## Lab 08:  Parse Different Data Types with Python

### Case Study

NetStream Networks, a next-generation telecommunications provider, offers high-speed internet and enterprise-grade fiber-optic connectivity across major urban centers. With over 1,000 skilled employees and a rapidly growing infrastructure, the company faced increasing challenges managing its complex networking environment. As the demand for uptime, scalability, and rapid configuration increased, NetStream recognized that its legacy practices, based on manual device access and siloed configuration files, could not support its vision for an agile, automated network.

The leadership team decided to adopt automation and modern data handling practices to enhance operational efficiency and streamline configuration management. Their goal was to leverage structured data formats, such as XML, JSON, and YAML, combined with Python scripting, to implement a standardized, automated, and centralized approach to network management.

### Business Challenge

NetStream’s network engineers were spending valuable time logging into each device individually to configure routers, perform audits, or retrieve operational data. The use of manual CLI commands resulted in configuration inconsistencies, human errors, and extended resolution times.

There was no centralized control to push bulk configuration changes or monitor device status in real time. Configuration backups were often managed ad hoc, and the lack of structured data made automation difficult. As more devices and cities were added, the operational overhead increased significantly, threatening scalability and service quality.

Furthermore, data extracted from network devices came in varying formats such as XML for device status, JSON from REST APIs, and YAML for configuration files. The absence of a standardized parsing and automation mechanism made integration between monitoring systems and configuration management tools highly inefficient.

### Solution

To improve data handling, NetStream adopted a Python-based automation approach to parse and convert XML, JSON, and YAML formats. This allowed the team to extract configuration parameters, validate settings, and transform data for integration with network tools and APIs.

The streamlined process enhanced visibility, reduced manual errors, and accelerated configuration tasks, laying the groundwork for advanced automation and consistent infrastructure management.

1. Parse XML in Python
2. Parse JSON in Python
3. Parse YAML in Python

|  |  |  |
| --- | --- | --- |
| **//Parse XML in Python**  1. Due to the flexibility of Extensible Markup Language (XML), parsing it can often be challenging. The all-text, tag-based structure of XML does not directly map to standard data types in Python or other commonly used programming languages. Additionally, it is not always clear how attribute values within XML should be interpreted or represented in code. However, in certain Cisco development environments, these challenges can be minimized. Cisco provides tools like YANG-CLI, which are specifically designed to handle XML data used in data modeling and related operations. For instance, the myfile.xml file located in **~/labs/devnet-src/parsing** demonstrates the type of XML data that YANG-CLI is capable of managing. You will use Python to parse this file and extract the valuable information it contains   |  | | --- | | <?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> |     2. To begin parsing the XML file, open the parsexml.py script located in the **~/labs/devnet-src/parsing** directory. First, import the ElementTree module from the xml library, which is used to parse XML data, and the re module for handling regular expressions. Then, use the parse() function from ElementTree to read the contents of myfile.xml and assign the resulting tree to a variable called xml. Retrieve the root element of the parsed XML structure using the getroot() method and store it in a variable named root. Since XML tags often contain namespaces, extract the namespace using a regular expression and store it in a variable ns. After that, search the root element for the <edit-config> tag using the namespace. Within that tag, locate and extract the values of the <default-operation> and <test-option> elements. Finally, print these values to display them. This approach allows Cisco developers to efficiently access key configuration values in XML, especially when working with tools like YANG-CLI.   |  | | --- | | import xml.etree.ElementTree as ET  import re  # Parse the XML file and get the root element  xml = ET.parse("myfile.xml")  root = xml.getroot()  # Extract the namespace using regular expression  ns = re.match('{.\*}', root.tag).group(0)  # Find the <edit-config> tag and its child elements  editconf = root.find(f"{ns}edit-config")  defop = editconf.find(f"{ns}default-operation")  testop = editconf.find(f"{ns}test-option")  # Print the values of <default-operation> and <test-option>  print("Default Operation:", defop.text)  print("Test Option:", testop.text) |     3. In the terminal session shown, the user begins by listing the contents of their home directory using the **ls** command and then navigates into the labs directory with **cd labs**. After verifying the contents of the labs directory using another **ls** command, they enter the devnet-src subdirectory using **cd devnet-src/.** Inside this directory, they again list the contents and locate the parsing folder, which they enter **using cd parsing/.** After listing the files in this folder with ls, the user identifies the script named **parsexml.py** along with the XML file myfile.xml. To parse the XML content and extract specific elements, they execute the script with the command **python3 parsexml.py.** The script runs successfully and prints the values of the <default-operation> and <test-option> XML tags, displaying the output: “The default-operation contains: **merge**” and “The test-option contains: **set**”. This confirms that the XML parsing was completed correctly and the desired information was retrieved. |
| **//Parse JSON in Python**  1. To parse JSON data and work with it in a Python environment, especially when interacting with REST APIs, a typical process involves retrieving a token through authentication, performing GET and POST (or PUT) requests, and handling JSON data. In this exercise, you will create a script that demonstrates how to parse JSON, extract key data points, and even convert the structure to YAML. First, open the parsejson.py file located in the **~/labs/devnet-src/parsing** directory. Begin by importing the json and yaml libraries, which are essential for working with JSON and converting it to YAML format. Then, use Python’s with statement to safely open the **myfile.json** file in read mode. Assign the opened file to a variable named json\_file, and use json.load(json\_file) to parse the JSON content into a Python dictionary stored in the variable ourjson. The with statement handles file closing automatically, making the code more robust and cleaner. To verify the structure and content of the parsed data, add a print statement to display the value of ourjson. This confirms that the JSON data has been successfully parsed and is ready for further manipulation, such as extracting specific values or converting it into YAML.   |  | | --- | | # Fill in this file with the code from the parsing JSON exercise  import json  import yaml  with open('myfile.json','r') as json\_file:  ourjson = json.load(json\_file)  print(ourjson) |     2.  Save and run your script. You should see the following output.    3. Add print statements that display the token value and how many seconds until the token expires.   |  | | --- | | # Fill in this file with the code from the parsing JSON exercise  import json  import yaml  with open('myfile.json','r') as json\_file:  ourjson = json.load(json\_file)  print(ourjson)  print("The access token is: {}".format(ourjson['access\_token']))  print("The token expires in {} seconds.".format(ourjson['expires\_in'])) |     4. Save and run your script. You should see the following output.    5. Add a print statement that will display the three dashes required for a YAML file. The two **\n** will add two lines after the previous output. Then add a statement to print **ourjson** as YAML data by using the **dump()** method of the **yaml** library.   |  | | --- | | # Fill in this file with the code from the parsing JSON exercise  import json  import yaml  with open('myfile.json','r') as json\_file:  ourjson = json.load(json\_file)  print(ourjson)  print("The access token is: {}".format(ourjson['access\_token']))  print("The token expires in {} seconds.".format(ourjson['expires\_in']))  print("\n\n---")  print(yaml.dump(ourjson)) |     6. Save and run your script. You should see the following output. |

|  |  |  |  |
| --- | --- | --- | --- |
| ***//* Parse YAML in Python**  1. To parse YAML data and convert it into a JSON structure in Python, you will create a script that reads the YAML file, processes its contents, and then serializes it to JSON. Start by opening the **parseyaml.py** file located in the **~/labs/devnet-src/parsing** directory. Begin by importing the json and yaml libraries, which are essential for working with both data formats. Use Python’s with statement to open the **myfile.yaml** file in read mode, assigning it to the variable yaml\_file. Then, apply the yaml.safe\_load() method to parse the YAML content into a native Python data structure, and store the result in a variable called ouryaml. This ensures safe and efficient parsing. To confirm that the file has been correctly read and parsed into a Python dictionary, add a print() statement for ouryaml. This script forms the foundation for further processing, such as converting the YAML data into JSON using json.dumps() in later steps.   |  | | --- | | import json  import yaml  # Open and parse the YAML file  with open('myfile.yaml', 'r') as yaml\_file:  ouryaml = yaml.safe\_load(yaml\_file)  # Print the parsed Python dictionary  print(ouryaml) |     2. Save and run your script. You should see the following output.    3. Add print statements that display the token value and how many seconds until the token expires.   |  | | --- | | Fill in this file with the code from the parsing YAML exercise  import json  import yaml  with open('myfile.yaml','r') as yaml\_file:  ouryaml = yaml.safe\_load(yaml\_file)  print(ouryaml)  print("The access token is {}".format(ouryaml['access\_token']))  print("The token expires in {} seconds.".format(ouryaml['expires\_in'])) |     4. Save and run your script. You should see the following output.    5. Add a print statement to add two blank lines after the previous output. Then add a statement to print **ouryaml** as JSON data by using the **dumps()** method of the **json** library. Add the indent parameter to prettify the JSON data.   |  | | --- | | # Fill in this file with the code from the parsing YAML exercise  import json  import yaml  with open('myfile.yaml','r') as yaml\_file:  ouryaml = yaml.safe\_load(yaml\_file)  print(ouryaml)  print("The access token is {}".format(ouryaml['access\_token']))  print("The token expires in {} seconds.".format(ouryaml['expires\_in']))  print("\n\n")  print(json.dumps(ouryaml, indent=4)) |     6. Save and run your script. You should see the following output. Notice that the output looks just like the **myfile.json.** |