PRACTICA CALIFICADA Nº 1
(Domiciliaria)

NOTA: Para cada item de los contenidos, explique con detalle cómo trabajan.

PARTE 1 (10 puntos)

Usando Paramiko para conexión a dispositivos de red a través de SSH

Contenido

- Iniciar una sesión SSH con Paramiko
- Ejecución de un comando a través de SSH
- Leer la salida de un comando ejecutado
- Ejecución de comandos en múltiples dispositivos
- Ejecución de una secuencia de comandos
- Usar claves públicas/privadas para la autenticación
- Cargando la configuración SSH local

Nota: Instalar python3 -m pip install paramiko o python3 -m pip install paramiko==2.7.1

Iniciar una sesión SSH con Paramiko

La base para conectarse a un dispositivo a través de SSH con Python y Paramiko es el objeto SSHClient de la biblioteca. Usaremos este objeto para crear una conexión inicial al servidor SSH y luego usaremos las funciones de este objeto para ejecutar comandos en el dispositivo. Aquí veremos cómo abrir mediante programación una conexión SSH.

Uso del archivo file1.py.

Comencemos importando la biblioteca Paramiko y creando un objeto cliente. También especificaremos el host, el nombre de usuario y la contraseña en variables y luego iniciaremos una conexión al host especificado:

1. Importe la biblioteca de Paramiko:

   from paramiko.client import SSHClient

2. Especifique el host, el nombre de usuario y la contraseña. Puede nombrar estas variables como desees. En la comunidad de Python, se ha vuelto estándar poner en mayúsculas estas variables globales. Las tres variables SSH_USER, SSH_PASSWORD y SSH_HOST son variables de tipo cadena y por lo tanto usan comillas dobles para marcarlas. La variable SSH_PORT es un número entero y, por lo tanto, no utiliza comillas dobles:

```
SSH_USER = "developer"                                  # your ssh user
SSH_PASSWORD = "Cisco12345"                             # your ssh password
SSH_HOST = "sandbox-iosxe-recomm-1.cisco.com"           # IP/host of your device/server
SSH_PORT = 22                                           # Change this if your SSH port is different
```

3. Crea un objeto SSHClient, que acabamos de importar de Paramiko:

```
client = SSHClient ()
```

4. Si bien hemos creado el objeto cliente, aún no nos hemos conectado al dispositivo. Usaremos el método *connect* del objeto cliente para hacerlo. Antes de conectarnos realmente, tendremos que asegurarnos de que el cliente conozca las claves de host:

```
client.load_system_host_keys()
try:
    client.connect(SSH_HOST, port=SSH_PORT,
                                    username=SSH_USER,
                                    password=SSH_PASSWORD,
                                   look_for_keys=False)
        print("Connected successfully!")
except Exception:
        print("Failed to establish connection.")
```

5. Finalmente, hemos creado nuestra conexión. Es un buen hábito cerrar las conexiones una vez que hayamos terminado de usarlas. Para hacerlo, podemos usar la función *close ()* del cliente:

```
finally:
        client.close()
```

6. Para ejecutar este script, vaya al terminal y ejecútalo con lo siguiente:

```
python3 file1.py
```

En este ejemplo, confiamos en que el usuario ya había iniciado sesión en el dispositivo desde la línea de comando para que se conociera el host. Si usamos el código anterior para conectarnos a un dispositivo que no se conocía anteriormente, el código fallará con una excepción. La forma en que Paramiko maneja las claves de host desconocidas se puede especificar mediante una política. Una de estas políticas, *AutoAddPolicy*, nos permite simplemente agregar claves de host desconocidas al conjunto de scripts de claves de host:

```
from paramiko.client import SSHClient, AutoAddPolicy
SSH_USER = "<Insert your ssh user here>"
SSH_PASSWORD = "<Insert your ssh password here>"
SSH_HOST = "<Insert the IP/host of your device/server here>"
SSH_PORT = 22 # Change this if your SSH port is different

client = SSHClient()
client.set_missing_host_key_policy(AutoAddPolicy())
client.connect(SSH_HOST, port=SSH_PORT,
        username=SSH_USER,
        password=SSH_PASSWORD)
```

El código anterior agregará automáticamente estas claves de host. Ten en cuenta que esto podría ser un riesgo de seguridad potencial, ya que no se está verificando que el host al que se está conectando sea al que se conectó la última vez. En este ejemplo, pasamos detalles de conexión como nombre de usuario, nombre de host y contraseña directamente como una variable en el script. Si bien esto es excelente para las pruebas, es posible que desees que tu script te solicite una variable al ejecutarse.

Para las variables no secretas como el nombre de usuario, el host y el puerto, podemos usar la función incorporada *input ()*, pero para las contraseñas, es mejor usar una solicitud de contraseña dedicada que oculte lo que ha escrito para que alguien que revise el historial de tu consola no puede recuperar su contraseña. Para este propósito, Python tiene el módulo incorporado *getpass*.

Analiza los pasos para recuperar las variables de configuración necesarias, no como información estática en el script, sino de forma interactiva del usuario mediante una combinación de entrada y el módulo getpass:

```
import getpass
SSH_PASSWORD = getpass.getpass(prompt='Password: ',
stream=None)
SSH_USER = input("Username: ")
SSH_HOST = input("Host: ")
```

```
SSH_PORT = int(input("Port: "))
```

Ejecutando un comando a través de SSH

Ahora, podemos seguir adelante y ejecutar un comando en el dispositivo remoto. De manera similar a la forma en que manejamos la emisión de comandos en un dispositivo remoto a mano, tenemos tres flujos diferentes que regresan a nosotros: la salida estándar (o *stdout*), que es la salida normal, el error estándar (o *stderr*), que es el flujo predeterminado para que el sistema devuelva errores, y el estándar in (o *stdin*), que es el flujo utilizado para enviar texto de vuelta al comando ejecutado. Esto puede resultar útil si, en tu flujo de trabajo, normalmente interactúas con la línea de comandos.

Uso del archivo file2.py

1. Importa la biblioteca de Paramiko:

   ```
   from paramiko.client import SSHClient
   ```

2. Especifique el host, el nombre de usuario y la contraseña. Puede nombrar estas variables como desees. En la comunidad de Python, se ha convertido en un estándar poner en mayúsculas estas variables globales:

   ```
   SSH_USER = "<Insert your ssh user here>"
   SSH_PASSWORD = "<Insert your ssh password here>"
   SSH_HOST = "<Insert the IP/host of your device/server
   here>"
   SSH_PORT = 22 # Change this if your SSH port is different
   ```

3. Crea un objeto SSHClient, que acabamos de importar de Paramiko:

   ```
   client = SSHClient()
   ```

4. Si bien hemos creado el objeto cliente, aún no nos hemos conectado al dispositivo. Usaremos el método de connect del objeto cliente para hacerlo. Antes de conectarnos realmente, tendremos que asegurarnos de que el cliente conozca las claves de host:

   ```
   client.load_system_host_keys()
   client.connect(SSH_HOST, port=SSH_PORT,
           username=SSH_USER,
           password=SSH_PASSWORD)
   ```

5. Finalmente, podemos usar el cliente para ejecutar un comando. La ejecución de un comando nos devolverá tres objetos similares a archivos diferentes que representan stdin, stdout y stderr:

   CMD = "show ip interface brief" # You can issue any command you want
   stdin, stdout, stderr = client.exec_command(CMD)
   client.close()

6. Para ejecutar este script, vaya a su terminal y ejecútalo con esto:

   python3 file2.py

Leer la salida de un comando ejecutado

Uso del archivo file3.py

En el código anterior, vimos cómo conectarse primero a un dispositivo y luego ejecutar el comando. Sin embargo, hasta ahora hemos ignorado la salida.

En esta parte, veremos cómo abrir mediante programación una conexión SSH, enviar un comando y luego escribir el resultado de ese comando en un archivo. Usaremos esto para hacer una copia de seguridad de la configuración en ejecución.

1. Importa la biblioteca de Paramiko:

   from paramiko.client import SSHClient

2. Especifica el host, el nombre de usuario y la contraseña. Puede nombrar estas variables como desees. En la comunidad de Python, se ha convertido en un estándar poner en mayúsculas estas variables globales:

   SSH_USER = "<Insert your ssh user here>"
   SSH_PASSWORD = "<Insert your ssh password here>"
   SSH_HOST = "<Insert the IP/host of your device/server here>"
   SSH_PORT = 22 # Change this if your SSH port is different

3. Crea un objeto SSHClient, que acabamos de importar de Paramiko:

   client = SSHClient()

4. Si bien hemos creado el objeto cliente, aún no nos hemos conectado al dispositivo. Usaremos el método de connect del objeto cliente para hacerlo. Antes de conectarnos realmente, tendremos que asegurarnos de que el cliente conozca las claves de host:

   ```
   client.load_system_host_keys()
   client.connect(SSH_HOST, port=SSH_PORT,
   username=SSH_USER,
   password=SSH_PASSWORD)
   ```

5. Finalmente, podemos usar el cliente para ejecutar un comando. La ejecución de un comando nos devolverá tres objetos similares a archivos diferentes que representan *stdin, stdout* y *stderr*:

   ```
   CMD = "show running-config"
   stdin, stdout, stderr = client.exec_command(CMD)
   ```

6. Usaremos el objeto stdout para recuperar lo que ha devuelto el comando:

   ```
   output = stdout.readlines()
   ```

7. A continuación, volvemos a escribir la salida en un archivo:

   ```
   with open("backup.txt", "w") as out_file
           for line in output:
              out_file.write(line)
   ```

8. Para ejecutar este script, vaya a su terminal y ejecútalo con esto:

   python3 file3.py

Ejecutando el mismo comando contra múltiples dispositivos

Uso del archivo file4.py, credencial.json

En esta parte, veremos cómo abrir mediante programación una conexión SSH a varios dispositivos, emitir el mismo comando para todos ellos y luego guardar la salida. Usaremos de nuevo este ejemplo para realizar una copia de seguridad de la configuración en ejecución de varios dispositivos.

Crearemos un archivo llamado ***credentials.json***. Usaremos este archivo para recuperar credenciales como el nombre de usuario y la contraseña de nuestros dispositivos.

Comenzamos creando el archivo de credenciales. Luego leeremos ese archivo de nuestro script de Python, crearemos clientes para cada uno de estos dispositivos y finalmente ejecutaremos el comando mientras también guardamos la salida en nuestro archivo:

1. Importa las bibliotecas necesarias, Paramiko y json

```
import json
from paramiko.client import SSHClient
```

2. Abre el archivo credentials.json y proporcione las credenciales a su (s) dispositivo (s) en el formato que se muestra en el siguiente código. Puede especificar tantos dispositivos como desees:

```
[
{
        "name": "<insert a unique name of your device>",
        "host": "<insert the host of your device>",
        "username": "<insert the username>",
        "password": "<insert the password",
        "port": 22
},
{
        "name": "<insert a unique name of your device>",
        "host": "<insert the host of your device>",
        "username": "<insert the username>",
        "password": "<insert the password",
        "port": 22
}
]
```

En nuestro caso:

```
[
{
        "name": "sandbox_one",
        "host": "sandbox-iosxe-recomm-1.cisco.com",
        "username": "developer",
```

```
        "password": "Cisco12345",
        "port": 22
}
]
```

3. Regresa al archivo file4.py. Ahora abriremos el archivo JSON en nuestro script de Python:

```
credentials = {}
with open("credentials.json") as fh:
    credentials = json.load(fh)
```

4. Crea una variable que contenga el comando que desea ejecutar. Luego, recorreremos todos los dispositivos especificados en el archivo credencial.json y crearemos un objeto de cliente SSH. Además, crearemos un archivo de salida individual para cada uno de nuestros dispositivos según el nombre que especificamos en el archivo JSON:

```
CMD = "show running-config"
for cred in credentials:
 out_file_name = str(cred['name']) + ".txt"
 client = SSHClient()
 client.load_system_host_keys()
 client.connect(cred['host'], port=cred['port'],
                                username=cred['username'],
                                 password=cred['password'])
  stdin, stdout, stderr = client.exec_command(CMD)

  out_file = open(out_file_name, "w")
  output = stdout.readlines()
  for line in output:
      out_file.write(line)
  out_file.close()
  client.close()
  print("Executed command on " + cred['name'])
```

5. Para ejecutar este script, vaya al terminal y ejecútalo con esto: python3 file4.py.

Ejecutando una secuencia de comandos

Uso del archivo file5.py

Aquí veremos cómo abrir mediante programación una conexión SSH a un dispositivo, abrir un shell y luego enviar una lista de comandos al dispositivo antes de cerrar la conexión.

Comenzamos creando el archivo de credenciales. Luego leeremos ese archivo de nuestro script de Python, crearemos clientes para cada uno de estos dispositivos y finalmente ejecutaremos el comando mientras también guardamos la salida en nuestro archivo:

1. Importamos la biblioteca de Paramiko. También necesitaremos la biblioteca de time incorporada:

   ```
   from paramiko.client import SSHClient
   import time
   ```

2. Especifica el host, el nombre de usuario y la contraseña. Puede nombrar estas variables como desee. En la comunidad de Python, se ha convertido en un estándar poner en mayúsculas estas variables globales:

   ```
   SSH_USER = "<Insert your ssh user here>"
   SSH_PASSWORD = "<Insert your ssh password here>"
   SSH_HOST = "<Insert the IP/host of your device/server here>"
   SSH_PORT = 22 # Change this if your SSH port is different
   ```

3. Crea un objeto SSHClient, que acabamos de importar de Paramiko:

   ```
   client = SSHClient()
   ```

4. Si bien hemos creado el objeto cliente, aún no nos hemos conectado al dispositivo. Usaremos el método *connect* del objeto cliente para hacerlo. Antes de conectarnos realmente, tendremos que asegurarnos de que el cliente conozca las claves de host:

   ```
   client.load_system_host_keys()
   client.connect(SSH_HOST, port=SSH_PORT,
                            username=SSH_USER,
                            password=SSH_PASSWORD)
   ```

5. Abre una sesión de shell interactiva y un canal que podamos usar para recuperar la salida:

   ```
   channel = client.get_transport().open_session()
   ```

```
shell = channel.invoke_shell()
```

6. A continuación, especifica la lista de comandos que queremos ejecutar en el dispositivo:

```
commands = [
    "configure terminal",
    "hostname test"
]
```

7. Repite cada uno de los comandos, ejecútelos y luego espera 2 segundos:

```
for cmd in commands:
    shell.send(cmd + "\n")
     out = shell.recv(1024)
     print(out)
    time.sleep(1)
```

8. Finalmente, necesitamos cerrar la conexión:

```
client.close()
```

9. Para ejecutar este script, vaya a su terminal y ejecútalo con esto:

```
python 3 file5.py
```

Usar claves públicas / privadas para la autenticación

Uso del archivo: file6.py

Aquí veremos cómo abrir mediante programación una conexión SSH utilizando una clave privada protegida por contraseña.

Comencemos importando las bibliotecas requeridas, definamos los nuevos detalles de conexión y finalmente abramos una conexión usando autenticación basada en claves:

1. Importa la biblioteca de Paramiko:

```
from paramiko.client import SSHClient
```

2. Especifica el host y el nombre de usuario. Puede nombrar estas variables como desee. En la comunidad de Python, se ha convertido en un estándar poner en mayúsculas estas variables globales. En lugar de la contraseña del dispositivo, ahora necesitaremos dos nuevas variables: la ruta al archivo de clave privada que queremos usar para autenticarnos y la contraseña para ese archivo de clave privada:

```
SSH_USER = "<Insert your ssh user here>"
SSH_HOST = "<Insert the IP/host of your device/server here>"
SSH_PORT = 22 # Change this if your SSH port is different
SSH_KEY = "<Insert the name of your private key here>"
SSH_KEY_PASSWORD = "<Insert the password here>
```

3. Crea un objeto *SSHClient*, que acabamos de importar de Paramiko:

```
client = SSHClient()
```

4. Si bien hemos creado nuestro objeto cliente, aún no nos hemos conectado al dispositivo. Usaremos el método *connect* del objeto cliente para hacerlo. Antes de conectarnos realmente, aún tendremos que asegurarnos de que nuestro cliente conozca las claves de host:

```
client.load_system_host_keys()
client.connect(SSH_HOST, port=SSH_PORT,
                         username=SSH_USER,
                         look_for_keys=True,
                         key_filename=SSH_KEY,
                         passphrase=SSH_KEY_PASSWORD)
```

5. Como vimos antes, ahora podemos ejecutar un comando una vez establecida la conexión:

```
stdin, stdout, stderr = client.exec_command('<your command>')
```

6. Finalmente, necesitamos cerrar la conexión:
client.close()

7. Para ejecutar este script, vaya al terminal y ejecútalo con: python 3 file6.py

Cargando la configuración SSH local

En el siguiente código veremos cómo analizar mediante programación el archivo S*SHConfig*, extraer la información relevante basada en un host y almacenarla en un diccionario.
También necesitarás un archivo de configuración SSH para el dispositivo al que está intentando conectarse. En este ejemplo, usaremos un archivo de configuración que tiene el siguiente contenido:

```
Host example
    Host <insert your host address here>
    User <insert your user here>
    Port <insert the port here>
    IdentityFile <insert the path to your private key here>
```

Uso del archivo: file7.py

Comencemos importando las bibliotecas requeridas y definiendo la ruta a nuestra configuración SSH:

1. Importe la biblioteca de Paramiko:

   ```
   from paramiko.client import SSHClient
   from paramiko import SSHConfig
   ```

2. Especifica la ruta a su archivo de configuración SSH y el nombre de su host tal como aparece en su configuración SSH (ejemplo en este fragmento).
   Completaremos todas las demás variables de la configuración que estamos leyendo:

   ```
   SSH_CONFIG = "/Users/yuri/.ssh/config"                    # path to ssh config
   SSH_HOST = "example"
   ```

3. Creamos un objeto SSHConfig, que acabamos de importar de Paramiko, y crea un objeto de archivo local con la ruta a nuestra configuración SSH:

   ```
   config = SSHConfig()
   config_file = open(SSH_CONFIG)
   ```

4. A continuación, necesitamos decirle al objeto SSHConfig que cargue y analice el archivo de configuración:

   ```
   config.parse(config_file)
   ```

5. Con la configuración analizada, ahora podemos hacer una búsqueda en este objeto de configuración para extraer toda la información almacenada en la propia configuración. La función de búsqueda devolverá un diccionario:

   ```
   dev_config = config.lookup(SSH_HOST)
   ```

6. Con la configuración de nuestro dispositivo extraída de la configuración SSH, podemos continuar y completar nuestros detalles de conexión con lo que hemos extraído del archivo de configuración SSH:

   ```
   client = SSHClient()
   client.load_system_host_keys()

   HOST = dev_config['hostname'],
   client.connect(HOST, port=int(dev_config['port']),
                        username=dev_config['user'],
                        password =dev_config['password'])
   ```

7. Con la conexión establecida, podemos hacer todas las cosas diferentes que descubrimos en código anteriores antes de finalmente cerrar la conexión:

   ```
   client.close()
   ```

8. Para ejecutar este script, vaya al terminal y ejecuta esto: python3 file7.py

PARTE 2 (10 puntos)

Configuración de dispositivos de red con Netmiko

En esta actividad, aprenderemos cómo usar la biblioteca netmiko para conectarnos a los dispositivos, emitir comandos, recuperar resultados y copiar archivos, todo desde Python. Con las técnicas aprendidas, podrás automatizar los flujos de trabajo existentes que implican aplicar uno o más comandos a un dispositivo de red desde Python, así como recuperar información de sus dispositivos de red de una manera estructurada.

Instala el paquete netmiko (python3 -m pip install netmiko). Además, necesitará un dispositivo (virtual o físico) en el que pueda iniciar sesión a través de SSH.

Contenido

- Conexión a un dispositivo de red usando netmiko
- Enviar comandos usando netmiko
- Recuperar salidas de comandos como datos estructurados de Python usando netmiko y Genie
- Recopilación de datos con netmiko
- Conexión a varios dispositivos

Conexión a un dispositivo de red usando netmiko

Uso del archivo: file1.py

En este código, verás cómo abrir mediante programación una conexión SSH creando una instancia de *ConnectHandler* de netmiko.

Comencemos importando las clases requeridas de la biblioteca netmiko. Luego, configuraremos un diccionario que contiene los detalles de nuestra conexión y luego iniciaremos una conexión con el dispositivo que acabamos de especificar. Siga estos pasos para establecer una conexión desde tu script de Python a un dispositivo de red usando netmiko:

1. Importar ConnectHandler desde netmiko:

   ```
   from netmiko import ConnectHandler
   ```

2. Crea un diccionario que contendrá nuestros detalles de conexión. En este ejemplo, nos estamos conectando a un dispositivo Cisco IOS. Asegúrese de utilizar exactamente las mismas claves de diccionario (como device_type o host) al especificar la información de su conexión:

   ```
   connection_info = {
       'device_type': 'cisco_ios',
       'host': 'sandbox-iosxe-recomm-1.cisco.com',
       'port': 22,
       'username': 'developer',
        'password': 'Cisco12345'
   ```

```
}
```

3. A continuación, usaremos un administrador de contexto de Python para abrir una conexión, que es similar a abrir un archivo:

```
with ConnectHandler(**connection_info) as conn:
    print("Successfully connected!")
```

4. Para ejecutar este script, vaya al terminal y ejecuta el siguiente comando:

```
python3 file1.py
```

Si bien sería excesivo reiterar aquí la lista completa de 105 controladores, los siguientes son los controladores de los proveedores de dispositivos (de red) más comunes:

- cisco_ios, cisco_xe y cisco_xr para conectarse a dispositivos Cisco IOS
- cisco_nxos para conectarse a dispositivos que ejecutan el sistema operativo Cisco NX-OS
- linux para conectarse a dispositivos linux genéricos como servidores
- juniper, juniper_junos y juniper_screenos para conectarse a dispositivos de Juniper
- arista_eos para conectarse a dispositivos Arista
- hp_comware y hp_procurve para conectarse a dispositivos de HP
- huawei, huawei_smartax, huawei_olt y huawei_vrpv8 para conectarse a dispositivos de Huawei

Con el tipo de dispositivo definido, ahora podemos continuar y especificar los atributos de conexión familiares, como host, nombre de usuario, contraseña y puerto.

A continuación, usamos la construcción Python de un administrador de contexto. Un administrador de contexto es una construcción de Python que nos permite realizar dos operaciones relacionadas una tras otra sin que el desarrollador escriba un código repetitivo excesivo. Esencialmente, los administradores de contexto permiten a los desarrolladores de bibliotecas especificar el código que se ejecutará cuando se cree el objeto, antes de que se ejecute el código proporcionado por el usuario y después de que se ejecute el código proporcionado por el usuario.

netmiko también define un administrador de contexto para ConnectHandler para que podamos usar la instrucción with. Con esto, netmiko también se encarga de cerrar la conexión al dispositivo una vez que hayamos ejecutado todos nuestros comandos:

```
connection_info = {
   'device_type': 'cisco_ios',
   'host': '<insert your host here>',
  'port': <insert your port number here>,
  'username': '<insert your username here>',
  'password': '<insert your password here>'
}
with ConnectHandler(device_type='cisco_ios',
    host='<your_host>',
   port=<insert your port number here>,
   username='<insert your username here>',
   password='<insert your password here>') as conn:
      print("Successful connection!")
```

Enviar comandos usando netmiko

Uso del archivo: file2.py

En esta parte veremos cómo abrir mediante programación una conexión SSH creando una instancia de ConnectHandler de netmiko, enviando un comando al dispositivo remoto y recuperando la salida del comando como una cadena.

Comencemos importando las clases requeridas de la biblioteca netmiko. Luego, configuraremos un diccionario que contiene los detalles de nuestra conexión y luego iniciaremos una conexión con el dispositivo que acabamos de especificar. Con la conexión cubierta, podemos proceder y emitir el comando e imprimir la salida al usuario.

Seguimos estos pasos para conectarse a un dispositivo de red y enviar comandos usando netmiko:

1. Importar ConnectHandler desde netmiko

```
from netmiko import ConnectHandler
```

2. Crea un diccionario que contendrá los detalles de conexión. En este ejemplo, nos estamos conectando a un dispositivo Cisco IOS. Asegúrese de utilizar exactamente las mismas claves de diccionario (como device_type o host) al especificar la información de su conexión:

```
connection_info = {
    'device_type': 'cisco_ios',
    'host': '<insert your host here>',
    'port': <insert your port number here>,
    'username': '<insert your username here>',
    'password': '<insert your password here>'
}
```

3. A continuación, usaremos un administrador de contexto de Python para abrir una conexión similar a cómo podemos abrir un archivo. Dentro de ese administrador de contexto, usaremos el método *send_command ()* de ConnectHandler para enviar el comando show interfaces al dispositivo e imprimir la salida:

```
with ConnectHandler(**connection_info) as conn:
    out = conn.send_command("show interfaces")
    print(out)
```

4. Para ejecutar este script utiliza el siguiente comando. La salida debe ser la misma que si se hubiera conectado al dispositivo mediante SSH directamente y hubiera emitido el siguiente comando:

```
python3 file2.py
```

Recuperar salidas de comandos como datos estructurados de Python usando netmiko y Genie

Usaremos el archivo **interfaces.py**.

Con los siguientes pasos, puedes recuperar la salida de un comando como datos estructurados de Python en lugar de texto sin formato:

1. Importe el *ConnectHandler* de netmiko. Además, importamos el módulo *pretty* de pprint que nos permite imprimir un diccionario con mejor formato:

```
from netmiko import ConnectHandler
```

2. Crea un diccionario que contendrá los detalles de conexión. En este ejemplo, nos estamos conectando a un dispositivo Cisco IOS.Asegúrese de utilizar exactamente las mismas claves de diccionario (como device_type o host) al especificar la información de su conexión:

```
connection_info = {
    'device_type': 'cisco_ios',
    'host': '<insert your host here>',
    'port': <insert your port number here>,
    'username': '<insert your username here>',
    'password': '<insert your password here>'
}
```

3. A continuación, usaremos un administrador de contexto de Python para abrir una conexión, similar a cómo abrimos un archivo. Dentro de ese administrador de contexto, usaremos el método *send_command ()* del *ConnectHandler* para enviar nuestro comando show interfaces al dispositivo e imprimir la salida. Observe cómo hemos establecido el indicador *use_genie* de *send_command* en True:

```
with ConnectHandler(**connection_info) as conn:
        out = conn.send_command("show interfaces",   use_genie=True)
```

4. A continuación, imprimimos el diccionario completo que recibimos usando prettyprint. Asegúrese de estar todavía dentro del administrador de contexto

```
        pprint.pprint(out)
```

5. Y finalmente, iteramos sobre todos los elementos del diccionario. Las claves de este diccionario específico serán los nombres de las interfaces. El resultado del curso depende del comando específico que estés emitiendo:

```
        for interface in out.keys():
            print(interface)
```

6. Para ejecutar este script utiliza el siguiente comando:

```
python3 interfaces.py
```

Lo que debería salir es una muestra de la salida del diccionario devuelto impreso por la biblioteca *prettyprint*. Como puede ver, el diccionario tiene un formato agradable y una sangría adecuada, lo que facilita su lectura.

Con la conexión netmiko establecida podemos enviar nuestro comando nuevamente y recuperar el resultado. La única diferencia con la forma en que invocamos *send_command ()* fue que cambiamos el indicador *use_genie* a True. Internamente, esto invocó al analizador Genie, y el objeto devuelto ya no es el texto sin formato devuelto por el dispositivo de red en sí, ¡sino la salida analizada en forma de diccionario!

Con nuestro comando show interfaces, el analizador Genie pone los nombres de las interfaces como clave para un diccionario anidado. En ese diccionario se encuentran todos los detalles devueltos por el comando en datos estructurados de Python. Por ejemplo, podría encontrar el ancho de banda de una interfaz llamada GigabitEthernet1 utilizando ['GigabitEthernet1'] ['bandwidth'].

Con nuestro conocimiento de cómo están estructurados los datos devueltos en el diccionario, podemos seguir adelante e imprimir nuestra lista deseada de interfaces recorriendo todas las claves en el diccionario devuelto por Genie.

Recopilación de datos con netmiko

Uso del archivo file3.py

Anteriormente vimos cómo recuperar la salida de un comando como datos estructurados en Python. En esta sección, nos basaremos en esta funcionalidad para obtener un perfil de todas nuestras interfaces. Esto nos mostrará cómo utilizar los datos analizados para imprimir e identificar rápidamente la información relevante. Para este ejemplo, usaremos el comando show interfaces nuevamente y, para cada interfaz, recuperaremos e imprimimos lo siguiente:

- El nombre de la interfaz
- El estado de la interfaz (habilitada o no habilitada)
- La dirección física de la interfaz
- El modo dúplex
- Cualquier contador que esté por encima de 0 preparándose

Abre tu editor de código y revisa file3.py.

Con los siguientes pasos, puedes recuperar el resultado de nuestro comando show interfaces como datos estructurados de Python que luego se analizarán más a fondo:

1. Importa ConnectHandler desde netmiko. Además, importamos el bonito módulo pprint que nos permite imprimir un diccionario con mejor formato:

```
from netmiko import ConnectHandler
import pprint
```

2. Crea un diccionario que contendrá los detalles de conexión. En este ejemplo, nos estamos conectando a un dispositivo Cisco IOS. Asegúrese de utilizar exactamente las mismas claves de diccionario (como device_type o host) al especificar la información de su conexión:

```
connection_info = {
    'device_type': 'cisco_ios',
    'host': '<insert your host here>',
    'port': <insert your port number here>,
    'username': '<insert your username here>',
    'password': '<insert your password here>'
}
```

3. A continuación, usaremos un administrador de contexto de Python para abrir una conexión, similar a abrir un archivo. Dentro de ese administrador de contexto, usaremos el método send_command () del ConnectHandler para enviar nuestro comando show interfaces al dispositivo e imprimir la salida. Observa que hemos establecido el indicador use_genie de send_command en True:

```
with ConnectHandler(**connection_info) as conn:
    out = conn.send_command("show interfaces",   use_genie=True)
```

4. Con nuestros datos estructurados recuperados, ahora podemos iterar sobre cada una de las interfaces e imprimir la información que queremos. Asegúrese de estar todavía dentro del administrador de contexto:

```
for name, details in out.items():
        print(f"{name}")
        print(f"- Status: {details.get('enabled', None)}")
        print(f"- Physical address: {details.get('phys_address', None)}")
        print(f"- Duplex mode: {details.get('duplex_mode', None)}")
```

5. Con nuestra información básica impresa, ahora podemos iterar sobre todos los contadores y sus valores para verificar si alguno de esos contadores está por encima de 0. Asegúrese de que todavía está dentro del administrador de contexto (primer nivel de indentado) así como dentro del for loop de las interfaces (segundo nivel de indentado):

```
for counter, count in details.get('counters', {}).items():
    if isinstance(count, int):
        if count > 0:
            print(f"- {counter}: {count}")
    elif isinstance(count, dict):
        for sub_counter, sub_count in count.items():
            if sub_count > 0:
                print(f"- {counter}::{sub_counter}: {sub_count}")
```

6. Para ejecutar este script, vaya al terminal y escribe: python3 file3.py

Con la conexión netmiko establecida podemos enviar el comando nuevamente y recuperar la salida como datos estructurados.

Con esta información básica impresa, podemos iterar a continuación sobre los contadores disponibles.

La salida exacta de los datos estructurados depende de cómo Genie analiza el comando. Puede ver la estructura devuelta echando un vistazo a los analizadores disponibles. Genie se actualiza constantemente y se agregan nuevos analizadores. Puede encontrar una lista actualizada de todos los analizadores incluidos en https://pubhub.devnetcloud.com/media/genie-feature-browser/docs/#/parsers.

Conexión a varios dispositivos

En esta sección vamos a especificar la información de conexión para los dispositivos en forma de archivo JSON. Según la información de conexión almacenada en este archivo JSON, luego nos conectaremos a cada uno de estos dispositivos, emitiremos un comando (*show running-config* en este ejemplo), recuperaremos la salida y la guardaremos en un archivo.

Primero importamos la biblioteca json incorporada, así como el módulo netmiko. A continuación, vamos a especificar los detalles de nuestra conexión en nuestro archivo *connections.json*, leer ese archivo en nuestro script de Python y abrir una conexión basada en

los detalles proporcionados. Emitiremos el comando *show running-config* en cada uno de los dispositivos y guardaremos la salida en un archivo de texto.

Con los siguientes pasos, podemos leer los detalles de la conexión de un archivo JSON, conectarnos a cada dispositivo especificado, ejecutar un comando y guardar la salida del comando por dispositivo:

Uso del archivo file4.py, connections.json

1. Abre el archivo *connections.json* en tu editor de código. Para cada dispositivo, vamos a definir la siguiente estructura.

```
[
 {
    "name": "device-1",
    "connection": {
          "device_type": "<insert your device type here>",
          "host": "<insert your host here>",
          "port": <insert your port number here>,
          "username": "<insert your username here>",
          "password": "<insert your password here>"
          }
     },
{
"name": "device-2",
"connection": {
   "device_type": "<insert your device type here>",
   "host": "<insert your host here>",
  "port": <insert your port number here>,
  "username": "<insert your username here>",
  "password": "<insert your password here>"
 }
 }
]
```

2. Con todos sus dispositivos especificados, puedes abrir el archivo file4.py. Primero vamos a importar las bibliotecas requeridas:

```
import json
from netmiko import ConnectHandler
```

3. A continuación, necesitamos leer los detalles de la conexión de nuestro archivo JSON:

```
devices = []
with open("connections.json", "r") as fh:
    devices = json.load(fh)
```

4. Ahora podemos iterar sobre cada dispositivo en la lista de dispositivos usando la información de conexión almacenada en el archivo JSON, emitir el comando definido y luego guardar la salida en un nuevo archivo basado en el nombre que le dimos al dispositivo en el archivo JSON:

```
CMD = "show running-config"
for device in devices:
        file_name = f"{device['name']}.out"
    print(f"Retrieving config for {device['name']}")
    with ConnectHandler(**device['connection']) as conn:
            out = conn.send_command(CMD)
        with open(file_name) as f:
            f.write(out)
```

5. Para ejecutar este script debes hacer: python file4.py

Deberías ver una salida impresa reconociendo que el comando se está emitiendo para cada uno de sus dispositivos especificados en el archivo connections.json.