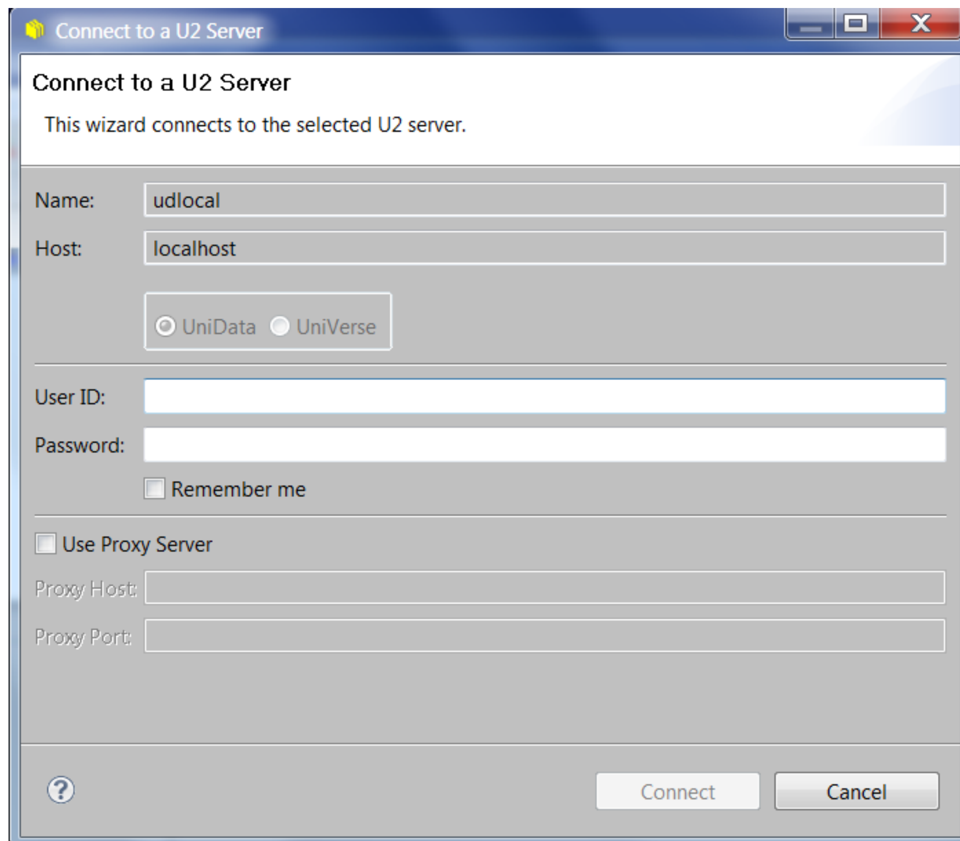
      ```
      LOGTO XDEMO
      BASIC BP SIMPLE_SUBROUTINE
      CATALOG BP SIMPLE_SUBROUTINE
      ```

3. Start Basic Developer Toolkit (BDT). If you do not have it, you can download the U2 DBTools and install it. To start BDT, select **START**> **All Programs** > **Rocket U2**> **Basic Developer Toolkit**.
4. In BDT, create a U2 Server Definition.

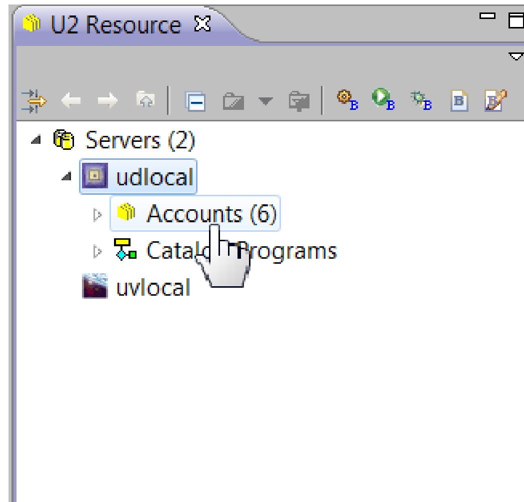


a. Use any name.
b. Host is `localhost`.
c. Select UniData or UniVerse.
d. Click **Finish**.

5. Connect to U2 Server.
   a. Select the UniData or UniVerse server.
   b. Double click or right-click **Connect**.
   c. Enter the User ID and Password for your login.
   d. Click **Connect**.



6. Explore XDEMO in U2 Resources.
   a. Select your U2 Server.
   b. Click **>** to expand **Accounts**.
   c. Click on **XDEMO**.
   d. Explore the Database Files.



The following are important files:

- BP – Basic programs used in examples
- PP – Python programs used in examples
- BP_SOLUTIONS - Basic Program Solutions
- PP_SOLUTIONS - Python Program Solutions

7. Run Python from a U2 session.
   a. From the U2 Shell environment opened in Step 2, LOGTO XDEMO.
   b. Create a RELP like environment at the ECL/TCL prompt by entering: **PYTHON**
   c. Enter the following command:

```
python> print('Hello Python')
Hello Python
```

   d. Enter **quit** to return to the ECL/TCL command line.

8. Edit a Python program.
   a. In BDT, from the XDEMO account, right-click on **PP** and select **New Basic Program**. (Yes, we know there is no option to create a Python program…yet!)
   b. In the **New Basic Program** window make sure the PP file is selected.
   c. Enter a name for the program (i.e. exercise_1.py).
   d. Make sure **Use Template** is blank.
   e. Click **Finish**.
   f. Enter the following lines in the editor for the program file you just created:

```
print('This is a simple Hello World program')
print('')
yourName = input('Enter your name: ')
print('')
print('Hello ' + yourName )
```

g. Press <Ctrl>+s to save your work.

9. Run your Python program from U2.

   a. In your U2 Database Shell window execute the following at ECL/TCL:

```
: RUNPY PP exercise_1.py
or
> RUNPY PP exercise_1.py
```

   Congratulations! You just wrote and executed a program from Python.

10. Run the program from the OS command prompt.

    a. Start a command prompt by clicking the **Start** icon, and entering `cmd` in the **Search programs and files** text box.

    b. In the cmd prompt window, change directory to the U2 XDEMO account.

```
'cd \U2\ud82\XDEMO' – for UniData
'cd \U2\UV\XDEMO' – for UniVerse
```

Note:

Note that Python was installed as part of the U2 releases, and is in a directory called `python` in the U2 home directory.

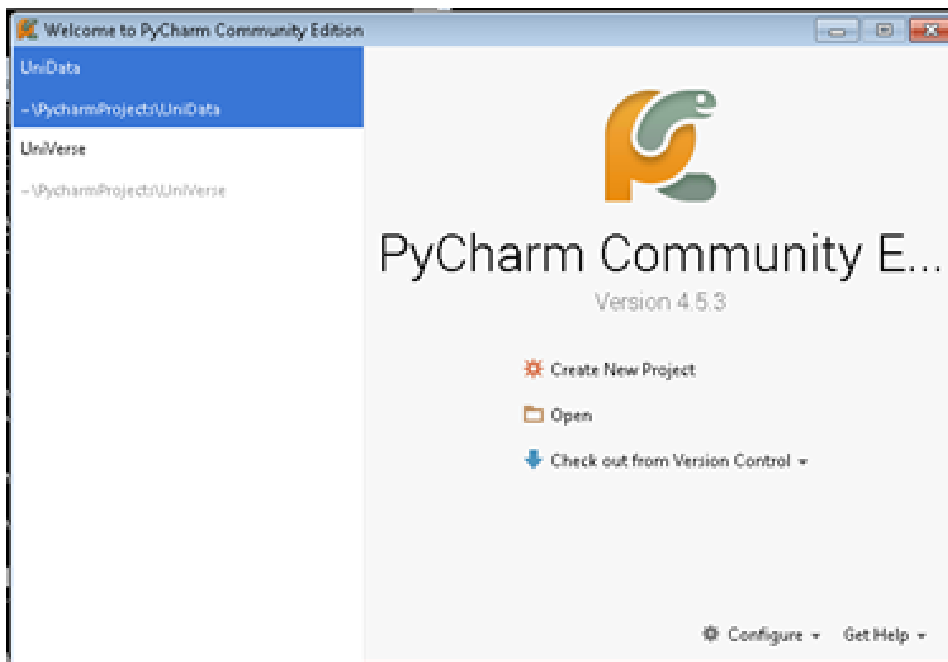
---

    c. Run the Python program from the OS command prompt.

```
..\python\python.exe PP\exercise_1.py
```

    Congratulations! You just executed a program from a cmd prompt.

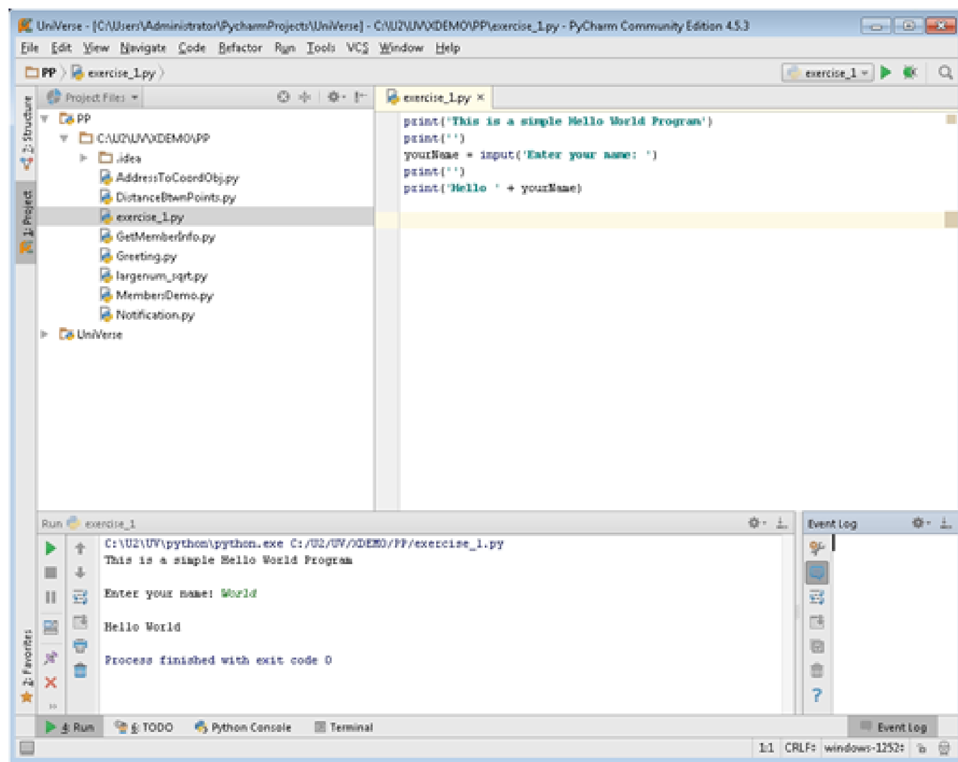
11. Open PyCharm or the Python editor of your choice (as long as it allows program execution) and create a project that points to the `U2 home directory\python\python.exe` for its interpreter and to XDEMO for its Python code.



12. Run the program from PyCharm (or your selected tool).

    a. Start PyCharm by clicking on the icon in the task bar.

b. Click **Run** and select exercise_1.



Part II: Exercise 2 - Learning the u2py Module

The u2py module is provided to allow Python developers to interact with the U2 Databases. This module will give a basic overview of the module and the object and methods it contains. In this exercise you will create several small programs that use these objects and methods.

After this exercise you will be able to:

- Import the `u2py` module and display the online help
- Call a U2 Basic subroutine
- Execute a U2 command from Python
- Work with U2 dynamic arrays in Python
- Understand U2 file handling from Python (open, read, write and delete items)

1. Enter your UniData or UniVerse Shell.
2. At the ECL (:) or TCL (>) prompt enter **PYTHON**.
3. Import the `u2py` module.

   ```
   python> import u2py
   ```

   When you want Python to have access to the U2 Database running on the server, you will first need to import the `u2py` module. If you leave the Python prompt and then come back to it, you will need to import the `u2py` module again.
4. Get help on `u2py`. As with any Python module, you can get help on the objects and methods by invoking `help` for the module.

   ```
   python> 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 U2 ECL/TCL commands
     3) read/write U2 files
     4) handle U2 dynamic arrays
     5) manage U2 SELECT list
     6) control U2 transactions
   CLASSES
     builtins.Exception(builtins.BaseException)
       U2Error
     builtins.object
       Command
       DynArray
       File
       List
         Subroutine
   …
   …
   ```

   Run the command on your lab to see the entire help output.

Note:

For the next few steps, you can either execute the statements from a Python command prompt, or create/edit a Python program in `PP` so you can easily re-execute them.

5. Call a U2 BASIC subroutine. The `u2py.Subroutine` object is a powerful tool in your U2 Python development environment. It allows you to leverage existing routines, and bring the data back into Python.
   a. The following is from the Python help for the `u2py` module, using the command `help(u2py.Subroutine).`

```
class Subroutine(builtins.object)
 | Subroutine(name, pnum) -> new Subroutine object -- to support the execution
of U2 programs
 |
 | name -- the name of a catalogued U2BASIC subroutine.
 | pnum -- the number of parameters that the subroutine requires.
 |
 | Methods defined here:
 |
 | __init__(self, /, *args, **kwargs)
 |   Initialize self. See help(type(self)) for accurate signature.
 |
 | __new__(*args, **kwargs) from builtins.type
 |   Create and return a new object. See help(type) for accurate signature.
 |
 | __repr__(self, /)
 |   Return repr(self).
 |
 | call(...)
 |   call() -> None -- call the catalogued subroutine and bring back any
changes to the arguments
 |
 | ----------------------------------------------------------------
```

   b. You should have cataloged the `SIMPLE_SUBROUTINE` when you installed the lab files. The subroutine takes text and applies a conversion code to do title casing. Enter the following commands at the Python command prompt.

```
python> import u2py
python> mySub = u2py.Subroutine("SIMPLE_SUBROUTINE", 2)
python> mySub.args[0] = ""
python> mySub.args[1] = "rocket software"
python> mySub.call()
python> output = mySub.args[0]
python> print(output)
Rocket Software- output with title case
```

Note:

Once you have the `u2py.Subroutine` object, you can keep modifying and using the call method.

```
python> mySub.args[1] = "john mcdoe iii"
python> mySub.call()
python> print(str(mySub.args[0]))
John Mcdoe Iii- output with title case
```

c. Create `exercise_2_1.py` file in `PP` and add the previous commands to it. Run it from either the OS command line or the TCL/ECL command line, as we saw in the previous exercise.

6. Execute a U2 statement.

a. The `u2py.Command` is also powerful, and allows you to execute U2 Commands as well as programs. You have the ability to save or use the output from the command in your program. The following is from the Python help for the `u2py` module, using the command `help(u2py.Command)`:

```
class Command(_u2py._Command)
 |  Command(cmdtext) -> new Command object -- to support the execution
of U2 com
mands
 |
 |  cmdtext -- ECL/TCL command text.
 |
 |  Method resolution order:
 |      Command
 |      _u2py._Command
 |      builtins.object
 |
 |  Methods defined here:
 |
 |  __repr__(self)
 |
 |  run(self, capture=False)
 |      run([capture=False]) -> None -- run the TCL/ECL command, similar to
running a command from U2 prompt
 |      if capture is True, return the output of the command as a string
 |
```

b. Modify the `exercise_2_2.py` program in PP file with the following:

UniData

```
cmd = u2py.Command("LISTUSER")
```

UniVerse

```
cmd = u2py.Command("LISTU")
```

c. Run the `exercise_2_2.py` program.

Note:

The `run` method in this example has no parameters, which means it defaults to False for capturing the output. You see the output as a direct outcome of the execution of the run. The output from the command is `None`. Do you want your user to see this output immediately on their screen or do you want to bring the output back to the program to work with it there?

d. To capture the output, change `.run` to `.run(True)`. Run the `exercise_2_2.py` program again.

Note:

You will not see any standard messages at the execution of the run. Look at the output from the program, this is the captured output being printed from the program.

e. Edit your program to allow the user to input the command to execute. Set the capture status you want – True or False.

```
usrcmd = input('Enter command you want to execute: ')

# change the text to a TCL command
cmd = u2py.Command(usrcmd)
```

7. Instantiate a U2 Dynamic Array object in Python.

Note:

The `u2py` module has a `DynArray` object. It not only provides you with the access methods to get information into and out of the dynamic array but also provides many of the multiValue functions as methods.

```
python> import u2py
python> myDArray = u2py.DynArray("")
python> myDArray
<u2py.DynArray value=b''>
```

Note:

An empty array was created.

You can use any Python variable that is a list.

```
python> A = [ "python", "simple", "list" ]
python> myDArray = u2py.DynArray( A )
python> myDArray
<u2py.DynArray value=b'python\xfesimple\xfelist'>
```

Note:

The \xfe is the escape sequence to represent a char(254) in the data.

We can take a `DynArray` and turn it back to a Python `list`.

```
python> B = myDArray.to_list()
python> B
['python', 'simple', 'list']
```

8. Open a file and read a record from Python to populate a U2 Dynamic Array with the read.

a. View the Python help for `u2py.File`.

```
class File(builtins.object)
 | File(name, [type]) -> new U2PY File object -- to support U2 file I/O
 |
 | name is the VOC name of a U2 hashed/DIR file
 | type is either DATA_FILE (default) or DICT_FILE.
 |
```

b. Create a File object using `u2py.File` method to open a U2 file.

```
python> myFile = u2py.File("MEMBERS")
```

9. Get a U2 Dynamic Array object from a file read. We use the read method of the `u2py.File` object to return `u2py.DynArray`.

```
python> myrec = myFile.read("0001")
python> myrec
```

Note:

The variable *myrec* by itself prints the value of the dynamic array.

10. Extract from a U2 Dynamic Array object. There are several ways to extract the data from the `u2py.DynArray`. The most direct is the `extract` method (using the variable created previously).

```
python> birthDate = myrec.extract(10, 0, 0)
python> birthDate
<u2py.DynArray value=b'2173'>
```

Since it is in internal format, the `oconv` method needs to be used to get it in output format.

```
python> print(birthDate.oconv("D-"))
12-12-1973
```

11. Replace data in a U2 Dynamic Array object. Replace the birth date in *myrec* by adding 1 day to it (12-13-1973) in the dynamic array.

```
python> myrec.replace(10,0,0,int(birthDate) + 1)
python> birthDate = myrec.extract(10, 0, 0)
python> birthDate
<u2py.DynArray value=b'2174'>
```

Since birthDate is a Dynamic Array, we needed to cast it as an integer before adding one to its value. This does not change the record, it only updates the U2 Dynamic Array object.

12. Read from a U2 file using Python.

a. Get help on method descriptor in `u2py.File.read`.

```
python> help(u2py.File.read)
F.read(recordid, [lockflag]) -> new DynArray object -- read a record in
the file
lockflag is either 0 (default), or [LOCK_EXCLUSIVE or LOCK_SHARED]
 [ + LOCK_WAIT]
```

b. In steps 8 and 9 we used the `u2py.File.read` method to get a dynamic array.

```
python> myFile = u2py.File("MEMBERS")
python> myrec = myFile.read("0001")
python> myrec
```

To set a lock while an item is read, add the optional lockflag:

UniVerse

```
python> myRec = myFile.read("0001", u2py.LOCK_EXCLUSIVE)
python> listReadu = u2py.Command("LIST.READU EVERY")
python> listReadu.run()
```

UniData

```
python> myRec = myFile.read("0001", u2py.LOCK_EXCLUSIVE)
```

```
python> listReadu = u2py.Command("LIST.READU")
python> listReadu.run()
```

Resulting in:

```
UniVerse:
Active Group Locks:                    Record Group Group Group

Device.... Inode..... Netnode Userno  Lmode G-Address. Locks ...RD ...SH ...EX

 851688904 4930231204    0   2  53 IN    2000   1   0   0   0


Active Record Locks:
Device.... Inode..... Netnode Userno  Lmode    Pid Item-ID................
...
 851688904 4930231204    0   2  53 RU    3008 0001


UniData:
 UNO UNBR   UID  UNAME   TTY   FILENAME RECORD_ID M   TIME  DATE
  1 3916     0 Administ Console   MEMBERS    0001 X 14:45:50 Aug 12
```

c. Try reading and setting a lock from another process.

```
python> import u2py
python> myFile = u2py.File("MEMBERS")
python> myRec = myFile.read("0001", u2py.LOCK_EXCLUSIVE)
Traceback (most recent call last):
 File "<console>", line 1, in <module>
u2py.U2Error: (30002) File or record is locked by another user
```

Note:

Exceptions are covered in greater detail in the next exercise.

13. Write to a U2 file from Python.

a. Get help on method descriptor in `u2py.File.write`.

```
F.write(recordid, record, [lockflag]) -> None -- write a record to the file
  lockflag is either 0, LOCK_RETAIN, LOCK_WAIT, LOCK_RETAIN + LOCK_WAIT.
```

b. Using the `u2py.DynArray` object you created with the read in the previous step, write a new item to the file.

```
python> myrec = myFile.write("9999", myrec)
```

You can verify that it wrote by reading it to a new object.

```
python> check = myFile.read("9999")
python> check
<u2py.DynArray value=b'Adcock\xfeLinda\xfeQ\xfeMrs\xfe4446 Andrea Street\xfe
Inglewood\xfeMA\xfe25580\xfeF\xfe2173\xfe3523000189\xfe3520185630\xfe
ladcock@mv.rs.com\xfekiller\xfeU2FsdGVkX1/tOGQj4DZvG3HgFCzRSbBOsqPN+gQPhXc=\n
\xfeladcock.jpg\xfeA\xfe9165766677843519\xfd1956344422067412\xfd
6411999977522077\xfeU2FsdGVkX1/Pxwv5Wfv7Cd5mNbjc9E710B76MRqRFfToxUPlEZF/
v+zrz2bom5Sv\n\xfdU2FsdGVkX18gLCRPUoUsMKjmy/
1bR5uUexCo61KmpknRQjCYcAj3HGKPL4l6AYTE\n\xfdU2FsdGVkX1/dJCjuvGLT+/
afct3NZNQh79qIh/iBzakWB/vRyUcm81PWD2jJPcij\n\xfeAMEX\xfdMC\xfdAMEX\xfe16954\xfd
18111\xfd18142\xfe530\xfd185\xfd641'>
```

14. Delete an item from the U2 file using Python.
    a. Get help on method_descriptor in `u2py.File.delete`.

    ```
    >python help(u2py.File.delete)
    F.delete(recordid, [lockflag]) -> None -- delete a record in the file

    lockflag is either 0 (default), LOCK_RETAIN, or LOCK_RETAIN + LOCK_WAIT.
    ```

    b. Delete the item created in the previous step.

    ```
    python> myFile.delete("9999")
    ```

    c. Try to delete the same item again, and notice the exception that is raised. We handle this later in the next exercise.

    ```
    python> myFile.delete("9999")
    Traceback (most recent call last):
    File "<console>", line 1, in <module>
    u2py.U2Error: (30001) Record not found
    ```

Part III: Exercise 3 - Exceptions in the u2py module

In exercise 2 we briefly covered several of the objects in the `u2py` module. One thing that was not covered is what happens when there are issues with the method you are trying to execute. Python handles issues in a module by raising an exception in the `u2py` module.

After this exercise you will be able to:

- Write Python code that deals with raised exceptions
- Know how to handle `u2py` exceptions

1. Try the following at the Python prompt.
   a. Assume you are trying to convert a variable to an integer.

```
python> a = 10
python> int(a)
python> a = "text"
python> int(a)
Traceback (most recent call last):
File "<console>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'text'
```

   As you see above, an error occurred. Since it was not handled, the execution stopped, and the error message displayed. Note that the error was a ValueError. In order to add code to handle the error, it will be easier to use an editor and create a function rather than doing all the necessary indentation at the Python prompt in ECL/TCL.

   b. The try/except structure is Python's way of handling exceptions. Create a module in the `PP` file and call it `convertException.py`.

   c. Enter the following function called `convert` in `convertException.py`, which includes exception handling.

```
def convert(s):
    ''' Convert to an integer '''
    try:
        x = int(s)
        print("Conversion succeeded!")
    except ValueError:
        print("Conversion failed!")
        x = -1
    return x
```

   Note this only handles a ValueError.

   d. To use this function, the `PP` directory should be in the path for `u2py`, as designated in the `u2.pth` file. To accomplish this, go to the default location of the `u2.pth` file.

   - On Windows (`$UDTHOME\python or $UVHOME\python`)
   - On Linux (`$UDTHOME/python/lib/python3.4/site-packages` or `$UVHOME/python/lib/python3.4/site-packages`)

   You can add additional paths to the `pth` file.

   UniData

```
u2.pth
C:\U2\ud82\bin
C:\u2\ud82\XDEMO\PP
```

UniVerse

```
u2.pth
C:\U2\UV\bin
C:\U2\UV\XDEMO\PP
```

e. Restart your U2 Shell session to pick up the new path and import the `convert` function from the `convertException` module and use it.

```
python> from convertException import convert
python> convert(10)
Conversion succeeded!
10
python> convert("text")
Conversion failed!
-1
```

f. Now try with a list, rather than a number or string.

```
python> convert([4, 5, 6])
```

Now there is a different type of error.

```
Traceback (most recent call last):
File "<console>", line 1, in <module>
File "C:\U2\UV\XDEMO\PP\convertException.py",line 4, in convert
   x = int(s)
TypeError: int() argument must be a string or a number, not 'list'
```

g. Add another except condition to your `convert` function for TypeError and try this again. You will need to restart your Python session and import the `convert` function again.

```
except TypeError:
    print("Conversion failed - TypeError!")
    x = -1
```

h. Many of the methods provided in the `u2py` objects raise exceptions. Try a `u2py` command that raises an exception, like opening a bad file name, to see what happens.

```
python>import u2py
python> bad = u2py.File("INVALIDFILE")
Traceback (most recent call last):
File "<console>", line 1, in <module>
u2py.U2Error: (14002) No such file or directory
```

Understanding these exceptions and handling them in your code minimizes the chances of the program aborting.

2. Next, combine two things we have learned – functions and exception processing. As you know, you may need to do the same commands over and over again. By creating a module that has common functions, you can greatly simplify your development effort. Add exception handling to this function to complete the module. If you would like to exit the program when the exception occurs, you can use `sys.exit(0)`. If you are executing this function within a U2 Shell, when the `sys.exit(0)` condition occurs, the U2 shell session is ended. Add an `import sys` to use the `sys` function.

a. Build a function with the `def` statement. The following is a simple function in Python for reading a U2 record. Build this in the `PP` file and call the module `u2Func`. Notice we are handling the exception U2Error.

```
def readU2Item(thefile, itemid):
    import u2py
    import sys
    try:
        myFile = u2py.File(thefile)

    except u2py.U2Error as e:
        print("Open File error " + str(thefile))
        print(e)
        sys.exit(0)

    # read the item to returned_value variable
    try:
        rec = myFile.read(itemid)

    except u2py.U2Error:
        rec = u2py.DynArray("")

    returned_value = rec

    return returned_value
```

You are storing this function in the `PP` directory file in an item called `u2Func.py`. This `PP` directory should be in the path for `u2py` as designated by the `u2.pth` file, as we showed earlier.

b. Now build a Python program, `exercise_3.py` in your `PP` directory that imports the function and calls the method and prints out the resulting `u2DynArray`.

```
from u2Func import readU2Item
a = readU2Item("MEMBERS", "0001")
print(a)
```

3. When you have it working, alter this program so that the end user is prompted for the filename and @ID. You now have a Python function that can be used by any program that needs to read a particular record.

Part IV: Exercise 4 - Basic Access to Python

So far you have learned about using Python to access U2 files, programs and subroutines. In this exercise you will learn how U2 BASIC programs can call Python functions and access Python objects.

After this exercise you will be able to:

- Write a U2 BASIC program to call a Python function
- Set up the Python call in a U2 Subroutine
- Handle Python exceptions in U2 BASIC

Note:

When working with Python from U2 BASIC, the first step is to load the required module. This is done with the `pyImport` function.

```
pyresult = PyImport (moduelName)
moduleName: the name of the module to be imported
pyresult: a PYOBJECT variable pointing to the Python module object
```

Once you have loaded the Python module, you can access it with one of 5 functions:

- Get the value of an attribute of a Python object

```
pyresult = PyGetAttr (pyobject, attrName)

pyobject: a PYOBJECT variable pointing to a Python object
           attrName: the name of the attribute whose value to be retrieved
pyresult: could be either a standard U2 BASIC variable or a PYOBJECT variable
```

- Set the value of an attribute of a Python object

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

pyobject: a PYOBJECT variable pointing to a Python object
attrName: the name of the attribute whose value is to be set
value: a value expression that can be evaluated to a string, number or a PYOBJECT
pyresult: a integer value, -1: failure
```

- Call a Python callable object

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

pycallableobject: a PYOBJECT variable pointing to a Python object that is callable
such as a function object, class object, method object
arg1,arg2,…: the arguments to the callable Python object that either can be evaluated
to a string, number or a PYOBJECT
pyresult: : could be either a standard U2 BASIC variable or a PYOBJECT variable
```

Note: The next two items are Convenience functions. They allow quick calling of a Python function or a method.

- Call a Python function on a Python module

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

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 either can be evaluated
to a string, number or a PYOBJECT
pyresult: : could be either a standard U2 BASIC variable or a PYOBJECT variable
```

- Call a method on a Python object

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

pyobject: a PYOBJECT variable pointing to a Python object
methodName: the name of the method to be called, it must be defined on the class of
the object
arg1,arg2,…: the arguments to the method that either can be evaluated to a string,
number or a PYOBJECT
pyresult: : could be either a standard U2 BASIC variable or a PYOBJECT variable
```

1. Use the Rocket U2 Basic Developer Toolkit (or other editor) to review the `BASIC_EXAMPLE_1` program in the `BP` file.

```
ModuleName = "largenum_sqrt"
FuncName = "getsqrt"
*
* import the module
pymodule = PyImport(ModuleName)
*
PRINT "Enter the number ":
INPUT NMBR
pyresult = PyCallFunction(ModuleName, FuncName, NMBR)
*
PRINT pyresult
END
```

The U2 BASIC code is doing the same as the following Python commands.

```
python> import largenum_sqrt
python> NMBR = input("Enter the number")
Enter the number1234567
python> pyresult = largenum_sqrt.getsqrt(NMBR)
python> print(pyresult)
1111.110705555481541639651584896590489796399283230370185750625649
```

   a. Run the commands shown above from the Python shell
   b. Compile the `BASIC_EXAMPLE_1` program

   ```
   BASIC BP BASIC_EXAMPLE_1
   ```

   c. Run the `BASIC_EXAMPLE_1` program.

   ```
   RUN BP BASIC_EXAMPLE_1


   Enter the number ?1234567
   1111.110705555481541639651584896590489796399283230370185750625649
   ```

2. Turn BASIC_EXAMPLE_1 into a subroutine and call it BASIC_SUB_1. Don't forget to compile and catalog it. Add another BASIC program to call that subroutine. Call it EXERCISE_4_1 and place it in BP.
3. Just like in the Python code, any exceptions raised from the code can be captured in U2 BASIC. New BASIC @variables capture the details of a Python exception when one occurs in the Python code called from the BASIC program. Modify the BASIC_SUB_1 to check for exceptions, and display the exception information if not null (empty string).

```
@PYEXCEPTIONTYPE: a string, stores the exception type; if no exception is
 thrown, its value is an empty string

@PYEXCEPTIONMSG: a string, stores the detailed exception message; if no exception
 is thrown, its value is an empty string

@PYEXCEPTIONTRACEBACK: a string, stores the trackback of the exception; if no
exception is thrown, its value is an empty string
```

4. Re-catalog and rerun EXERCISE_4_1.
5. Check the type of Python results in U2 BASIC.

Note: When the results are returned to U2 BASIC, you may get a Python object. There is a new BASIC variable type, PYOBJECT. A PYOBJECT variable is used internally to receive a Python object (U2 BASIC does not have a way to tell the type of a variable). A PYOBJECT variable cannot be printed or otherwise manipulated in U2 BASIC, but it can be passed to another U2 BASIC Python API function.

   a. Modify program EXERCISE_4_2 in BP and complete the missing steps.

   Note: For this example, use the myType module that is in the PP directory. myType is an example of a module you might want to create for your system. In this module we create a Python list that we can retrieve for this exercise, but also contains the function that will tell us what type of an object it is. The final function will convert the object to string, making it able to print, if it can be converted.

Look at the help from the module

```
python>import myType
python>help(myType)
```

Congratulations - you have completed this lab!