Script Infrastructure and Sample Templates

Table 1: Feature History Table

Feature Name	Release Information	Description
Contextual Script Infrastructure	Release 7.3.2	When you create and run Python scripts on the router, this feature enables a contextual interaction between the scripts, the IOS XR software, and the external servers. This context, programmed in the script, uses Cisco IOS XR Python packages, modules, and libraries to: • obtain operational data from the router • set configurations and conditions • detect events in the network and trigger an appropriate action

You can create Python scripts and execute the scripts on routers running Cisco IOS XR software. The software supports the Python packages, libraries and dictionaries in the software image. For more informtion about the script types and to run the scripts using CLI commands To run the same actions using NETCONF RPCs,

Cisco IOS XR, Release 7.3.2 supports creating scripts using Python version 3.5.

Cisco IOS XR, Release 7.5.1 supports creating scripts using Python version 3.9.

Cisco IOS XR Python Packages

Table 2: Feature History Table

Feature Name	Release Information	Description
Upgraded IOS XR Python from Version 3.5 to Version 3.9	Release 7.5.1	This upgrade adds new modules and capabilities to create Python scripts and execute the scripts on routers running Cisco IOS XR software. Some of the modules added as part of the upgraded IOS XR Python 3.9 are: hashlib, idna, packaging, pyparsing, six, yaml.

With on-box Python scripting, automation scripts that was run from an external controller is now run on the router. To achieve this functionality, Cisco IOS XR software provides contextual support using SDK libraries and standard protocols.

The following Python third party application packages are supported by the scripting infrastructure and can be used to create automation scripts.

Package	Description	Support Introduced in Release
appdirs	Chooses the appropriate platform-specific directories for user data.	Release 7.3.2
array	Defines an object type that can compactly represent an array of basic values: characters, integers, floating point numbers.	Release 7.3.2
asn1crypto	Parses and serializes Abstract Syntax Notation One (ASN.1) data structures.	Release 7.3.2
chardet	Universal character encoding auto-detector.	Release 7.3.2
concurrent.futures	Provides a high-level interface for asynchronously executing callables.	Release 7.3.2
ecdsa	Implements Elliptic Curve Digital Signature Algorithm (ECDSA) cryptography library to create keypairs (signing key and verifying key), sign messages, and verify the signatures.	Release 7.3.2

Package	Description	Support Introduced in Release
enum	Enumerates symbolic names (members) bound to unique, constant values.	Release 7.3.2
email	Manages email messages.	Release 7.3.2
google.protobuf	Supports language-neutral, platform-neutral, extensible mechanism for serializing structured data.	Release 7.3.2
hashlib	Implements a common interface to many different secure hash and message digest algorithms.	Release 7.5.1
idna	Supports the Internationalized Domain Names in Applications (IDNA) protocol as specified in RFC 5891.	Release 7.5.1
ipaddress	Provides capability to create, manipulate and operate on IPv4 and IPv6 addresses and networks.	Release 7.3.2
jinja2	Supports adding functionality useful for templating environments.	Release 7.3.2
json	Provides a lightweight data interchange format.	Release 7.3.2
markupsafe	Implements a text object that escapes characters so it is safe to use in HTML and XML.	Release 7.3.2
netaddr	Enables system-independent network address manipulation and processing of Layer 3 network addresses.	Release 7.3.2
packaging	Add the necessary files and structure to create the package.	Release 7.5.1
pdb	Defines an interactive source code debugger for Python programs.	Release 7.3.2
pkg_resources	Provides runtime facilities for finding, introspecting, activating and using installed distributions.	Release 7.3.2

Package	Description	Support Introduced in Release
psutil	Provides library to retrieve information on running processes and system utilization such as CPU, memory, disks, sensors and processes.	Release 7.3.2
pyasn1	Provides a collection of ASN.1 modules expressed in form of pyasn1 classes. Includes protocols PDUs definition (SNMP, LDAP etc.) and various data structures (X.509, PKCS).	Release 7.3.2
pyparsing	Provides a library of classes to construct the grammar directly in Python code.	Release 7.5.1
requests	Allows sending HTTP/1.1 requests using Python.	Release 7.3.2
shellescape	Defines the function that returns a shell-escaped version of a Python string.	Release 7.3.2
six	Provides simple utilities for wrapping over differences between Python 2 and Python 3.	Release 7.5.1
subprocess	Spawns new processes, connects to input/output/error pipes, and obtain return codes.	Release 7.3.2
urllib3	HTTP client for Python.	Release 7.3.2
xmltodict	Makes working with XML feel like you are working with JSON.	Release 7.3.2
yaml	Provides a human-friendly format for structured data, that is both easy to write for humans and still parsable by computers.	Release 7.5.1

Cisco IOS XR Python Libraries

Cisco IOS XR software provides support for the following SDK libraries and standard protocols.

Library	Syntax

xrlog	`# To generate syslogs #`
	`from cisco.script_mgmt import xrlog`
	
	`syslog = xrlog.getSysLogger('template_exec')`
	For more information, see Xrlog Python Module, on page 12.

netconf	`#To connect to netconf client #`
	`from iosxr.netconf.netconf_lib import NetconfClient`
	
	`nc = NetconfClient(debug=True)`

xrclihelper	`# To run native xr cli and config commands`
	`from iosxr.xrcli.xrcli_helper import *`
	
	`helper = XrcliHelper(debug = True)`
	For more information, see Xrcli_helper Python Module, on page 8.

config_validation	`# To validate configuration #`
	`import cisco.config_validation as xr`
	For more information, see *Config Scripts* Chapter.

eem	`# For EEM operations #`
	`from iosxr import eem`
	For more information, see *EEM Scripts* Chapter.

precommit	`# For Precommit script operations #`
	`from cisco.script_mgmt import precommit`
	For more information, see *Precommit Scripts* Chapter.

Sample Script Templates

Table 3: Feature History Table

Feature Name	Release Information	Description
Github Repository for Automation Scripts	Release 7.5.1	You now have access to sample scripts and templates published on the Github repository. You can leverage these samples to use the python packages and libraries developed by Cisco to build your custom automation scripts for your network

Use these sample script templates based on script type to build your custom script.

To get familiar with IOS XR Python scripts, see the samples and templates on the Cisco Devnet developer program and Github repository.

Follow these instructions to download the sample scripts from the Github repository to your router, and run the scripts:

1. Clone the Github repository.

   ```
   $git clone https://github.com/CiscoDevNet/iosxr-ops.git
   ```

2. Copy the Python files to the router's harddisk or a remote repository.

Config Script

The following example shows a code snippet for config script. Use this snippet in your script to import the libraries required to validate configuration and also generate syslogs.

```
#Needed for config validation
import cisco.config_validation as xr

#Used for generating syslogs
from cisco.script_mgmt import xrlog
syslog = xrlog.getSysLogger('Add script name here')

def check_config(root):
    #Add config validations
    pass

xr.register_validate_callback([<Add config path here>],check_config)
```

Exec Script

Use this sample code snippet in your exec script to import Python libraries to connect to NETCONF client and also to generate syslogs.

```
#To connect to netconf client
from iosxr.netconf.netconf_lib import NetconfClient

#To generate syslogs
syslog = xrlog.getSysLogger('template_exec')

def test_exec():
    """
    Testcase for exec script
    """
    nc = NetconfClient(debug=True)
    nc.connect()
    #Netconf or processing operations
    nc.close()


if __name__ == '__main__':
    test_exec()
```

Process Script

Use the following sample code snippet to trigger a process script and perform various actions on the script. You can leverage this snippet to create your own custom process script. Any exec script can be used as a process script.

```
To trigger script
Step 1: Add and configure script as shown in README.MD
```

```
Step 2: Register the application with Appmgr

Configuraton:
appmgr process-script my-process-app
executable test_process.py
run args --threshold <threshold-value>

Step 3: Activate the registered application
appmgr process-script activate name my-process-app

Step 4: Check script status
show appmgr process-script-table

Router#show appmgr process-script-table
Name             Executable         Activated    Status     Restart Policy   Config Pending
---------------  ------------------ ---------- ------------- ---------------- --------------
my-process-app   test_process.py      Yes       Running    On Failure            No

Step 5: More operations
Router#appmgr process-script ?
  activate    Activate process script
  deactivate  Deactivate process script
  kill        Kill process script
  restart     Restart process script
  start       Start process script
  stop        Stop process script
"""

#To connect to netconf client
from iosxr.netconf.netconf_lib import NetconfClient

#To generate syslogs
syslog = xrlog.getSysLogger('template_exec')

def test_process():
    """
    Testcase for process script
    """
    nc = NetconfClient(debug=True)
    nc.connect()
    #Netconf or any other operations
    nc.close()


if __name__ == '__main__':
    test_process()
```

EEM Script

You can leverage the following sample code to import Python libraries to create your custom eem script and also generate syslogs.

```
Required configuration:
User and AAA configuration

event manager event-trigger <trigger-name>
type syslog pattern "PROC_RESTART_NAME"

event manager action <action-name>
username <user>
type script script-name <script-name> checksum sha256 <checksum>

event manager policy-map policy1
```

```
trigger event <trigger-name>
action <action-name>

To verify:
Check for syslog EVENT SCRIPT EXECUTED: User restarted <process-name>

"""
#Needed for eem operations
from iosxr import eem

#Used to generate syslogs
from cisco.script_mgmt import xrlog
syslog = xrlog.getSysLogger(<add your script name here>)

# event_dict consists of details of the event
rc, event_dict = eem.event_reqinfo()

#You can process the information as needed and take action for example: generate a syslog.
#Syslog type can be emergency, alert, critical, error, exception, warning, notification,
info, debug

syslog.info(<Add you syslog here>)
```

Xrcli_helper Python Module

Overview of xrcli_helper Python Module

The `XrcliHelper` is a utility class designed to facilitate the execution of IOS-XR CLI commands and configuration changes programmatically. It provides methods to:

- Execute native IOS-XR commands.
- Apply configurations from files or strings.

Prerequisites of xrcli_helper Python Module

- Python 2.7 or higher.
- Ensure that you are on Cisco IOS XR Release 7.4.x or higher.
- Access to Cisco IOS XR device with AAA Authorization enabled. Use the **aaa authorization exec default group tacacs+ local** command to enable AAA Authorization.
- Ensure that the iosxr.xrcli.xrcli_helper module is available in your Python environment.

Import Library Information

To use the XrcliHelper class in your Python script, you need to import it from the appropriate module. The import statement provided allows you to bring the XrcliHelper class into your script so you can create instances of it and use its methods.

```
from iosxr.xrcli.xrcli_helper import XrcliHelper
```

Library/API Initialization

By initializing the XrcliHelper class, you establish the environment needed to execute IOS-XR commands and apply configurations programmatically. This serves as the initial step in automating network management tasks, enabling you to utilize the class's methods to efficiently interact with your IOS-XR devices.

```
<object name> = XrcliHelper([debug=True/False(default)])
```

This example shows how to initialize Xrclihelper class.

```
helper = XrcliHelper()
```

Xrcli_helper Script APIs

xrcli_exec

The xrcli_exec API executes IOS-XR exec commands to obtain the output.

Parameter

cmd: A String representing the IOS- XR exec command to be executed.

Result

The result of the xrcli_exec API is a dictionary containing:

- `status`: Indicates whether the command execution was `error` or `success`.
- `output`: The output of the executed command.

Example

The following example shows the sample output of xrcli_exec API:

```
>>> result = helper.xrcli_exec("show filesystem ")

>>> print(result)
{'output': '\n'
           '------------------------------ show filesystem '
           '------------------------------\n'
           'File Systems:\n'
           '\n'
           '      Size(b)      Free(b)        Type  Flags  Prefixes\n'
           '   4275265536   4274974720  flash-disk     rw  disk0:\n'
           '  67301322752  67266158592    harddisk     rw  harddisk:\n'
           '            0            0     network     rw  ftp:\n'
           '  60264796160  51056054272       flash     rw  /misc/config\n'
           '            0            0     network     rw  tftp:\n',
'status': 'success'}
```

xr_apply_config_file

The xr_apply_config_file API applies configuration to IOS-XR using a file.

Parameter

- `filename`: Path to a configuration file containing XR config commands with the following structure:

  ```
  !
  XR config command
  ```

```
!
end
```

- `comment`: A comment for the configuration commit, which will be visible in the output of **show configuration commit list detail**.

Result

The result of xr_apply_config_file is a dictionary specifying the effect of the configuration change:

- `status`: Indicates whether the configuration application was `error` or `success`.
- `output`:
  - If `status` is `error`: use the **show configuration failed** command.
  - If `status` is `success`: use the **show configuration commit changes last 1** command.

Example

The following example shows sample output of xr_apply_config_file API.

```
[node0_RP0_CPU0:~]$more /harddisk:/noshut_int.cfg
!
interface hundredGigE 0/0/0/24
no shutdown
interface hundredGigE 0/0/0/25
no shutdown
!
end

>>> result = helper.xr_apply_config_file("/harddisk:/noshut_int.cfg")
>>> print(result)
{'output': '\n'
           '------------------- show configuration commit changes last 1 '
           '--------------------\n'
           '!! Building configuration...\n'
           '!! IOS XR Configuration x.y.z \n'
           'interface HundredGigE0/0/0/24\n'
           ' no shutdown\n'
           '!\n'
           'interface HundredGigE0/0/0/25\n'
           ' no shutdown\n'
           '!\n'
           'end\n'
           '\n',
'status': 'success'}
>>>
```

xr_apply_config_string

The xr_apply_config_string applies configuration to XR using a single line string.

Parameter

`cmd`: Single line string representing an XR config command.

`comment`:Reason for the config commit, visible in **show configuration commit list detail**.

Result

The result of xr_apply_config_string is a dictionary specifying the effect of the configuration change:

- **status**: Indicates whether the configuration application was `error` or `success`.
- **output**:
  - If `status` is `error`: use the **show configuration failed** command.
  - If `status` is `success`: use the **show configuration commit changes last 1** command.

Example

The following example shows sample output of xr_apply_config_file API.

```
>>> cmd = """
... interface HundredGigE0/0/0/25
... description "shut down by scriptx"
... shut
... """
>>> result = helper.xr_apply_config_string(cmd)
>>> print(result)
{'output': '\n'
           '------------------- show configuration commit changes last 1 '
           '-------------------\n'
           '!! Building configuration...\n'
           '!! IOS XR Configuration x.y.z\n'
           'interface HundredGigE0/0/0/25\n'
           ' description "shut down by scriptx"\n'
           ' shutdown\n'
           '!\n'
           'end\n'
           '\n',
'status': 'success'
>>>
```

user

The user is an XrcliHelper Object Attribute (not API) which contains the username to authorize the XR commands.

Example

The following example shows sample output of `user`.

```
Example:

>>> helper.user
'cisco'
```

toggle_debug

The toggle_debug enables or disables debug logging.

Example

The following example shows sample output of `toggle_debug`.

```
>>> helper.toggle_debug(True)
>>>
```

Xrlog Python Module

Overview of Xrlog Python Module

The `xrlog` Python module is a utility designed for generating syslog messages and script logs within Cisco IOS XR environments. It provides methods to do the following:

- Module to generate XR syslog messages from scripts.
- Provides a logger for generating script logs.

Prerequisites of Xrlog Python Module

- Python 2.7 or higher.
- Ensure that you are on Cisco IOS XR Release 7.4.x or higher.
- Ensure that the `cisco.script_mgmt.xrlog` is available in your Python environment.

Import Library Information

To use the `xrlog` module in your Python script, you need to import it from the appropriate module. The import statement provided allows you to bring the `xrlog` functionalities into your script so you can create instances of syslog and script loggers.

```
from cisco.script_mgmt import xrlog
```

Library/API Initialization

By initializing the `xrlog` module, you establish the environment needed to generate syslog messages and script logs programmatically. This serves as the initial step in automating logging tasks, enabling you to utilize the module's methods to efficiently log events and messages.

```
<object_name> = xrlog.getSysLogger([logger_name [default: root]])
<object_name> = xrlog.getScriptLogger([logger_name [default: root]])
```

Example:

This is the example of generating syslog and script logs.

```
syslog = xrlog.getSysLogger('myscript')
log = xrlog.getScriptLogger('myscript')
```

getSysLogger Script APIs

The `getSysLogger` API returns a syslogger object with APIs to print to XR syslog.

Parameter

name: A string representing the module name of the syslogger. This parameter is optional, and the default value is "root".

Result

The result of the `getSysLogger` API is a syslogger object.

Result

The result of the `alert` API is the message being logged to the XR syslog with severity level 1.

Syslogger APIs

The following are the list of Syslogger APIs:

- emergency
- alert
- critical
- error
- warning
- notification
- info
- debug
- log
- setlevel

emergency

The `emergency` API prints a message string to the XR syslog with severity level 0, indicating that the system is unusable.

Parameters

- **self**: The syslogger object.
- **msg_string**: A string representing the syslog message to be printed.

Result

The result of the `emergency` API is the message being logged to the XR syslog with severity level 0.

Example

The following example shows how to use the `emergency` API to log a message indicating a system emergency and the system logging message that you can see on the router.

```
>>> syslog.emergency("script generated syslog message")
RP/0/RP0/CPU0: scripting_python3[67965]: %OS-SCRIPT_LOG-0-EMERGENCY : Script-myscript:
script generated syslog message
```

alert

The `alert` API prints a message string to the XR syslog with severity level 1, indicating that immediate action is needed.

After getting the object, the following are the list of APIs that belong to that object.

Parameters

- **self**: The syslogger object.
- **msg_string**: A string representing the syslog message to be printed.

Example

The following example shows how to use the `alert` API to log a message indicating that immediate action is needed and the system logging message that you can see on the router:

```
>>> syslog.alert("script generated syslog message")
RP/0/RP0/CPU0: scripting_python3[67965]: %OS-SCRIPT_LOG-1-ALERT : Script-myscript: script
generated syslog message
```

critical

The `critical` API prints a message string to the XR syslog with severity level 2, indicating critical conditions.

Parameters

- **self**: The syslogger object.
- **msg_string**: A string representing the syslog message to be printed.

Result

The result of the `critical` API is the message being logged to the XR syslog with severity level 2.

Example

The following example shows how to use the `critical` API to log a message indicating critical conditions and the system logging message that you can see on the router:

```
>>> syslog.critical("script generated syslog message")
RP/0/RP0/CPU0: scripting_python3[67965]: %OS-SCRIPT_LOG-2-CRITICAL : Script
```

error

The `error` API prints a message string to the XR syslog with severity level 3, indicating error conditions.

Parameters

- **self**: The syslogger object.
- **msg_string**: A string representing the syslog message to be printed.

Result

The result of the `error` API is the message being logged to the XR syslog with severity level 3.

Example

The following example shows how to use the `error` API to log a message indicating error conditions and the system logging message that you can see on the router:

```
>>> syslog.error("script generated syslog message")
RP/0/RP0/CPU0: scripting_python3[67965]: %OS-SCRIPT_LOG-3-ERROR : Script-myscript: script
generated syslog message
```

warning

The `warning` API prints a message string to the XR syslog with severity level 4, indicating a warning condition.

Parameters

- **self**: The syslogger object.
- **msg_string**: A string representing the syslog message to be printed.

Result

The result of the `warning` API is the message being logged to the XR syslog with severity level 4.

Example

The following example shows how to use the `warning` API to log a message indicating a warning condition and the system logging message that you can see on the router:

```
>>> syslog.warning("script generated syslog message")
RP/0/RP0/CPU0: scripting_python3[67965]: %OS-SCRIPT_LOG-4-WARNING : Script-myscript: script
 generated syslog message
```

notification

The `notification` API prints a message string to the XR syslog with severity level 5, indicating a normal but significant condition.

Parameters

- **self**: The syslogger object.
- **msg_string**: A string representing the syslog message to be printed.

Result

The result of the `notification` API is the message being logged to the XR syslog with severity level 5.

Example

The following example shows how to use the `notification` API to log a message indicating a normal but significant condition and the system logging message that you can see on the router:

```
>>> syslog.notification("script generated syslog message")
RP/0/RP0/CPU0:scripting_python3[67965]: %OS-SCRIPT_LOG-5-NOTIFICATION : Script-myscript:
script generated syslog message
```

info

The `info` API prints a message string to the XR syslog with severity level 6, indicating an informational message only.

Parameters

- **self**: The syslogger object.
- **msg_string**: A string representing the syslog message to be printed.

Result

The result of the `info` API is the message being logged to the XR syslog with severity level 6.

Example

The following example shows how to use the `info` API to log an informational message and the system logging message that you can see on the router:

```
>>> syslog.info("script generated syslog message")
RP/0/RP0/CPU0:scripting_python3[67965]: %OS-SCRIPT_LOG-6-INFO : Script-myscript: script
generated syslog message
```

debug

The `debug` API prints a message string to the XR syslog with severity level 7, indicating a debugging message only.

Parameters

- **self**: The syslogger object.
- **msg_string**: A string representing the syslog message to be printed.

Result

The result of the `debug` API is the message being logged to the XR syslog with severity level 7.

Example

The following example shows how to use the `debug` API to log a debugging message and the system logging message that you can see on the router:

```
>>> syslog.debug("script generated syslog message")
RP/0/RP0/CPU0:scripting_python3[67965]: %OS-SCRIPT_LOG-7-DEBUG : Script-myscript: script
generated syslog message
```

log

The `log` API prints a message string to the XR syslog at the provided severity level.

Parameters

- **self**: The syslogger object.
- **level**: An integer representing the syslog logging level.
- **msg_string**: A string representing the syslog message to be printed.

Result

The result of the `log` API is the message being logged to the XR syslog at the specified severity level.

Example

The following example shows how to use the `log` API to log a message at a specified severity level and the system logging message that is generated on the router:

```
syslog.log(xrlog.WARNING, "script generated syslog message")
RP/0/RP0/CPU0: scripting_python3[67965]: %OS-SCRIPT_LOG-4-WARNING : Script-myscript: script
 generated syslog message
>>> syslog.log(30, "script generated syslog message")
RP/0/RP0/CPU0:scripting_python3[67965]: %OS-SCRIPT_LOG-4-WARNING : Script-myscript: script
 generated syslog message
```

setlevel

The `setLevel` API sets the level of messages that should be written to syslogs. Messages with a lower level than the specified level will be discarded.

Parameters

level: An integer representing the syslog logging level.

Result

The result of the `setLevel` API is that only messages with a severity level equal to or higher than the specified level will be logged to the XR syslog.

Example

The following example shows how to use the `setLevel` API to set the logging level:

```
syslog = xrlog.getSysLogger('myscript')
syslog.setLevel(4)
```

Script Logger API

getScriptLogger

The `getScriptLogger` API returns a Python Logger object.

Parameters

name: A string representing the module name of the logger. This parameter is optional.

Result

The result of the `getScriptLogger` API is a Python Logger object.

For more information on Python Logger, refer to the Python logger documentation.

Example

The following example shows how to initialize a script logger using the `getScriptLogger` API:

```
log = xrlog.getScriptLogger('myscript')
```