Laboratorio 3b - Analizar diferentes tipos de datos con Python

Objetivos

Parte 1: Iniciar la Maquina Virtual (Virtual Machine) DEVASC.

Parte 2: Analizar XML en Python.

Parte 3: Analizar JSON en Python.

Parte 4: Analizar YAML en Python.

Aspectos básicos/Situación.

Examinar significa analizar un mensaje, dividiéndolo en sus partes componentes y comprender el propósito de cada parte en contexto. Cuando los mensajes se transmiten entre equipos, viajan como una secuencia de caracteres. Esos caracteres son efectivamente una string (cadena). Ese mensaje debe ser analizado en una estructura de datos semánticamente equivalente que contenga datos de tipos reconocidos (por ejemplo, integers, floats, strings, and booleans) antes de que los datos puedan ser interpretados y actuados sobre ellos.

En este laboratorio, usará Python para analizar cada formato de datos a su vez: XML, JSON y YAML. Analizaremos ejemplos de código y hablaremos sobre cómo funciona cada analizador.

Recursos necesarios

- 1 Computadora con sistema operativo de su elección
- Virtual Box o VMWare
- Máquina virtual (Virtual Machine) DEVASC

Instrucciones

Parte 1: Iniciar la Máquina virtual (Virtual Machine) de DEVASC

Si no ha completado el laboratorio **- Instale el Entorno de Laboratorio de la Máquina Virtual**, hágalo ahora. Si se ha completado ya, inicie la máquina virtual DEVASC.

Parte 2: Analizar XML en Python

Debido a la flexibilidad proporcionada por el Lenguaje de marcado extensible (XML), puede ser complicado de analizar. Los campos de datos etiquetados de texto completo de XML no se asignan inequívocamente a los tipos de datos predeterminados en Python u otros lenguajes populares. Además, no siempre es obvio cómo se deben representar los valores de atributo en los datos.

Estos problemas pueden ser eludidos por los desarrolladores de Cisco que trabajan en algunos contextos, ya que Cisco ha proporcionado herramientas como YANG-CLI, que valida y consume XML relevante para el modelado de datos y las tareas relacionadas. A continuación se muestra el contenido del archivo **myfile.xml** que se encuentra en **~/labs/devnet-src/parsing**. Este es un ejemplo del tipo de archivo que administra YANG-CLI. Analizar este archivo en Python para obtener acceso a la información que contiene.

```
<?xml version="1.0" encoding="UTF-8"?>
<rpc message-id="1"
 xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
 <edit-config>
  <target>
   <candidate/>
  </target>
  <default-operation>fusionar </default-operation>
  <test-option>set </test-option>
  <config>
   <int8.1
    xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"
    nc:operation="crear»
    xmlns=» http://netconfcentral.org/ns/test">9 </int8.1>
  </config>
 </edit-config>
</rpc>
```

Paso 1: Crar un script para analizar los datos XML.

a. Abrir el archivo **parsexml.py** que se encuentra en el directorio **~/labs/devnet-src/parsing**.

b. Importar el módulo ElementTree de la librería XML y el motor de expresiones regulares. El módulo ElementTree se utilizará para realizar el análisis. El motor de expresión regular se utilizará para buscar datos específicos.

Nota: Si no tiene experiencia con el uso de expresiones regulares en Linux, Python u otros lenguajes de programación orientados a objetos, busque tutoriales en Internet.

```
Import XML.Etree.ElementTree as ET
Import re
```

c. A continuación, utilice la función de **parse** de ET (ElementTree) para analizar el archivo **myfile.xml** y asignarlo a una variable (**xml**). Luego, obtenga el elemento raíz con la función **getroot** y asígnelo a una variable (**root**).

```
xml = et.parse («myfile.xml»)
root = xml.getroot ()
```

d. Ahora el nivel superior del árbol se puede buscar la etiqueta que contiene <edit-config>, y cuando se encuentra, ese bloque etiquetado se puede buscar por dos valores con nombre que contiene: <default-operation> y <test-option>. Crear una expresión regular para obtener el contenido del contenido root (raíz) XML en la <rpc> etiqueta y, a continuación, agregue expresiones regulares adicionales para profundizar en el contenido con el fin de encontrar el valor de <edit-config> <default-operation>, y **<test-option>** elementos.

```
ns = re.match ('{.*}', root.tag) .group (0)
editconf = root.find («{} edit-config» .format (ns))
defop = editconf.find («{} default-operation» .format (ns))
testop = editconf.find («{} test-option» .format (ns))
```

e. Agregar instrucciones de impresión para imprimir el valor de los **<default-operation>**y **<test-option>**

```
print ("The default-operation contains: {}".format(defop.text))
print ("The test-option contains: {}".format(testop.text))
```

Paso 2: Ejecute el script

Guarde y ejecute **parsexml.py**. Debería ver el siguiente resultado:

```
devasc @labvm: ~/labs/devnet-src/parsing$ python3 parsexml.py
La operación predeterminada contiene: merge
La opción de prueba contiene: set
devasc @labvm: ~/labs/devnet-src/parsing$
```

Parte 3: Analizar JSON en Python

El análisis de la notación de objetos JavaScript (JSON) es un requisito frecuente para interactuar con las API REST. Los pasos son generalmente los siguientes:

1) Autenticar usando una combinación de usuario/contraseña para recuperar un token que caducará después de un período de tiempo establecido. Este token se utiliza para autenticar solicitudes posteriores.
2) Ejecutar una solicitud GET a la REST API, autenticando según sea necesario, para recuperar el estado de un recurso, solicitando JSON como formato de salida.
3) Modificar el JSON devuelto, según sea necesario.
4) Ejecutar un POST (o PUT) en la misma API REST (de nuevo, autenticando según sea necesario) para cambiar el estado del recurso, nuevamente solicitando JSON como formato de salida e interpretándolo según sea necesario para determinar si la operación se realizó correctamente.

El ejemplo JSON para analizar es esta respuesta de una solicitud de token:

```
{
 «access_token»
:"zdi3mgeyyzqtnmflns00ndnhlwflnzatzgvjnje0mgu1ogzWnmzdewn2itytu3",
 «expires_in» :1209600,
 «refresh_token»
:"mdeymzq1njc4otaxmjm0nty3odkwmtizndu2nzg5mdeymzq1njc4otEymzq1njc4",
 «refreshtokenexpires_in» :7776000
}
```

En los scripts de Python, la librería Python **json** se puede usar para analizar JSON en estructuras de datos nativas de Python y serializar estructuras de datos como JSON. La librería Python **yaml** se puede utilizar para convertir los datos a YAML.

El siguiente programa utiliza ambos módulos para analizar los datos JSON anteriores, extraer e imprimir valores de datos, y generar una versión YAML del archivo. Utilizar el método **json** library **loads ()** para analizar una string en la que se ha leído el archivo. A continuación, utilizar referencias de datos normales de Python para extraer valores de la estructura de datos de Python resultante. Finalmente, utilizar la función de **yamllibraryl dump()** para serializar los datos de Python de nuevo como YAML, al terminal.

Paso 1: Crear un script para analizar los datos JSON.

a. Abrir el archivo **parsejson.py** que se encuentra en el directorio **~/labs/devnet-src/parsing**.

b. Importar las librerías **json** y **yaml**.

```
Import json
Import yaml
```

c. Usar la instrucción Python **with** para abrir **myfile.json** y establecerlo en el nombre de la variable **json_file**. Luego use el método **json.load** para cargar el archivo JSON en una string establecida con el nombre de variable **ourjson**.

Nota: No hay necesidad de cerrar explícitamente el archivo, ya que la instrucción **with** garantiza la apertura y el cierre adecuados del archivo.

```
with open ('myfile.json', 'r') as json_file:
    ourjson = json.load (json_file)
```

d. Agregar una declaración de impresión para **ourjson** para ver que ahora es un diccionario de Python.

```
print (ourjson)
```

Paso 2: Ejecutar el script para imprimir los datos JSON y luego modificarlo para imprimir datos de interés.

a. Guardar y ejecutar el script. Debería ver el siguiente resultado:

```
devasc @labvm: ~/labs/devnet-src/parsing$ python3 parsejson.py
{'access_token': 'zdi3mgeyzqtnmflns00ndnhlwflnzatzgvjnje0mgu1ogzmzwnmzDewn2itytu3',
'expires_in': 1209600, 'refresh_token':
'mdeymzq1njc4otaxmJm0nty3OdY3kWnty3jmJm0nty3kWnty3kWnty3kWnjmJM0njm_token'
Mtizndu2nzg5mdeymzq1njc4oteymzq1njc4 ',' refreshtokenexpires_in ': 7776000}
devasc @labvm: ~/labs/devnet-src/parsing$
```

b. Agregar instrucciones de impresión que muestren el valor del token y cuántos segundos hasta que caduque el token.

```
print("The access token is: {}".format(ourjson['access_token']))
print("The token expires in {} seconds.".format(ourjson['expires_in']))
```

c. Guardar y ejecutar el script. Debería ver el siguiente resultado:

```
devasc @labvm: ~/labs/devnet-src/parsing$ python3 parsejson.py
{'access_token': 'zdi3mgeyzqtnmflns00ndnhlwflnzatzgvjnje0mgu1ogzmzwnmzDewn2itytu3',
'expires_in': 1209600, 'refresh_token':
'mdeymzq1njc4otaxmJm0nty3OdY3kWnty3jmJm0nty3kWnty3kWnty3kWnjmJM0njm_token'
Mtizndu2nzg5mdeymzq1njc4oteymzq1njc4 ',' refreshtokenexpires_in ': 7776000}
1209600
El token de acceso es zdi3mgeyyzqtnmflns00ndnhlwflnzatzgvjnje0mgu1ogzWnmzDewn2itytu3
El token caduca en 1209600 segundos
devasc @labvm: ~/labs/devnet-src/parsing$
```

Paso 3: Muestra los datos JSON analizados en un formato de datos YAML.

a. Agregue una instrucción de impresión que mostrará los tres guiones necesarios para un archivo YAML. Los dos **\ n** agregarán dos líneas después de la salida anterior. Luego agregue una declaración para imprimir **ourjson** como datos YAML utilizando el método **dump ()** de la librería **yaml**.

```
print(«\ n\ n—»)
print (yaml.dump (ourjson))
```

b. Guardar y ejecutar el script. Debería ver el siguiente resultado:

```
devasc@labvm:~/labs/devnet-src/parsing$ python3 parsejson.py
<output from previous steps omitted>
---
access_token: zdi3mgeyzqtnmflns00ndnhlwflnzatzgvjnje0mgu1ogzWnmzDewn2itytu3
expires_in: 1209600
refresh_token: mdeymzq1njc4otaxmjm0nty3odkwmtizndu2nzg5mdeymzq1njc4otEymzq1njc4
refreshtokenexpires_in: 7776000

devasc @labvm: ~/labs/devnet-src/parsing$
```

Parte 4: Analizar YAML en Python

El siguiente programa importa las librerías **json** y **yaml**, utiliza PyYaml para analizar un archivo YAML, extraer e imprimir valores de datos y generar una versión JSON del archivo. Utilizar el método de librería **yamlsafe_load ()** para analizar la secuencia de archivos y las referencias normales de datos de Python para extraer valores de la estructura de datos de Python resultante. A continuación, utilice la función **json librarydumps()** para serializar los datos de Python de nuevo como JSON.

El ejemplo de YAML para analizar es el mismo archivo YAML que se ha salido en la Parte 3:

```
---
access_token: zdi3mgeyzqtnmflns00ndnhlwflnzatzgvjnje0mgu1ogzWnmzDewn2itytu3
expires_in: 1209600
refresh_token: mdeymzq1njc4otaxmjm0nty3odkwmtizndu2nzg5mdeymzq1njc4otEymzq1njc4
refreshtokenexpires_in: 7776000
```

Paso 1: Crear un script para analizar los datos de YAML.

a. Abrir el archivo **parseyaml.py** que se encuentra en el directorio **~/labs/devnet-src/parsing**.

b. Importar las librerías **json** y **yaml**.

```
Import json
Import yaml
```

c. Utilice la instrucción Python **with** para abrir **myfile.yaml** y establecerlo en el nombre de la variable **yaml_file**. A continuación, utilice el método **yaml.safe_load** para cargar el archivo YAML en una string (cadena) establecida con el nombre de variable **ouryaml**.

```
with open('myfile.yaml','r') as yaml_file:
    ouryaml = yaml.safe_load (yaml_file)
```

d. Agregue una declaración de impresión para **ouryaml** para ver que ahora es un diccionario de Python.

```
print (ouryaml)
```

Paso 2: Ejecutar el script para imprimir los datos de YAML y luego modificarlo para imprimir datos de interés.

a. Guardar y ejecutar el script. Debería ver el siguiente resultado:

```
devasc @labvm: ~/labs/devnet-src/parsing$ python3 parseyaml.py
{'access_token': 'zdi3mgeyzqtnmflns00ndnhlwflnzatzgvjnje0mgu1ogzmzwnmzDewn2itytu3',
'expires_in': 1209600, 'refresh_token':
'mdeymzq1njc4otaxmJm0nty3OdY3kWnty3jmJm0nty3kWnty3kWnty3kWnjmJM0njm_token'
Mtizndu2nzg5mdeymzq1njc4oteymzq1njc4 ',' refreshtokenexpires_in ': 7776000}
```

```
devasc @labvm: ~/labs/devnet-src/parsing$
```

b. Agregar instrucciones de impresión que muestren el valor del token y cuántos segundos hasta que caduque el token.

```
print("The access token is {}".format(ouryaml['access_token']))
print("The token expires in {} seconds.".format(ouryaml['expires_in']))
```

c. Guardar y ejecutar el script. Debería ver el siguiente resultado:

```
devasc @labvm: ~/labs/devnet-src/parsing$ python3 parseyaml.py
{'access_token': 'zdi3mgeyzqtnmflns00ndnhlwflnzatzgvjnje0mgu1ogzmzwnmzDewn2itytu3',
'expires_in': 1209600, 'refresh_token':
'mdeymzq1njc4otaxmJm0nty3OdY3kWnty3jmJm0nty3kWnty3kWnty3kWnjmJM0njm_token'
Mtizndu2nzg5mdeymzq1njc4oteymzq1njc4 ',' refreshtokenexpires_in ': 7776000}
El token de acceso es zdi3mgeyyzqtnmflns00ndnhlwflnzatzgvjnje0mgu1ogzWnmzDewn2itytu3
El token caduca en 1209600 segundos.
devasc @labvm: ~/labs/devnet-src/parsing$
```

Paso 3: Muestra los datos YAML analizados en un formato de datos JSON.

a. Agregue una instrucción de impresión para agregar dos líneas en blanco después de la salida anterior. Luego agregue una declaración para imprimir **ouryaml** como datos JSON utilizando el método **dumps ()** de la librería **json**. Agregar el parámetro de sangría para especificar los datos JSON.

```
print («\ n\ n»)
print(json.dumps(ouryaml, indent=4))
```

b. Guardar y ejecutar el script. Debería ver el siguiente resultado: Observar que la salida se parece al **myfile.json**.

```
devasc @labvm: ~/labs/devnet-src/parsing$ python3 parseyaml.py
<output from previous steps omitted>
{
  «access_token» :"zdi3mgeyyzqtnmflns00ndnhlwflnzatzgvjnje0mgu1ogzWnmzdewn2itytu3",
  «expires_in» :1209600,
  «refresh_token» :"mdeymzq1njc4otaxmjm0nty3odkwmtizndu2nzg5mdeymzq1njc4otEymzq1njc4",
  «refreshtokenexpires_in» :7776000
}
devasc @labvm: ~/labs/devnet-src/parsing$
```