Helion

Zautomatyzuj swoją firmę z Pythonem

Praktyczne rozwiązania dla firmowej sieci

Packt›

Bassem Aly

Tytuł oryginału: Hands-On Enterprise Automation with Python: Automate common administrative and security tasks with Python

Tłumaczenie: Łukasz Wójcicki

ISBN: 978-83-283-5331-2

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
ftp://ftp.helion.pl/przyklady/zafipy.zip

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/zafipy
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści

4

ROZDZIAŁ

Zarządzanie urządzeniami sieciowymi za pomocą języka Python

Do tej pory nauczyliśmy się, jak korzystać z Pythona w różnych systemach operacyjnych, i dowiedzieliśmy się, jak zbudować topologię sieci za pomocą narzędzia EVE-NG. W tym rozdziale nauczymy się, jak korzystać z bibliotek sieciowych, które służą do automatyzacji różnych zadań sieciowych. Python może współpracować z urządzeniami na wielu warstwach sieci.

Na początku nauczymy się obsługiwać warstwę niskiego poziomu za pomocą programowania gniazd sieciowych i **modułów `socket`**. Moduły `socket` dostarczają interfejsów niskiego poziomu pomiędzy systemem operacyjnym (na którym pracuje Python) a urządzeniem sieciowym. Ponadto moduły Pythona mogą współpracować na wyższym poziomie poprzez Telnet, SSH i API. Z tego rozdziału dowiemy się także, jak za pomocą Pythona ustanowić zdalne połączenia i jak wykonać operację na zdalnym urządzeniu, korzystając z Telnetu i modułów SSH.

Tematy omówione w tym rozdziale:

- Jak połączyć się z urządzeniem za pomocą protokołu Telnet i języka Python
- Współpraca Pythona z SSH
- Praca z sieciami za pomocą biblioteki netaddr
- Przykłady wykorzystania metodologii automatyzacji pracy w sieciach (ang. *network automation*)

Wymagania techniczne

Należy mieć zainstalowane następujące narzędzia:

- Python 2.7.1x;
- darmowa wersja PyCharm Community lub płatna PyCharm Pro Edition;
- topologia EVE-NG (zobacz rozdział 3., „Konfigurowanie sieciowego środowiska laboratoryjnego", aby się dowiedzieć, jak zainstalować i skonfigurować emulator).

Przytoczone w tym rozdziale źródła skryptów możesz znaleźć na GitHubie pod adresem *https://github.com/TheNetworker/EnterpriseAutomation.git*.

Python i SSH

W odróżnieniu od Telnetu protokół SSH tworzy bezpieczny kanał komunikacji między klientem i serwerem. Utworzony tunel zabezpieczony jest za pomocą różnych algorytmów szyfrujących, które skutecznie utrudniają deszyfrację przesyłanych treści. Specjaliści ds. sieci w pierwszej kolejności decydują się na użycie SSH do zabezpieczania swoich końcówek sieciowych.

Python może komunikować się z urządzeniami sieciowymi za pomocą protokołu SSH, wykorzystując popularną bibliotekę Paramiko. Paramiko wspiera mechanizmy uwierzytelnienia, algorytmy wymiany klucza (DSA, RSA, ECDSA i ED25519) oraz wiele innych właściwości SSH (np. komendę `proxy` i SFTP).

Moduł Paramiko

Paramiko jest najpopularniejszym modułem do obsługi protokołu SSH za pomocą języka Python. Na oficjalnej stronie modułu (na GitHubie) możemy dowiedzieć się, że nazwa Paramiko powstała jako kombinacja dwóch słów z języka esperanto oznaczających „paranoidalny" i „przyjaciel". Moduł został napisany w języku Python — poza kilkoma funkcjami (np. kryptograficznymi), które zostały napisane w języku C. Więcej informacji na temat twórców i historii Paramiko możesz znaleźć na oficjalnej stronie modułu, pod adresem *https://github.com/paramiko/paramiko*.

Instalacja modułu

Jeżeli pracujesz pod systemem Windows, otwórz konsolę poleceń za pomocą polecenia `cmd`. Jeżeli pracujesz pod systemem Linux, skorzystaj z powłoki. Musimy pobrać najnowszy moduł Paramiko z PyPI (Python Package Index). Dodatkowo zostaną pobrane pakiety powiązane z naszym modułem, tj. `cryptography`, `ipaddress` oraz `six`. Wykonujemy polecenie:

```
pip install paramiko
```

```
bassim@me-inside:~$ pip install paramiko
Collecting paramiko
  Using cached paramiko-2.4.0-py2.py3-none-any.whl
Collecting cryptography>=1.5 (from paramiko)
  Using cached cryptography-2.1.4-cp27-cp27mu-manylinux1_x86_64.whl
Collecting pynacl>=1.0.1 (from paramiko)
  Using cached PyNaCl-1.2.1-cp27-cp27mu-manylinux1_x86_64.whl
Collecting pyasn1>=0.1.7 (from paramiko)
  Using cached pyasn1-0.4.2-py2.py3-none-any.whl
Collecting bcrypt>=3.1.3 (from paramiko)
  Using cached bcrypt-3.1.4-cp27-cp27mu-manylinux1_x86_64.whl
Collecting cffi>=1.7; platform_python_implementation != "PyPy" (from cryptography>=1.5->paramiko)
  Downloading cffi-1.11.4-cp27-cp27mu-manylinux1_x86_64.whl (406kB)
    100% |                                | 409kB 1.2MB/s
Collecting enum34; python_version < "3" (from cryptography>=1.5->paramiko)
  Using cached enum34-1.1.6-py2-none-any.whl
Collecting idna>=2.1 (from cryptography>=1.5->paramiko)
  Using cached idna-2.6-py2.py3-none-any.whl
Collecting asn1crypto>=0.21.0 (from cryptography>=1.5->paramiko)
  Using cached asn1crypto-0.24.0-py2.py3-none-any.whl
Collecting six>=1.4.1 (from cryptography>=1.5->paramiko)
  Using cached six-1.11.0-py2.py3-none-any.whl
Collecting ipaddress; python_version < "3" (from cryptography>=1.5->paramiko)
Collecting pycparser (from cffi>=1.7; platform_python_implementation != "PyPy"->cryptography>=1.5->pa
ramiko)
Installing collected packages: pycparser, cffi, enum34, idna, asn1crypto, six, ipaddress, cryptograph
y, pynacl, pyasn1, bcrypt, paramiko
```

Następnie możesz zweryfikować proces instalacji poprzez uruchomienie powłoki Pythona i spróbować zaimportować moduł Paramiko — tak jak pokazano na zrzucie poniżej. Po wykonaniu importu Python nie powinien pokazać żadnych błędów.

```
bassim@me-inside:~$ python
Python 2.7.14 (default, Sep 23 2017, 22:06:14)
[GCC 7.2.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import paramiko
>>>
```

Połączenie SSH z urządzeniem sieciowym

Na początku skryptu musimy zaimportować moduł, z którego chcemy skorzystać. Następnie utworzymy klienta SSH, korzystając z funkcji `SSHClient()`. W następnej kolejności skonfigurujemy moduł Paramiko, aby automatycznie dodawał klucz każdego hosta jako zaufany i ustanawiał połączenie między klientem i serwerem. Następnie użyjemy funkcji `connect` i wprowadzimy dane uwierzytelniające:

```
#!/usr/bin/python
__author__ = "Bassim Aly"
__EMAIL__ = "basim.alyy@gmail.com"
import paramiko
import time
Channel = paramiko.SSHClient()
Channel.set_missing_host_key_policy(paramiko.AutoAddPolicy())
Channel.connect(hostname="10.10.88.112", username='admin',
password='access123', look_for_keys=False, allow_agent=False)
shell = Channel.invoke_shell()
```

`AutoAddPolicy()` to tylko jedna z polityk, jaką można wykorzystać w funkcji `set_missing_host_key_policy()`. Jest ona akceptowalna jedynie w naszym środowisku testowym. W środowisku produkcyjnym powinniśmy użyć bardziej restrykcyjnych polityk typu `WarningPolicy()` lub `RejectPolicy()`.

Funkcja `invoke_shell()` wywoływana na końcu skryptu tworzy interaktywną sesję powłoki z serwerem SSH. Jako argumenty do funkcji można podać dodatkowe parametry: rodzaj terminala, szerokość i wysokość.

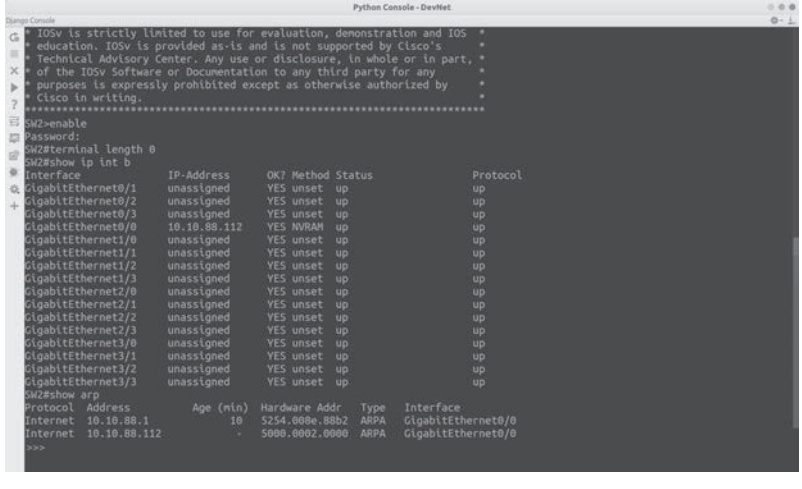
Argumenty funkcji `connect`:

- `Look_For_Keys` — domyślnie ma wartość `True`, co wymusza na module Paramiko stosowanie uwierzytelnienia za pomocą pary kluczy (klucz prywatny, klucz publiczny) w stosunku do urządzenia sieciowego. W przypadku gdy parametr ten ma wartość `False`, stosowane jest uwierzytelnienie za pomocą hasła.
- `allow_agent paramiko` — dzięki tej opcji połączenie następuje do lokalnego agenta SSH. Opcja wymagana jest przy uwierzytelnieniu za pomocą kluczy. W przypadku kiedy do uwierzytelnienia stosujemy login/hasło opcja jest ustawiana na `False`.

Ostatecznym działaniem skryptu jest wywołanie serii komend do terminala urządzenia, takich jak `show ip int b` i `show arp`, oraz uzyskanie wyniku w powłoce Pythona:

```
shell.send("enable\n")
shell.send("access123\n")
shell.send("terminal length 0\n")
shell.send("show ip int b\n")
shell.send("show arp\n")
time.sleep(2)
print shell.recv(5000)
Channel.close()
```

Wynik działania skryptu:



Gdy chcemy wywołać komendy, których przetwarzanie zajmuje więcej czasu, zalecane jest użycie funkcji `time.sleep()`. Jeśli nie użyjemy takiej funkcji i Python za szybko zwróci nam wynik, możemy spodziewać się pustego wyniku.

Moduł Netmiko

Moduł Netmiko jest bardziej rozbudowaną wersją modułu Paramiko i potrafi łączyć się z konkretnymi rodzajami urządzeń sieciowych. Podczas połączenia SSH moduł Paramiko sprawdza jedynie w sposób ogólny, czy dane urządzenie (z którym się łączy) jest na przykład serwerem czy drukarką. Natomiast moduł Netmiko łączy się z konkretnym typem urządzenia. Dzięki temu połączenia są obsługiwane dużo wydajniej. Moduł ten obsługuje szeroki zakres urządzeń wielu producentów.

Netmiko jest swego rodzaju nakładką na Paramiko i rozszerza jego możliwości. Możemy skorzystać z możliwości urządzenia dostarczanych bezpośrednio od producenta, odczytywać i zapisywać informacje w pliku konfiguracyjnym lub wysyłać znak powrotu karetki `\n` po każdej komendzie.

Wsparcie producentów

Moduł Netmiko wspiera wielu producentów i systematycznie dodawani są nowi. Poniższa lista urządzeń została podzielona na trzy grupy: regularnie testowane, testowane w ograniczonym zakresie oraz eksperymentalne. Listę można znaleźć także na GitHubie pod adresem *https://github.com/ktbyers/netmiko#supports*.

Oto lista wspieranych producentów z grupy „regularnie testowane”:

Regularnie testowane:
Arista vEOS
Cisco ASA
Cisco IOS
Cisco IOS-XE
Cisco IOS-XR
Cisco NX-OS
Cisco SG300
HP Comware7
HP ProCurve
Juniper Junos
Linux

Oto lista wspieranych producentów z grupy „testowane w ograniczonym zakresie”:

Testowane w ograniczonym zakresie:
Alcatel AOS6/AOS8
Avaya ERS
Avaya VSP
Brocade VDX
Brocade MLX/NetIron
Calix B6
Cisco WLC
Dell-Force10
Dell PowerConnect
Huawei
Mellanox
NetApp cDOT
Palo Alto PAN-OS
Pluribus
Ruckus ICX/FastIron
Ubiquiti EdgeSwitch
Vyatta VyOS

Oto lista wspieranych producentów z grupy „eksperymentalne”:

Eksperymentalne:
A10
Accedian
Aruba
Ciena SAOS
Cisco Telepresence
Check Point GAiA
Coriant
Eltex
Enterasys
Extreme EXOS
Extreme Wing
F5 LTM
Fortinet
MRV Communications OptiSwitch
Nokia/Alcatel SR-OS
QuantaMesh

Instalacja i weryfikacja

Jeżeli pracujesz pod systemem Windows, otwórz konsolę poleceń za pomocą polecenia `cmd`. Jeżeli pracujesz pod systemem Linux, skorzystaj z powłoki. Musimy pobrać najnowszy moduł Netmiko z PyPI (Python Package Index). Wykonujemy polecenie:

```
pip install netmiko
```

```
bassim@me-inside:~$ pip install netmiko
Collecting netmiko
  Downloading netmiko-2.0.1.tar.gz (68kB)
    100% |████████████████████████████████| 71kB 450kB/s
Collecting paramiko>=2.0.0 (from netmiko)
  Using cached paramiko-2.4.0-py2.py3-none-any.whl
Collecting scp>=0.10.0 (from netmiko)
  Using cached scp-0.10.2-py2.py3-none-any.whl
Collecting pyyaml (from netmiko)
Collecting pyserial (from netmiko)
  Using cached pyserial-3.4-py2.py3-none-any.whl
Collecting textfsm (from netmiko)
  Downloading textfsm-0.3.2.tar.gz
Collecting cryptography>=1.5 (from paramiko>=2.0.0->netmiko)
  Using cached cryptography-2.1.4-cp27-cp27mu-manylinux1_x86_64.whl
Collecting pynacl>=1.0.1 (from paramiko>=2.0.0->netmiko)
  Using cached PyNaCl-1.2.1-cp27-cp27mu-manylinux1_x86_64.whl
Collecting pyasn1>=0.1.7 (from paramiko>=2.0.0->netmiko)
  Using cached pyasn1-0.4.2-py2.py3-none-any.whl
Collecting bcrypt>=3.1.3 (from paramiko>=2.0.0->netmiko)
  Using cached bcrypt-3.1.4-cp27-cp27mu-manylinux1_x86_64.whl
Collecting cffi>=1.7; platform_python_implementation != "PyPy" (from cryptography>=1.5->param
iko>=2.0.0->netmiko)
  Using cached cffi-1.11.4-cp27-cp27mu-manylinux1_x86_64.whl
Collecting enum34; python_version < "3" (from cryptography>=1.5->paramiko>=2.0.0->netmiko)
  Using cached enum34-1.1.6-py2-none-any.whl
Collecting idna>=2.1 (from cryptography>=1.5->paramiko>=2.0.0->netmiko)
```

Następnie importujemy moduł Netmiko z poziomu powłoki Pythona, aby przekonać się, czy nie ma błędów:

```
$ python
>>> import netmiko
```

Połączenie SSH za pomocą Netmiko

Przeszedł czas, aby poznać możliwości Netmiko podczas połączenia SSH. Domyślnie w trakcie sesji Netmiko wykonuje wiele operacji w tle: zajmuje się obsługą nieznanych kluczy SSH, ustala typ i rozmiar terminala oraz wykonuje operacje związane z konkretnym typem urządzenia (np. włączenie trybu uprzywilejowanego). W pierwszej kolejności musisz zdefiniować urządzenie (z którym się łączy) i wprowadzić wartości dla pięciu obowiązkowych parametrów:

```
R1 = {
    'device_type': 'cisco_ios',
    'ip': '10.10.88.110',
    'username': 'admin',
    'password': 'access123',
    'secret': 'access123',
}
```

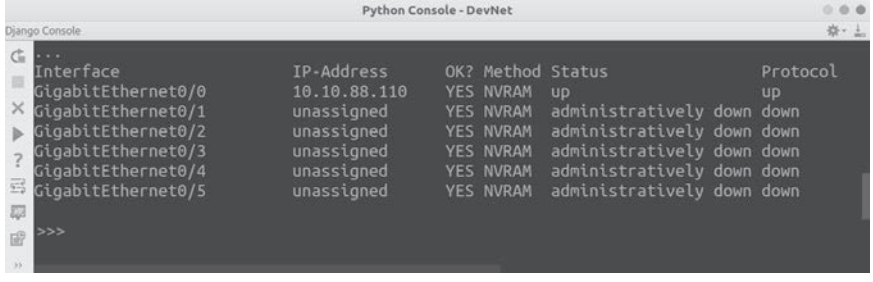
Pierwszy parametr, `device_type`, określa typ urządzenia. Dzięki temu określamy zakres prawidłowych komend. Następnie definiujemy adres IP (pole `ip`) urządzenia. W polu `ip` możemy zamiast adresu IP podać nazwę hosta — o ile może być poprawnie rozwiązana przez serwer DNS. W następnych polach (`username`, `password`, `secret`) podajemy login, hasło oraz hasło dla trybu uprzywilejowanego. W tym miejscu warto zaznaczyć, że możesz użyć funkcji `getpass()`, dzięki której nie podajemy hasła w sposób jawny (w kodzie skryptu), lecz w czasie wykonywania skryptu.

Należy pamiętać, że kolejność parametrów (wewnątrz zmiennej R1) nie jest ważna. Nie można za to zmienić nazw parametrów — muszą być dokładnie takie, jak w powyższym przykładzie.

W następnej kolejności zaimportujemy z modułu Netmiko funkcję `ConnectHandler` i jako argument do niej podamy zdefiniowaną wcześniej strukturę `R1`. Jeżeli wcześniej nasze urządzenia zostały skonfigurowane z hasłem trybu uprzywilejowanego (ang. *enable-mode*), do stworzenia połączenia wystarczy użyć funkcji `.enable()`. Aby wywołać polecenia w terminalu routera, należy skorzystać z funkcji `.send_command()`. Wynik zapisujemy w zmiennej `output`:

```
from netmiko import ConnectHandler
connection = ConnectHandler(**R1)
connection.enable()
output = connection.send_command("show ip int b")
print output
```

Wynik działania skryptu:

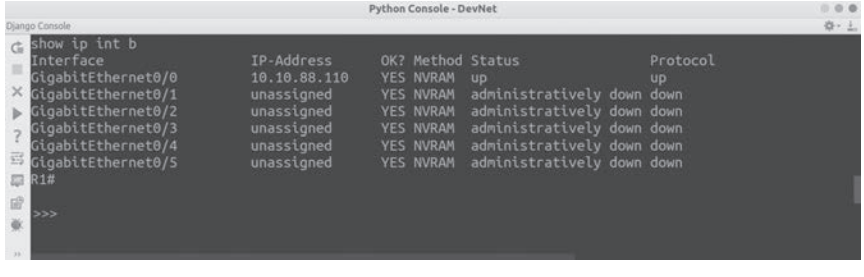


Domyślnie Netmiko zwraca sam wynik (bez znaku zachęty i wykonywanej komendy), który później może zostać obrobiony przez wyrażenia regularne (będzie o tym mowa w następnym rozdziale).

Jeżeli chcemy wyłączyć tę funkcjonalność i otrzymać w odpowiedzi znak zachęty i wykonywane komendy, należy w funkcji `.send_command()` określić wartość odpowiednich flag. Zmieniamy wartości `strip_command` i `strip_prompt` na `False` (domyślnie mają wartość `True`):

```
output = connection.send_command("show ip int b", strip_command=False,
strip_prompt=False)
```

Wynik działania:



Konfiguracja urządzeń za pomocą Netmiko

Netmiko może zostać wykorzystany do zdalnej konfiguracji urządzenia sieciowego (poprzez SSH). Konfigurację urządzenia (w formacie listy) uzyskujemy poprzez metodę `.config`. Uzyskana lista może być dołączona do skryptu Pythona lub odczytana z pliku, a następnie przekształcona w listę za pomocą metody `readlines()`:

```
from netmiko import ConnectHandler
SW2 = {
    'device_type': 'cisco_ios',
    'ip': '10.10.88.112',
    'username': 'admin',
    'password': 'access123',
    'secret': 'access123',
}
core_sw_config = ["int range gig0/1 - 2", "switchport trunk encapsulation dot1q",
"switchport mode trunk", "switchport trunk allowed vlan 1,2"]
print "########## Connecting to Device {0} ############".format(SW2['ip'])
net_connect = ConnectHandler(**SW2)
net_connect.enable()
print "***** Sending Configuration to Device *****"
net_connect.send_config_set(core_sw_config)
```

W skrypcie wykonaliśmy te same czynności co w poprzednim zadaniu — poza jedną. Wykorzystaliśmy funkcję `send_config_set`. Jako argument pobiera ona konfigurację w postaci listy, przełącza urządzenie w tryb konfiguracyjny i ładuje nowe dane.

W naszym przykładzie zmodyfikowaliśmy interfejsy `gig0/1` i `gig0/2` poprzez załadowanie nowej konfiguracji portów `trunk`. Możesz sprawdzić, czy konfiguracja przebiegła pomyślnie, poprzez wywołanie komendy `show run` na urządzeniu. Powinieneś otrzymać rezultat podobny do poniższego:

```
interface GigabitEthernet0/1
 switchport trunk allowed vlan 1,2
 switchport trunk encapsulation dot1q
 switchport mode trunk
 media-type rj45
 negotiation auto
!
interface GigabitEthernet0/2
 switchport trunk allowed vlan 1,2
 switchport trunk encapsulation dot1q
 switchport mode trunk
 media-type rj45
 negotiation auto
!
```

Obsługa wyjątków w Netmiko

Kiedy tworzymy skrypt w Pythonie, zakładamy, że urządzenie jest sprawne i włączone oraz wprowadzone zostały poprawne dane uwierzytelniające. Często jednak tak nie jest. Czasami występują problemy z łącznością między Pythonem i zdalnym urządzeniem, a czasami użytkownik wprowadził nieprawidłowe dane uwierzytelniające. Zazwyczaj w takim wypadku Python wyrzuci wyjątek i działanie skryptu zostanie przerwane. Nie jest to prawidłowe podejście.

Moduł Netmiko o nazwie `netmiko.ssh_exception` dostarcza odpowiednie klasy wyjątków, dzięki którym możemy obsłużyć powyższe przypadki. Pierwszą z nich jest klasa `Authentication ↪Exception`, która obsługuje wyjątki związane z błędami uwierzytelnienia na zdalnym urządzeniu. Następna klasa, `NetMikoTimeoutException`, wychwytuje wyjątki związane z błędami przekroczenia czasu oczekiwania (ang. *timeout*) i innymi problemami komunikacyjnymi. Teraz musimy jedynie wychwycić wyjątki (związane z czasem oczekwania i błędami uwierzytelnienia) rzucane przez metodę `ConnectHandler()`. Dokonamy tego za pomocą instrukcji `try-except`:

```
from netmiko import ConnectHandler
from netmiko.ssh_exception import AuthenticationException, NetMikoTimeoutException
device = {
    'device_type': 'cisco_ios',
    'ip': '10.10.88.112',
    'username': 'admin',
    'password': 'access123',
    'secret': 'access123',
}
print "########## Connecting to Device {0} ############".format(device['ip'])
try:
    net_connect = ConnectHandler(**device)
    net_connect.enable()
    print "***** show ip configuration of Device *****"
    output = net_connect.send_command("show ip int b")
    print output
    net_connect.disconnect()
except NetMikoTimeoutException:
    print "=========== SOMETHING WRONG HAPPEN WITH
    ↪{0} ============".format(device['ip'])
except AuthenticationException:
    print "========= Authentication Failed with
    ↪{0} ============".format(device['ip'])
except Exception as unknown_error:
    print "============ SOMETHING UNKNOWN HAPPEN WITH {0} ============"
```

Automatyczne wykrywanie urządzenia

Netmiko wprowadza mechanizmy pozwalające wykrywać typ urządzenia. Do tego celu wykorzystywana jest baza wartości **OID** (ang. *object identifier*) protokołu SNMP. Na zdalnym urządzeniu wykonywanych jest kilka komend `show`, aby porównać zwracane wartości. Ma to na celu określenie rodzaju urządzenia i jego systemu operacyjnego. Dzięki temu Netmiko może załadować odpowiedni sterownik w klasie `ConnectHandler()`:

```
#!/usr/local/bin/python
__author__ = "Bassim Aly"
__EMAIL__ = "basim.alyy@gmail.com"
from netmiko import SSHDetect, Netmiko
device = {
    'device_type': 'autodetect',
    'host': '10.10.88.110',
    'username': 'admin',
    'password': "access123",
}
```

```
detect_device = SSHDetect(**device)
device_type = detect_device.autodetect()
print(device_type)
print(detect_device.potential_matches)
device['device_type'] = device_type
connection = Netmiko(**device)
```

W powyższym skrypcie:

- Wartość `autodetect` parametru `device_type` mówi Netmiko, że musi poczekać, dopóki nie zostanie rozpoznany rodzaj urządzenia.
- Następnie, używając klasy `SSHDetect()`, moduł Netmiko przystępuje do wykrywania. Za pośrednictwem protokołu SSH klasa łączy się z urządzeniem i wykonuje kilka komend w celu rozpoznania systemu operacyjnego. Otrzymany wynik będzie miał strukturę słownika i najlepsze dopasowanie zostanie przypisane do zmiennej `device_type` w wyniku użycia funkcji `autodetect()`.
- Używając `potential_matches`, możemy uzyskać wszystkie wyniki, które były brane pod uwagę w momencie dopasowywania najlepszego wyniku.
- W tym momencie możemy zaktualizować parametr `device` (w naszej strukturze słownikowej), przypisując zmienną `device_type`.

Wykorzystanie protokołu Telnet za pomocą Pythona

Telnet jest jednym z najstarszych protokółów używanych w stosie TCP/IP. Stosowany jest głównie do wymiany danych podczas połączenia typu klient – serwer. Do obsługi żądań klientów serwer Telnetu wykorzystuje port 23.

W tym punkcie skrypt Pythona posłuży nam do stworzenia klienta Telnetu. Naszymi serwerami będą routery i switche. Python oferuje wsparcie dla Telnetu poprzez **bibliotekę telnetlib**. Nie musimy jej dodatkowo instalować — jest wbudowana.

Po stworzeniu obiektu klienta (za pomocą klasy `Telnet()` z biblioteki telnetlib) mamy do dyspozycji dwie ważne funkcje: `read_until()` (odczyt z urządzenia sieciowego) i `write()` (zapis do urządzenia sieciowego).

Należy wziąć pod uwagę to, że funkcja `read_until()` czyści dane w buforze, więc nie będą one dostępne podczas następnych odczytów. Jeżeli odczytujesz ważne dane, które chcesz zapamiętać, musisz je zapisać w zmiennej. Jeżeli tego nie zrobisz i dane zostaną przetworzone w dalszej części skryptu, nie będziesz mógł wrócić do oryginalnych danych.

Dane przesyłane za pośrednictwem protokołu Telnet są jawne, więc wszelkie przesyłane hasła i loginy są łatwym celem do przechwycenia (np. w wyniku ataku man-in-the-middle). Protokół Telnet w dalszym ciągu wykorzystywany jest przez producentów. Często integrowany jest z VPN-em i protokołami radius/tacacs, aby zapewnić w miarę lekki i bezpieczny dostęp do urządzenia.

Aby zrozumieć cały skrypt, prześledź następujące kroki:

1. Na początku zaimportujemy moduł telnetlib i zdefiniujemy zmienne przechowujące nazwę użytkownika (`username`) i hasła (`password`, `enable_password`):

```
import telnetlib
username = "admin"
password = "access123"
enable_password = "access123"
```

2. Zdefiniujemy zmienną tworzącą połączenie ze zdalnym hostem. Warto zauważyć, że podczas tworzenia połączenia nie podajemy nazwy użytkownika i hasła, lecz jedynie adres IP zdalnego hosta:

```
cnx = telnetlib.Telnet(host="10.10.88.110") # połączenie telnet do bramki
```

3. W tym momencie podajemy nazwę użytkownika. Poprzez połączenie Telnet odczytujemy odpowiedź serwera i w ciągu zwracanych danych szukamy słowa `Username`. Gdy tylko je otrzymamy, możemy przesłać na serwer nasz login. Podobnie postępujemy w przypadku wprowadzania hasła użytkownika i hasła dla trybu uprzywilejowanego (ang. *enable password*):

```
cnx.read_until("Username:")
cnx.write(username + "\n")
cnx.read_until("Password:")
cnx.write(password + "\n")
cnx.read_until(">")
cnx.write("en" + "\n")
cnx.read_until("Password:")
cnx.write(enable_password + "\n")
```

Ważne jest, aby czekać na dokładnie takie same słowa (w funkcji `read_until()`), jakie pojawiają się w konsoli urządzenia. W przeciwnym razie będziemy czekać na wynik do momentu, w którym skrypt zwróci błąd z powodu przekroczenia czasu oczekiwania na połączenie.

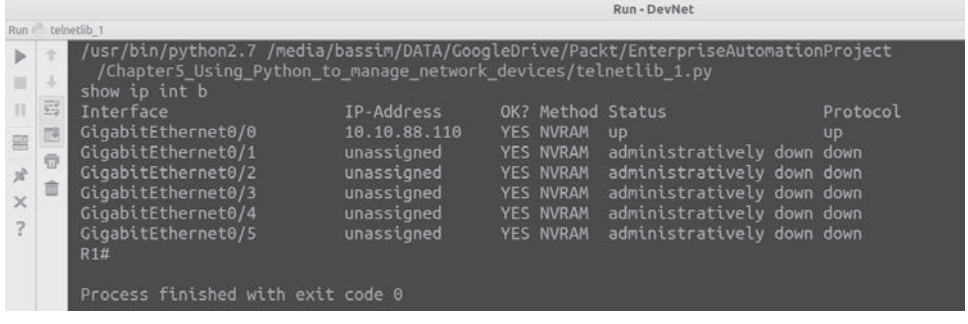
4. Na końcu wykonamy komendę `show ip interface brief`. Następnie odczytamy ze zdalnego urządzenia wszystkie dane aż do znaku zachęty `#`. Dzięki temu otrzymamy dane konfiguracyjne interfejsu routera:

```
cnx.read_until("#")
cnx.write("show ip int b" + "\n")
output = cnx.read_until("#")
print output
```

Pełny kod skryptu:

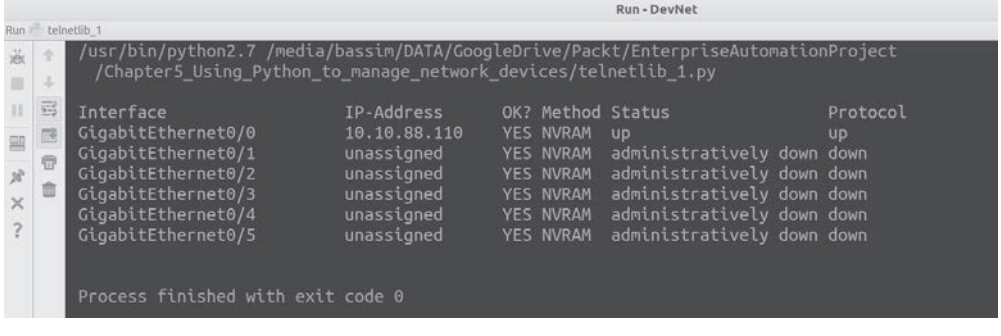
```
__author__ = "Bassim Aly"
__EMAIL__ = "basim.alyy@gmail.com"

import telnetlib
username = "admin"
password = "access123"
enable_password = "access123"
cnx = telnetlib.Telnet(host="10.10.88.110")
cnx.read_until("Username:")
cnx.write(username + "\n")
cnx.read_until("Password:")
cnx.write(password + "\n")
cnx.read_until(">")
cnx.write("en" + "\n")
cnx.read_until("Password:")
cnx.write(enable_password + "\n")
cnx.read_until("#")
cnx.write("show ip int b" + "\n")
output = cnx.read_until("#")
print output
```

Wynik działania skryptu:



Zauważmy, że wynik działania zawiera wykonywaną komendę `show ip int b` i znak zachęty routera `"R1#"`. Aby się ich pozbyć, powinniśmy użyć wbudowanej funkcji do operacji na stringach, np. `replace()`:

```
cleaned_output = output.replace("show ip int b", "").replace("R1#", "")
print cleaned_output
```



W naszym kodzie dane uwierzytelniające (login i hasła) wprowadziliśmy w postaci jawnej. Nie jest to poprawna metoda z punktu widzenia bezpieczeństwa. W dalszej części rozdziału dowiemy się, jak ukryć takie dane, i stworzymy taki mechanizm, aby uwierzytelnienie następowało tylko w czasie działania skryptu.

Ponadto jeżeli chcesz wywołać komendę typu `show running config`, która spowoduje wyświetlenie kilku stron ekranu, warto na początku połączenia wykonać komendę `terminal length 0`, która wyłączy funkcję przewijania ekranu.

Zmiana konfiguracji poprzez telnetlib

W poprzednim punkcie rozdziału dowiedzieliśmy się, jak można wykonać proste operacje typu `show ip int brief` za pomocą **modułu telnetlib**. Teraz musimy wykorzystać tę wiedzę, aby wysłać konfigurację VLAN do czterech switchy w naszej sieci. Za pomocą funkcji `range()` tworzymy listę identyfikatorów VLAN. Następnie dany identyfikator VLAN ID wysyłamy do konkretnego switcha (w czasie iteracji). Adresy IP switchy zdefiniowaliśmy jako listę, a lista przetwarzana jest w pętli dzięki instrukcji `for`. Ponadto skorzystaliśmy z modułu `getpass`, aby nie pokazywać hasła w konsoli — podajemy je tylko w czasie działania skryptu:

```
#!/usr/bin/python
import telnetlib
import getpass
import time
switch_ips = ["10.10.88.111", "10.10.88.112", "10.10.88.113", "10.10.88.114"]
username = raw_input("Please Enter your username:")
password = getpass.getpass("Please Enter your Password:")
enable_password = getpass.getpass("Please Enter your Enable Password:")
for sw_ip in switch_ips:
    print "\n#################### Working on Device " + sw_ip + "####################"
    connection = telnetlib.Telnet(host=sw_ip.strip())
    connection.read_until("Username:")
    connection.write(username + "\n")
    connection.read_until("Password:")
    connection.write(password + "\n")
    connection.read_until(">")
    connection.write("enable" + "\n")
    connection.read_until("Password:")
    connection.write(enable_password + "\n")
    connection.read_until("#")
    connection.write("config terminal" + "\n") #  tryb konfiguracji
    vlans = range(300,400)
    for vlan_id in vlans:
        print "\n********* Adding VLAN " + str(vlan_id) + "**********"
        connection.read_until("#")
        connection.write("vlan " + str(vlan_id) + "\n")
        time.sleep(1)
        connection.write("exit" + "\n")
        connection.read_until("#")
connection.close()
```

W najbardziej zewnętrznej pętli `for` przechodzimy przez wszystkie switche. W środku pętli dla każdego switcha generujemy identyfikatory VLAN z zakresu od 300 do 400 i wysyłamy je do konkretnego urządzenia.

Wynik działania skryptu:

```
bassim@me-inside:~$ /usr/bin/python2.7 /media/bassim/DATA/GoogleDrive/Packt/EnterpriseAutomationProje
ct/Chapter5_Using_Python_to_manage_network_devices/telnetlib_push_vlans.py
Please Enter your username:admin
Please Enter your Password:
Please Enter your Enable Password:

################### Working on Device 10.10.88.111 ###################

********* Adding VLAN 300**********

********* Adding VLAN 301**********

********* Adding VLAN 302**********

********* Adding VLAN 303**********

********* Adding VLAN 304**********

********* Adding VLAN 305**********

********* Adding VLAN 306**********

********* Adding VLAN 307**********

********* Adding VLAN 308**********
```

Wynik działania skryptu możesz także sprawdzić, posługując się konsolą switcha:

```
SW1#show vlan

VLAN Name                             Status    Ports
---- -------------------------------- --------- -------------------------------
1    default                          active    Gi0/1, Gi0/2, Gi0/3, Gi1/0
                                                Gi1/1, Gi1/2, Gi1/3, Gi2/0
                                                Gi2/1, Gi2/2, Gi2/3, Gi3/0
                                                Gi3/1, Gi3/2, Gi3/3
300  VLAN0300                         active
301  VLAN0301                         active
302  VLAN0302                         active
303  VLAN0303                         active
304  VLAN0304                         active
305  VLAN0305                         active
306  VLAN0306                         active
307  VLAN0307                         active
308  VLAN0308                         active
309  VLAN0309                         active
310  VLAN0310                         active
311  VLAN0311                         active
312  VLAN0312                         active
313  VLAN0313                         active
314  VLAN0314                         active
315  VLAN0315                         active
316  VLAN0316                         active
317  VLAN0317                         active
```

Praca z sieciami z wykorzystaniem biblioteki netaddr

Inżynierowie zajmujący się sieciami teleinformatycznymi bardzo często w swojej pracy posługują się adresami IP. Twórcy Pythona stworzyli wspaniałą bibliotekę, o nazwie **netaddr**, która bardzo ułatwia tę pracę. Na przykład aby w aplikacji w prosty sposób uzyskać adres sieci i adres broadcast dla adresu *129.183.1.55/21*, można posłużyć się wbudowanymi w moduł netaddr funkcjami `network` i `broadcast`:

```
net.network
129.183.0.
net.broadcast
129.183.0.0
```

Generalnie moduł netaddr udostępnia wsparcie dla:

Adresów warstwy 3:

- adresy IPv4 i IPv6, podsieci, maski, prefiksy;
- iteracja, wycinanie, sortowanie, agregacja i klasyfikacja adresów IP;
- wsparcie dla różnych formatów zapisu (CIDR, dowolne adresy i wzorce, nmap);
- operacje na zbiorach (unie, intersekcje i wiele innych) adresów IP i podsieci;
- parsowanie różnorodnych formatów i notacji;
- uzyskiwanie informacji o adresie IP z **IANA** (ang. *Internet Assigned Numbers Authority*);
- uzyskiwanie odwrotnych translacji adresów **DNS** (ang. *DNS reverse lookups*);
- supernetting i subnetting (grupowanie sieci).

Adresów warstwy 2:

- manipulacja adresami MAC i identyfikatorami EUI-64;
- uzyskiwanie informacji IEEE (OUI, IAB);
- generowanie adresów IPv6.

Instalowanie modułu netaddr

Moduł netaddr można zainstalować za pomocą narzędzia **pip:**

```
pip install netaddr
```

Do weryfikacji poprawności instalacji można skorzystać z PyCharma lub konsoli Pythona i spróbować zaimportować moduł. Jeżeli podczas importu nie będzie żadnych błędów, oznacza to, że moduł został prawidłowo zainstalowany:

```
$ python
>>> import netaddr
```

Metody modułu netaddr

Moduł netaddr ma dwie ważne metody do pracy z adresami IP. Pierwsza z nich, `IPAddress()`, służy do definiowania klasowego adresu IP z domyślną maską podsieci. Druga metoda, `IPNetwork()`, służy do definiowania bezklasowego adresu IP z notacją CIDR.

Obie metody przekształcają adres IP w postaci stringa na obiekty. Mamy do dyspozycji wiele operacji, które możemy wykonać na takich obiektach. Możemy na przykład sprawdzić, czy adres IP jest unicastowy, multicastowy, typu loopback, prywatny czy publiczny oraz czy jest prawidłowy. Wynikiem takich operacji jest wartość logiczna `True` lub `False`, która może później zostać wykorzystana w operacjach typu `if`.

Moduł wspiera również operatory porównania dla adresów IP: `==`, `<`, `>`. Ponadto może generować podsieci, a także możliwe jest przeglądanie listy nadsieci w celu uzyskania informacji, czy dany adres IP lub podsieć do niej należy. Moduł netaddr może ponadto wygenerować pełną listę poprawnych hostów (adres IP sieci i broadcast):

```
#!/usr/bin/python
__author__ = "Bassim Aly"
__EMAIL__ = "basim.alyy@gmail.com"
from netaddr import IPNetwork,IPAddress
def check_ip_address(ipaddr):
    ip_attributes = []
    ipaddress = IPAddress(ipaddr)

    if ipaddress.is_private():
        ip_attributes.append("IP Address is Private")
    else:
        ip_attributes.append("IP Address is public")
    if ipaddress.is_unicast():
        ip_attributes.append("IP Address is unicast")
    elif ipaddress.is_multicast():
        ip_attributes.append("IP Address is multicast")
    if ipaddress.is_loopback():
        ip_attributes.append("IP Address is loopback")

    return "\n".join(ip_attributes)

def operate_on_ip_network(ipnet):
    net_attributes = []
    net = IPNetwork(ipnet)
    net_attributes.append("Network IP Address is " + str(net.network) + "and
    ↪Netowrk Mask is " + str(net.netmask))

    net_attributes.append("The Broadcast is " + str(net.broadcast))
    net_attributes.append("IP Version is " + str(net.version))
    net_attributes.append("Information known about this network is " + str
    ↪(net.info))
    net_attributes.append("The IPv6 representation is " + str(net.ipv6()))
    net_attributes.append("The Network size is " + str(net.size))
    net_attributes.append("Generating a list of ip addresses inside the subnet")
```

```
        for ip in net:
            net_attributes.append("\t" + str(ip))
        return "\n".join(net_attributes)

ipaddr = raw_input("Please Enter the IP Address: ")
print check_ip_address(ipaddr)

ipnet = raw_input("Please Enter the IP Network: ")
print operate_on_ip_network(ipnet)
```

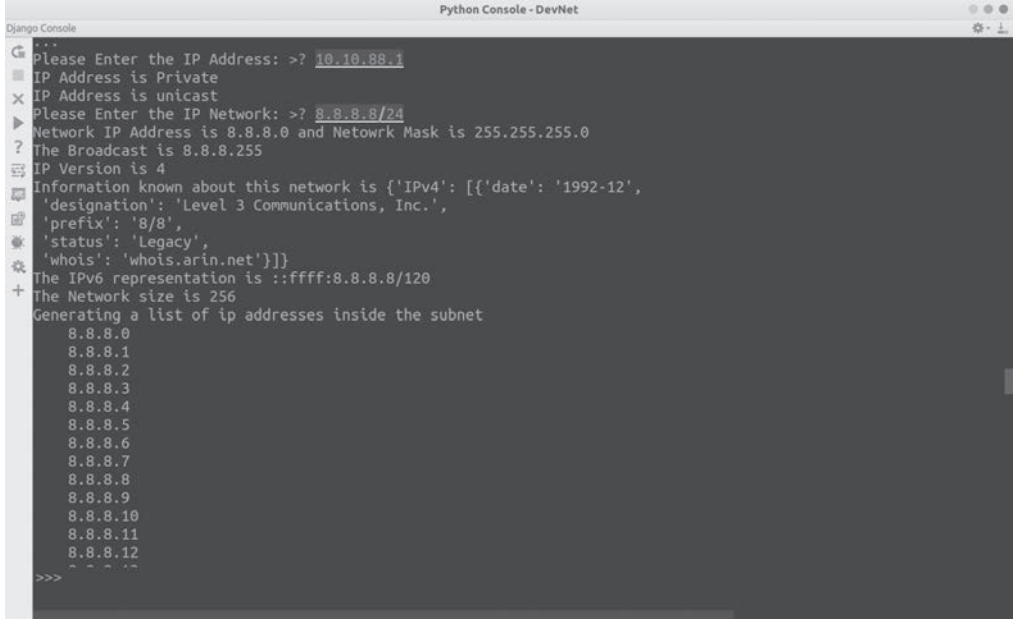
W powyższym skrypcie za pomocą funkcji `raw_input()` prosimy użytkownika o podanie adresu IP i IP sieci. Następnie wywołujemy dwie metody, `check_ip_address()` i `operate_on_ip_network()`, i przekazujemy do nich wartości podane przez użytkownika. Pierwsza funkcja, `check_ip_address()`, sprawdza, czy podany adres IP jest unicastowy, multicastowy, prywatny lub czy jest loopbackiem. Wynik sprawdzenia zwracany jest użytkownikowi.

Druga funkcja, `operate_on_ip_network()`, pobiera IP sieci i zwraca ID sieci, maskę sieci, broadcast, wersję, wszystkie znane informacje na temat sieci, reprezentację IPv6 oraz generuje wszystkie adresy IP z podsieci.

Należy zapamiętać, że net.info zwraca użyteczne informacje jedynie w przypadku, gdy adres IP jest publiczny.

Musimy także zaimportować funkcje `IPNetwork` i `IPAddress` z modułu netaddr, zanim użyjemy ich w programie.

Wynik działania skryptu:



Przykładowe przypadki użycia

Z czasem, gdy nasza sieć się rozrasta i w naszej sieci pojawia się wiele nowych urządzeń od różnych producentów, potrzebujemy stworzyć skrypt, który zautomatyzuje pewne czynności. W następnych rozdziałach przeanalizujemy trzy przykłady, które mogą służyć do zbierania różnych informacji na temat sieci, skrócenia czasu potrzebnego do usunięcia problemu lub odzyskania ostatniej dobrej konfiguracji sieci. Dzięki temu specjaliści ds. sieci będą mogli bardziej skupić się na swojej pracy, a nasza sieć szybciej odzyska sprawność po awarii.

Konfiguracja kopii zapasowej urządzenia

Konfiguracja kopii zapasowej urządzenia jest jednym z najważniejszych zadań dla specjalisty zajmującego się sieciami. W tym przykładzie napiszemy skrypt, który będzie tworzył kopię zapasową (ang. *backup*) różnych urządzeń. Do tego zadania wykorzystamy bibliotekę Netmiko.

W nazwie pliku (z kopią zapasową) powinien być zawarty adres backupowanego urządzenia — dla łatwiejszego odszukania w przyszłości. Na przykład plik kopii zapasowej urządzenia SW1 powinien nazywać się `dev_10.10.88.111_.cfg`.

Tworzymy skrypt

Na początku zdefiniujemy w pliku dostęp do switchy. Do wykonania kopii zapasowej konfiguracji każdego z urządzeń potrzebne będą dane dostępowe. Porozdzielamy je przecinkami, aby później w łatwy sposób sparsować dane za pomocą funkcji `split()` i przekazać je do funkcji `ConnectHandler`. Taki format zapisu ułatwi nam również zarówno eksport, jak i import danych z programu Microsoft Excel lub dowolnej bazy danych.

Oto struktura pliku:

```
<device_ipaddress>,<username>,<password>,<enable_password>,<vendor>
```

```
1  10.10.88.110,admin,access123,access123,cisco
2  10.10.88.111,admin,access123,access123,Cisco
3  10.10.88.112,admin,access123,access123,Cisco
4  10.10.88.113,admin,access123,access123,Cisco
5  10.10.88.114,admin,access123,access123,Cisco
```

Do importu pliku (wewnątrz skryptu) wykorzystamy instrukcję `with open`. Do wczytania każdej linii pliku użyjemy funkcji `readlines()`. Do parsowania wczytanej linii pliku zastosujemy funkcję `split()`. Dzięki temu uzyskamy dostęp do każdego pola rozdzielonego przecinkiem:

```
from netmiko import ConnectHandler
from datetime import datetime
```

```
with open("/media/bassim/DATA/GoogleDrive/Packt/EnterpriseAutomationProject/
↪Chapter5_Using_Python_to_manage_network_devices/UC1_devices.txt") as devices_file:
    devices = devices_file.readlines()

for line in devices:
    line = line.strip("\n")
    ipaddr = line.split(",")[0]
    username = line.split(",")[1]
    password = line.split(",")[2]
    enable_password = line.split(",")[3]

    vendor = line.split(",")[4]

    if vendor.lower() == "cisco":
        device_type = "cisco_ios"
        backup_command = "show running-config"

    elif vendor.lower() == "juniper":
        device_type = "juniper"
        backup_command = "show configuration | display set"
```

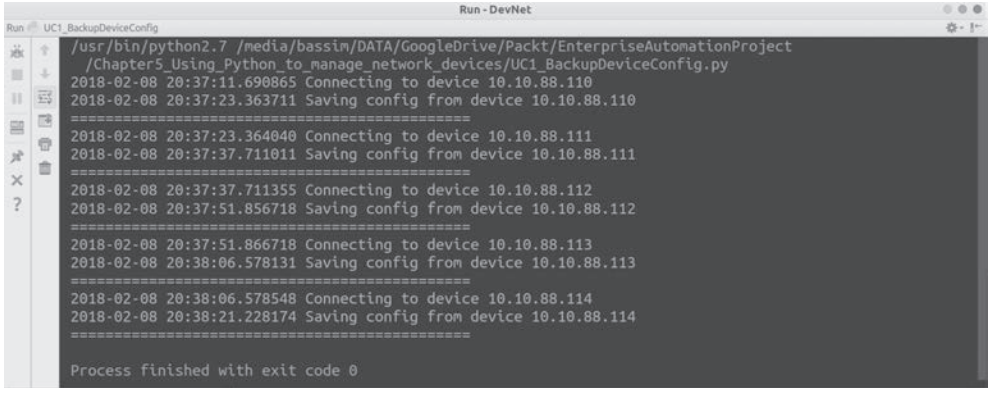
Chcemy stworzyć skrypt uniwersalny (obsługujący wielu różnych producentów sprzętu), dlatego musimy sprawdzić parametry analizowanego switcha (za pomocą komendy `if`) i przypisać właściwe wartości do zmiennych `device_type` i `backup_command`.

Na tym etapie jesteśmy gotowi, aby nawiązać z urządzeniem połączenie SSH i wykonać polecenie zdefiniowane w zmiennej `backup_command`. Komendę wysyłamy, korzystając z metody `.send_command()` dostępnej za pośrednictwem modułu Netmiko:

```
print str(datetime.now()) + " Connecting to device {}" .format(ipaddr)

net_connect = ConnectHandler(device_type=device_type,
                             ip=ipaddr,
                             username=username,
                             password=password,
                             secret=enable_password)
net_connect.enable()
running_config = net_connect.send_command(backup_command)

print str(datetime.now()) + " Saving config from device {}" .format(ipaddr)

f = open( "dev_" + ipaddr + "_.cfg", "w")
f.write(running_config)
f.close()
print "================================================"
```

Powyższy fragment skryptu otwiera plik o nazwie, która zawarta jest w zmiennej `ipaddr`, i ustawia jego atrybut do zapisu.

Wynik działania skryptu:



Zwróć uwagę, że pliki kopii zapasowych zostały utworzone w katalogu domowym projektu i każdy plik zawiera w nazwie adres IP urządzenia:

```
bassim@me-inside:portal$ ls dev*
dev_10.10.88.110_.cfg  dev_10.10.88.112_.cfg  dev_10.10.88.114_.cfg
dev_10.10.88.111_.cfg  dev_10.10.88.113_.cfg
bassim@me-inside:portal$
bassim@me-inside:portal$ more dev_10.10.88.110_.cfg
Building configuration...


Current configuration : 3994 bytes
!
version 15.6
service timestamps debug datetime msec
service timestamps log datetime msec
no service password-encryption
!
hostname R1
!
boot-start-marker
boot-end-marker
!
!
enable password access123
!
aaa new-model
!
!
aaa authentication login default local
```

Uruchomienie skryptu możesz zaplanować na określoną godzinę. Możesz to zrobić, tworząc proste zadanie `cron` w systemie Linux lub wykorzystać harmonogram zadań w Windowsie. Przykładowo możesz zaplanować uruchamianie skryptu codziennie o północy i przechowywać kopie zapasowe w katalogu, którego nazwa może określać dzień backupu.

Utworzenie własnego terminala dostępowego

Jako twórca oprogramowania masz możliwość napisania takiego kodu, który spełni wszystkie Twoje oczekiwania. W tym przykładzie stworzymy własny terminal, który uzyska dostęp do routera za pośrednictwem modułu telnetlib. Kombinacja znaków wpisywana w terminalu zostanie przetłumaczona na komendy zrozumiałe dla urządzenia. Wynik ich działania będzie mógł zostać wyświetlony w terminalu lub zapisany do pliku:

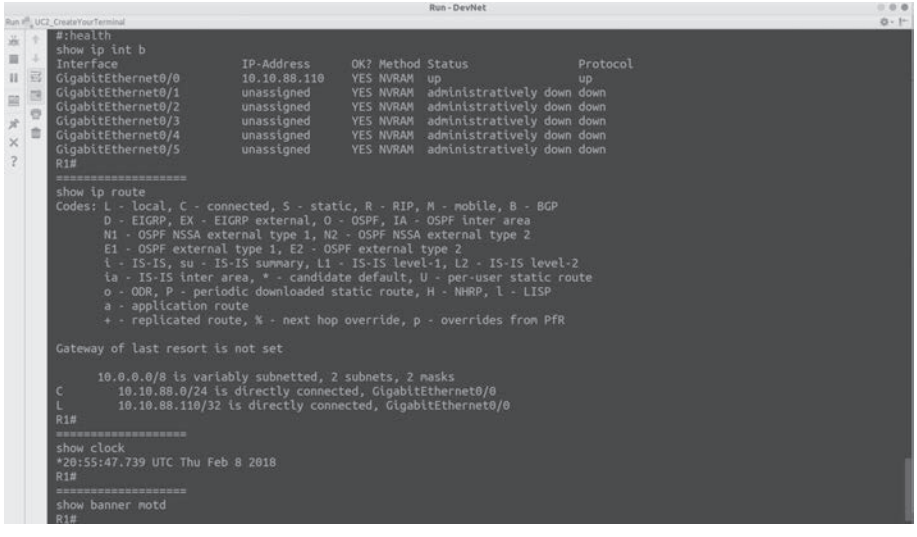
```
#!/usr/bin/python
__author__ = "Bassim Aly"
__EMAIL__ = "basim.alyy@gmail.com"

import telnetlib

connection = telnetlib.Telnet(host="10.10.88.110")
connection.read_until("Username:")
connection.write("admin" + "\n")
connection.read_until("Password:")
connection.write("access123" + "\n")
connection.read_until(">")
connection.write("en" + "\n")
connection.read_until("Password:")
connection.write("access123" + "\n")
connection.read_until("#")
connection.write("terminal length 0" + "\n")
connection.read_until("#")
while True:
    command = raw_input("#:")
    if "health" in command.lower():
        commands = ["show ip int b",
                    "show ip route",
                    "show clock",
                    "show banner motd"
                   ]

    elif "discover" in command.lower():
        commands = ["show arp",
                    "show version | i uptime",
                    "show inventory",
                    ]
    else:
        commands = [command]
    for cmd in commands:
        connection.write(cmd + "\n")
        output = connection.read_until("#")
        print output
        print "==================="
```

Na początku tworzymy połączenie telnetowe z routerem. Podajemy dane dostępowe i wchodzimy w tryb uprzywilejowany. Następnie w nieskończonej pętli (`while True`) czekamy na komendy od użytkownika (funkcja `raw_input()`). Jeśli użytkownik wprowadzi komendę, to jest ona bezpośrednio przesyłana do urządzenia sieciowego.

Jednak w przypadku, gdy użytkownik wprowadzi słowa `health` lub `discover`, zostanie wysłana zaprogramowana przez nas seria komend. Takie rozwiązanie może być niezwykle użyteczne w razie problemów. Znacznie przyśpiesza czas reakcji. Wyobraźmy sobie, że mamy problem z OSPF pomiędzy dwoma sąsiednimi routerami. Wówczas w stworzonym przez nas terminalu będziemy mogli wydać polecenie typu `tshoot_ospf` (o ile wcześniej je oprogramujemy w naszym skrypcie), które wykona serię poleceń typu sprawdzenie statusu sąsiadów, sprawdzenie MTU interfejsów, sprawdzenie rozgłaszanych sieci i wiele innych. Pozwoli to sprawnie odnaleźć problem.

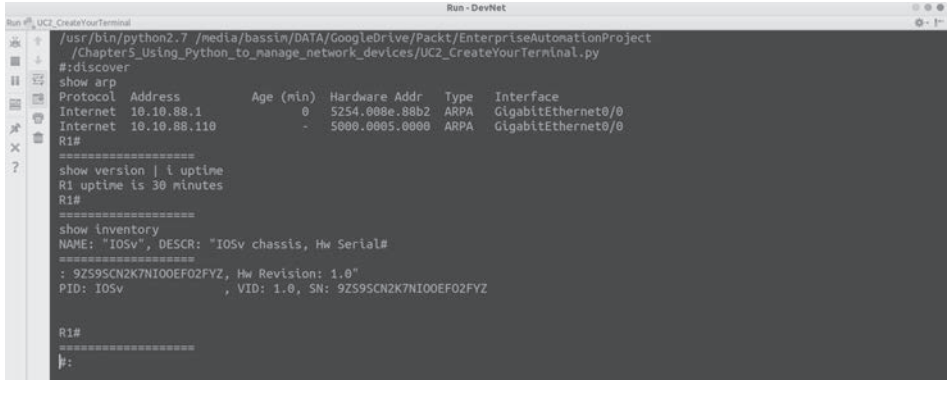
Wynik działania skryptu:

Na początku wypróbuj działanie skryptu poprzez wpisanie w terminalu komendy `health`:



Jak widzisz, wykonało się kilka poleceń zrozumiałych dla urządzenia.

Następnie spróbuj wpisać komendę `discover`:



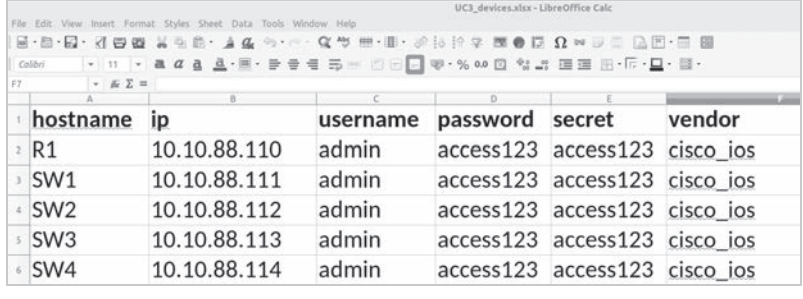
Ponownie zobaczymy w wyniku serię wykonanych poleceń — tym razem zaprogramowanych dla komendy `discover`. W następnym rozdziale nauczymy się parsować wynik działania skryptu w celu uzyskania użytecznych danych.

Odczyt danych z arkusza Excela

Arkusz Excela często wykorzystywany jest przez specjalistów IT do przechowywania użytecznych informacji na temat infrastruktury teleinformatycznej (adresy IP, rodzaj urządzenia, dane dostępowe). Python wspiera obsługę arkuszy Excela. Dzięki temu możemy takie dane wykorzystać w naszym skrypcie.

W naszym przypadku użyjemy **modułu Excel Read** (**xlrd**) do odczytu danych z arkusza `UC3_devices.xlsx`. Zawiera on nazwę hosta, IP, nazwę użytkownika, hasło, hasło dla trybu uprzywilejowanego i rodzaj sprzętu używanego w infrastrukturze. Uzyskane dane wykorzystamy w module netmiko.

Oto zrzut arkusza Excela:



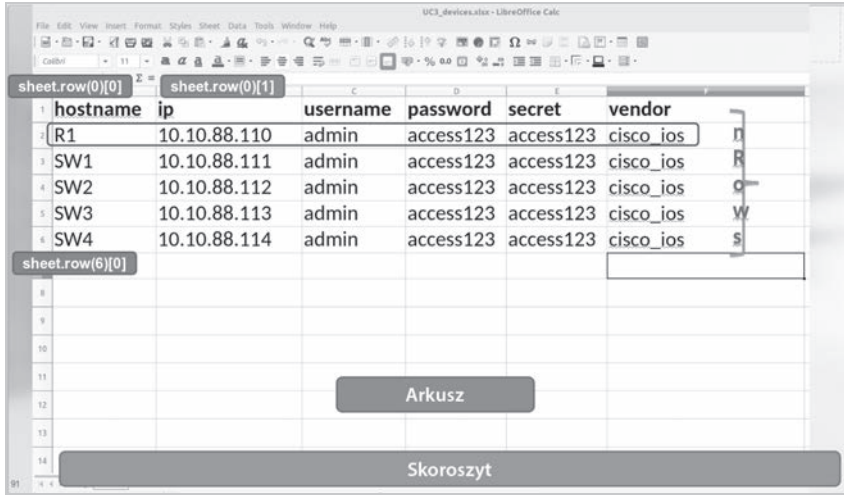
	hostname	ip	username	password	secret	vendor
2	R1	10.10.88.110	admin	access123	access123	cisco_ios
3	SW1	10.10.88.111	admin	access123	access123	cisco_ios
4	SW2	10.10.88.112	admin	access123	access123	cisco_ios
5	SW3	10.10.88.113	admin	access123	access123	cisco_ios
6	SW4	10.10.88.114	admin	access123	access123	cisco_ios

Na początku musimy zainstalować moduł xlrd, z którego będziemy korzystać przy próbie odczytu danych z arkusza Excela. Do instalacji modułu używamy narzędzia `pip`:

```
pip install xlrd
```

Moduł XLRD wczyta arkusz Excela i przekonwertuje rzędy i kolumny do postaci matrycy. Na przykład jeżeli chcemy odczytać wartość pierwszego elementu z lewej, to musimy odwołać się do pola `row[0][0]`. Dostęp do następnego elementu leżącego po prawej uzyskamy przez odwołanie do pola `row[0][1]`. I tak dalej.

W czasie odczytu arkusza moduł xlrd używa dwóch specjalnych liczników: `nrows` (ang. *number of rows*) i `ncols` (ang. *number of columns*). Wartość licznika `nrows` zwiększana jest o 1 w przypadku odczytu kolejnego rzędu, a wartość licznika `ncols` zwiększana jest o 1 w przypadku odczytu kolejnej kolumny. Dzięki tym dwóm parametrom znamy rozmiar matrycy.



Ścieżkę do pliku z arkuszem podajemy w funkcji `open_workbook()` (dostępna w **module xlrd**). Następnie uzyskujemy dostęp do arkusza, używając funkcji `sheet_by_index()` lub funkcji `sheet_by_name()`. W naszym przypadku dane przechowywane są w pierwszym arkuszu (`index=0`), a ścieżką dostępu do pliku jest nazwa rozdziału. Następnie będziemy przechodzić przez poszczególne wiersze, a dostęp do konkretnego wiersza uzyskamy, stosując funkcję `row()`. Zwracane informacje są przedstawiane w postaci listy, więc możemy uzyskać dostęp do każdej wartości poprzez jej indeks.

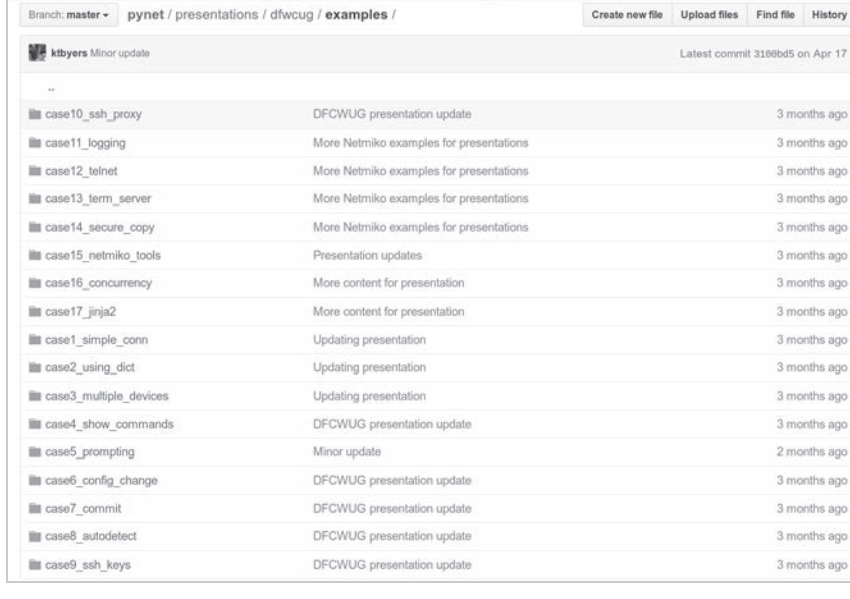
Oto skrypt:

```
__author__ = "Bassim Aly"
__EMAIL__ = "basim.alyy@gmail.com"

from netmiko import ConnectHandler
from netmiko.ssh_exception import AuthenticationException, NetMikoTimeoutException
import xlrd

from pprint import pprint

workbook = xlrd.open_workbook(r"/media/bassim/DATA/GoogleDrive/Packt/
↪EnterpriseAutomationProject/Chapter4_Using_Python_to_manage_network_devices/
↪UC3_devices.xlsx")

sheet = workbook.sheet_by_index(0)

for index in range(1, sheet.nrows):
    hostname = sheet.row(index)[0].value
    ipaddr = sheet.row(index)[1].value
    username = sheet.row(index)[2].value
    password = sheet.row(index)[3].value
    enable_password = sheet.row(index)[4].value
    vendor = sheet.row(index)[5].value

    device = {
        'device_type': vendor,
```

```
    'ip': ipaddr,
    'username': username,
    'password': password,
    'secret': enable_password,
}
print "########## Connecting to Device {0} ############".format(device['ip'])
try:
    net_connect = ConnectHandler(**device)
    net_connect.enable()

    print "***** show ip configuration of Device *****"
    output = net_connect.send_command("show ip int b")
    print output

    net_connect.disconnect()
except NetMikoTimeoutException:
    print "=======SOMETHING WRONG HAPPEN WITH {0}=======".format(device['ip'])
except AuthenticationException:
    print "=======Authentication Failed with {0}=======".format(device['ip'])
except Exception as unknown_error:
    print "=======SOMETHING UNKNOWN HAPPEN WITH {0}======="
```

Więcej przykładów

Moduł Netmiko można stosować w bardzo wielu przypadkach, które automatyzują pracę w sieci. Może to być ładowanie, pobieranie plików aktualizacyjnych na zdalne urządzenie, ładowanie konfiguracji z wykorzystaniem szablonów Jinja2, uzyskiwanie dostępu do terminali serwerowych i wiele innych. Więcej użytecznych przykładów znajdziesz pod adresem *https://github.com/ktbyers/pynet/tree/master/presentations/dfwcug/examples*:



Podsumowanie

W tym rozdziale rozpoczęliśmy praktyczną wędrówkę po świecie zautomatyzowanych zadań sieciowych. Poznaliśmy wiele dostępnych w języku Python narzędzi do nawiązywania połączenia ze zdalnymi urządzeniami (za pośrednictwem protokołów Telnet i SSH) oraz do wykonywania zdalnych poleceń. Poznaliśmy także moduł netaddr, który ułatwia pracę w sieci (praca z adresami IP i podsieciami). Na końcu pogłębiliśmy wiedzę, analizując dwa praktyczne przypadki.

Z następnego rozdziału dowiesz się, jak przetworzyć otrzymane wyniki działań, aby uzyskać z nich praktyczne informacje.

Skorowidz

A

B

C

D

E

F

G

I

J

K

L

M

R

S, Ś

T

U

V

W

Y

Z

Automatyzacja zadań — przyjaciel najlepszych adminów!

Żadna współczesna firma nie może funkcjonować bez rozwiązań IT. Co więcej, stale trzeba mieć na uwadze wzrost wymagań wobec systemów informatycznych i konieczność wdrażania kolejnych aplikacji. Poza tym nie można ani na chwilę zapominać o kwestiach cyberbezpieczeństwa — ryzyko naruszenia integralności i bezpieczeństwa danych spędza sen z powiek wielu administratorom. Jeśli do tego dodać standardowe zadania związane z bieżącym działaniem firmowej sieci, okaże się, że praca administratora nie jest ani łatwa, ani prosta. Ratunkiem może być automatyzacja, która znacznie usprawnia czynności administracyjne.

Książka jest przeznaczona dla administratorów sieci, którzy chcą wykorzystać Pythona do zautomatyzowania części swoich obowiązków. Pokazano tu kilka praktycznych przykładów takiego zastosowania Pythona oraz narzędzi Ansible i Python Fabric, opisano też techniki konfiguracji serwera. Szczegółowo przedstawiono sposoby automatyzacji zadań związanych z zarządzaniem użytkownikami, bazą danych oraz procesami. Znalazły się tu również wskazówki ułatwiające pisanie skryptów dla usług testowych oraz przygotowanie automatyzacji pracy na maszynach wirtualnych i w środowisku chmurowym. W końcowych rozdziałach zaprezentowano niezwykle ważne zagadnienia bezpieczeństwa wraz z możliwościami automatyzacji w tym zakresie.

Najciekawsze zagadnienia przedstawione w książce:

- Python, jego IDE PyCharm oraz biblioteki używane do automatyzacji
- korzystanie z protokołów telnet i SSH (biblioteki: netmiko, paramiko i telnetlib)
- monitorowanie systemu i generowanie raportów
- maszyny wirtualne i wykorzystanie hipernadzorcy VMWare
- automatyzacja usług AWS za pomocą Boto3
- tworzenie skanera sieci w Pythonie

Bassem Aly — od dziewięciu lat pracuje w branży telekomunikacyjnej. Zajmował się projektowaniem i wdrażaniem rozwiązań wykorzystujących różne techniki automatyzacji oraz frameworki DevOps. Ma także duże doświadczenie w projektowaniu i wdrażaniu aplikacji telekomunikacyjnych w OpenStack. Prowadzi szkolenia korporacyjne w zakresie automatyzacji sieci i programowania sieciowego z wykorzystaniem Pythona i Ansible.

ISBN 978-83-283-5331-2
9 788328 353312
Cena: 67,00 zł

Packt