



Python API

- [Information About the Python API, on page 1](#)
- [Using Python, on page 1](#)

Information About the Python API

The Cisco Nexus 3500 platform switches support Python v2.7.11 in both interactive and noninteractive (script) modes and are available in the Guest Shell.

Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

The Python interpreter and the extensive standard library are freely available in source or binary form for all major platforms from the Python website:

<http://www.python.org/>

The same site also contains distributions of and pointers to many free third-party Python modules, programs and tools, and more documentation.

The Python scripting capability gives programmatic access to the device's command-line interface (CLI) to perform various tasks and Power On Auto Provisioning (POAP) or Embedded Event Manager (EEM) actions. Python can be accessed from the Bash shell.

The Python interpreter is available in the Cisco NX-OS software.

Using Python

This section describes how to write and execute Python scripts.

Cisco Python Package

Cisco NX-OS provides a Cisco Python package that enables access to many core network-device modules, such as interfaces, VLANs, VRFs, ACLs, and routes. You can display the details of the Cisco Python package by entering the **help()** command. To obtain additional information about the classes and methods in a module,

you can run the help command for a specific module. For example, `help(cisco.interface)` displays the properties of the `cisco.interface` module.

The following is an example of how to display information about the Cisco Python package:

```
>>> import cisco
>>> help(cisco)
Help on package cisco:

NAME
    cisco

FILE
    /isan/python/scripts/cisco/__init__.py

PACKAGE CONTENTS
    acl
    bgp
    cisco_secret
    cisco_socket
    feature
    interface
    key
    line_parser
    md5sum
    nxcli
    ospf
    routemap
    routes
    section_parser
    ssh
    system
    tacacs
    vrf

CLASSES
    __builtin__.object
    cisco.cisco_secret.CiscoSecret
    cisco.interface.Interface
    cisco.key.Key
```

Using the CLI Command APIs

The Python programming language uses three APIs that can execute CLI commands. The APIs are available from the Python CLI module.

These APIs are listed in the following table. You must enable the APIs with the `from cli import *` command. The arguments for these APIs are strings of CLI commands. To execute a CLI command through the Python interpreter, you enter the CLI command as an argument string of one of the following APIs:

Table 1: CLI Command APIs

API	Description
cli() Example: <pre>string = cli ("cli-command")</pre>	Returns the raw output of CLI commands, including control or special characters. Note The interactive Python interpreter prints control or special characters 'escaped'. Carriage return is printed as '\n' and gives results that can be difficult to read. The cli() API gives results that are more readable.
clid() Example: <pre>json_string = clid ("cli-command")</pre>	Returns JSON output for cli-command , if XML support exists for the command, otherwise an exception is thrown. Note This API can be useful when searching the output of show commands.
clip() Example: <pre>clip ("cli-command")</pre>	Prints the output of the CLI command directly to stdout and returns nothing to Python. Note <code>clip ("cli-command")</code> is equivalent to <pre>r=cli ("cli-command") print r</pre>

When two or more commands are run individually, the state is not persistent from one command to subsequent commands.

In the following example, the second command fails because the state from the first command does not persist for the second command:

```
>>> cli("conf t")
>>> cli("interface eth4/1")
```

When two or more commands are run together, the state is persistent from one command to subsequent commands.

In the following example, the second command is successful because the state persists for the second and third commands:

```
>>> cli("conf t ; interface eth4/1 ; shut")
```



Note Commands are separated with ";" as shown in the example. The semicolon (;) must be surrounded with single blank characters.

Invoking the Python Interpreter from the CLI

The following example shows how to invoke Python 2 from the CLI:



Note The Python interpreter is designated with the ">>>" or "..." prompt.

```
switch# python
switch# python
```

Warning: Python 2.7 is End of Support, and future NXOS software will deprecate python 2.7 support. It is recommended for new scripts to use 'python3' instead. Type "python3" to use the new shell.

```
Python 2.7.11 (default, Jun  4 2020, 09:48:24)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
>>> from cli import *
>>> import json
>>> cli('configure terminal ; interface loopback 1 ; no shut')
''
>>> intflist=json.loads(clid('show interface brief'))
>>> i=0
>>> while i < len(intflist['TABLE_interface']['ROW_interface']):
...     intf=intflist['TABLE_interface']['ROW_interface'][i]
...     i=i+1
...     if intf['state'] == 'up':
...         print intf['interface']
...
mgmt0
loopback1
>>>
```

Display Formats

The following examples show various display formats using the Python APIs:

Example 1:

```
>>> from cli import *
>>> cli("conf ; interface loopback 1")
''
>>> clip('where detail')
mode:
username:          admin
vdc:               switch
routing-context vrf: default
```

Example 2:

```
>>> from cli import *
>>> cli("conf ; interface loopback 1")
''
>>> cli('where detail')
' mode:          \n username:          admin\n vdc:
switch\n routing-context vrf: default\n'
```

```
>>>
```

Example 3:

```
>>> from cli import *
>>> cli("conf ; interface loopback 1")
''
>>> r = cli('where detail') ; print r
mode:
username:          admin
vdc:               EOR-1
routing-context vrf: default
>>>
```

Example 4:

```
>>> from cli import *
>>> import json
>>> out=json.loads(clid('show version'))
>>> for k in out.keys():
...     print "%30s = %s" % (k, out[k])
...
kern_uptm_secs = 21
kick_file_name = bootflash:///nxos.9.2.1.bin.S246
rr_service = None
module_id = Supervisor Module
kick_tmstamp = 07/11/2018 00:01:44
bios_cmpl_time = 05/17/2018
bootflash_size = 20971520
kickstart_ver_str = 9.2(1)
kick_cmpl_time = 7/9/2018 9:00:00
chassis_id = Nexus9000 C9504 (4 Slot) Chassis
proc_board_id = SAL171211LX
memory = 16077872
manufacturer = Cisco Systems, Inc.
kern_uptm_mins = 26
bios_ver_str = 05.31
cpu_name = Intel(R) Xeon(R) CPU D-1528 @ 1.90GHz
kern_uptm_hrs = 2
rr_usecs = 816550
rr_sys_ver = 9.2(1)
rr_reason = Reset Requested by CLI command reload
rr_ctime = Wed Jul 11 20:44:39 2018
header_str = Cisco Nexus Operating System (NX-OS) Software
TAC support: http://www.cisco.com/tac
Copyright (C) 2002-2018, Cisco and/or its affiliates.
All rights reserved.
The copyrights to certain works contained in this software are
owned by other third parties and used and distributed under their own
licenses, such as open source. This software is provided "as is," and unless
otherwise stated, there is no warranty, express or implied, including but not
limited to warranties of merchantability and fitness for a particular purpose.
Certain components of this software are licensed under
the GNU General Public License (GPL) version 2.0 or
GNU General Public License (GPL) version 3.0 or the GNU
Lesser General Public License (LGPL) Version 2.1 or
Lesser General Public License (LGPL) Version 2.0.
A copy of each such license is available at
http://www.opensource.org/licenses/gpl-2.0.php and
http://opensource.org/licenses/gpl-3.0.html and
http://www.opensource.org/licenses/lgpl-2.1.php and
http://www.gnu.org/licenses/old-licenses/library.txt.
host_name = switch
```

```

        mem_type = kB
        kern_uptm_days = 0
    >>>

```

Non-Interactive Python

A Python script can run in non-interactive mode by providing the Python script name as an argument to the Python CLI command. Python scripts must be placed under the bootflash or volatile scheme. A maximum of 32 command-line arguments for the Python script are allowed with the Python CLI command.

The Cisco Nexus 3500 platform switches also support the source CLI command for running Python scripts. The `bootflash:scripts` directory is the default script directory for the source CLI command.

This example shows the script first and then executing it. Saving is like bringing any file to the bootflash.

```

switch# show file bootflash:deltaCounters.py
#!/isan/bin/python

from cli import *
import sys, time

ifName = sys.argv[1]
delay = float(sys.argv[2])
count = int(sys.argv[3])
cmd = 'show interface ' + ifName + ' counters'

out = json.loads(clid(cmd))
rxuc = int(out['TABLE_rx_counters']['ROW_rx_counters'][0]['eth_inucast'])
rxmc = int(out['TABLE_rx_counters']['ROW_rx_counters'][1]['eth_inmcast'])
rxbc = int(out['TABLE_rx_counters']['ROW_rx_counters'][1]['eth_inbcast'])
txuc = int(out['TABLE_tx_counters']['ROW_tx_counters'][0]['eth_outucast'])
txmc = int(out['TABLE_tx_counters']['ROW_tx_counters'][1]['eth_outmcast'])
txbc = int(out['TABLE_tx_counters']['ROW_tx_counters'][1]['eth_outbcast'])
print 'row rx_ucast rx_mcast rx_bcast tx_ucast tx_mcast tx_bcast'
print '=====
print '      %8d %8d %8d %8d %8d %8d' % (rxuc, rxmc, rxbc, txuc, txmc, txbc)
print '=====

i = 0
while (i < count):
    time.sleep(delay)
    out = json.loads(clid(cmd))
    rxucNew = int(out['TABLE_rx_counters']['ROW_rx_counters'][0]['eth_inucast'])
    rxmcNew = int(out['TABLE_rx_counters']['ROW_rx_counters'][1]['eth_inmcast'])
    rxbcNew = int(out['TABLE_rx_counters']['ROW_rx_counters'][1]['eth_inbcast'])
    txucNew = int(out['TABLE_tx_counters']['ROW_tx_counters'][0]['eth_outucast'])
    txmcNew = int(out['TABLE_tx_counters']['ROW_tx_counters'][1]['eth_outmcast'])
    txbcNew = int(out['TABLE_tx_counters']['ROW_tx_counters'][1]['eth_outbcast'])
    i += 1
    print '%-3d %8d %8d %8d %8d %8d' % \
        (i, rxucNew - rxuc, rxmcNew - rxmc, rxbcNew - rxbc, txucNew - txuc, txmcNew - txmc,
        txbcNew - txbc)

switch# python bootflash:deltaCounters.py Ethernet1/1 1 5
row rx_ucast rx_mcast rx_bcast tx_ucast tx_mcast tx_bcast
=====
              0       791         1         0    212739         0
=====
1              0         0         0         0         26         0
2              0         0         0         0         27         0

```

```

3          0          1          0          0          54          0
4          0          1          0          0          55          0
5          0          1          0          0          81          0
switch#

```

The following example shows how a **source** command specifies command-line arguments. In the example, *policy-map* is an argument to the `cgrep python` script. The example also shows that a **source** command can follow the pipe operator (`|`).

```

switch# show running-config | source sys/cgrep policy-map

policy-map type network-qos nw-pfc
policy-map type network-qos no-drop-2
policy-map type network-qos wred-policy
policy-map type network-qos pause-policy
policy-map type qos foo
policy-map type qos classify
policy-map type qos cos-based
policy-map type qos no-drop-2
policy-map type qos pfc-tor-port

```

Running Scripts with Embedded Event Manager

On Cisco Nexus 3500 platform switches, Embedded Event Manager (EEM) policies support Python scripts.

The following example shows how to run a Python script as an EEM action:

- An EEM applet can include a Python script with an action command.

```

switch# show running-config eem

!Command: show running-config eem
!Running configuration last done at: Thu Jun 25 15:29:38 2020
!Time: Thu Jun 25 15:33:19 2020

version 9.3(5) Bios:version 07.67
event manager applet a1
  event cli match "show clock"
  action 1 cli python bootflash:pydate.py

switch# show file logflash:vdc_1/event_archive_1 | last 33

eem_event_time:06/25/2020,15:34:24 event_type:cli event_id:24 slot:active(1) vdc
:1 severity:minor applets:a1
eem_param_info:command = "exshow clock"
Starting with policy a1
stty: standard input: Inappropriate ioctl for device
Executing the following commands succeeded:
    python bootflash:pydate.py
Completed executing policy a1
Event Id:24 event type:10241 handling completed

```

- You can search for the action that is triggered by the event in the log file by running the **show file logflash:event_archive_1** command.

```

switch# show file logflash:event_archive_1 | last 33

eem_event_time:05/01/2011,19:40:28 event_type:cli event_id:8 slot:active(1)

```

```

vdc:1 severity:minor applets:a1
eem_param_info:command = "exshow clock"
Starting with policy a1
Python

2011-05-01 19:40:28.644891
Executing the following commands succeeded:
python bootflash:pydate.py

PC_VSH_CMD_TLV(7679) with q

```

Python Integration with Cisco NX-OS Network Interfaces

On Cisco Nexus 3500 platform switches, Python is integrated with the underlying Cisco NX-OS network interfaces. You can switch from one virtual routing context to another by setting up a context through the `cisco.vrf.set_global_vrf()` API.

The following example shows how to retrieve an HTML document over the management interface of a device. You can also establish a connection to an external entity over the in-band interface by switching to a desired virtual routing context.

```

switch# python
Python 2.7.5 (default, Oct 8 2013, 23:59:43)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import urllib2
>>> from cisco.vrf import *
>>> set_global_vrf('management')
>>> page=urllib2.urlopen('http://172.23.40.211:8000/welcome.html')
>>> print page.read()
Hello Cisco Nexus 9000

>>>
>>> import cisco
>>> help(cisco.vrf.set_global_vrf)
Help on function set_global_vrf in module cisco.vrf:

set_global_vrf(vrf)
    Sets the global vrf. Any new sockets that are created (using socket.socket)
    will automatically get set to this vrf (including sockets used by other
    python libraries).

    Arguments:
        vrf: VRF name (string) or the VRF ID (int).

    Returns: Nothing

>>>

```

Cisco NX-OS Security with Python

Cisco NX-OS resources are protected by the Cisco NX-OS Sandbox layer of software and by the CLI role-based access control (RBAC).

All users who are associated with a Cisco NX-OS network-admin or dev-ops role are privileged users. Users who are granted access to Python with a custom role are regarded as nonprivileged users. Nonprivileged users have limited access to Cisco NX-OS resources, such as the file system, guest shell, and Bash commands. Privileged users have greater access to all the resources of Cisco NX-OS.

Examples of Security and User Authority

-

Example of Running Script with Scheduler

-

