Python Getting Started Guide

For Summit and BlackDiamond Products

Abstract: This document provides a quick tutorial for installing and setting up Python, introduces Python scripting, and discusses Python application creation, showing examples with both scripting and daemon.

Published: February 2015

Extreme Networks, Inc.
145 Rio Robles
San Jose, California 95134
Phone / +1 408.579.2800
Toll-free / +1 888.257.3000
www.extremenetworks.com

Contents

Disclaimer

Scripts can be found on Extreme's website and are provided free of charge by Extreme. We hope such scripts are helpful when used in conjunction with Extreme products and technology; however, scripts are provided simply as an accommodation and are not supported nor maintained by Extreme.

ANY SCRIPTS PROVIDED BY EXTREME ARE HEREBY PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL EXTREME OR ITS THIRD PARTY LICENSORS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE USE OR DISTRIBUTION OF SUCH SCRIPTS.

Introduction

Python is a very popular programming language and is gaining a lot of traction in the switching industry as well. Python version 2.7.3 has been integrated into EXOS since EXOS 15.6.1.

This document will detail installing a working Python environment on most laptop systems, including installing Python and its necessary tools to an IDE. This guide will try to present the very basics of Python language, for those who are new to the language. The aim of this guide is not to teach Python, and readers are strongly encouraged to consult existing resources.

If you are already familiar with Python, then you should skip the teaching sections and go directly to the EXOS CLI Modules section.

References

The following documents were used extensively in the preparation of this document:

- *ExtremeXOS User Guide* (Release 15.6)
- CLI Python Scripting
- Python User Scripting Functional Specification
- ExtremeXOS API Documentation (Release 15.7.1)
- https://docs.python.org/2/
- Several Books/eBooks on Python (many are free)

Installing Python

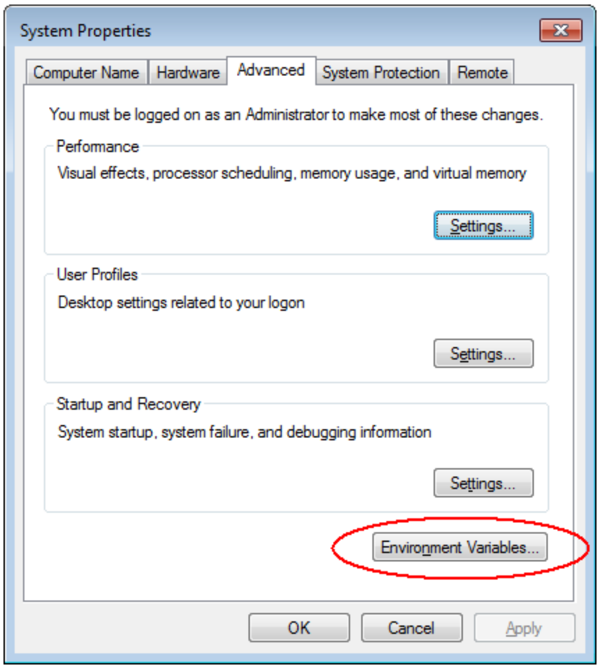
Windows

1. Download the windows MSI installer for the latest 2.7 version from the Python website: https://www.python.org/downloads/

NOTE

At the time of this writing, the 2.7.9 version is available at: https://www.python.org/downloads/release/python-279/

2. Follow the Python installer wizard, leaving the default directory (C:\Python27) unchanged for now.
3. When Python installation is complete, modify your computer's Path environment variable so that Python can be called from anywhere in the directory. To do this:
   a. From your computer's System Properties, (Control Panel > System > Advanced System Settings), click the Advanced tab.
   b. Click the Environment Variables button.



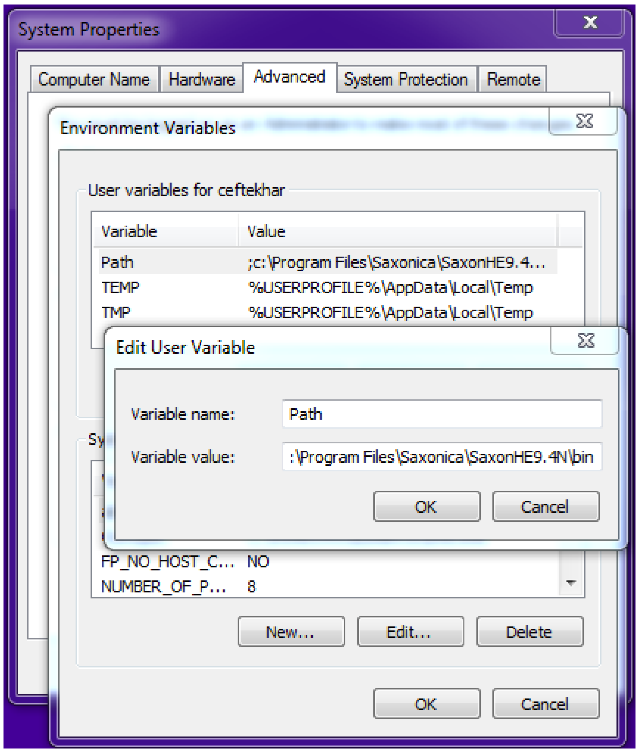
You must be logged on as an Administrator to make most of these changes.

Visual effects, processor scheduling, memory usage, and virtual memory

Desktop settings related to your logon

System startup, system failure, and debugging information

c. Select **Path** and click **Edit**.



User variables for ceftekhar

Variable	Value
Path	;c:\Program Files\Saxonica\SaxonHE9.4...
TEMP	%USERPROFILE%\AppData\Local\Temp
TMP	%USERPROFILE%\AppData\Local\Temp

Variable name: Path

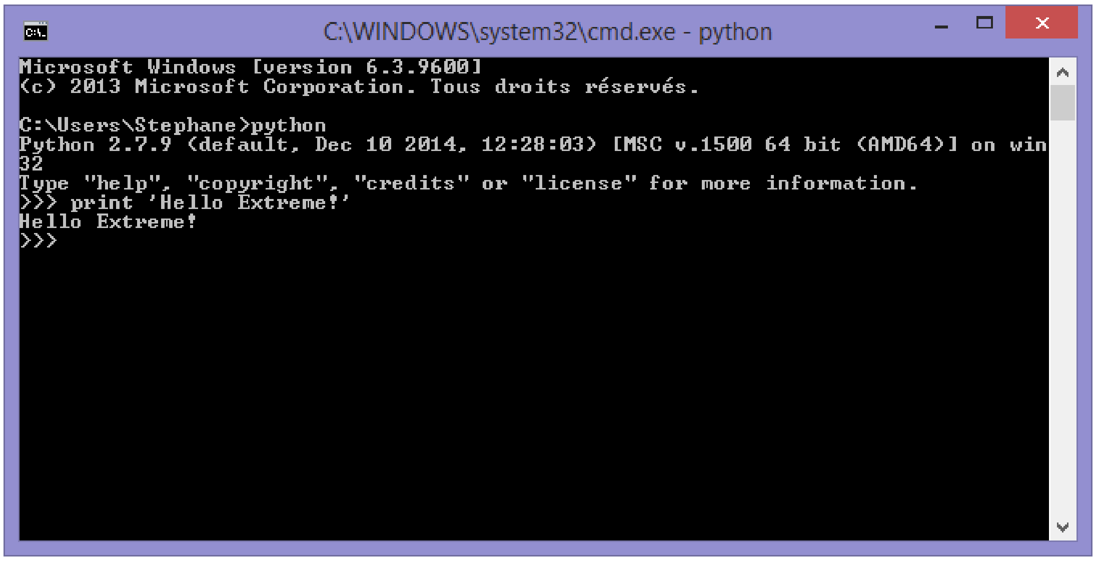
Variable value: :\Program Files\Saxonica\SaxonHE9.4N\bin

FP_NO_HOST_C...	NO
NUMBER_OF_P...	8

d. Add the following entries to the existing path:

`C:\Python27\;C:\Python27\Lib\site-packages\;C:\Python27\Scripts\;`

4. Once Python is installed, invoke the Python interpreter from a command shell, and run any program from it.



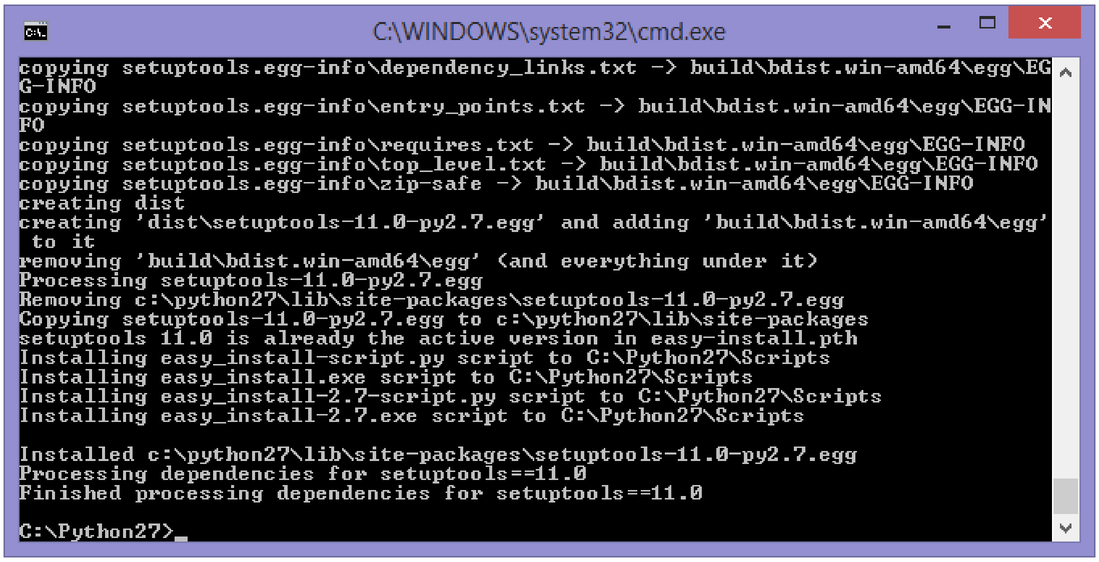
```
Microsoft Windows [version 6.3.9600]
(c) 2013 Microsoft Corporation. Tous droits réservés.

C:\Users\Stephane>python
Python 2.7.9 (default, Dec 10 2014, 12:28:03) [MSC v.1500 64 bit (AMD64)] on win
32
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'Hello Extreme!'
Hello Extreme!
>>>
```

Before we can begin writing Python code, we'll first configure the system to easily install libraries using *easy_install* and *pip*.

Installing the easy_install

1. Navigate to https://pypi.python.org/pypi/setuptools and download the `ez_setup.py` file from the Windows (simplified) section and save it to the C:\Python27 directory.

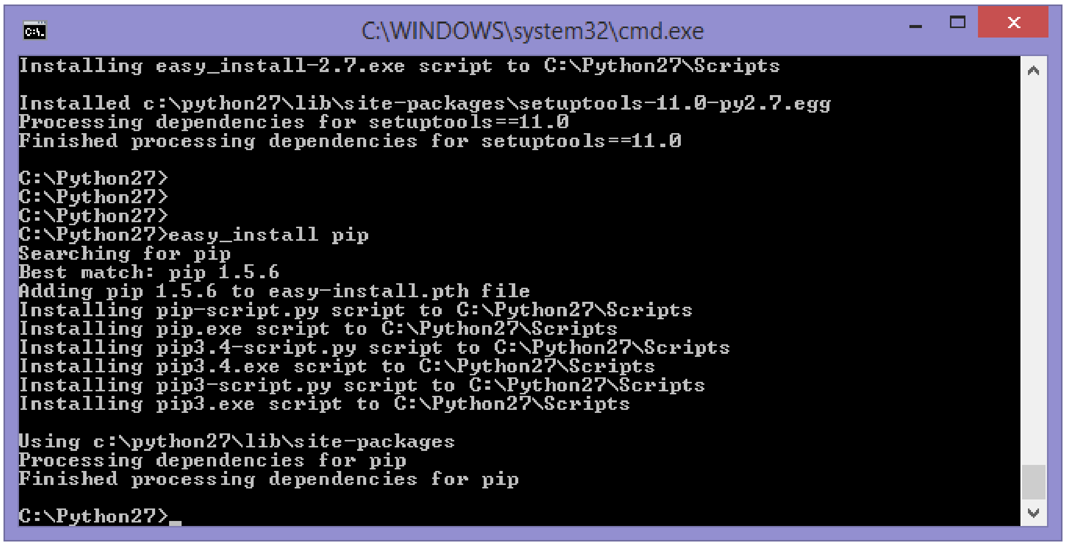
2. From the Python27 directory, execute the program `python ez_setup.py`.



```
copying setuptools.egg-info\dependency_links.txt -> build\bdist.win-amd64\egg\EG
G-INFO
copying setuptools.egg-info\entry_points.txt -> build\bdist.win-amd64\egg\EGG-IN
FO
copying setuptools.egg-info\requires.txt -> build\bdist.win-amd64\egg\EGG-INFO
copying setuptools.egg-info\top_level.txt -> build\bdist.win-amd64\egg\EGG-INFO
copying setuptools.egg-info\zip-safe -> build\bdist.win-amd64\egg\EGG-INFO
creating dist
creating 'dist\setuptools-11.0-py2.7.egg' and adding 'build\bdist.win-amd64\egg'
 to it
removing 'build\bdist.win-amd64\egg' (and everything under it)
Processing setuptools-11.0-py2.7.egg
Removing c:\python27\lib\site-packages\setuptools-11.0-py2.7.egg
Copying setuptools-11.0-py2.7.egg to c:\python27\lib\site-packages
setuptools 11.0 is already the active version in easy-install.pth
Installing easy_install-script.py script to C:\Python27\Scripts
Installing easy_install.exe script to C:\Python27\Scripts
Installing easy_install-2.7-script.py script to C:\Python27\Scripts
Installing easy_install-2.7.exe script to C:\Python27\Scripts

Installed c:\python27\lib\site-packages\setuptools-11.0-py2.7.egg
Processing dependencies for setuptools==11.0
Finished processing dependencies for setuptools==11.0

C:\Python27>_
```

Installing pip

We'll use *easy_install* to install *pip*. From the Python directory, execute the command `easy_install pip`.



```
Installing easy_install-2.7.exe script to C:\Python27\Scripts

Installed c:\python27\lib\site-packages\setuptools-11.0-py2.7.egg
Processing dependencies for setuptools==11.0
Finished processing dependencies for setuptools==11.0

C:\Python27>
C:\Python27>
C:\Python27>
C:\Python27>easy_install pip
Searching for pip
Best match: pip 1.5.6
Adding pip 1.5.6 to easy-install.pth file
Installing pip-script.py script to C:\Python27\Scripts
Installing pip.exe script to C:\Python27\Scripts
Installing pip3.4-script.py script to C:\Python27\Scripts
Installing pip3.4.exe script to C:\Python27\Scripts
Installing pip3-script.py script to C:\Python27\Scripts
Installing pip3.exe script to C:\Python27\Scripts

Using c:\python27\lib\site-packages
Processing dependencies for pip
Finished processing dependencies for pip

C:\Python27>_
```

Mac OS X

Installation on Mac OS X is more straightforward than on Windows. Simply download and install the latest 2.7 .dmg installer from the Python website (https://www.python.org/downloads/mac-osx/)

Once Python is installed, ensure correct setup by opening a terminal and executing the *easy_install* program. Unlike Windows, *easy_install* is part of the installation of Python on Mac.

In your console, type the following command: `sudo easy_install pip`.

Ubuntu

Python is everywhere in Linux, and as such, installation is the simplest one: Python is already installed! However it is possible that Python 2.7 is not part of touch images of Ubuntu, replaced by Python 3. https://wiki.ubuntu.com/Python/3

With Ubuntu 14.10, python 2.7.8 is integrated by default, along with Python 3.4.2. Python 2.7 is still part of the archive, and should be until 2020, which is the targeted EOL of Python 2.

Depending on your version, you may have to install/upgrade Python 2.7. As both version are available, you'll need to specify which one to use in your script, with the usual following line at the start of your script.

```
#!/usr/bin/python
```

In either case, you have to install *easy_install* and *pip*:

```
sudo apt-get install python-setuptools
sudo apt-get install python-pip
```

Choosing an IDE

Included with Python, you can find the IDLE GUI IDE. You can use it for small programs, but it is recommended to find a more complete IDE to ease development.

Choosing an IDE is a matter of taste, so you are encouraged to find the one that suits you best. Here are a few options:

- Eclipse: available on every system and very common on Linux and Mac OS X.
- Visual Studio Express: the free version of Microsoft program is very complete and powerful (http://www.visualstudio.com/en-us/products/free-developer-offers-vs).
- Wing IDE 101: the free version of this IDE is very simple to use. This is the one we'll use through this document (http://wingware.com/downloads/wingide-101).
- Vim: Vi Improved is also a good IDE under Unix environment and is part of Ubuntu installation.

Scripting with Python

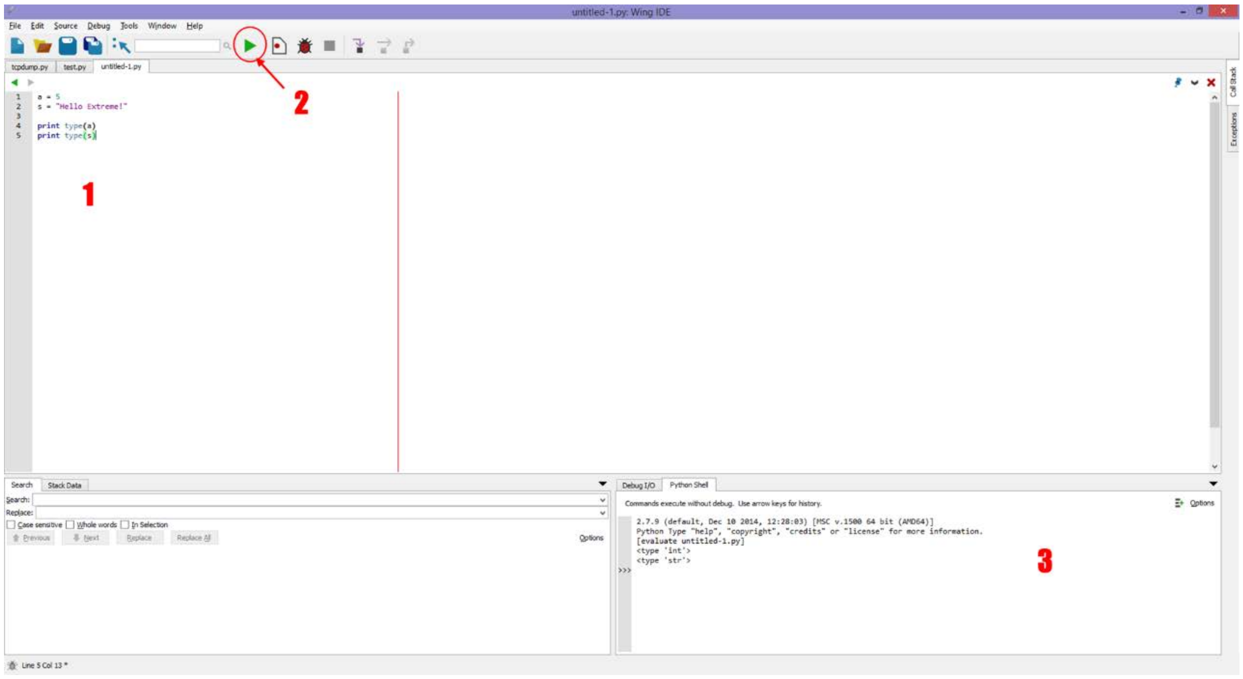
With Python it is now possible to create scripts that can be executed manually or triggered by an event through UPM. In this respect, Python can be considered as an equivalent to TCL. However, Python has several advantages over TCL:

- Python is a more widely used programming tool and is more powerful (with strings, manipulation, etc.).
- Python is modern programming tool that universities are teaching to their students and used in Networking/Datacenter environment (Google, Facebook, etc.).
- Python is easier to learn.
- A lot of code and libraries already exist and can be reused easily.
- Communication between systems is possible, either through events or network sockets.

Basic Python

For those who don't know Python, this section provides a quick presentation of some common, basic concepts.

Using your IDE, type the code and execute it. The result should be displayed inside the Python Shell window integrated.


untitled-1.py: Wing IDE
File Edit Source Debug Tools Window Help
tcpdump.py test.py untitled-1.py
1
2
Search Stack Data
Debug I/O Python Shell
Commands execute without debug. Use arrow keys for history.
Options
3
Line 5 Col 13 *

```
a = 5
s = "Hello Extreme!"

print type(a)
print type(s)
```

```
2.7.9 (default, Dec 10 2014, 12:28:03) [MSC v.1500 64 bit (AMD64)]
Python Type "help", "copyright", "credits" or "license" for more information.
[evaluate untitled-1.py]
<type 'int'>
<type 'str'>
>>>
```

The image above shows the typical Wing IDE GUI. You can write your Python code in the main window (1), where several tabs can be opened for several files. For testing the result of your code, you can execute the code of the active window (2) and see the result in the Python Shell window (3).

Variables

Python doesn't require having a variable type definition; it figures it out. You can simply declare it and affect a value without any other burden. Within the Python interpreter, you can see the value of a variable by simply typing its name.

```
>>> a = 5
>>> a
5
```

We can do all the math we want on integer or float type variable. We can also add strings together easily:

```
>>> a = 5
>>> b = 42
>>> a = a+b
>>> a
47

>>> a = 5
>>> a += 1
>>> a
6

>>> a = 5
>>> b = 42
>>> a,b = b,a
>>> a
42
>>> b
5

>>> s1 = “Hello”
>>> s2 = “Extreme!”
>>> s1 = s1+” “+s2
>>> s1
‘Hello Extreme!’
```

The function `type()` confirms the variable type:

```
a = 5
s = "Hello Extreme!"

print a,type(a)
print s,type(s)
```

The above script returns the following result:

```
5 <type 'int'>
Hello Extreme! <type 'str'>
```

The “int” type means integer. This is a number, without a decimal. The “str” type means string. This is a character string.

You can convert a type to another using the function with the same names:

- `int()`: converts a variable to an integer
- `str()`: converts a variable to a string

```
a = 5
s = "Hello Extreme!"

print a,type(a)
print s,type(s)

a = str(a)
print a,type(a)

a = int(a)
print a,type(a)
```

The result is:

```
5 <type 'int'>
Hello Extreme! <type 'str'>
5 <type 'str'>
5 <type 'int'>
```

Print

To print something to the screen (the terminal), we use the built-in function `print`. We can use it differently to achieve the same result.

The following simple script is an example:

```
a = 5
s = "Hello Extreme!"

print 'The value of a is',a,'and the value of s is:',s
```

The result of this script is:

```
The value of a is 5 and the value of s is: Hello Extreme!
```

The following examples are displaying the same thing:

```
print "The value of a is",a,"and the value of s is:",s
print "The value of a is {0} and the value of s is: {1}".format(a,s)
print "The value of a is {} and the value of s is: {}".format(a,s)
```

In the second example in the above code, if we swap the value in curly braces, we see that we change the order of the print. It would have been similar if we'd changed the order in the `format()` function:

```
>>> print "The value of a is {1} and the value of s is: {0}".format(a,s)
The value of a is Hello Extreme! and the value of s is: 5
```

Loop and Condition Statements

If, elif, else

The usual **`if, elif, else`** conditional instructions are present in Python. If you're not familiar with Python, remember to:

- Add the colon (:) at the end of the line.
- Take care of the indentation of your code.

Indentation is critical because unlike most of other programming languages, this is not just for ease of reading, but necessary for Python to understand the instruction blocks.

Here's a simple example:

```
a = 42

if a > 0:
    print 'a is positive'
elif a < 0:
    print 'a is negative'
else:
    print 'a is null'
```

While

You can also use the **`while`** loop condition.

```
i = 0

while i < 10:
    print i,'*',i,'=',i*i
    i += 1
```

For

Another very useful loop condition is the **`for`** loop. If you have already done some coding in C or similar languages,, you will notice that this instruction is different in Python. It's close to the **`foreach`** instruction in Perl or PHP.

The **`for`** instruction works on a sequence. It can be some list, like a character string. For example:

```
for element in sequence:
```

The **`element`** variable is created by the **`for`** instruction. You don't have to create it. It takes the value of each successive entry in the *sequence*.

Here is a simple example:

```
s = "Hello Extreme!"

for letter in s:
    print letter
```

Try

You can define a block of program within a **`try`** statement. The block tries to execute, but if there's a problem, like an exception, it will jump to the **`except`** statement that matches the exception (error type).

You can have several except statements, one for each error type that could be returned and/or an except statement without any error type that would catch any error. It's also possible to have several error types for a same except statement.

The **`finally`** statement is a way to execute the code it contains whatever happened. This try statement is a nice way to handle interruption or error (like a wrong argument passed or not enough):

```
>>> def foo():
...    try:
...        1/0
...    finally:
...        return 42
...
>>> foo()
42
```

With

The **`with`** statement, introduced in Python 2.5, is a nice way to use an expression, execute a block of program, and exit cleanly, because the **`with`** statement takes care of everything. This is very helpful for file management.

```
def readStatus():
    try:
        with open(STATUS_FILE,'r') as fd:
            status = fd.read()
    except:
        status = 0

    try:
        return int(status)
    except:
        return 0
```

The file opened in the **`with`** statement is automatically closed when the second **`try`** is reached.

Creating Functions

As the examples in the previous section show, we can create functions to make more complex programs. A new function is defined with the keyword `def` followed by the name of the function, two brackets (< >) with an optional list of arguments, and the semi-colon.

Respecting the indentation necessary for Python to execute properly, we can then enter our code for that function. At the end of the block, we can return a value, using the keyword `return`.

```
def foo():
    #<instructions> note that this is a comment
    return 42
```

Functions can be made more complex by using multiple arguments of different types, and an undefined number of optional arguments. Likewise, you can return several values of different type as well as lists of arguments (tuples).

```
>>> def foo(a,b):
...    for letter in b:
...        print letter
...    a = a*a
...    b = b+" Extreme!"
...    return a,b
...
>>> foo(42,"Hello")
H
e
l
l
o
(1764, 'Hello Extreme!')
```

You can also have arguments with a predefined value. This way, if the arguments are not passed, they will have a default value.

Here's an example using the previous script but it omits sending a string to the `foo()` function. As you can see, the definition of the `foo()` function has a predefined value.

```
>>> def foo(a,b="Hello"):
...    for letter in b:
...        print letter
...    a = a*a
...    b = b+" Extreme!"
...    return a,b
...
... def main():
...    print foo(42)
...
>>> main()
H
e
l
l
o
(1764, 'Hello Extreme!')
```

The main Function

With a more complex program dealing with several functions, we have to start the program at one precise point: the **`main`** function. This is similar to the typical **`void main()`** function in C.

```
if __name__ == '__main__':
    try:
        main()
    except SystemExit:
        #catch SystemExit to prevent EXOS shell from exiting to
        #the login prompt
        pass
```

When the script is executed, it will start in the **`main()`** function.

Arguments

Variable Arguments

We already saw that we can send arguments to a function, but what if we want a variable number of arguments? Hopefully, we can define such a function. It's also possible to mix the function with mandatory arguments, as long as the mandatory ones are passed first.

```
>>> def manyArgs(*arg):
...  print "I was called with", len(arg), "arguments:", arg
...
>>> manyArgs("Extreme")
I was called with 1 arguments: ('Extreme',)
>>> manyArgs(42,"Hello", "Extreme", 5)
I was called with 4 arguments: (42, 'Hello', 'Extreme', 5)
```

System Arguments

When creating a script/program in Python, we may need to receive some arguments from the command line. These arguments can be flags to format the expected output, or some parameters needed for treatment in the program.

The basic way to catch these arguments is to use the **`system`** module. You must first import that module (using the keyword **`import`**) so we can use its functions. Each argument passed in the command line is listed in the **`sys.argv`** list. Here's an example:

```
import sys

def main():
    i = 0
    while i < len(sys.argv):
        print sys.argv[i]
        i += 1

if __name__ == '__main__':
    try:
        main()
    except SystemExit:
        pass
```

Looking ahead to the next section slightly, we can run that script (named **`test.py`**) on an Extreme switch running EXOS 15.6.1, we get the following output:

```
X440-8t.5 # run script test
exsh
X440-8t.6 #
X440-8t.6 # run script test 5
exsh
5
X440-8t.7 # run script test 5 42 Hello
exsh
5
42
Hello
X440-8t.8 #
```

But Python offers more than that. Using the **`argparse`** module, we have a better tool to handle arguments and can offer a built-in help mode.

With **`argparse`**, it's possible to list all the possible arguments, generate a help message, and define which are mandatory.

```
import argparse

def getParams():
    parser = argparse.ArgumentParser(prog='param')
    parser.add_argument('-i','--remote_address',
            help='Remote IP address to ping',
            action='append',
            required=True)
    parser.add_argument('-p','--policy',
            help='Name of ACL policy',
            required=True)
    parser.add_argument('-f','--from_address',
            help="'From' IP address in ping command")
    parser.add_argument('-r','--virtual_router',
            help='Virtual Router used for ping')

    args = parser.parse_args()
    return args
```

A simple script (named **`param.py`**) calling this function returns the following output if we do not pass any argument:

```
X440-8t.6 # run script param
usage: param [-h] -i REMOTE_ADDRESS -p POLICY [-f FROM_ADDRESS]
             [-r VIRTUAL_ROUTER]
param: error: argument -i/--remote_address is required
X440-8t.7 #
```

As we specified two mandatory parameters ("required=True"), there's an error generated because the parameters are missing. Nonetheless, we have the help output displayed.

If we pass the valid **–h** argument, we have a nice help output. The text used in this help message is the one defined in each help parameter in the code.

```
X440-8t.10 # run script param -h
usage: param [-h] -i REMOTE_ADDRESS -p POLICY [-f FROM_ADDRESS]
             [-r VIRTUAL_ROUTER]

optional arguments:
  -h, --help            show this help message and exit
  -i REMOTE_ADDRESS, --remote_address REMOTE_ADDRESS
                        Remote IP address to ping
  -p POLICY, --policy POLICY
                        Name of ACL policy
  -f FROM_ADDRESS, --from_address FROM_ADDRESS
                        'From' IP address in ping command
  -r VIRTUAL_ROUTER, --virtual_router VIRTUAL_ROUTER
                        Virtual Router used for ping
X440-8t.11 #
```

Finally, it is important to note that the arguments can be passed in any order.

User Input

To catch user input in a program, we can use the `raw_input()` function. It reads a line from input, converts it to a string value, and returns that. Here's an example:

```
>>> s = raw_input('Enter a sentence: ')
Enter a sentence: Hello Extreme!
>>> s
'Hello Extreme!'
```

Standard Library

The Python implementation in EXOS uses the standard library, but not all the library is included for size reasons. Many more modules are available, and what they do and how they work are provided in the online documentation at: https://docs.python.org/2/library/index.html.

Among the different modules you may need the most, the following should be the most common:

- os
- sys
- socket
- io
- argparse

The *os* module gives you access to operating system depend functionalities.

The *sys* module is a system specific module. As stated in the Python documentation: "It provides access to some variables used or maintained by the interpreter and to function that interacts strongly with the interpreter."

Using *socket*, you can create a network socket to establish a communication between devices. The *socket* module supports IPv4, IPv6, UDP, TCP and UDS for the most popular ones.

The *io* module is related to file management and anything else related to typical input/output mechanism.

The *argparse* module is described in the System Arguments section.

EXOS CLI Modules

Once you are familiar with the basic concepts and possibilities of Python, which already gives us a great tool, let's see how we can interact with EXOS.

Exsh Module

Like every other modules described previously, EXOS CLI has a Python module ready. To start using it, we have to import it. This module is called *exsh*.

What this module offers is the ability to send a CLI command to EXOS, and capture the result. We can work either in xml, text, both or none.

Here's the description of this module:

`exsh.clicmd(cmd, capture=False, xml=False, args=None)`

Send a CLI command to EXOS.

Parameters:

– `cmd (str)` – any valid EXOS CLI command
– `capture` (True or False) – if ***`capture = True`***, CLI output text is returned
– `xml` (True or False) – if ***`xml = True`***, the XML that EXOS used to create the CLI output is returned
– `args (str)` – used to provide additional input to some EXOS commands that prompt for more information

Returns:

– None – if both `capture` and `xml` are False.
– Captured text – if `capture` is True.
– XML – if `xml` is True.
– Captured text and XML – if both `capture` and `xml` are True.

Raises:

– `RuntimeError` – EXOS command is invalid or encountered an error

The display text may change from release to release which makes it more difficult to maintain a script that parses CLI responses from one release to another. Using the XML is a more machine-readable method of processing the CLI output. The XML tends to be stable from one release to another and is not subject to formatting changes.

Let’s have some simple examples using this module.

No CLI capture is required:

```
exsh.clicmd('create vlan testvlan')
```

This example captures the CLI display text:

```
cliReply = exsh.clicmd('show ports configuration no-refresh',
capture=True)
```

Same command but captures the XML instead of the CLI display text:

```
xmlReply = exsh.clicmd('show ports configuration no-refresh',
xml=True)
```

Same command, but captures both CLI display text and XML:

```
cliReply, xmlReply = exsh.clicmd('show ports configuration no-
refresh', capture=True, xml=True)
```

When capturing the XML CLI response, the Python *xml.etree.ElementTree* module can be used to process the results.

Below is an example:

```
try:
    import xml.etree.cElementTree as ET
except ImportError:
    import xml.etree.ElementTree as ET

REPLY_TAG = '</reply>'

#an ExtremeXOS ping command
cmd = 'ping count 2 10.10.10.10'

#give the command to ExtremeXOS, capture the response in XML
xmlout = exsh.clicmd(cmd, xml=True)

# look for totalTx and totalRx elements in the XML response
if xmlout != None:
    while True:
        # each level 0 XML transaction is sandwiched by <reply></reply>
        idx = str.find(xmlout, REPLY_TAG)
        if idx == -1:
            break
        idx += len(REPLY_TAG)
        xmlextract = xmlout[:idx]
        tree = ET.fromstring(xmlextract)
        for elem in tree.iter():
            if 'totalTx' == elem.tag:
                totalTx = int(elem.text)
            if 'totalRx' == elem.tag:
                totalRx = int(elem.text)
        #strip off the first XML transaction from group and continue
        xmlout = xmlout[idx:]
```

Exshexpect Module

It may be desirable to interact with EXOS using expect scripting functionality. The Python community offers a **pexpect.py** module that can provide this capability. The home page for **pexpect** can be found at: http://pexpect.readthedocs.org/en/latest/

EXOS uses **pexpect.py** version 3.2. Documentation can be found at: https://pypi.python.org/pypi/pexpect/

The **pexpect.py** module provides interfaces to run interactive shell commands and to process the results through expect like functions. The challenge with **exsh.clicmd()** is that it is a synchronous call and not a separate process.

The **exshexpect.py** module was developed to provide a wrapper for **pexpect.py** that interfaces to **exsh.clicmd()**.

The following is an example of using pexpect together with the **exshexpect** module:

```
import pexpect
import exshexpect

exosPrompt = '::--:--::' # <- make up a prompt for expect that will not
match any command output.

p = exshexpect.exshspawn(exosPrompt, exsh.clicmd)  # <- create an expect
object. Pass in the prompt and the back end function to call

p.sendline('show fan')  # <- use sendline to send commands to EXOS (in
this case exsh.clicmd)

idx = p.expect([exosPrompt, pexpect.EOF, pexpect.TIMEOUT])

print 'idx=',idx  # <-see pexpect.py for details on working with python
expect
print 'before->',p.before
print 'after->',p.after
```

Special case for passing in a command with an additional argument described in **exsh.clicmd()**:

```
p.send('enable debug-mode\nC211-6BB4-E6F5-B579\n')
```

In the line above, use **send()** and include **\n** terminations for the command and the additional parameter. This is the same as calling:

```
exsh.clicmd('enable debug-mode', capture=True, args='C211-6BB4-
E6F5-B579')
```

but using expect to process any response.

```
idx = p.expect([exosPrompt, pexpect.EOF, pexpect.TIMEOUT])
```

Dealing with Socket

Socket creation defaults to the Mgmt VR in EXOS. If you need to create a socket on another Virtual Router, you need to determine the VR ID of the desired VR. For example, VR-Default has a VR ID of 2. To use VR-Default, follow the below example for socket creation:

```
import socket

EXTREME_SO_VRID = 37 #special Extreme ioctl to set the VR

udp_sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

try:
    f = open('/proc/self/ns_id', 'w')
    f.write('2\n')
    f.close()
except:
    udp_sock.setsockopt(socket.SOL_SOCKET, EXTREME_SO_VRID, 2)
```

The `try/except` statement is necessary as the `/proc` recommended method to set the VR ID is known to have a Permission Error with at least EXOS 15.6.1, 15.6.2, and 15.7.1. For future compatibility, it is then recommended to follow this way of coding.

The value in the `f.write()` function, as the last parameter in the `setsockopt()` function, sets the desired VR we want to use for the socket.

You can find the numeric VR ID for any Virtual Router name on a system using the following script:

```
def vrNameToNumber(vrName):
    vrId = None
    with open('/proc/net/vr','r') as f:
        for l in f.readlines():
            rslt = l.split()
            if rslt[3] == vrName:
                vrId = int(rslt[0])
                break
    return vrId
```

By printing the the result of this function, you can find the numeric VR ID for any VR name:

```
VM1.3 # vi findVR.py

def vrNameToNumber(vrName):
    vrId = None
    with open('/proc/net/vr','r') as f:
        for l in f.readlines():
            rslt = l.split()
            if rslt[3] == vrName:
                vrId = int(rslt[0])
                break
    return vrId

print vrNameToNumber('VR-Mgmt')
print vrNameToNumber('VR-Default')
print vrNameToNumber('VR-Control')
print vrNameToNumber('VR-Unknown')
~
~
```

```
VM1.4 # run script findVR
0
2
1
None
VM1.5 #
```

Installing and Executing Python Scripts on EXOS

Installation

The Python script may be edited directly on the switch using the embedded *vi* editor in EXOS. When doing this, ensure the resulting file has a `.py` suffix. Another approach is to transfer a Python script from a PC to the switch using the `tftp` command.

As a quick reminder, the *vi* text editor in EXOS is a simplified version of the Unix *vi* editor. Once in the editor, you can navigate the file with your keyboard arrows, and you can insert text using the `i` command. Once complete, exit the insert mode by pressing the [Esc] key, and then save and quit with the command `:wq`.

To use the `tftp` EXOS command, start a TFTP server on your PC, preferably pointing directly to the directory where your script is located, and type the following EXOS CLI command on your switch:

```
X440-8t.2 # tftp get 192.168.56.1 vr vr-mgmt TermSize.py
Downloading TermSize.py on primary Node ... done!
X440-8t.3 # ls
-rw-r--r--    1 admin    admin         476 Jan  1 09:23 CheckkVerPython.py
-rw-r--r--    1 admin    admin        1073 Jan  5 21:57 TermSize.py
-rw-r--r--    1 admin    admin        2230 Jan  2 11:01 cat.py
drw-r--r--    2 root     root            0 Nov 10 10:26 dhcp
-rw-r--r--    1 root     root       193919 Dec 29 17:46 primary.cfg
-rw-r--r--    1 admin    admin         267 Jan  2 11:03 showFile.py
drwxr-xr-x    2 root     root            0 Jan  5 21:54 vmt

 1K-blocks      Used Available Use%
    121824      3484    118340   3%
X440-8t.4 #
```

Execution

A Python script is a text file with a `.py` extension. Starting with EXOS 15.6.1, such a file is recognized by EXOS. It is possible to execute it manually so that it runs to completion, with several commands:

- The existing EXOS command for TCL, `load script <script>`, has been enhanced to support Python script.
- A new command has been introduced: `run script <script>`.

These commands are synonymous and perform the same function.

Using the provided commands, we can run any script and see the result. Writing a simple script, we can verify the Python version in EXOS:

```
X440-8t.5 # run script CheckVerPython
Python version is 2.7.3
X440-8t.6 #
```

Adding Third-Party Modules

One very nice thing about Python is its large community of programmers that have been around for many years. Because of this, many Python libraries and documentation exist, from Scientific Calculation to Game Programming.

For your need, you may have to install new libraries on EXOS to use them for your code. This can be easily accomplished by TFTP-ing the files in the `/config` directory. Python will automatically search that directory. This is the default directory where files are sent when you do a tftp.

As an example, if you need the **`eventd.py`** program and its configuration file **`eventd.conf`**, you simply tftp both files to the switch. You need a TFTP server on your PC, pointing to the directory where the files are located, and you type the **`tftp`** command on the switch.

```
tftp 192.168.56.1 –g –r eventd.py
tftp 192.168.56.1 –g –r eventd.conf
```

You can now use the library in your Python script.

Python Scripts Triggered by UPM

Just like TCL scripting, we can associate a Python script with UPM. UPM gives the opportunity to execute the Python script based on an event in the logs (for example, a Port Up/Down), or to execute it periodically (every user-defined time period).

Here's an example. Assuming we have a Python script (**`ping.py`**) doing some ping as a health-check mechanism used to add or remove a given PBR, we can create a UPM profile to execute the script every ten seconds.

```
X460G2-24t-10G4.91 # create upm profile ping
Start typing the profile and end with a . as the first and the only
character on a line.
Use - edit upm profile <name> - for block mode capability
load script ping.py -i10.68.9.7 -i10.69.100.200 –rVR-Default –f10.10.10.10
–pMyPBRa –vProd
load script ping.py -i10.20.20.1 –i10.20.20.2 –rVR-Default –f10.10.10.10 –
pMyPBRb -vSales
.
X460G2-24t-10G4.92 # create upm timer pingtimer
X460G2-24t-10G4.93 # config upm timer "pingtimer" profile "ping"
X460G2-24t-10G4.94 # config upm timer "pingtimer" after 10 every 10
```

In this example, the **`ping.py`** Python script requires various parameters, and can be executed several times to provide its services to different resources. UPM also offers the advantage to pass environment variables that can be used in a script. This turns a static script into a dynamic script that can be generic enough to work on various cases.

With TCL, the UPM environment variables are embedded. With Python scripting these variables have to be passed as arguments.

Below is a simple script (`lldp-device.py`) detecting that a LLDP device has been connected (or removed) on a switch.

```
#!/usr/bin/python

import exsh
import sys


def main():
    voiceVLAN = 'ToIP'
    port = sys.argv[2]

    if sys.argv[1] == 'DEVICE-DETECT':
        exsh.clicmd('create log entry \"LLDP device detected on port
{}\"'.format(port))
        exsh.clicmd('config vlan {} add port {} tagged'.format(voiceVLAN,
port))
        exsh.clicmd('config lldp port {} advertise system-
name'.format(port))
        exsh.clicmd('config lldp port {} advertise vendor-specific dot3
mac-phy'.format(port))
        exsh.clicmd('config lldp port {} advertise vendor-specific med
capabilities'.format(port))
        exsh.clicmd('config lldp port {} advertise vendor-specific med
power-via-mdi'.format(port))

    elif sys.argv[1] == 'DEVICE-UNDETECT':
        exsh.clicmd('config vlan {} del port {}'.format(voiceVLAN, port))
        exsh.clicmd('unconfig lldp port {}'.format(port))
        exsh.clicmd('create log entry \"LLDP device removed from port
{}\"'.format(port))


if __name__ == '__main__':
    try:
        main()
    except SystemExit:
        pass
```

With a UPM profile configured to launch our Python script when a device is detected/undetected by LLDP, we have a very dynamic script.

The voiceVLAN variable in our Python script could be another argument to make it even more generic. We could also test if the VLAN already exists (and create it if needed), and we should first validate the correct parameters have been entered if we wanted to productize this script.

```
X440-8p.6 # create upm profile lldp-detect
Start typing the profile and end with a . as the first and the only
character on a line.
Use - edit upm profile <name> - for block mode capability
run script lldp-device $EVENT.NAME $EVENT.USER_PORT
.
X440-8p.7 # configure upm event device-detect profile lldp-detect ports 1-
8
```

```
X440-8p.8 # configure upm event device-undetect profile lldp-detect ports
1-8
```

When connecting an IP Phone on port 5 of a Summit x440-8p, running EXOS 15.6.1, we see the following in the logs:

```
X440-8p.10 # sh log
01/14/2015 17:09:43.36 <Info:System.userComment> LLDP device detected on
port 5
01/14/2015 17:09:42.12 <Noti:UPM.Msg.upmMsgExshLaunch> Launched profile
lldp-detect for the event device-detect
01/14/2015 17:09:42.12 <Noti:UPM.Msg.LLDPDevDetected> LLDP Device
detected. Mac is 00:07:3B:D3:27:3E, IP is 0.0.0.0, on port 5, device type
is 5, max power is 5600
01/14/2015 17:09:10.38 <Noti:POE.port_delivering> Port 5 is delivering
power
01/14/2015 17:09:10.33 <Info:vlan.msgs.portLinkStateUp> Port 5 link UP at
speed 100 Mbps and full-duplex

A total of 5 log messages were displayed.
X440-8p.11 #
```

The port has been correctly added to the VLAN:

```
X440-8p.39 # sh vlan
-----------------------------------------------------------------------
Name      VID  Protocol Addr       Flags     Proto  Ports  Virtual
                                                    Active router
                                                    /Total
-----------------------------------------------------------------------
Default   1   -----------------------T------- ANY    1 /12  VR-Default
Mgmt      4095 10.1.1.2       /24  --------- ANY    1 /1   VR-Mgmt
ToIP      10   ----------------------------- ANY    1 /1   VR-Default
-----------------------------------------------------------------------
```

There's no need to try to use the Python `print` function with UPM because the output is redirected and nothing is displayed (neither in console nor in the active terminal).

Python Apps

With EXOS 15.7.1 onwards, the use of Python is not limited anymore to “simply” scripting. It’s now possible to create native applications written in Python that can be launched manually or automatically and be persistent. These applications are executed as EXOS processes and are managed by EPM. As such it’s possible to show, start, terminate, restart, etc. any process created.

The Python environment for Native Apps, the daemon one, is a different environment than the scripting one presented previously. They both are based on Python 2.7, but each has a different set of modules available. Over time, they should be less different.

EXOS Python API

The EXOS Python API provides a high-level interface to EXOS functionality. It is part of the EXOS SDK, which allows third-parties and customers to develop native applications which extend the functionality of EXOS. The SDK is not necessary to write Python apps; a developer only requires the API documentation to start writing applications.

To use the EXOS Python API, it should be imported as follows:

```
from exos import api
```

The EXOS Python API provides several functionalities to interact with EXOS.

Runtime Environment

Python applications are run in a “Python container” known as EXPY. It’s a simple C-based EXOS application that is linked against the Python runtime environment. It also implements the EXOS Python API.

In most cases, EXPY will be transparent to a Python developer. However, if packaging an XMOD, you will need to understand its parameters:

```
Usage: expy [OPTION] -c <SCRIPT>
   or: expy [OPTION] -m <MODULE>

Run python code in a SCRIPT or MODULE.

-c                Chunk of python code to run.
-m                Module to be run via the runpy module.
-n <name>         Name to give exosSetProcessName().
-d                Run the code as a daemon under EPM.  The code must init
                  itself (register callbacks, spawn threads, etc.) and then
                  exit.  expy will then go into the dispatcher forever.
                  Without this option, expy runs the code in a thread and
                  exits when the thread exits.
```

System Functions

On the system level, a few functions are available:

`exos.api.`**`force_exit`**`(`*`retcode`*`)`

Forcibly exit the Python container (expy).

Parameters:

retcode (int): code to return to the exit() system function.

Returns:

None.

`exos.api.get_process_name()`

Get the name of this process.

Returns:

Process_Name (str)

`exos.api.`**`set_process_name`**`(`*`process_name`*`)`

Set the name of this process.

Parameters:

process_name (str): name of the process

Returns:

None

Logging Functions

Trace buffers are circular in-memory log buffers and are typically used for collecting debug data, which can then be displayed in the CLI or dumped to a file. A process can have multiple trace buffers. It is typically convenient to use one for each component or library within a process.

`class exos.api.`**`TraceBuffer`**`(`*`buffer_name, buffer_size`*`)`

A circular in-memory log buffer

`log(`*`msg`*`)`

Log *msg* to the trace buffer

`dump(file_name)`

Dump this trace buffer to *file_name*. When *file_name* is `None`, the filename is generated. The file is always written under `/usr/local/tmp`.

`clear()`

Clear this trace buffer. All existing log entries will be deleted.

`class exos.api.`**`TraceBufferHandler`**`(`*`buffer_name, buffer_size`*`)`

A `logging.Handler` that's backed by a `TraceBuffer`. Allows the standard python logging mechanism to be used with trace buffers. *Buffer_name* and *buffer_size* are used to create the `TraceBuffer`.

`emit(`*`record`*`)`

Write the *record* to the trace buffer.

`trace_buffer = None`

The underlying trace buffer

`exos.api.`**`dump_trace_buffer`**`(`*`buffer_name, file_name`*`)`

Dump a trace buffer *buffer_name* to a file *file_name*. The trace buffer will be written to `/usr/local/tmp/<file_name>`

Parameters:

buffer_name (str): if None, then all buffers will be dumped.

file_name (str): if None, the file name will be generated.

Raises:

`UnknownTraceBufferError`: the *buffer_name* does not exist.

`exos.api.`**`clear_trace_buffer`**`(`*`buffer_name`*`)`

Clear the trace buffer *buffer_name*.

Parameters:

Buffer_name (str): if None, then all buffers are cleared.

Raises:

`UnknownTraceBufferError`: the *buffer_name* does not exist.

Trace buffers can be plugged into the logging Python module with a *TraceBufferHandler*:

```
import logging

logger = logging.getLogger("myapp")
logger.setLevel(logging.DEBUG)
logHandler = exosapi.TraceBufferHandler("mybuf", 20480)
logHandler.setLevel(logging.DEBUG)
logger.addHandler(logHandler)
```

To display a trace buffer in the CLI:

```
debug ems show trace <process> <bufname>|all
```

Authentication and Session Management

A user is authenticated by providing a username and password. The currently configured mechanisms (local, Radius) are used to verify the user. If authenticated successfully, an instance of ExosUser is returned. From there, the user's permission level can be obtained.

`exos.api.`**`authenticate_user`**`(`*`username, password`*`)`

Authenticate a user, given a username and password.

Parameters:

username (str): the name of the user to authenticate

password (str): the password to use

Returns:

`ExosUser`: successful authentication.

`None`: authentication failed

`exos.api.`**`authenticate_user_async`**`(`*`callback, username, password`*`)`

Asynchronous version of **`authenticate_user()`**. Returns immediately and calls *callback* with the result. *Callback* must accept 1 argument, which will be an **`ExosUser`** if successful or **`None`** if failed.

Parameters:

callback: function to call.

username (str): the username.

password (str): the password.

Returns:

`ExosUser`: successful authentication.

`None`: authentication failed

`class exos.api.`**`ExosUser`**`(`*`username`*`)`

An authenticated EXOS user. Instances of this class should be obtained via **`authenticate_user()`**, **`authenticated_user_async()`** or **`validate_session()`**. They should not be created directly.

`start_session(`*`remote_addr`*`)`

Start a session for this user. ***`Remote_addr`*** must be a string in IPv4 or IPv6 format. If this user already has a session, `AaaSessionExistsError` is thrown.

`end_session()`

End a session for this user. If this user does not have a session, `AaaSessionExistsError` is thrown.

`read_write = None`

True if the user has read/write privileges

`session_cookie = None`

The user's session cookie, if a session has been started

`session_start = None`

The time, as returned by time.time(), the session was started

`username = None`

The user's name

A session can be created, given an instance of `ExosUser`. Each session will be assigned a random session cookie. Applications can return that cookie to their client and later use it to retrieve the `ExosUser` again.

`exos.api.`**`validate_session`**`(`*`session_cookie`*`)`

Given a session_cookie, find the session.

Parameters:

session_cookie:

Returns:

`ExosUser`: if the session is found

`None`: if the session cannot be found

`exos.api.get_sessions()`

Return the list of active processes sessions. Will only return sessions created with *create_session()*.

Returns:

List of active sessions.

CLI Access

`exos.api.`**`exec_cli`**`(`*`cmds, timeout=0`*`)`

Send a list of commands to the CLI and return the output. This call will block until all commands have completed. This is a non-interactive, session-less version of the CLI. It doesn't prompt or page, meaning some CLI commands are not valid in it (example: `dis clipaging`).

Parameters:

cmds (str): a list of commands.

timeout (int): timeout value, defaults to 0.

Returns:

A string of the CLI output (str).

`exos.api.`**`exec_cli_async`**`(`*`cmds, callback, timeout=0`*`)`

Send a list of commands to the CLI and return the output via a callback. This function will return immediately.

Parameters:

cmds (str): a list of commands.

timeout (int): timeout value, defaults to 0.

callback will be called with each block of output.

The first parameter to *callback* is either `CLI_EVENT_EXEC_REPLY`, indicating this is just another block of output, or `CLI_EVENT_EXEC_DONE`, indicating a command has completed. A `CLI_EVENT_EXEC_DONE` will be passed for each command sent.

The second parameter to callback is the output itself, as a string.

`exos.api.CLI_EVENT_EXEC_REPLY`

Used in an `exec_cli_async()` callback to indicate this is part of the output.

`exos.api.`**`CLI_EVENT_EXEC_DONE`**

Used in an `exec_cli_async()` callback to indicate the current command has finished.

NOTE

The first release of the Python API included in EXOS 15.7.1 has a few limitations. One of them is about the `show config` CLI command. The implementation of this command is done in such a way that we cannot display the output yet.

Packet Manipulation Functions

The `expkt` module provides packet manipulation capabilities that can be used by a Python application to implement a network protocol.

- At Layer 2, data plane packets can be identified by MAC address or Ethertype and received into the Python application. From there, the application can process the packet as it pleases. It can also generate new packets and inject them back into the data plane.
- At Layer 3, the application, or individual sockets within the application, can bound to a virtual router, thereby scoping it.

It subclasses Python's `socket` module and can be used as a drop-in replacement.

```
from exos import api
from exos.api import expkt as socket
```

Layer 2

A Layer 2 socket is typically created as follows:

```
sock = socket.socket(socket.PF_EXOS_PACKET, socket.SOCK_RAW,
socket.htons(socket.ETH_P_ALL))
```

This socket will use the EXOS packet interface and will be able to send and receive raw packets to and from the data plane.

Receive

Prior to receiving packets on an L2 socket, a filter must be set up:

```
sock.setup_filter("filter1",
                  action = socket.UP_AND_CONTINUE,
                  dstmac = [0x00,0xE0,0x2B,0x00,0x00,0x00])
```

This filter means that any packet coming into the data plane that is destined for MAC address 00:E0:2B:00:00:00 will be put on the socket and also forwarded (UP_AND_CONTINUE).

Ethertype could also be used:

```
sock.setup_filter("filter2", ethertype=0x0842)
```

Once the filter is set up, the socket behaves as any other socket. For example, `select()` can be used to block for the next packet and `recvfrom()` can be used to retrieve a packet.

The following could be used to receive a packet:

```
pkt, addr = sock.recvfrom(8192)
```

In the above, `pkt` is a string containing the entire packet, including L2 headers, and `addr` is an `ExosSockAddr`.

Send

When sending on an L2 socket using the EXOS packet interface, a few extra attributes are available:

`slot_ports`: A sequence of tuples. Each tuple is a (slot, port) pair identifying a port on which the packet should be sent.

The following example sends on ports 1:1 and 1:2:

```
sock.sendto(pkt, slot_ports=[(1,1),(1,2)])
```

`vlan_name`: The VLAN on which the packet will be sent. The following example will forward the packet on VLAN foo:

```
sock.sendto(pkt, vlan_name="foo")
```

Layer 3

Layer 3 sockets are created as normal. Examples include:

```
udp_sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
tcp_sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

Every EXOS process has an affinity to a particular virtual router. This means the process will tend to do things, like open a socket, in the scope of that virtual router.

In the EXOS Python API, the VR affinity defaults to VR-Mgmt. However, this can be overridden at process startup via the `vr` option. How this option is passed depends on the startup mechanism. For example, when running the Python container (`expy`) directly, it is the `-v` option and when using *create process*, it is the `vr` parameter.

An individual socket can be bound to a different virtual router:

```
tcp_sock.set_vr("VR-Default")
```

Impacket

`Impacket` is a Python library that helps encode and decode packets. It is included as a convenience, but is not part of the EXOS Python API. More information can be found at the *Impacket* website at:
http://corelabs.coresecurity.com/index.php?module=Wiki&action=view&type=tool&name=Impacket

The following example uses `Impacket` to build an ICMP packet. The packet is then sent:

```
ip = ImpactPacket.IP()
ip.set_ip_src("10.11.12.13")
ip.set_ip_dst("13.12.11.10")
icmp = ImpactPacket.ICMP()
icmp.set_icmp_type(icmp.ICMP_ECHO)
icmp.contains(ImpactPacket.Data("PYTHON_COMPLETES_EXOS"*5))
icmp.set_icmp_id(42)
icmp.set_icmp_cksum(0)
icmp.auto_checksum = 1
ip.contains(icmp)

sock = socket.socket(socket.AF_INET, socket.SOCK_RAW, socket.IPPROTO_ICMP)
sock.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)
sock.sendto(ip.get_packet(), ("13.12.11.10", 0))
```

Asynchronous Callback

Sockets can be used asynchronously by setting a callback. When data is received on the socket, the callback is called. Typically, the callback would then receive on the socket and do something with the packet:

```
def packet_handler(handled_sock):
    pkt, addr = handled_sock.recvfrom(8192)
    # Do something with pkt

sock.set_callback(packet_handler)
```

Registering a socket callback has a similar result as creating an extra thread to block on the socket, except that the extra thread is not needed. Instead, a shared thread is used to block on many sockets, including ones internal to EXOS.

API

`class exos.api.expkt.`**`socket`**`(*args, **kwargs)`

An EXPKT socket. This socket object subclasses Python's socket class and can be used as a drop-in replacement. It adds EXOS-specific features, such as virtual router awareness and the ability to send and receive data plane traffic.

`detach_all_filters``()`

Removes all filters attached to this socket. Runs on *close()*.

`set_callback``(func, buf_size=9000, flags)`

Allows asynchronous reading of the socket. Takes a function to call back when a packet is received on the socket. The function must accept one argument, the socket.

`set_vr``(vr_name)`

Set the virtual router that the socket operates on. For L3 sockets only.

`setup_filter``(name, action=UP_AND_CONTINUE, dstmac=None, ethertype=None, vr=None, slot-Port=False, ethertype_offset=24)`

Set up the necessary filter to control what packets are sent to the socket.

`name` is required. Also, either `dstmac`, `ethertype`, or both are required.

Parameters:

`name` Name to use when creating an access-list for this filter. This name will be visible in the `show access-list` CLI command, for example.

`dstmac` Destination MAC address to filter on. If not provided, then ethertype must be provided.

`ethertype` Ethertype to filter on. If not provided, then dstmac must be provided.

`vr` Name of a virtual router to filter on. By default, the filter will run on all virtual routers.

`action` Action to take when the filter is matched. Valid actions include UP_AND_CONTINUE and UP_AND_DROP.

`ethertype_offset` The location of the ethertype in the packet (measured in bytes). The default is 24.

`unset_callback()`

Removes the callback associated with the socket. Runs on *close()*.

`exos.api.expkt.PF_EXOS_PACKET`

Socket domain for the EXOS packet (EXPKT) interface. Sockets created in this domain will be able to send and receive raw packets to and from the data plane.

`exos.api.expkt.`**`ETH_P_ALL`**

Socket protocol that indicates the socket receives all protocols.

`exos.api.expkt.`**`UP_AND_CONTINUE`**

Filter action indicating a matched packet must be put on the socket and also forwarded by the data plane. The packet on the socket is thus a copy of the packet.

`exos.api.expkt.`**`UP_AND_DROP`**

Filter action indicating a matched packet must be put on the socket and not forwarded by the data plane. This is not a copy of the packet, but the packet itself, ready for processing.

`exos.api.expkt.`**`EGRESS_AND_INGRESS`**

Filter direction indicating the filter matches both incoming and outgoing packets.

`exos.api.expkt.`**`SO_VRID`**

Socket option to get or set a socket's virtual router, to be used with *getsockopt()* and *setsockopt()*. The value is the VR's id (vrid), not the VR's name. To set the virtual router by name, see *set_vrid()*.

For example, the following puts sock2 in the same VR as sock1:

```
vrid = sock1.getsockopt(SOL_SOCKET, socket.SO_VRID)
sock2.setsockopt(SOL_SOCKET, socket.SO_VRID, vrid)
```

Process Creation

Once we know how to use the EXOS Python API, we need to be able to create processes from our Python programs. To create an EXOS process for a Python module, you need to use the following command in EXOS CLI:

```
create process <name> python-module <python-module> {start [auto | on-
demand]} {vr <vr-name>} {description <description>} {<arg1> {<arg2>
{<arg3> {<arg4> {<arg5> {<arg6> {<arg7> {<arg8> {<arg9>}}}}}}}}}
```

The `start` parameter specifies the starting behavior of the process:

- If `auto` is used, a daemon process is created and started immediately.
- If `on-demand` is used, it is necessary to use the *start process <name>* command to start it. The process is then a "run to completion" process.

The default behavior is `on-demand`.

If not specified, the default VR is "VR-Mgmt."

If you want to delete a process that you have created, you need to use the following command in EXOS: `delete process <name>`

This command doesn't prompt for a confirmation; only a dynamically created process can be deleted. This EXOS command cannot delete a non user-created process. Here's an example:

```
(Engineering) VM1.2 # create process CheckVersionPy python-module
CheckVersionPython start auto
creating CheckVersionPy...
* (Engineering) PE2.3 #
* (Engineering) PE2.3 # show process
Process Name     Version  Restart    State            Start Time
-------------------------------------------------------------------------
CheckVersionPy   User        0    Ready        Wed Jan  7 17:58:53 2015
aaa              3.0.0.3     0    Ready        Wed Jan  7 17:53:57 2015
acl              3.0.0.2     0    Ready        Wed Jan  7 17:53:57 2015
[…]
```

NOTE

While it is perfectly fine to add the `.py` suffix of the filename in the scripting environment, the `.py` suffix must be *excluded* from the `create process` CLI command when creating a daemon.

It is then possible to have a detailed view of the process:

```
* (Engineering) VM1.4 # show process "CheckVersionPy"
  <cr>            Execute the command
  description     Process name description
  detail          Detailed information
  |               Filter the output of the command
* (Engineering) VM1.4 #
* (Engineering) VM1.4 # show process "CheckVersionPy" detail
Name          PID     Path    Type Link Date                    Build By
Peer
--------------------------------------------------------------------------
CheckVersionPy 1870   /exos/bin/expy App  Thu Dec 18 18:37:25 EST 2014
release-manager 70
Virtual Router(s):
--------------------------------------------------------------------------

Configuration:
Start Priority  SchedPolicy  Stack  TTY  CoreSize  Heartbeat  StartSeq
--------------------------------------------------------------------------
1        10        0         0       0     0           1         1
Memory Usage Configuration:
Memory(KB) Zones: Green Yellow Orange Red
--------------------------------------------------------------------------
 500000000          0     0     0     0

Recovery policies
--------------------------------------------------------------------------
none
--------------------------------------------------------------------------
Statistics:
ConnectionLost  Timeout  Start  Restart  Kill  Register  Signal  Hello
Hello Ack
```

```
--------------------------------------------------------------------------
0              0         0     0         0      1       0        0
5
Memory Zone  Green   Yellow   Orange    Red
--------------------------------------------------------------------------
 Green       0          0        0        0
--------------------------------------------------------------------------
Commands:
  Start        Stop       Resume        Shutdown          Kill
--------------------------------------------------------------------------
  0             0            0            0                 0
--------------------------------------------------------------------------
Resource Usage:
UserTime SysTime  PageReclaim PageFault Up Since                Up Date
Up Time
--------------------------------------------------------------------------
0.1      0.6         1256      15     Wed Jan  7 17:58:53 2015 00/00/00
00:00:44
--------------------------------------------------------------------------

  Thread Name              Pid         Tid       Delay  Timeout Count
--------------------------------------------------------------------------
 main                    1870       -1213282624   10         0
--------------------------------------------------------------------------
* (Engineering) VM1.5 #
```

If required, the process can be terminated:

```
* (Engineering) VM1.6 # terminate process "CheckVersionPy" graceful
Do you want to save configuration changes to currently selected
configuration
file (primary.cfg)? (y or n) No
You will lose CheckVersionPy's configuration if you save the configuration
after
terminating this process. Do you want to continue? (y/N) Yes
Successful graceful termination for CheckVersionPy
* (Engineering) VM1.7 #
* (Engineering) VM1.7 # show process
Process Name     Version  Restart    State            Start Time
-------------------------------------------------------------------------
CheckVersionPy   User        0    Stopped      Wed Jan  7 17:58:53 2015
aaa              3.0.0.3     0    Ready        Wed Jan  7 17:53:57 2015
acl              3.0.0.2     0    Ready        Wed Jan  7 17:53:57 2015
bfd              1.0.0.1     0    Ready        Wed Jan  7 17:53:57 2015
bgp              4.0.0.1     0    Ready        Wed Jan  7 17:53:57 2015
bgp-3            4.0.0.1     0    Ready        Wed Jan  7 17:54:05 2015
brm              1.0.0.0     0    Ready        Wed Jan  7 17:53:57 2015
[…]
```

Some logs are provided by default:

```
(Engineering) VM1.8 # sh log | inc CheckVersionPy
01/07/2015 18:11:56.79 <Noti:DM.Notice> Process CheckVersionPy Stopped
01/07/2015 18:11:56.79 <Info:EPM.Msg.proc_shutdown> Requested process
CheckVersionPy shutdown
```

Python Native App, when started automatically, are persistent upon a reboot of the system. They are part of the config file:

```
(Engineering) VM1.5 # sh config | inc CheckVersion
create process CheckVersionPy python-module CheckVersionPython start auto
```

We can also easily list all the user-created processes running on a switch:

```
(Engineering) X440-8t.3 # show process | inc User
mySNTP           User        0    Ready        Tue Jan 20 21:12:47 2015
testapi          User        0    Ready        Tue Jan 20 20:58:05 2015
(Engineering) X440-8t.4 #
```

Precautions with Python App

When creating processes with EXOS Python API, those processes are managed by EPM, just like any other EXOS processes. EPM, the process manager, is generally responsible for throttling processes and avoiding switch death. These Python daemons run under EPM as well, so they inherit that functionality.

It's the developer's responsibility to take care to not "kill" the switch. Memory is a limited resource and the amount available is system- and configuration-dependent. If too much memory is consumed, the switch will not operate correctly. By default, EPM limits a process to 50 MB of memory. If the process goes over the limit, EPM kills the process.

To monitor the amount of memory available and used by a process, there are no functions available in the API. This can be monitored via `show process <name> detail or show memory process <name>` CLI command.

Examples

When dealing with the daemon environment, a few habits need to be changed compared to the scripting environment.

- The `.py` suffix must not be part of the process creation.
- The `print` function can only print to the console, as no other output is guaranteed in the system when dealing with daemons. Instead of using the `print` function, it is advised to use the `trace logging` capability.
- The daemon should validate it is running in the Python container (`expy`), not as a script.

Basic Pattern

Below is an example showing the minimal pattern to follow:

```
import sys
import logging
from exos import api

# Setup logging.
logger = logging.getLogger('myprog')
logger.setLevel(logging.DEBUG)
logHandler = api.TraceBufferHandler("mybuf", 5120)
logHandler.setLevel(logging.DEBUG)
logHandler.setFormatter(logging.Formatter("%(threadName)s:%(name)s:%(funcN
ame)s.%(lineno)s:: %(message)s"))
logger.addHandler(logHandler)

try:
    def main():
        # Verify we're under EXPY.  We can't live without it.
        if not hasattr(sys, 'expy') or not sys.expy:
            print "Must be run within expy"
            return

        # your program runs here

    main()

except BaseException, e:
    logger.error("Exception on startup, {}".format(e), exc_info=True)
```

Application Exercise

Let's write a first Python App that will simply give us the Python version running on the daemon environment. This app will check that it is running in `expy` and, if so, will log the Python version in a log file.

```
import sys
import logging
from exos import api

logger = logging.getLogger('MyFirstApp')
logger.setLevel(logging.DEBUG)

logHandler = api.TraceBufferHandler("MyBuffer", 5120)
logHandler.setLevel(logging.DEBUG)
logHandler.setFormatter(logging.Formatter("%(levelname)s:%(threadName)s:%(
name)s:%(funcName)s.%(lineno)s:: %(message)s"))

logger.addHandler(logHandler)


try:

    def main():
        if not hasattr(sys, 'expy') or not sys.expy:
            print "Must be run within expy"
            return

        version =
str(sys.version_info[0])+'.'+str(sys.version_info[1])+'.'+str(sys.version_
info[2])
        logger.info('Python version is: {}'.format(version))


    main()

except BaseException, e:
    logger.error("Exception on startup, {}".format(e), exc_info=True)
```

Let’s detail what is happening here.

We first start by importing all the required modules. This is very straight-forward so we directly jump to the next block, the logging definition.

In the first line, we initialize a logger.

```
import sys
import logging
from exos import api

logger = logging.getLogger('MyFirstApp')
logger.setLevel(logging.DEBUG)

logHandler = api.TraceBufferHandler("MyBuffer", 5120)
logHandler.setLevel(logging.DEBUG)
logHandler.setFormatter(logging.Formatter("%(levelname)s:%(threadName)s:%(
name)s:%(funcName)s.%(lineno)s:: %(message)s"))

logger.addHandler(logHandler)


try:

    def main():
        if not hasattr(sys, 'expy') or not sys.expy:
            print "Must be run within expy"
            return

        version =
str(sys.version_info[0])+'.'+str(sys.version_info[1])+'.'+str(sys.version_
info[2])
        logger.info('Python version is: {}'.format(version))


    main()

except BaseException, e:
    logger.error("Exception on startup, {}".format(e), exc_info=True)
```

Then we define what logging level will be able to be registered in the logger. This is the usual Unix logging levels:

`Debug (10):` typically any information useful for debugging.

`Info (20):` to give information on the application.

`Warning (30)`: to give some information on a specific case.

`Error (40)`: something went wrong.

`Critical (50)`: this is very bad.

We can write these information with the appropriate level using:

```
logger.(info | warning | debug | error | critical)
```

After that, we create a log handler that will redirect the information that need to be written into a given buffer. In this case, we are using the EXOS `TraceBufferHandler()` function, with a buffer name and a buffer size. Once again we need to set the logging level to the handler, then we define the formatting of the logs and we can finally associate the handler with the *logger* created.

The `main()` function is included into a `Try/Except` statement to catch any errors.

The first thing we check is that we are running in the daemon environment (`expy`). If this is not the case, we exit the function, and so the program.

```
import sys
import logging
from exos import api

logger = logging.getLogger('MyFirstApp')
logger.setLevel(logging.DEBUG)

logHandler = api.TraceBufferHandler("MyBuffer", 5120)
logHandler.setLevel(logging.DEBUG)
logHandler.setFormatter(logging.Formatter("%(levelname)s:%(threadName)s:%(
name)s:%(funcName)s.%(lineno)s:: %(message)s"))

logger.addHandler(logHandler)


try:

    def main():
        if not hasattr(sys, 'expy') or not sys.expy:
            print "Must be run within expy"
            return

        version =
str(sys.version_info[0])+'.'+str(sys.version_info[1])+'.'+str(sys.version_
info[2])
        logger.info('Python version is: {}'.format(version))


    main()

except BaseException, e:
    logger.error("Exception on startup, {}".format(e), exc_info=True)
  return
```

Finally, we run our code to check the Python version. We are logging the result into the logger, with an `info` logging level.

```
import sys
import logging
from exos import api

logger = logging.getLogger('MyFirstApp')
logger.setLevel(logging.DEBUG)

logHandler = api.TraceBufferHandler("MyBuffer", 5120)
logHandler.setLevel(logging.DEBUG)
logHandler.setFormatter(logging.Formatter("%(levelname)s:%(threadName)s:%(
name)s:%(funcName)s.%(lineno)s:: %(message)s"))

```

```
logger.addHandler(logHandler)


try:

    def main():
        if not hasattr(sys, 'expy') or not sys.expy:
            print "Must be run within expy"
            return

        version =
str(sys.version_info[0])+'.'+str(sys.version_info[1])+'.'+str(sys.version_
info[2])
        logger.info('Python version is: {}'.format(version))


    main()

except BaseException, e:
    logger.error("Exception on startup, {}".format(e), exc_info=True)
  return
```

This is what we have on a switch:

```
* (Engineering) VM1.6 # tftp get 192.168.56.1 MyFirstApp.py
Downloading MyFirstApp.py on primary Node ... done!
* (Engineering) VM1.7 #
* (Engineering) VM1.7 # create process MyFirstApp python-module MyFirstApp
start auto
creating MyFirstApp...
* (Engineering) VM1.8 #
* (Engineering) VM1.8 #
* (Engineering) VM1.8 # debug ems show trace "MyFirstApp" MyBuffer
01/12/2015 23:48:46.264771 [183] <MyFirstApp:MyBuffer> Begin trace buffer
01/12/2015 23:48:46.264862 [184] <MyFirstApp:MyBuffer>
INFO:ExpyRunner:MyFirstApp:main.24:: Python version is: 2.7.3
* (Engineering) VM1.9 #
```

Of course, this example is not really useful as an application because it runs only once, at startup. Doing it that way, the process will last needlessly in EXOS. It would be more appropriate to execute it *on-demand*.

mySNTP

Let's use a simple example to demonstrate the basic use of *sockets*. In the below Python code, we are creating a SNTP Client program that will log the UTC time when executed. We have added the use of an optional external file to display the correct time depending on the timezone. Once again, this App is more like a script in its spirit, as it is executed only once, at startup. The goal here is more to show how to use all the mechanisms of the API, rather than building a real and useful App.

```
#!/usr/bin/python

import sys
import struct
import time
from exos import api
from exos.api import expkt as socket
```

```
import logging

logger = logging.getLogger('mySNTP')
logger.setLevel(logging.DEBUG)

logHandler = api.TraceBufferHandler("mySNTPBuffer", 5120)
logHandler.setLevel(logging.DEBUG)
logHandler.setFormatter(logging.Formatter("%(levelname)s:%(name)s:%(funcNa
me)s.%(lineno)s:: %(message)s"))

logger.addHandler(logHandler)

NTP_SERVER = '194.57.169.1'   # ntp.univ-angers.fr
TIME1970 = 2208988800L
CONFIG_FILE = '/usr/local/cfg/mySNTP.txt'

try:
    def main():
        if not hasattr(sys, 'expy') or not sys.expy:
            print "Must be run within expy"
            return

        try:
            with open(CONFIG_FILE,'r') as fd:
                s = fd.read()
        except:
            s = None

        if s:
            if s[3] == '+':
                table = s.split('+')
                GMT = int(table[1])
            elif s[3] == '-':
                table = s.split('-')
                GMT = int(table[1])
                GMT *= -1
            else:
                GMT = 0
        else:
            GMT = 0


        client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        client.set_vr('VR-Default')
        data = '\x1b' + 47 * '\0'
        client.sendto(data, (NTP_SERVER, 123))
        data, address = client.recvfrom(1024)

        if data:
            logger.info('Response received from {}'.format(address))

        t = struct.unpack('!12I', data)[10]
        t -= TIME1970

        logger.info('UTC Time is {}'.format(time.ctime(t)))

        if GMT != 0:
            t += GMT*3600
            logger.info('Configured time in {} is {}: Time is
{}'.format(CONFIG_FILE, s, time.ctime(t)))
```

```
    main()

except BaseException, e:
    logger.error("Exception on startup, {}".format(e), exc_info=True)
```

To better demonstrate the EXOS Pyhton API, we create the process in VR VR-Mgmt (the default VR used if not specified). The switch running that Python app has internet connectivity only through VR-Default, so we must change the VR in our code.

```
(Engineering) X460G2-24p-10G4.6 # create process mySNTP python-module
mySNTP start auto
creating mySNTP...
* (Engineering) X460G2-24p-10G4.7 #
```

Our program first tries to read the config file. If it gets an error trying to open it, we catch that error through the `try/except` statement. We consider in that case that the file is not present, and we continue the execution of the code.

```
try:
    with open(CONFIG_FILE,'r') as fd:
        s = fd.read()
except:
    s = None
```

If we are able to open the file, we parse it to find what GMT offset is in it.

```
if s:
    if s[3] == '+':
        table = s.split('+')
        GMT = int(table[1])
    elif s[3] == '-':
        table = s.split('-')
        GMT = int(table[1])
        GMT *= -1
    else:
        GMT = 0
else:
    GMT = 0
```

If we cannot find the expected data, or if the file is not present, we simply set the GMT variable to 0.

The most interesting part is how we establish a network communication between our app and a NTP server in the internet.

We first create a L3 socket, using IPv4 and UDP, named *client*. Then we make sure the socket is bound to the required VR (VR-Default), so that communication is possible (our switch can communicate with the outside only through this VR).

```
client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
client.set_vr('VR-Default')
data = '\x1b' + 47 * '\0'
client.sendto(data, (NTP_SERVER, 123))
data, address = client.recvfrom(1024)
```

After creating the data we need to send (as defined by NTP), we send it to the configured server on the appropriate UDP port (the NTP port).

Our program then waits for an answer (1024 bytes long) from the NTP server.

The rest of the code is just processing of this information to log it into our logger. We can see the result looking at our logger:

```
(Engineering) X460G2-24p-10G4.7 # debug ems show trace mySNTP mySNTPBuffer
01/20/2015 22:11:18.003005 [185] <mySNTP:mySNTPBuffer> Begin trace buffer
01/20/2015 22:11:18.058736 [208] <mySNTP:mySNTPBuffer>
INFO:mySNTP:main.55:: Response received from ('194.57.169.1', 123)
01/20/2015 22:11:18.062694 [209] <mySNTP:mySNTPBuffer>
INFO:mySNTP:main.60:: UTC Time is Tue Jan 20 14:06:56 2015
01/20/2015 22:11:18.065896 [210] <mySNTP:mySNTPBuffer>
INFO:mySNTP:main.64:: Configured time in /usr/local/cfg/mySNTP.txt is
GMT+1: Time is Tue Jan 20 15:06:56 2015
(Engineering) X460G2-24p-10G4.8 #
```

Future

Over time, the differences between the two environments—scripting and daemon—should be less and less present.

Also, the version of the Python API included in 15.7.1 is the first release. There are a few limitations (see the note for the **`show config`** example) as it addresses a few simple use cases. Extreme Networks has already done quite a bit of work since this release and will cover more use cases in the future.

More importantly, the Python SDK should evolve to allow more interaction with EXOS. Among the key features that shall be included, we have:

- CM backend
- Extending CLI
- Events

While Python enhancements are targeted for a future release, earlier tech releases might include some of this functionality.

With CM backend support and CLI, the user/third-party has the opportunity to create new CLI commands. For example, creating a process *foo*, it would be possible to do things like **`show foo`** and so on.

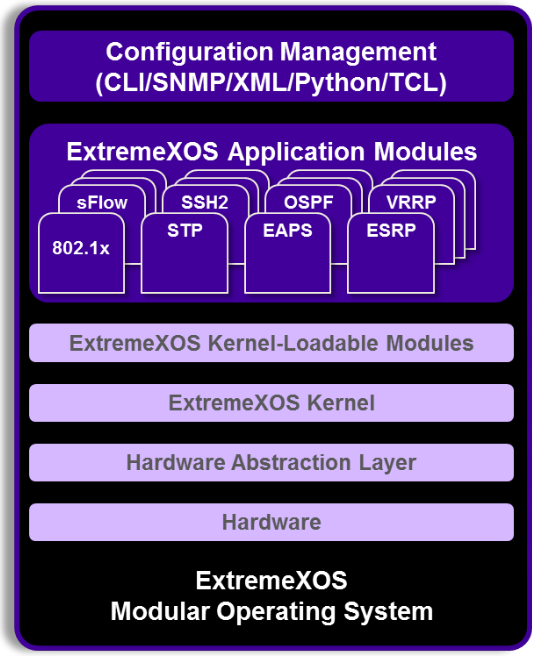
The three of these features will allow the creation of a "mini-controller," embedded in EXOS, allowing the switch to do things when other things happen.

EXOS SDK

The EXOS Python API is another way to interact with EXOS. It interacts with the C API. The current version of the C API allows for CLI creation and could be made available internally upon request.

This is extending the Open Architecture of EXOS, and provides an easy way around the controlled release cycle of EXOS. It gives the opportunity to deliver prototype quickly and work on formalizing it in a later release.

- Northbound API
  - OF, XML, SNMP, CLI, Python
- Open SDK
  - XML, C, Python
- Event-Driven Agent
  - UPM
- Automation
  - TCL, Python
- Linux Core
  - Independent processes
- Hardware Abstraction Layer
  - Single OS



Publication History

Date	Version	Author	Notes
January 2015	0.1	Stéphane Grosjean	Initial release
February 2015	1.0	Christina E. Mayr	Edit and format