Travaux pratiques - Analyser différents types de données avec Python

# Objectifs

Partie 1 : Lancer la DEVASC VM

Partie 2 : Analyser (Parse) XML en Python

Partie 3 : Analyser (Parse) JSON en Python

Partie 4 : Analyser (Parse) YAML en Python

# Contexte/scénario

Parsing signifie analyser un message, le diviser en ses parties constitutives et comprendre le but de chaque partie dans son contexte. Lorsque les messages sont transmis entre ordinateurs, ils voyagent comme un flux de caractères. Ces caractères sont effectivement une chaîne. Ce message doit être analysé dans une structure de données sémantique équivalente contenant des données de types reconnus (par exemple, entiers, flottants, chaînes et booléens) avant que les données puissent être interprétées et traitées.

Dans ce laboratoire, vous allez utiliser Python pour analyser chaque format de données à tour de rôle : XML, JSON et YAML. Nous allons parcourir des exemples de code et parler du fonctionnement de chaque analyseur.

# Ressources requises

* 1 PC avec système d'exploitation de votre choix
* Boîte virtuelle ou VMWare
* Machine virtuelle DEVASC

# Instructions

## Lancer la machine virtuelle DEVASC

Si vous n'avez pas encore terminé **TP - Installer l'environnement de laboratoire de machines virtuelles**, faites-le maintenant. Si vous avez déjà terminé ce laboratoire, lancez la machine virtuelle DEVASC maintenant.

## Analyser XML en Python

En raison de la flexibilité fournie par le langage XML (Extensible Markup Language), il peut être difficile d'analyser. Les champs de données balisés en texte intégral de XML ne sont pas mappés sans ambiguïté avec les types de données par défaut en Python ou dans d'autres langues populaires. En outre, il n'est pas toujours évident comment les valeurs attributaires doivent être représentées dans les données.

Ces problèmes peuvent être évités par les développeurs Cisco travaillant dans certains contextes, car Cisco a fourni des outils tels que YANG-CLI, qui valide et consomme du XML pertinent pour la modélisation des données et les tâches connexes. Voici le contenu du fichier **myfile.xml** trouvé dans **~/labs/devnet-src/parsing**. Ceci est un exemple du type de fichier géré par YANG-CLI. Vous allez analyser ce fichier en Python pour accéder aux informations qu'il contient.

<?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>merge</default-operation>

<test-option>set</test-option>

<config>

<int8.1

xmlns:nc="urn:ietf:params:xml:ns:netconf:base:1.0"

nc:operation="create"

xmlns="http://netconfcentral.org/ns/test">9</int8.1>

</config>

</edit-config>

</rpc>

### Créez un script pour analyser les données XML.

* + - 1. Ouvrez le fichier **parsexml.py** situé dans le répertoire **~/labs/devnet-src/parsing**.
      2. Importez le module ElementTree de la bibliothèque xml et le moteur d'expression régulière. Le module ElementTree sera utilisé pour effectuer l'analyse. Le moteur d'expression régulière sera utilisé pour rechercher des données spécifiques.

**Remarque** : Si vous n'avez aucune expérience avec l'utilisation d'expressions régulières dans Linux, Python ou d'autres langages de programmation orientés objet, recherchez des didacticiels sur Internet.

importer Xml.etree.elementTree en tant que ET

Importer re

* + - 1. Ensuite, utilisez la fonction **parse** de ET (ElementTree) pour analyser le fichier **myfile.xml** et l'affecter à une variable (**xml**). Ensuite, obtenez l'élément racine avec la fonction **getroot** et attribuez-le à une variable (**root**).

xml = ET.parse("myfile.xml")

root = xml.getroot()

* + - 1. Maintenant, le niveau supérieur de l'arborescence peut être recherché pour la balise contenant **<edit-config>**, et une fois trouvé, ce bloc balisé peut être recherché pour deux valeurs nommées qu'il contient : **<default-operation>** et **<test-option>**. Créez une expression régulière pour obtenir le contenu du contenu racine XML dans la **<rpc>** balise, puis ajoutez des expressions régulières supplémentaires pour explorer le contenu afin de trouver la valeur de **<edit-config>**, **<default-operation>**, et **<test-option** éléments.

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))

* + - 1. Ajoutez des instructions d'impression pour imprimer la valeur des éléments**<default-operation>** et **<test-option>**.

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

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

### Exécutez le script.

Enregistrez et exécutez le fichier **parsexml.py**. Vous devriez obtenir le résultat suivant.

devasc@labvm:~/labs/devnet-src/parsing$ **python3 parsexml.py**

The default-operation contains: merge

The test-option contains: set

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

## Analyser JSON en Python

L'analyse de JavaScript Object Notation (JSON) est une exigence fréquente d'interaction avec les API REST. Les étapes sont généralement les suivantes :

* + - * 1. Authentifiez à l'aide d'une combinaison utilisateur/mot de passe pour récupérer un jeton qui expirera après un certain temps. Ce jeton est utilisé pour authentifier les demandes suivantes.
        2. Exécutez une requête GET à l'API REST, en s'authentifiant si nécessaire, pour récupérer l'état d'une ressource, en demandant JSON comme format de sortie.
        3. Modifiez le JSON retourné, si nécessaire.
        4. Exécutez un POST (ou PUT) sur la même API REST (encore une fois, en s'authentifiant si nécessaire) pour modifier l'état de la ressource, en demandant à nouveau JSON comme format de sortie et en l'interprétant selon les besoins pour déterminer si l'opération a réussi.

L'exemple JSON à analyser est la réponse d'une requête de jeton :

{

"access\_token":"ZDI3MGEyYzQtNmFlNS00NDNhLWFlNzAtZGVjNjE0MGU1OGZmZWNmZDEwN2ItYTU3",

"expires\_in":1209600,

"refresh\_token":"MDEyMzQ1Njc4OTAxMjM0NTY3ODkwMTIzNDU2Nzg5MDEyMzQ1Njc4OTEyMzQ1Njc4",

"refreshtokenexpires\_in":7776000

}

Dans les scripts Python, la bibliothèque **json** Python peut être utilisée pour analyser JSON dans les structures de données natives Python, et sérialiser les structures de données en tant que JSON. La bibliothèque Python **yaml** peut être utilisée pour convertir les données en YAML.

Le programme suivant utilise les deux modules pour analyser les données JSON ci-dessus, extraire et imprimer les valeurs de données, et générer une version YAML du fichier. Il utilise la méthode **json** library **loads ()** pour analyser une chaîne dans laquelle le fichier a été lu. Il utilise ensuite des références de données Python normales pour extraire des valeurs de la structure de données Python résultante. Enfin, il utilise la fonction **yaml** library **dump ()** pour sérialiser les données Python en tant que YAML, vers le terminal.

### Créez un script pour analyser les données JSON.

* + - 1. Ouvrez le fichier **parsejson.py** situé dans le répertoire **~/labs/devnet-src/parsing**.
      2. Importez les bibliothèques **json** et **yaml**.

importer json

importer yaml

* + - 1. Utilisez l'instruction Python **with** pour ouvrir **myfile.json** et définissez-le sur le nom de variable **json\_file**. Ensuite, utilisez la méthode **json.load** pour charger le fichier JSON dans une chaîne définie sur le nom de variable **ourjson**.

**Remarque** : Il n'est pas nécessaire de fermer explicitement le fichier car l'instruction **with** assure une ouverture et une fermeture correctes du fichier.

with open('myfile.json','r') as json\_file:

ourjson = json.load(json\_file)

* + - 1. Ajoutez une instruction print pour **ourjson** pour voir qu'il s'agit maintenant d'un dictionnaire Python.

print(ourjson)

### Exécutez le script pour imprimer les données JSON, puis modifiez-la pour imprimer les données d'intérêt.

* + - 1. Enregistrez et exécutez votre script. Vous devriez voir la sortie suivante.

devasc@labvm:~/labs/devnet-src/parsing$ **python3 parsejson.py**

{'access\_token': 'ZDI3MGEyYzQtNmFlNS00NDNhLWFlNzAtZGVjNjE0MGU1OGZmZWNmZDEwN2ItYTU3', 'expires\_in': 1209600, 'refresh\_token': 'MDEyMzQ1Njc4OTAxMjM0NTY3ODkwMTIzNDU2Nzg5MDEyMzQ1Njc4OTEyMzQ1Njc4', 'refreshtokenexpires\_in': 7776000}

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

* + - 1. Ajoutez des instructions d'impression qui affichent la valeur du jeton et le nombre de secondes jusqu'à l'expiration du jeton.

print("The access token is: {}".format(ourjson['access\_token']))

print("The token expires in {} seconds.".format(ourjson['expires\_in']))

* + - 1. Enregistrez et exécutez votre script. Vous devriez voir la sortie suivante.

devasc@labvm:~/labs/devnet-src/parsing$ **python3 parsejson.py**

{'access\_token': 'ZDI3MGEyYzQtNmFlNS00NDNhLWFlNzAtZGVjNjE0MGU1OGZmZWNmZDEwN2ItYTU3', 'expires\_in': 1209600, 'refresh\_token': 'MDEyMzQ1Njc4OTAxMjM0NTY3ODkwMTIzNDU2Nzg5MDEyMzQ1Njc4OTEyMzQ1Njc4', 'refreshtokenexpires\_in': 7776000}

1209600

The access token is ZDI3MGEyYzQtNmFlNS00NDNhLWFlNzAtZGVjNjE0MGU1OGZmZWNmZDEwN2ItYTU3

The token expires in 1209600 seconds

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

### Affiche les données JSON analysées dans un format de données YAML.

* + - 1. Ajoutez une instruction d'impression qui affichera les trois tirets requis pour un fichier YAML. Les deux **\ n** ajouteront deux lignes après la sortie précédente. Ensuite, ajoutez une instruction pour imprimer **ourjson** en tant que données YAML en utilisant la méthode **dump ()** de la bibliothèque **yaml**.

print("\n\n---")

print (yaml.dump (ourjson))

* + - 1. Enregistrez et exécutez votre script. Vous devriez voir la sortie suivante.

devasc@labvm:~/labs/devnet-src/parsing$ **python3 parsejson.py**

<output from previous steps omitted>

---

access\_token: ZDI3MGEyYzQtNmFlNS00NDNhLWFlNzAtZGVjNjE0MGU1OGZmZWNmZDEwN2ItYTU3

expires\_in: 1209600

refresh\_token: MDEyMzQ1Njc4OTAxMjM0NTY3ODkwMTIzNDU2Nzg5MDEyMzQ1Njc4OTEyMzQ1Njc4

refreshtokenexpires\_in: 7776000

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

## Analyser YAML en Python

Le programme suivant importe les bibliothèques **json** et **yaml**, utilise PyYAML pour analyser un fichier YAML, extraire et imprimer des valeurs de données et générer une version JSON du fichier. Il utilise la méthode **yaml** library **safe\_load ()** pour analyser le flux de fichiers et les références de données Python normales pour extraire les valeurs de la structure de données Python résultante. Il utilise ensuite la fonction **json** library **dumps ()** pour sérialiser les données Python en tant que JSON.

L'exemple YAML à analyser est le même fichier YAML que vous avez produit dans la partie 3 :

---

access\_token: ZDI3MGEyYzQtNmFlNS00NDNhLWFlNzAtZGVjNjE0MGU1OGZmZWNmZDEwN2ItYTU3

expires\_in: 1209600

refresh\_token: MDEyMzQ1Njc4OTAxMjM0NTY3ODkwMTIzNDU2Nzg5MDEyMzQ1Njc4OTEyMzQ1Njc4

refreshtokenexpires\_in: 7776000

### Créez un script pour analyser les données YAML.

* + - 1. Ouvrez le fichier **parseyaml.py** situé dans le répertoire **~/labs/devnet-src/parsing**.
      2. Importez les bibliothèques **json** et **yaml**.

importer json

importer yaml

* + - 1. Utilisez l'instruction Python **with** pour ouvrir **monfichier.yaml** et définissez-le sur le nom de variable **yaml\_file**. Ensuite, utilisez la méthode **yaml.safe\_load** pour charger le fichier YAML dans une chaîne définie sur le nom de variable **ouryaml**.

with open('myfile.yaml','r') as yaml\_file:

ouryaml = yaml.safe\_load(yaml\_file)

* + - 1. Ajoutez une instruction print pour **ouryaml** pour voir qu'il s'agit maintenant d'un dictionnaire Python.

print(ouryaml)

### Exécutez le script pour imprimer les données YAML, puis modifiez-la pour imprimer les données d'intérêt.

* + - 1. Enregistrez et exécutez votre script. Vous devriez voir la sortie suivante.

devasc@labvm:~/labs/devnet-src/parsing$ **python3 parseyaml.py**

{'access\_token': 'ZDI3MGEyYzQtNmFlNS00NDNhLWFlNzAtZGVjNjE0MGU1OGZmZWNmZDEwN2ItYTU3', 'expires\_in': 1209600, 'refresh\_token': 'MDEyMzQ1Njc4OTAxMjM0NTY3ODkwMTIzNDU2Nzg5MDEyMzQ1Njc4OTEyMzQ1Njc4', 'refreshtokenexpires\_in': 7776000}

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

* + - 1. Ajoutez des instructions d'impression qui affichent la valeur du jeton et le nombre de secondes jusqu'à l'expiration du jeton.

print("The access token is {}".format(ouryaml['access\_token']))

print("The token expires in {} seconds.".format(ouryaml['expires\_in']))

* + - 1. Save and run your script. You should see the following output.

devasc@labvm:~/labs/devnet-src/parsing$ **python3 parseyaml.py**

{'access\_token': 'ZDI3MGEyYzQtNmFlNS00NDNhLWFlNzAtZGVjNjE0MGU1OGZmZWNmZDEwN2ItYTU3', 'expires\_in': 1209600, 'refresh\_token': 'MDEyMzQ1Njc4OTAxMjM0NTY3ODkwMTIzNDU2Nzg5MDEyMzQ1Njc4OTEyMzQ1Njc4', 'refreshtokenexpires\_in': 7776000}

The access token is ZDI3MGEyYzQtNmFlNS00NDNhLWFlNzAtZGVjNjE0MGU1OGZmZWNmZDEwN2ItYTU3

The token expires in 1209600 seconds.

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

### Affiche les données YAML analysées dans un format de données JSON.

* + - 1. Ajoutez une instruction d'impression pour ajouter deux lignes vides après la sortie précédente. Ensuite, ajoutez une instruction pour imprimer **ouryaml** en tant que données JSON en utilisant la méthode **dumps ()** de la bibliothèque **json**. Ajoutez le paramètre indent pour préttifier les données JSON.

print("\n\n")

print (json.dumps (ouryaml, indent=4))

* + - 1. Sauvegardez et exécutez votre scénario. Vous devriez voir le résultat suivant. Notez que la sortie ressemble à **myfile.json**.

devasc@labvm:~/labs/devnet-src/parsing$ **python3 parseyaml.py**

<output from previous steps omitted>

{

"access\_token": "ZDI3MGEyYzQtNmFlNS00NDNhLWFlNzAtZGVjNjE0MGU1OGZmZWNmZDEwN2ItYTU3",

"expires\_in": 1209600,

"refresh\_token": "MDEyMzQ1Njc4OTAxMjM0NTY3ODkwMTIzNDU2Nzg5MDEyMzQ1Njc4OTEyMzQ1Njc4",

"refreshtokenexpires\_in": 7776000

}

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

Fin du document