Entwicklung, Debuggen und Bereitstellen der NX-SDK-Python-Anwendung in Nexus 3000-/9000-Switches

Inhalt

Einführung

Dieses Dokument bietet einen Workflow für die Anwendungsentwicklung mit Python mit dem Cisco NX-Software Development Kit (SDK) unter Verwendung von Cisco NX-OS auf den Plattformen Nexus 3000 und Nexus 9000.

Voraussetzungen

Anforderungen

Für dieses Dokument bestehen keine speziellen Anforderungen.

Verwendete Komponenten

Die Informationen in diesem Dokument basieren auf den folgenden Software- und Hardwareversionen:

- In diesem Dokument werden NX-SDK v1.0.0 und NX-SDK v1.5.0 verwendet.
- NX-SDK v1.0.0 kann auf der Nexus 9000-Plattform ab NX-OS 7.0(3)I6(1) und Nexus 3000-Plattform ab NX-OS 7.0(3)I7(1) verwendet werden.
- NX-SDK v1.5.0 kann ab NX-OS Version 7.0(3)I7(3) sowohl auf der Nexus 9000-Plattform als auch auf der Nexus 3000-Plattform verwendet werden.

Die Informationen in diesem Dokument wurden von den Geräten in einer bestimmten Laborumgebung erstellt. Alle in diesem Dokument verwendeten Geräte haben mit einer leeren (Standard-)Konfiguration begonnen. Wenn Ihr Netzwerk in Betrieb ist, stellen Sie sicher, dass Sie die potenziellen Auswirkungen eines Befehls verstehen.

Hintergrundinformationen

Das Cisco NX-SDK ermöglicht die Entwicklung benutzerdefinierter Anwendungen, die nativ in Cisco NX-OS auf Nexus 9000- und Nexus 3000-Plattformen ausgeführt werden können. NX-SDK bietet Kunden die Möglichkeit, eigene CLI-Befehle und -Ausgaben zu erstellen, angepasste Syslogs als Reaktion auf bestimmte Ereignisse zu generieren, maßgeschneiderte Telemetriedaten zu streamen und vieles mehr.

NX-SDK verfügt über eine C++-API, die mithilfe von Simplified Wrapper und Interface Generator (SWIG) in andere Sprachen übersetzt wird. So kann der Kunde NX-SDK in jeder beliebigen Sprache einsetzen. Dieses Dokument demonstriert die Implementierung gängiger NX-SDK-Funktionen in Python und bietet Kunden einen Workflow für die Entwicklung eigener NX-SDK-Python-Anwendungen.

Entwicklung einer Python-Anwendung mit NX-SDK

NX-SDK aktivieren

Damit eine NX-SDK-Anwendung ausgeführt werden kann, muss die NX-SDK-Funktion zuerst auf dem Gerät aktiviert werden:

```
switch(config)# feature nxsdk
```

Erstellen einer Python-Datei

Eine Python-Datei kann mit der NX-OS Bash-Shell erstellt und bearbeitet werden. Damit die Bash-Shell verwendet werden kann, muss sie zunächst auf dem Gerät aktiviert werden:

```
switch(config)# feature bash-shell
```

Geben Sie die Bash-Shell ein, und verwenden Sie den vi-Text-Editor, um die Python-Datei zu erstellen und zu bearbeiten:

```
switch(config)# run bash
```

```
bash-4.2$ vi /isan/bin/nxsdk-app.py
```

Hinweis: Best Practice ist es, Python-Dateien im **/isan/bin/**directory zu erstellen. Python-Dateien benötigen Ausführungsberechtigungen, um ausgeführt zu werden. Platzieren Sie Python-Dateien nicht im **/bootflash-**Verzeichnis oder einem der Unterverzeichnisse.

Hinweis: Es ist nicht erforderlich, Python-Dateien über NX-OS zu erstellen und zu bearbeiten. Der Entwickler kann die Anwendung mithilfe der lokalen Umgebung erstellen und die ausgefüllten Dateien mithilfe eines Dateiübertragungsprotokolls seiner Wahl auf das Gerät übertragen. Es kann jedoch für Entwickler effizienter sein, ihr Skript mithilfe von NX-OS-Dienstprogrammen zu debuggen und Fehler zu beheben.

Implementieren von NX-SDK-Komponenten

Es wird empfohlen, dass Entwickler beginnen, ihre NX-SDK-Python-Anwendung aus der benutzerdefinierten CliPyApp-Vorlage auf dem Cisco DevNet NX-SDK GitHub zu erstellen. In diesen Abschnitten dieses Dokuments werden die erforderlichen Komponenten mit ihren Namen in dieser Vorlage beschrieben.

Für eine NX-SDK-Python-Anwendung sind vier Hauptkomponenten erforderlich:

1. NX-SDK muss über die **import nx_sdk_py-**Anweisung in die Anwendung importiert werden.
2. Eine Funktion (in der Regel **sdkThread**), die die NX-SDK-Anwendung startet und verschiedene Optionen für die Anwendung ändert.
3. Die Erstellung benutzerdefinierter CLI-Befehle sowie die Definition der benutzerdefinierten CLI-Befehlssyntax in der **sdkThread-**Funktion.
4. Eine Klasse mit dem Namen **pyCmdHandler** mit der Methode **postCliCb**, die benutzerdefinierte CLI-Befehle verarbeitet, die von der NX-SDK-Anwendung hinzugefügt werden.

sdkThread-Funktion

Die sdkThread-Funktion initialisiert, fügt Funktionen zur NX-SDK-Anwendung hinzu und startet diese. Für die Funktion müssen keine Parameter übergeben werden. Für alle Python NX-SDK-Anwendungen müssen drei Methoden aus der nx_sdk_py-Bibliothek aufgerufen werden:

1. **nx_sdk_py.NxSdk.getSdkInst(len(sys.argv), sys.argv)** gibt entweder ein SDK-Instanzobjekt zurück oder gibt None zurück. Wenn ein SDK-Instanzobjekt zurückgegeben wird, wurde die NX-SDK-Anwendung erfolgreich in der NX-OS-Infrastruktur registriert. Wenn None (Keine) zurückgegeben wird, treten bei diesem Registrierungsvorgang Fehler auf, und im Syslog des Geräts wird ein Eintrag in das Fehlerprotokoll angezeigt. Diese Methode ist hier dokumentiert.

   Hinweis: Ab NX-SDK v1.5.0 kann ein dritter boolescher Parameter an die **NxSdk.getSdkInst-**

Methode übergeben werden, die erweiterte Ausnahmen aktiviert, wenn True und Advanced Exceptions bei False deaktiviert werden. Diese Methode ist hier dokumentiert.

1. **sdk.startEventLoop()**-Methode, wobei **sdk** das SDK-Instanzobjekt ist, das von der **nx_sdk_py.NxSdk.getSdkInst()**-Methode zurückgegeben wird. Diese Methode startet die NX-SDK-Anwendung und ermöglicht die Interaktion mit der NX-OS-Infrastruktur. Diese Methode ist hier dokumentiert.
2. **nx_sdk_py.NxSdk.__swig_zerstören_(sdk)**-Methode, wobei **sdk** das SDK-Instanzobjekt ist, das von der zuvor erläuterten Methode **nx_sdk_py.NxSdk.getSdkInst()** zurückgegeben wurde. Diese Methode, die am Ende der **sdkThread-**Funktion platziert wird, ermöglicht das fehlerfreie Beenden der NX-SDK-Anwendung.

Einige häufig verwendete Methoden sind:

- **sdk.getTracer()**, wobei **sdk** das SDK-Instanzobjekt ist, das von der **nx_sdk_py.NxSdk.getSdkInst()**-Methode zurückgegeben wird. Diese Methode gibt ein NxTrace-Objekt zurück, das zum Generieren benutzerdefinierter Syslogs sowie zum Protokollieren von Ereignissen und Fehlern im Ereignisverlauf der Anwendung verwendet werden kann. Benutzerdefinierte Syslog-Meldungen werden im Systemprotokoll des Geräts angezeigt (sichtbar über den Befehl **show logging logfile**), während die Ereignisse, die im Ereignisverlauf der Anwendung protokolliert werden, über die Option **show <application-name> nxsdk event-history events** oder **show <application-name> nxsdk event-history-**befehle angezeigt werden. Diese Methode ist hier dokumentiert. Das von dieser Methode zurückgegebene NxTrace-Objekt und die zugehörigen Methoden werden hier dokumentiert. Beispielsweise können Sie die Ereignishistorie einer Anwendung mit dem Namen **Transceiver_DOM.py** über die **Anzeige Transceiver_DOM.py nxsdk-Ereignisverlaufsereignisse** anzeigen und **Transceiver_DOM.py nxsdk-**Befehle für **Ereignisverlaufsfehler** anzeigen.
- **sdk.getCliParser()**, wobei **sdk** das SDK-Instanzobjekt ist, das von der **nx_sdk_py.NxSdk.getSdkInst()**-Methode zurückgegeben wird. Diese Methode gibt ein NxCliParser-Objekt zurück, mit dem die bereits über Python vorhandenen CLI-Befehle ausgeführt sowie benutzerdefinierte CLI-Befehle erstellt werden können. Diese Methode ist hier dokumentiert. Das von dieser Methode zurückgegebene NxCliParser-Objekt und die zugehörigen Methoden werden hier dokumentiert.
- **cliP.execShowCmd("cmd", return_type)**, wobei **cliP** das NxCliParser-Objekt ist, das von der **sdk.getCliParser()**-Methode zurückgegeben wird, **cmd** ist der Befehl show, den Sie in Anführungszeichen und **return_type ausführen möchten ist das Datenformat, das ausgegeben werden soll.** Das Datenformat kann entweder **R_TEXT**, **R_JSON** oder **R_XML sein**. Diese Methode ist hier dokumentiert.

  Hinweis: Die Datenformate **R_JSON** und **R_XML** funktionieren nur, wenn der Befehl die Ausgabe in diesen Formaten unterstützt. In NX-OS können Sie überprüfen, ob ein Befehl die Ausgabe in einem bestimmten Datenformat unterstützt, indem Sie die Ausgabe in das angeforderte Datenformat leiten. Wenn der Befehl piped aussagekräftige Ausgaben zurückgibt, wird dieses Datenformat unterstützt. Wenn Sie z. B. **show mac address-table** ausführen **| gibt** in NX-OS die JSON-Ausgabe zurück, und das R_JSON-Datenformat wird auch in NX-SDK unterstützt.

- **cliP.execConfigCmd(cmd_filename)**, wobei **cliP** das NxCliParser-Objekt ist, das von der **sdk.getCliParser()**-Methode zurückgegeben wird, und **cmd_filename** ist der absolute

Dateipfad zu einer Datei, die durch Zeilen getrennte Befehle enthält. Diese Methode gibt eine Zeichenfolge zurück, die den Erfolg der Befehlsausführung anzeigt. Wenn "SUCCESS" in der Zeichenfolge enthalten ist, werden alle Befehle erfolgreich ausgeführt. Andernfalls enthält die Zeichenfolge die Ausnahme, die beschreibt, warum die Befehle nicht ausgeführt wurden. Diese Methode ist hier dokumentiert.

Folgende optionale Methoden können hilfreich sein:

- **sdk.setAppDesc('description string')**, wobei **sdk** das SDK-Instanzobjekt ist, das von der **nx_sdk_py.NxSdk.getSdkInst()**-Methode zurückgegeben wird. Diese Methode legt die Beschreibung der NX-SDK-Anwendung fest. Die Beschreibung wird im kontextsensitiven Hilfemenü von NX-OS angezeigt, auf das über ein Fragezeichen in der CLI zugegriffen werden kann. Diese Methode ist hier dokumentiert. Beispielsweise wird eine Anwendung mit dem Namen **Transceiver_DOM.py**, eine Anwendungsbeschreibung für die Rückgabe aller Schnittstellen mit DOM-fähigen Transceivern, die in der kontextsensitiven Hilfe von NX-OS eingefügt wurden, wie folgt angezeigt:

```
 N9K-C93180LC-EX# show Tra?
track Tracking information
Transceiver_DOM.py Returns all interfaces with DOM-capable transceivers inserted
```

- **sdk.getAppName()**, wobei **sdk** das SDK-Instanzobjekt ist, das von der **nx_sdk_py.NxSdk.getSdkinst()**-Methode zurückgegeben wird. Diese Methode gibt den Namen der Python-Anwendung zurück. Diese Methode ist hier dokumentiert.

Erstellen benutzerdefinierter CLI-Befehle

In einer Python-Anwendung mit NX-SDK werden benutzerdefinierte CLI-Befehle in der sdkThread-Funktion erstellt und definiert. Es gibt zwei Arten von Befehlen: **Anzeigen von** Befehlen und **Konfigurationsbefehlen**.

1. **Anzeigen** von Befehlen zur Anzeige von Informationen über das Gerät, seine Konfiguration oder Umgebung
2. **Konfigurationsbefehle** ändern die Konfiguration des Geräts, wodurch die Reaktion des Geräts auf das umliegende Netzwerk geändert wird.

Diese beiden Methoden ermöglichen die Erstellung von show-Befehlen bzw. Konfigurationsbefehlen:

- **cliP.newShowCmd("cmd_name", "syntax")**, wobei **cliP** das NxCliParser-Objekt ist, das von der **sdk.getCliParser()**-Methode zurückgegeben wird, **cmd_name** ist ein eindeutiger Name für den in der benutzerdefinierten NX-SDK-Anwendung integrierten Befehl und Syntax beschreibt, welche Schlüsselwörter und Parameter im Befehl verwendet werden können. Diese Methode gibt ein NxCliCmd-Objekt zurück, das hier dokumentiert ist. Diese Methode ist hier dokumentiert.

  Hinweis: Dieser Befehl ist eine Unterklasse von **cliP.newCliCmd("cmd_type", "cmd_name", "syntax")**, wobei **cmd_type** entweder **CONF_CMD** oder **SHOW_CMD** ist (je nach konfiguriertem Befehlstyp), **cmd_name ist ein eindeutiger Name für den internen Befehl Die benutzerdefinierte NX-SDK-Anwendung und die Syntax beschreiben, welche Schlüsselwörter und Parameter im Befehl verwendet werden können.** Daher kann die API-Dokumentation für diesen Befehl hilfreicher sein.

- **cliP.newConfigCmd("cmd_name", "syntax")**, wobei **cliP** das NxCliParser-Objekt ist, das von der **sdk.getCliParser()**-Methode zurückgegeben wird, **cmd_name** ist ein eindeutiger Name für den in der benutzerdefinierten NX-SDK-Anwendung integrierten Befehl beschreibt, welche Schlüsselwörter und Parameter im Befehl verwendet werden können. Diese Methode gibt ein NxCliCmd-Objekt zurück, das hier dokumentiert ist. Diese Methode ist hier dokumentiert.

  Hinweis: Dieser Befehl ist eine Unterklasse von **cliP.newCliCmd("cmd_type", "cmd_name", "syntax")**, wobei **cmd_type** entweder **CONF_CMD** oder **SHOW_CMD** ist (er hängt vom konfigurierten Befehlstyp ab), **cmd_name ist ein eindeutiger Name für den -Befehl für die benutzerdefinierte NX-SDK-Anwendung integriert ist, und die Syntax beschreibt, welche Schlüsselwörter und Parameter im Befehl verwendet werden können.** Daher kann die API-Dokumentation für diesen Befehl hilfreicher sein.

Beide Befehlstypen haben zwei verschiedene Komponenten: Parameter und Schlüsselwörter:

1. **Parameter** sind Werte, mit denen die Ergebnisse des Befehls geändert werden. Im Befehl **show ip route 192.168.1.0** gibt es beispielsweise ein **route**-Schlüsselwort gefolgt von einem Parameter, der eine IP-Adresse akzeptiert, das angibt, dass nur Routen angezeigt werden sollen, die die angegebene IP-Adresse enthalten.

2. **Schlüsselwörter** verändern die Ergebnisse des Befehls allein durch ihre Präsenz. Im Befehl **show mac address-table dynamic** gibt es beispielsweise ein **dynamisches** Schlüsselwort, das angibt, dass nur dynamisch erlernte MAC-Adressen angezeigt werden sollen.

Beide Komponenten werden bei der Erstellung in der Syntax eines NX-SDK-Befehls definiert. Es gibt Methoden für das NxCliCmd-Objekt, um die spezifische Implementierung beider Komponenten zu ändern.

- **nx_cmd.updateParam("<parameter>", "help_str", type)**, wobei **nx_cmd** das NxCliCmd-Objekt ist, das von der **cliP.newShowCmd()** oder der **cliP.newConfigCmd()**-Methode zurückgegeben wird**<parameter** >ist Der Name des Befehlsparameters, der durch eckige Klammern (<>) geändert werden kann, help_str legt die Hilfszeichenfolge des benutzerdefinierten Befehls fest und wird im kontextsensitiven Hilfemenü von NX-OS angezeigt, auf das über ein Fragezeichen in der CLI zugegriffen werden kann. Der Typ des Parameters ist der Typ. Zusätzliche optionale Parameter für diese Methode sind hier verfügbar und dokumentiert. Dies sind gültige Typen für Parameter, die im Typargument angegeben werden können:

  P_INTEGER - Gibt eine ganze Zahl an**P_STRING**: Gibt eine beliebige Zeichenfolge an**P_INTERFACE** - Legt eine beliebige Netzwerkschnittstelle fest**P_IP_ADDR** - Gibt eine beliebige IP-Adresse an**P_MAC_ADDR** - Gibt eine beliebige MAC-Adresse an**P_VRF**: Gibt eine beliebige Virtual Routing and Forwarding (VRF)-Instanz an.
- **nx_cmd.updateKeyword("Schlüsselwort", "help_str", is_key)**, wobei **nx_cmd** das NxCliCmd-Objekt ist, das von der **cliP.newShowCmd()** oder der **cliP.newConfigCmd()**-Methoden zurückgegeben wird, **ist Wenn Sie den Namen des zu ändernden Befehlsschlüsselworts eingeben, legt help_str die Hilfzeichenfolge des benutzerdefinierten Befehls fest und wird im kontextsensitiven Hilfemenü von NX-OS angezeigt, auf das über ein Fragezeichen in der CLI zugegriffen wird. Is_key ist ein optionaler boolescher Wert, der auf False festgelegt wird.** Wenn **is_key** True ist, wird durch die vom Befehl mit diesem Schlüsselwort erstellte eindeutige Konfiguration keine andere eindeutige Konfiguration überschrieben, die vom Befehl erstellt

wird. Wenn **is_key** False ist, überschreibt die vom Befehl mit diesem Schlüsselwort erstellte Konfiguration die andere durch den Befehl erstellte Konfiguration. Diese Methode ist hier dokumentiert.

Codebeispiele für häufig verwendete Befehlskomponenten können Sie im Abschnitt Custom CLI Command Examples dieses Dokuments anzeigen.

Nachdem benutzerdefinierte CLI-Befehle erstellt wurden, muss ein Objekt aus der später in diesem Dokument beschriebenen **pyCmdHandler-**Klasse erstellt und als CLI-Rückrufhandlerobjekt für das NxCliParser-Objekt festgelegt werden. Dies wird wie folgt gezeigt:

```
 cmd_handler = pyCmdHandler()
cliP.setCmdHandler(cmd_handler)
```

Anschließend muss das NxCliParser-Objekt der NX-OS CLI-Parserstruktur hinzugefügt werden, damit benutzerdefinierte CLI-Befehle für den Benutzer sichtbar sind. Dies erfolgt mit dem Befehl **cliP.addToParseTree()**, wobei **cliP** das NxCliParser-Objekt ist, das von der **sdk.getCliParser()**-Methode zurückgegeben wird.

sdkThread-Funktionsbeispiel

Hier ist ein Beispiel für eine typische sdkThread-Funktion mit der Verwendung der zuvor beschriebenen Funktionen. Diese Funktion (unter anderem in einer typischen benutzerdefinierten Python-Anwendung des NX-SDK) verwendet globale Variablen, die bei der Skriptausführung instanziiert werden.

```
cliP = ""
sdk = ""
event_hdlr = ""
tmsg = ""

def sdkThread():
    global cliP, sdk, event_hdlr, tmsg

    sdk = nx_sdk_py.NxSdk.getSdkInst(len(sys.argv), sys.argv)
    if not sdk:
       return

    sdk.setAppDesc("Returns all interfaces with DOM-capable transceivers inserted")

    tmsg = sdk.getTracer()
    tmsg.event("[{}] Started service".format(sdk.getAppName()))

    cliP = sdk.getCliParser()

    nxcmd = cliP.newShowCmd("show_port_bw_util_cmd", "port bw utilization [<port>]")
    nxcmd.updateKeyword("port", "Port Information")
    nxcmd.updateKeyword("bw", "Port Bandwidth Information")
    nxcmd.updateKeyword("utilization", "Port BW utilization in (%)")
    nxcmd.updateParam("<port>", "Optional Filter Port Ex) Ethernet1/1", nx_sdk_py.P_INTERFACE)

    nxcmd1 = cliP.newConfigCmd("port_bw_threshold_cmd", "port bw threshold <threshold>")
    nxcmd1.updateKeyword("threshold", "Port BW Threshold in (%)")

    int_attr = nx_sdk_py.cli_param_type_integer_attr()
    int_attr.min_val = 1;
    int_attr.max_val = 100;
```

```
    nxcmd1.updateParam("<threshold>", "Threshold Limit. Default 50%", nx_sdk_py.P_INTEGER,
int_attr, len(int_attr))

    mycmd = pyCmdHandler()
    cliP.setCmdHandler(mycmd)

    cliP.addToParseTree()

    sdk.startEventLoop()

    # If sdk.stopEventLoop() is called or application is removed from VSH...
    tmsg.event("Service Quitting...!")

    nx_sdk_py.NxSdk.__swig_destroy__(sdk)
```

pyCmdHandler-Klasse

Die **pyCmdHandler-**Klasse wird von der **NxCmdHandler-**Klasse in der Bibliothek nx_sdk_py geerbt. Die **PostCliCb(self, clicmd)**-Methode, die in der **pyCmdHandler-**Klasse definiert ist, wird immer aufgerufen, wenn CLI-Befehle aus einer NX-SDK-Anwendung stammen. Die **postCliCb(self, clicmd)**-Methode definiert daher, wie sich die in der sdkThread-Funktion definierten benutzerdefinierten CLI-Befehle auf dem Gerät verhalten.

Die **postCliCb(self, clicmd)**-Funktion gibt einen booleschen Wert zurück. Wenn **True** zurückgegeben wird, wird angenommen, dass der Befehl erfolgreich ausgeführt wurde. **False** sollte zurückgegeben werden, wenn der Befehl aus irgendeinem Grund nicht erfolgreich ausgeführt wurde.

Der **clicmd**-Parameter verwendet den eindeutigen Namen, der für den Befehl definiert wurde, als er in der sdkThread-Funktion erstellt wurde. Wenn Sie beispielsweise einen neuen **show**-Befehl mit dem eindeutigen Namen **show_xcvr_dom** erstellen, wird empfohlen, auf diesen Befehl mit demselben Namen in der **postCliCb(self, clicmd)**-Funktion zu verweisen, nachdem Sie überprüft haben, ob der Name des clicmd-Arguments **show_xcvr_dom enthält**. Es wird hier gezeigt:

```
def sdkThread():
    <snip>
    sh_xcvr_dom = cliP.newShowCmd("show_xcvr_dom", "dom")
    sh_xcvr_dom.updateKeyword("dom", "Show all interfaces with transceivers that are DOM-
capable")
    </snip>

class pyCmdHandler(nx_sdk_py.NxCmdHandler):
    def postCliCb(self, clicmd):
        if "show_xcvr_dom" in clicmd.getCmdName():
            get_dom_capable_interfaces()
```

Wenn ein Befehl erstellt wird, der Parameter verwendet, dann müssen Sie diese Parameter wahrscheinlich irgendwann in der **postCliCb(self, clicmd)** Funktion verwenden. Dies kann mit der **clicmd.getParamValue("<parameter>")**-Methode erfolgen, wobei **<parameter>** der Name des Befehlsparameters ist, der den Wert von eckigen Klammern (**<>**) abrufen soll. Diese Methode ist hier dokumentiert. Der von dieser Funktion zurückgegebene Wert muss jedoch in den von Ihnen benötigten Typ konvertiert werden. Dies kann mit den folgenden Methoden erfolgen:

- **nx_sdk_py.void_to_int** konvertiert einen Wert in einen ganzzahligen Typ.
- **nx_sdk_py.void_to_string** konvertiert einen Wert in einen String-Typ.

Die **PostCliCb(self, clicmd)**-Funktion (oder alle nachfolgenden Funktionen) wird in der Regel auch

dort eingesetzt, wo die Ausgabe des Befehls show auf die Konsole gedruckt wird. Dies erfolgt mit der **clicmd.printConsole()**-Methode.

> **Hinweis**: Wenn die Anwendung einen Fehler, eine unbehandelte Ausnahme oder auf andere Weise plötzlich beendet, wird die Ausgabe der Funktion **clicmd.printConsole()** überhaupt nicht angezeigt. Aus diesem Grund empfiehlt es sich beim Debuggen der Python-Anwendung, entweder Debug-Meldungen im Syslog unter Verwendung eines von der **sdk.getTracer()**-Methode zurückgegebenen NxTrace-Objekts zu protokollieren oder Print-Anweisungen zu verwenden und die Anwendung über die **/isan/bin/python** Binärdatei der Bash-Shell auszuführen.

pyCmdHandler-Klassenbeispiel

Der folgende Code dient als Beispiel für die oben beschriebene **pyCmdHandler-**Klasse. Dieser Code stammt aus der Datei ip_move.py in der hier verfügbaren Anwendung ip-move NX-SDK. Diese Anwendung verfolgt die Bewegung einer benutzerdefinierten IP-Adresse über die Schnittstellen eines Nexus-Geräts. Hierzu findet der Code die MAC-Adresse der IP-Adresseingabe über den **<ip>**-Parameter im ARP-Cache des Geräts und überprüft dann, in welchem VLAN sich die MAC-Adresse befindet, mithilfe der MAC-Adresstabelle des Geräts. Mithilfe dieser MAC- und VLAN-Adresse zeigt der **Befehl show system internal l2fm l2dbg macdb address <mac> vlan <vlan>** eine Liste von SNMP-Schnittstellenindizes an, die dieser Kombination in letzter Zeit zugeordnet wurden. Der Code verwendet dann den Befehl **show interface snmp-ifindex**, um aktuelle SNMP-Schnittstellenindizes in lesbare Schnittstellennamen zu übersetzen.

```
class pyCmdHandler(nx_sdk_py.NxCmdHandler):
    def postCliCb(self, clicmd):
        global cli_parser

        if "show_ip_movement" in clicmd.getCmdName():
            target_ip = nx_sdk_py.void_to_string(clicmd.getParamValue("<ip>"))

            target_mac = get_mac_from_arp(cli_parser, clicmd, target_ip)
            mac_vlan = ""
            if target_mac:
                mac_vlan = get_vlan_from_cam(cli_parser, clicmd, target_mac)
                if mac_vlan:
                    find_mac_movement(cli_parser, clicmd, target_mac, mac_vlan)
                else:
                    print("No entires in MAC address table")
                    clicmd.printConsole("No entries in MAC address table for
{}".format(target_mac))
            else:
                clicmd.printConsole("No entries in ARP table for {}".format(target_ip))
        return True

def get_mac_from_arp(cli_parser, clicmd, target_ip):
    exec_cmd = "show ip arp {}".format(target_ip)
    arp_cmd = cli_parser.execShowCmd(exec_cmd, nx_sdk_py.R_JSON)
    if arp_cmd:
        try:
            arp_json = json.loads(arp_cmd)
        except ValueError as exc:
            return None
        count = int(arp_json["TABLE_vrf"]["ROW_vrf"]["cnt-total"])
        if count:
            intf = arp_json["TABLE_vrf"]["ROW_vrf"]["TABLE_adj"]["ROW_adj"]
```

```
            if intf.get("ip-addr-out") == target_ip:
                target_mac = intf["mac"]
                clicmd.printConsole("{} is currently present in ARP table, MAC address
{}\n".format(target_ip, target_mac))
                return target_mac
            else:
                return None
        else:
            return None
    else:
        return None

def get_vlan_from_cam(cli_parser, clicmd, target_mac):
    exec_cmd = "show mac address-table address {}".format(target_mac)
    mac_cmd = cli_parser.execShowCmd(exec_cmd, nx_sdk_py.R_JSON)
    if mac_cmd:
        try:
            cam_json = json.loads(mac_cmd)
        except ValueError as exc:
            return None
        mac_entry = cam_json["TABLE_mac_address"]["ROW_mac_address"]
        if mac_entry:
            if mac_entry["disp_mac_addr"] == target_mac:
                egress_intf = mac_entry["disp_port"]
                mac_vlan = mac_entry["disp_vlan"]
                clicmd.printConsole("{} is currently present in MAC address table on interface
{}, VLAN {}\n".format(target_mac, egress_intf, mac_vlan))
                return mac_vlan
            else:
                return None
        else:
            return None
    else:
        return None

def find_mac_movement(cli_parser, clicmd, target_mac, mac_vlan):
    exec_cmd = "show system internal l2fm l2dbg macdb address {} vlan {}".format(target_mac,
mac_vlan)
    l2fm_cmd = cli_parser.execShowCmd(exec_cmd)
    if l2fm_cmd:
        event_re = re.compile(r"^\s+(\w{3}) (\w{3}) (\d+) (\d{2}):(\d{2}):(\d{2}) (\d{4})
(0x\S{8}) (\d+)\s+(\S+) (\d+)\s+(\d+)\s+(\d+)")
        unique_interfaces = []
        l2fm_events = l2fm_cmd.splitlines()
        for line in l2fm_events:
            res = re.search(event_re, line)
            if res:
                day_name = res.group(1)
                month = res.group(2)
                day = res.group(3)
                hour = res.group(4)
                minute = res.group(5)
                second = res.group(6)
                year = res.group(7)
                if_index = res.group(8)
                db = res.group(9)
                event = res.group(10)
                src=res.group(11)
                slot = res.group(12)
                fe = res.group(13)
                if "MAC_NOTIF_AM_MOVE" in event:
                    timestamp = "{} {} {} {}:{}:{} {}".format(day_name, month, day, hour,
minute, second, year)
                    intf_dict = {"if_index": if_index, "timestamp": timestamp}
```

```
                unique_interfaces.append(intf_dict)
        if not unique_interfaces:
            clicmd.printConsole("No entries for {} in L2FM L2DBG\n".format(target_mac))
        if len(unique_interfaces) == 1:
            clicmd.printConsole("{} has not been moving between
interfaces\n".format(target_mac))
        if len(unique_interfaces) > 1:
                clicmd.printConsole("{} has been moving between the following interfaces, from
most recent to least recent:\n".format(target_mac))
                unique_interfaces = get_snmp_intf_index(unique_interfaces)
                clicmd.printConsole("\t{} - {} (Current interface)\n".format(unique_interfaces[-
1]["timestamp"], unique_interfaces[-1]["intf_name"]))
                for intf in unique_interfaces[-2::-1]:
                    clicmd.printConsole("\t{} - {}\n".format(intf["timestamp"],
intf["intf_name"]))
def get_snmp_intf_index(if_index_dict_list): global cli_parser snmp_ifindex =
cli_parser.execShowCmd("show interface snmp-ifindex", nx_sdk_py.R_JSON) snmp_ifindex_json =
json.loads(snmp_ifindex) snmp_ifindex_list =
snmp_ifindex_json["TABLE_interface"]["ROW_interface"] for index_dict in if_index_dict_list:
index = index_dict["if_index"] for ifindex_json in snmp_ifindex_list: if index ==
ifindex_json["snmp-ifindex"]: index_dict["intf_name"] = ifindex_json["interface"] return
if_index_dict_list
```

Beispiele für benutzerdefinierte CLI-Befehlssyntax

In diesem Abschnitt werden einige Beispiele des Syntaxparameters dargestellt, der beim Erstellen benutzerdefinierter CLI-Befehle mit der **cliP.newShowCmd()**-Methode oder der **cliP.newConfigCmd()**-Methode verwendet wird, wobei **cliP** das von der **sdk.getCliParser() zurückgegebene NxParser-Objekt ist.** Methode.

> **Hinweis**: Die Unterstützung für Syntax mit öffnenden und schließenden Klammern ("(" und ")") wird in NX-SDK v1.5.0 eingeführt, das in NX-OS 7.0(3)I7(3) enthalten ist. Es wird davon ausgegangen, dass der Benutzer NX-SDK v1.5.0 verwendet, wenn er eines der genannten Beispiele befolgt, das auch die Syntax zum Öffnen und Schließen von Klammern umfasst.

Zentrales Schlüsselwort

Dieser Befehl show verwendet ein einzelnes Schlüsselwort mac und fügt dem Schlüsselwort eine Hilfszeichenfolge mit Zeigt alle falsch programmierten MAC-Adressen auf diesem Gerät hinzu.

```
 nx_cmd = cliP.newShowCmd("show_misprogrammed", "mac")
nx_cmd.updateKeyword("mac", "Shows all misprogrammed MAC addresses on this device")
```

Einzelner Parameter

Dieser Befehl show verwendet einen einzigen Parameter **<mac>**. Die umschließenden spitzen Klammern um das Wort mac bedeuten, dass es sich um einen Parameter handelt. Dem Parameter wird eine Hilfszeichenfolge von MAC-Adressen hinzugefügt, die auf Fehlprogrammierung überprüft werden soll. Der **nx_sdk_py.P_MAC_ADDR-**Parameter in der **nx_cmd.updateParam()**-Methode wird verwendet, um den Parametertyp als MAC-Adresse zu definieren, wodurch Endbenutzereingaben eines anderen Typs wie eine Zeichenfolge, eine ganze Zahl oder eine IP-Adresse verhindert werden.

```
 nx_cmd = cliP.newShowCmd("show_misprogrammed_mac", "<mac>")
nx_cmd.updateParam("<mac>", "MAC address to check for misprogramming", nx_sdk_py.P_MAC_ADDR)
```

Optionales Schlüsselwort

Dieser Befehl show kann optional ein einzelnes Schlüsselwort **[mac]** enthalten. Die umschließenden Klammern um das Wort mac geben an, dass dieses Schlüsselwort optional ist. Eine Hilfszeichenfolge von Zeigt alle falsch programmierten MAC-Adressen auf diesem Gerät wird dem Schlüsselwort hinzugefügt.

```
 nx_cmd = cliP.newShowCmd(  "show_misprogrammed_mac"  ,   "[mac]"  )
nx_cmd.updateKeyword( "mac" , "Shows all misprogrammed MAC addresses on this device" )
```

Optionaler Parameter

Dieser Befehl show kann optional einen einzigen Parameter annehmen **[<mac>]**. Die umschließenden Klammern um das Wort **<** mac **>** geben an, dass dieser Parameter optional ist. Die umschließenden spitzen Klammern um das Wort mac bedeuten, dass es sich um einen Parameter handelt. Dem Parameter wird eine Hilfszeichenfolge von MAC-Adressen hinzugefügt, die auf Fehlprogrammierung überprüft werden soll. Der **nx_sdk_py.P_MAC_ADDR-**Parameter in der **nx_cmd.updateParam()**-Methode wird verwendet, um den Parametertyp als MAC-Adresse zu definieren, wodurch Endbenutzereingaben eines anderen Typs wie eine Zeichenfolge, eine ganze Zahl oder eine IP-Adresse verhindert werden.

```
 nx_cmd = cliP.newShowCmd("show_misprogrammed_mac", "[<mac>]")
nx_cmd.updateParam("<mac>", "MAC address to check for misprogramming", nx_sdk_py.P_MAC_ADDR)
```

Zentrales Schlüsselwort und Parameter

Dieser show-Befehl verwendet sofort ein einzelnes Schlüsselwort MAC, gefolgt vom Parameter **<mac-address>**. Die eckigen Klammern um das Wort MAC-Adresse zeigen an, dass es sich um einen Parameter handelt. Dem Schlüsselwort wird eine Hilfszeichenfolge der Check MAC-Adresse für Fehlprogrammierung hinzugefügt. Dem Parameter wird eine Hilfszeichenfolge von MAC-Adressen hinzugefügt, die auf Fehlprogrammierung überprüft werden soll. Der **nx_sdk_py.P_MAC_ADDR-**Parameter in der **nx_cmd.updateParam()**-Methode wird verwendet, um den Parametertyp als MAC-Adresse zu definieren. Dies verhindert die Eingabe eines anderen Typs durch den Endbenutzer, z. B. eine Zeichenfolge, eine ganze Zahl oder eine IP-Adresse.

```
 nx_cmd = cliP.newShowCmd("show_misprogrammed", "mac <mac-address>")
nx_cmd.updateKeyword("mac", "Check MAC address for misprogramming")
nx_cmd.updateParam("<mac-address>", "MAC address to check for misprogramming",
nx_sdk_py.P_MAC_ADDR)
```

Mehrere Schlüsselwörter und Parameter

Dieser Befehl show kann eines von zwei Schlüsselwörtern verwenden, die beide über zwei verschiedene Parameter verfügen. Das erste Schlüsselwort MAC hat einen Parameter von **<mac-address>**, das zweite Schlüsselwort ip hat einen Parameter von **<ip-address>**. Die umschließenden spitzen Klammern um die Wörter MAC-Adresse und IP-Adresse geben an, dass es sich um Parameter handelt. Dem MAC-Schlüsselwort wird eine Hilfszeichenfolge der Check MAC-Adresse für Fehlprogrammierung hinzugefügt. Dem **<mac-address>** Parameter wird eine Hilfszeichenfolge von MAC-Adressen hinzugefügt, um auf Fehlprogrammierung zu überprüfen.

Der **nx_sdk_py.P_MAC_ADDR-**Parameter in der **nx_cmd.updateParam()**-Methode wird verwendet, um den Typ des **<mac-address>**Parameters als MAC-Adresse zu definieren, wodurch die Eingabe eines anderen Typs durch den Endbenutzer verhindert wird, z. B. eine Zeichenfolge, eine ganze Zahl oder eine IP-Adresse. Eine Hilfszeichenfolge der Check-IP-Adresse für Fehlprogrammierung wird dem ip-Schlüsselwort hinzugefügt. Dem **<ip-address>**-Parameter wird eine Hilfszeichenfolge von IP-Adressen hinzugefügt, um eine Fehlprogrammierung zu überprüfen. Der **nx_sdk_py.P_IP_ADDR-**Parameter in der **nx_cmd.updateParam()**-Methode wird verwendet, um den Typ des **<ip-address>**Parameters als IP-Adresse zu definieren, wodurch die Eingabe eines anderen Typs durch den Endbenutzer verhindert wird, z. B. eine Zeichenfolge, eine ganze Zahl oder eine IP-Adresse.

```
 nx_cmd = cliP.newShowCmd("show_misprogrammed", "(mac <mac-address> | ip <ip-address>)")
nx_cmd.updateKeyword("mac", "Check MAC address for misprogramming")
nx_cmd.updateParam("<mac-address>", "MAC address to check for misprogramming",
nx_sdk_py.P_MAC_ADDR)
nx_cmd.updateKeyword("ip", "Check IP address for misprogramming")
nx_cmd.updateParam("<ip-address>", "IP address to check for misprogramming",
nx_sdk_py.P_IP_ADDR)
```

Mehrere Schlüsselwörter und Parameter mit optionalem Schlüsselwort

Dieser Befehl show kann eines von zwei Schlüsselwörtern verwenden, die beide über zwei verschiedene Parameter verfügen. Das erste Schlüsselwort MAC hat einen Parameter von **<mac-address>**, das zweite Schlüsselwort ip hat einen Parameter von **<ip-address>**. Die umschließenden spitzen Klammern um die Wörter MAC-Adresse und IP-Adresse geben an, dass es sich um Parameter handelt. Dem MAC-Schlüsselwort wird eine Hilfszeichenfolge der Check MAC-Adresse für Fehlprogrammierung hinzugefügt. Dem **<mac-address>** Parameter wird eine Hilfszeichenfolge von MAC-Adressen hinzugefügt, um auf Fehlprogrammierung zu überprüfen. Der **nx_sdk_py.P_MAC_ADDR-**Parameter in der **nx_cmd.updateParam()**-Methode wird verwendet, um den Typ des **<mac-address>**Parameters als MAC-Adresse zu definieren, wodurch die Eingabe eines anderen Typs durch den Endbenutzer verhindert wird, z. B. eine Zeichenfolge, eine ganze Zahl oder eine IP-Adresse. Eine Hilfszeichenfolge der Check-IP-Adresse für Fehlprogrammierung wird dem ip-Schlüsselwort hinzugefügt. Dem **<ip-address>**-Parameter wird eine Hilfszeichenfolge von IP-Adressen hinzugefügt, um eine Fehlprogrammierung zu überprüfen. Der **nx_sdk_py.P_IP_ADDR-**Parameter in der **nx_cmd.updateParam()**-Methode wird verwendet, um den Typ des **<ip-address>**Parameters als IP-Adresse zu definieren, wodurch die Eingabe eines anderen Typs durch den Endbenutzer verhindert wird, z. B. eine Zeichenfolge, eine ganze Zahl oder eine IP-Adresse. Dieser Befehl show kann optional ein Schlüsselwort [clear] verwenden. Diesem optionalen Schlüsselwort wird eine Hilfszeichenfolge Clears-Adressen hinzugefügt, die als falsch programmiert erkannt wurden.

```
 nx_cmd = cliP.newShowCmd("show_misprogrammed", "(mac <mac-address> | ip <ip-address>) [clear]")
nx_cmd.updateKeyword("mac", "Check MAC address for misprogramming")
nx_cmd.updateParam("<mac-address>", "MAC address to check for misprogramming",
nx_sdk_py.P_MAC_ADDR)
nx_cmd.updateKeyword("ip", "Check IP address for misprogramming")
nx_cmd.updateParam("<ip-address>", "IP address to check for misprogramming",
nx_sdk_py.P_IP_ADDR)
nx_cmd.updateKeyword("clear", "Clears addresses detected to be misprogrammed")
```

Mehrere Schlüsselwörter und Parameter mit optionalem Parameter

Dieser Befehl show kann eines von zwei Schlüsselwörtern verwenden, die beide über zwei

verschiedene Parameter verfügen. Das erste Schlüsselwort MAC hat einen Parameter von **<mac-address>**, das zweite Schlüsselwort ip hat einen Parameter von **<ip-address>**. Die umschließenden spitzen Klammern um die Wörter MAC-Adresse und IP-Adresse geben an, dass es sich um Parameter handelt. Eine Hilfszeichenfolge der Check-MAC-Adresse für Fehlprogrammierung wird dem MAC-Schlüsselwort hinzugefügt. Dem **<mac-address>** Parameter wird eine Hilfszeichenfolge von MAC-Adressen hinzugefügt, um auf Fehlprogrammierung zu überprüfen. Der **nx_sdk_py.P_MAC_ADDR-**Parameter in der **nx_cmd.updateParam()**-Methode wird verwendet, um den Typ des **<mac-address>**Parameters als MAC-Adresse zu definieren, wodurch die Eingabe eines anderen Typs durch den Endbenutzer verhindert wird, z. B. eine Zeichenfolge, eine ganze Zahl oder eine IP-Adresse. Eine Hilfszeichenfolge der Check-IP-Adresse für Fehlprogrammierung wird dem ip-Schlüsselwort hinzugefügt. Dem **<ip-address>**-Parameter wird eine Hilfszeichenfolge von IP-Adressen hinzugefügt, um eine Fehlprogrammierung zu überprüfen. Der **nx_sdk_py.P_IP_ADDR-**Parameter in der **nx_cmd.updateParam()**-Methode wird verwendet, um den Typ des **<ip-address>**Parameters als IP-Adresse zu definieren, wodurch die Eingabe eines anderen Typs durch den Endbenutzer verhindert wird, z. B. eine Zeichenfolge, eine ganze Zahl oder eine IP-Adresse. Dieser Befehl show kann optional einen Parameter **[<module>]** annehmen. Dem optionalen Parameter wird eine Hilfszeichenfolge hinzugefügt, die nur klare Adressen auf dem angegebenen Modul enthält.

```
 nx_cmd = cliP.newShowCmd("show_misprogrammed", "(mac <mac-address> | ip <ip-address>)
[<module>]")
nx_cmd.updateKeyword("mac", "Check MAC address for misprogramming")
nx_cmd.updateParam("<mac-address>", "MAC address to check for misprogramming",
nx_sdk_py.P_MAC_ADDR)
nx_cmd.updateKeyword("ip", "Check IP address for misprogramming")
nx_cmd.updateParam("<ip-address>", "IP address to check for misprogramming",
nx_sdk_py.P_IP_ADDR)
nx_cmd.updateParam("<module>", "Clears addresses detected to be misprogrammed",
nx_sdk_py.P_INTEGER)
```

Debuggen einer Python-Anwendung mit NX-SDK

Nachdem eine NX-SDK-Python-Anwendung erstellt wurde, muss sie häufig gedebuggt werden. NX-SDK informiert Sie, falls Syntaxfehler im Code auftreten, aber da die Python NX-SDK-Bibliothek SWIG verwendet, um C++-Bibliotheken in Python-Bibliotheken zu übersetzen, führen alle Ausnahmen, die bei der Codeausführung auftreten, zu einem Anwendungskern-Dump, der dieser ähnlich ist:

```
 terminate called after throwing an instance of 'Swig::DirectorMethodException'
what(): SWIG director method error. Error detected when calling 'NxCmdHandler.postCliCb'
Aborted (core dumped)
```

Aufgrund der mehrdeutigen Natur dieser Fehlermeldung besteht die beste Methode zum Debuggen von Python-Anwendungen darin, Debug-Meldungen im Syslog unter Verwendung eines von der **sdk.getTracer()**-Methode zurückgegebenen NxTrace-Objekts zu protokollieren. Dies wird wie folgt gezeigt:

```
#! /isan/bin/python

tracer = 0

def evt_thread():
```

```
    <snip>
    tracer = sdk.getTracer()
    tracer.event("[NXSDK-APP][INFO] Started service")
<snip>
class pyCmdHandler(nx_sdk_py.NxCmdHandler):
    def postCliCb(self, clicmd):
        global tracer
        tracer.event("[NXSDK-APP][DEBUG] Received command: {}".format(clicmd))
        if "show_test_command" in clicmd.getCmdName():
            tracer.event("[NXSDK-APP][DEBUG] `show_test_command` recognized")
```

Wenn die Protokollierung von Debug-Meldungen im Syslog keine Option ist, besteht eine Alternative darin, Print-Anweisungen zu verwenden und die Anwendung über die Bash-Shell **/isan/bin/python-**Binärdatei auszuführen. Die Ausgabe dieser Druckanweisungen wird jedoch nur sichtbar, wenn sie auf diese Weise ausgeführt wird. Die Ausführung der Anwendung über die VSH-Shell erzeugt keine Ausgabe. Ein Beispiel für die Verwendung von Druckaussagen ist hier:

```
#! /isan/bin/python

tracer = 0

def evt_thread():
    <snip>
    print("[NXSDK-APP][INFO] Started service")
<snip>
class pyCmdHandler(nx_sdk_py.NxCmdHandler):
    def postCliCb(self, clicmd):
        print("[NXSDK-APP][DEBUG] Received command: {}".format(clicmd))
        if "show_test_command" in clicmd.getCmdName():
            print("[NXSDK-APP][DEBUG] `show_test_command` recognized")
```

Bereitstellen einer Python-Anwendung mit NX-SDK

Nachdem eine Python-Anwendung vollständig in der Bash-Shell getestet und für die Bereitstellung bereit ist, sollte die Anwendung über VSH in der Produktion installiert werden. Auf diese Weise kann die Anwendung bei einem erneuten Laden des Geräts oder bei einem Systemwechsel in einem Szenario mit zwei Supervisoren beibehalten werden. Um eine Anwendung über VSH bereitzustellen, müssen Sie ein RPM-Paket mit einer NX-SDK- und ENXOS SDK-Buildumgebung erstellen. Cisco DevNet bietet ein Docker-Image, das die Erstellung von RPM-Paketen vereinfacht.

> **Hinweis**: Wenn Sie Hilfe bei der Installation von Docker auf Ihrem Betriebssystem benötigen, lesen Sie die Installationsdokumentation von Docker.

Ziehen Sie auf einem Docker-fähigen Host die gewünschte Image-Version mit dem Befehl **docker ull dockercisco/nxsdk:<tag>** an, wobei **<tag>** das Tag der gewünschten Image-Version ist. Sie können die verfügbaren Bildversionen und die zugehörigen Tags hier anzeigen. Dies wird mit dem **v1-**Tag hier veranschaulicht:

```
 docker pull dockercisco/nxsdk:v1
```

Starten Sie einen Container mit dem Namen **nxsdk** aus diesem Bild, und fügen Sie ihn an. Wenn das Tag Ihrer Wahl anders ist, ersetzen Sie **v1** durch Ihr Tag:

```
 docker run -it --name nxsdk dockercisco/nxsdk:v1 /bin/bash
```

Aktualisieren Sie auf die neueste Version von NX-SDK, navigieren Sie zum **NX-SDK-**Verzeichnis, und rufen Sie dann die neuesten Dateien von git ab:

```
 cd /NX-SDK/
git pull
```

Wenn Sie eine ältere Version von NX-SDK verwenden möchten, können Sie die NX-SDK-Zweigstelle mit dem entsprechenden Version-Tag mit dem **Git Clone -b v<version>** https://github.com/CiscoDevNet/NX-SDK.git klonen, wobei **<version>** die gewünschte Version von NX-SDK ist. Dies wird hier mit NX-SDK v1.0.0 veranschaulicht:

```
 cd /
rm -rf /NX-SDK
git clone -b v1.0.0 https://github.com/CiscoDevNet/NX-SDK.git
```

Übertragen Sie anschließend Ihre Python-Anwendung auf den Docker-Container. Dafür gibt es verschiedene Möglichkeiten.

- Beenden Sie den Docker-Container (der den Container stoppt und ihn erneut starten muss), übertragen Sie die Python-Anwendung auf den Docker-Host, und verwenden Sie dann den Befehl **docker cp**, um die Anwendung vom Host in den Container zu kopieren. Dies wird hier unter der Annahme nachgewiesen, dass die Python-Anwendung an den Docker-Host unter **/app/python_app.py** übertragen wurde.

```
 root@2dcbe841742a:~# exit
[root@localhost ~]# docker cp /app/python_app.py nxsdk:/root/
[root@localhost ~]# docker start nxsdk
nxsdk
[root@localhost ~]# docker attach nxsdk
root@2dcbe841742a:/# ls /root/
python_app.py
```

- Kopieren Sie den Inhalt der Python-Anwendung in die System-Zwischenablage, und fügen Sie den Inhalt mithilfe von vim in eine Datei ein, die im Docker-Container erstellt wurde.

Verwenden Sie anschließend das Skript **rpm_gen.py** in **/NX-SDK/scripts/**, um ein RPM-Paket aus der Python-Anwendung zu erstellen. Dieses Skript verfügt über ein erforderliches Argument und zwei erforderliche Switches:

- Der Dateiname der Python-Anwendung. Beispielsweise würde eine Python-Anwendung in einer Datei mit dem Namen **python_app.py** zu einem Argument von **python_app.py** führen. Dieser Dateiname wird später als Anwendungsname für NX-SDK verwendet und auch von NX-OS verwendet, um auf die von dieser Anwendung erstellten Befehle zu verweisen.

  Hinweis: Der Dateiname muss keine Dateierweiterungen wie **.py** enthalten. Wenn in diesem Beispiel der Dateiname **python_app** anstelle von **python_app.py** lautete, würde das RPM-Paket ohne Problem generiert.

- Der -s-Switch verwendet ein Argument für den absoluten Dateipfad, der zum Speicherort des zuvor erwähnten Dateinamens führt. Wenn sich beispielsweise **python_app.py** in **/root/**befindet**,** wäre das richtige Argument **-s /root/**.
- Der -u-Schalter gibt an, dass der Quelldateiname mit dem ausführbaren Dateinamen identisch ist.

Die Verwendung des Skripts **rpm_gen.py** wird hier demonstriert.

```
root@7bfd1714dd2f:~# python /NX-SDK/scripts/rpm_gen.py test_python_app -s /root/ -u
##############################################################################################################
####
Generating rpm package...
<snip>
RPM package has been built
##############################################################################################################
####

SPEC file: /NX-SDK/rpm/SPECS/test_python_app.spec
RPM file : /NX-SDK/rpm/RPMS/test_python_app-1.0-1.0.0.x86_64.rpm
```

Der filepath zum RPM-Paket ist in der letzten Zeile der Ausgabe des Skripts **rpm_gen.py** angegeben. Diese Datei muss aus dem Docker-Container auf den Host kopiert werden, damit sie auf das Nexus-Gerät übertragen werden kann, auf dem die Anwendung ausgeführt werden soll. Nachdem Sie den Docker-Container verlassen haben, kann er mit dem Befehl **docker cp <container>:<container_filepath> <host_filepath>** durchgeführt werden, wobei **<container>** der Name des NX-SDK-Docker-Containers ist (in diesem Fall nxsdk), **<container_filepath>** der vollständige Pfad des RPM-Pakets im Container ist **/NX-SDK/rpm/RPMS/test_python_app-1.0-1.0.0.x86_64.rpm**) und **<host_filepath> ist der vollständige Pfad auf unserem Docker-Host, auf den das RPM-Paket übertragen werden soll (in diesem Fall /root/).** Dieser Befehl wird hier veranschaulicht:

```
root@7bfd1714dd2f:/# exit
[root@localhost ~]# docker cp nxsdk:/NX-SDK/rpm/RPMS/test_python_app-1.0-1.0.0.x86_64.rpm /root/
[root@localhost ~]# ls /root/
anaconda-ks.cfg             test_python_app-1.0-1.0.0.x86_64.rpm
```

Übertragen Sie dieses RPM-Paket auf das Nexus-Gerät unter Verwendung Ihrer bevorzugten Dateiübertragungsmethode. Sobald sich das RPM-Paket auf dem Gerät befindet, muss es ähnlich wie eine SMU installiert und aktiviert werden. Dies wird folgendermaßen gezeigt, unter der Annahme, dass das RPM-Paket auf den Bootflash des Geräts übertragen wurde.

```
N9K-C93180LC-EX# install add bootflash:test_python_app-1.0-1.0.0.x86_64.rpm
[####################] 100%
Install operation 27 completed successfully at Tue May  8 06:40:13 2018
N9K-C93180LC-EX# install activate test_python_app-1.0-1.0.0.x86_64
[####################] 100%
Install operation 28 completed successfully at Tue May  8 06:40:20 2018
```

> **Hinweis**: Wenn Sie das RPM-Paket mit dem Befehl **install add** installieren, geben Sie das Speichergerät und den genauen Dateinamen des Pakets ein. Wenn Sie das RPM-Paket nach der Installation aktivieren, schließen Sie das Speichergerät und den Dateinamen nicht an - verwenden Sie den Namen des Pakets selbst. Sie können den Paketnamen mit dem Befehl **show install inactive** überprüfen.

Nachdem das RPM-Paket aktiviert wurde, können Sie die Anwendung mit NX-SDK mit dem **nxsdk-Service <application-name>**-Konfigurationsbefehl starten, wobei **<application-name>** der Name des Python-Dateinamens (und anschließend die Anwendung) ist, der definiert wurde, als rpm_gen.py-Skript früher verwendet wurde. Dies wird wie folgt gezeigt:

```
N9K-C93180LC-EX# conf
```

```
Enter configuration commands, one per line. End with CNTL/Z.
N9K-C93180LC-EX(config)# nxsdk service-name test_python_app
% This could take some time. "show nxsdk internal service" to check if your App is Started &
Runnning
```

Sie können überprüfen, ob die Anwendung aktiv ist und mit dem Befehl **show nxsdk internal service** ausgeführt wurde:

```
N9K-C93180LC-EX# show nxsdk internal service

NXSDK Started/Temp unavailabe/Max services : 1/0/32
NXSDK Default App Path         : /isan/bin/nxsdk
NXSDK Supported Versions       : 1.0

Service-name              Base App        Started(PID)      Version    RPM Package
------------------------- --------------- ----------------- ---------- -----------------------
test_python_app           nxsdk_app4      VSH(23195)        1.0        test_python_app-1.0-
1.0.0.x86_64
```

Sie können auch überprüfen, ob die von dieser Anwendung erstellten benutzerdefinierten CLI-Befehle in NX-OS verfügbar sind:

```
N9K-C93180LC-EX# show test?
  test_python_app    Nexus Sdk Application
```

Zugehörige Informationen

- NX-SDK GitHub
- Cisco Nexus NX-OS-Programmierhandbuch der Serie 9000, Version 7.x
- Cisco Nexus 3000 NX-OS-Programmierhandbuch, Version 7.x
- Cisco Nexus 3500 NX-OS-Programmierhandbuch, Version 7.x
- Whitepaper: Netzwerkprogrammierbarkeit und -automatisierung mit Cisco Nexus Switches der Serie 9000
- Programmierbarkeit und Automatisierung mit Cisco Open NX-OS (PDF)
- Technischer Support und Dokumentation - Cisco Systems