Lab – Using the APIC-EM Path Trace API (Instructor Version)

**Instructor Note**: Red font color or gray highlights indicate text that appears in the instructor copy only.

1. Objectives

In this lab, you will create a program interacts with APIC-EM Path Trace application.

1. Background / Scenario

In this lab, you will utilize the APIC-EM API to perform a Path Trace by creating a program in Python, and by using the functions that have been created in the workshop.

The APIC-EM includes a robust Path Trace application that is accessible through the API. We will complete a program that does the following:

* Displays lists of all of the hosts and network devices on the APIC-EM network.
* Accepts user input for the source and destination devices for the path trace.
* Initiates the path trace.
* Monitors the status of the path trace until it has been completed by the APIC-EM.
* Displays details of the completed path trace to the user.

As we know, software developers frequently collaborate with others in the community to share code and access solutions in order to work more efficiently. In this lab, you are encouraged to seek help from others and refer to code that you have already developed. In addition, if necessary, solution code is available that you can explore in order to perform the tasks in this lab.

**Note:** In order to achieve the objectives of this workshop, the Python code you will work with has been simplified. This simplification may result in code that is less efficient, and less eloquent, than that created by professional software developers. As you gain skills in software development and network programmability, you will discover techniques that improve the usability of code. You are encouraged to explore the exemplar Path Trace solution that is available for download from GitHub to experience some of these techniques. This file and other tools can be found here: <https://github.com/CiscoDevNet/apic-em-samples-aradford>.

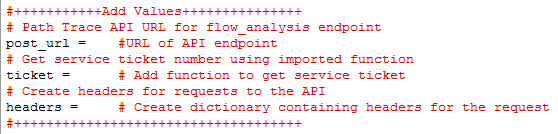
1. Required Resources

* Access to the APIC-EM in the DevNet sandbox at <https://sandboxapicem.cisco.com>
* IDLE Python IDE
* Python modules:
* json
* requests
* sys
* time
* tabulate
* module of functions previously created in the workshop or the apic\_
* JSON sample data file json\_data.json

1. Procedure

You will work to complete a small Python application that communicates with the APIC-EM Path Trace application. You are provided with two Python programs, the file that you will work in and the solution file. Depending on your familiarity with Python, you may want to have both files open. As we work through the lab, you will finish functional blocks of code that work together as the Python application. In order to run and test your working code, you will copy each completed code block into a new Python file of your own. As the code is completed, you can run the code blocks that you have finished in this file by copying and pasting them in. In this way you will compile the application from the completed code.

Work between the instructions in the lab and the lab code file named **04\_path\_trace.py**. You will add your own code to the code provided in that file. In order to locate the places were your input is required, look for comments surrounded by "+" characters, as shown in the example below.



In the **04\_path\_trace\_sol.py** solution file, the areas of completed code are highlighted similarly, however "Add Values" has been removed from the comments because this code is complete.

* + 1. Gather Requirements
       1. Open the APIC-EM sandbox and access the API documentation page. Use this URL: <https://sandboxapicem.cisco.com> Username: **devnetuser**, P/W: **Cisco123**!
       2. Open the documentation for the Flow Analysis endpoint. Look at the model schema for the **/flow-analysis** **POST** method.
       3. Make note of the schema for the request body. You will need to create a variable to hold dictionary key-value pairs for the source and destination IP addresses only.
       4. Look at the response class model schema. You will need to parse the returned JSON for the **flowAnalyisId** key of the response object. You will use this value to monitor the progress of the path trace and request the data for the completed path trace from the APIC-EM. A number of different path traces may have been running on the APIC-EM. The **flowAnalysisID** identifies the specific path trace that we have requested.
    2. Setup the Code Environment

Open the **04\_path\_trace.py** work file in IDLE. Refer to **Section 1** of the code. Import code modules and initialize variables to be used in the program.

* + - 1. Import the modules required for this application using the **import** and **from…import** commands, as was done previously. The list of required modules appears above.

**Note:** If you were unable to create the functions from earlier labs, import the **apic\_em\_functions\_sol** module.

* + - 1. Disable SSL certificate warnings.
      2. Create the **post\_url** variable and assign it the URL of the Flow Analysis endpoint of the API. The base URL is **https://sandboxapicem.cisco.com/api/v1**.
      3. Add code to get a fresh service ticket using the function you created earlier. Assign the returned value to the variable **ticket** to be used in the POST request headers.
      4. Create a dictionary called **headers** to hold the JSON header information that will be supplied to the POST request, as was done in previous labs.
    1. Display the Inventory of Hosts and Network Devices

Refer to **Section 2** of the code. In this step, you will reuse the functions that you created previously to display lists of the available hosts and network devices that may be included as endpoints in a Path Trace.

* + - 1. Display a message that tells the user what is being displayed followed by the function that displays the list of hosts in the topology on a new line. Follow this pattern:

print("**string**")

**function\_name**()

* + - 1. Display a message that tells the user what is being displayed followed by the function that displays the list of network devices in the topology.
      2. Test your code so far. Copy and paste parts one through three of the code you have completed so far into a new IDLE document, save the file to a temporary location and then run the code.

What should happen?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

The messages you created followed by tables of hosts and network devices that exist in the APIC-EM topology should be displayed.

* + - 1. Copy and paste the code you completed in Steps 2 and 3 into a new IDLE document. Run the code to verify that it works. If you encounter errors that you can't immediately identify and repair, you can compare your work to the solution code or ask your neighbor for help with figuring out what went wrong.
    1. Request the Path Trace Source and Destination IP Addresses

Refer to **Section 3** of the code. As you can see from the Path Trace API documentation, requests to the **/flow-analysis** API can accept a number of values. For this lab, you will only be using source and destination IP addresses, but the additional parameters can be added by modifying the JSON that is submitted to the API.

* + - 1. Prompt the user for the required IP addresses and assign the input to the **s\_ip** and **d\_ip** variables following the pattern below:

**variable** = input("**prompt string:** ")

* + - 1. Remember that you are working within a **while** loop structure. Indentation is important. Be sure that your level of indentation is consistent with other code that is present in the **while** loop.

Look at the **if…else** condition that is used to evaluate the user input. What condition is it testing?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

It is testing that something was entered at each prompt. If the user typed nothing and then pressed the <Enter> key, for either value, a message would prompt the user to make entries and then the code would be loop again due to the **continue** command.

This condition is meant to trap errors in the entries supplied by the user. What other errors could the code detect in the user input in order to ensure that only valid IP addresses have been entered?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Since the user should only enter IP addresses, the code could evaluate various features of IP addresses such as the presence of four octets separated by periods and that the values in octets are legal. Ideally, the code would look at a list of available IPs that is obtained from the inventory API and verify that each supplied address is present in that list. Due to time constraints, this functionality is not addressed in this activity.

* + - 1. Additional challenge: Add two statements that will get the values of the **path\_data** dictionary keys and print out a message that verifies the source and destination IP addresses that will be posted to the APIC-EM Path Trace endpoint.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

print('Source IP address: ' + path\_data['sourceIP'])

print(Destination IP address: ' + path\_data['destIP'])

* + - 1. Copy your completed code to the Python file you created in the step above. Run, test, and debug the code. Try entering IP addresses at the prompts and try skipping one of the entries by pressing the <Enter> key without entering an address. What should happen?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

When IP addresses are entered, nothing extra will be displayed on the screen unless the challenge in the step above has been completed. If nothing has been entered at either prompt, the error message should be displayed and the prompt for the source IP should be displayed again.

* + 1. Initiate the Path Trace and Get the Flow Analysis ID

Refer to **Section 4** of the code. In order to initiate the Path Trace, the **requests.post()** method is used to submit the source and destination IP addresses that have been submitted by the user to the **/flow-analysis** API. The **request.post()** method works similarly to the other requests module methods. We pass the method the four parameters with the variables shown in the table below:

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Variable/Parameter Name** | **Explanation** |
| URL | path\_url | The URL of the API endpoint |
| body | path | JSON formatted data for the source and destination IP addresses |
| headers | headers | The content type and authentication token represented as JSON data. |
| verification | verify | Used to verify the server’s security (TLS) certificate. False means the certificate is not verified. |

* + - 1. In the last step we requested the source and destination addresses for the Path Trace from the user. The values were stored in a dictionary held by the **path\_data** variable. We need to create a the variable from this data. However, the Python dictionary needs to be converted to JSON format. This is done with the **json.dumps()** method, which takes the object to be converted as its argument. In this case, that object is the **path\_data** dictionary.

Create the **path** variable and assign the converted **path\_data** dictionary variable to it. The pattern is:

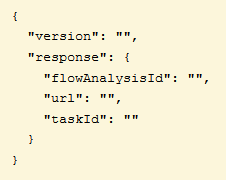
variable\_json = json.dumps(**dictionary**)

* + - 1. Build the **requests.post()** method to submit the path trace request, and store the response in the **resp** variable. Look at the example of the use of the **requests.post()** method from the **get\_ticket** code for guidance and use the information in the table above. You supply the correct variables to the statement. The pattern is:

**response\_variable** = requests.post(**URL**,**body**,headers=**variable**,verify=False)

* + - 1. The important element of the returned data is the flow analysis id, which we need to use to check the status of the path trace request and to get the completed path trace data from the APIC-EM. We will use the **resp\_json** variable that has been created in the code to translate the JSON into a Python dictionary from which we can extract the value we need.

Look at the structure of the JSON that is returned by the API.



From this we can see that that the JSON exists at two levels. The version: and response: objects are at the first level. At the second level, we can see the flowAnalysisId and other values. Our job is to access the value for the **flowAnalysisId** object and assign it to a variable that we call **flowAnalysisId**. Write a line of code that follows the pattern below:

flowAnalysisId = resp\_json[**"level one key**"]["**level 2 key**"]

Run your code and fix any errors that may appear. What should be output to the screen at this point in the program?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

In addition to what has been displayed previously, the flow analysis ID should also be displayed.

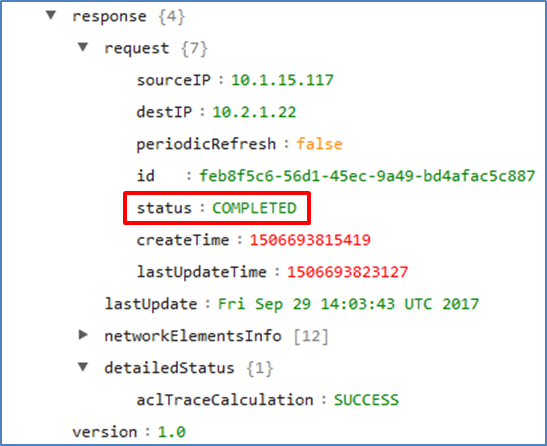
* + - 1. Test your code as directed above.
    1. Check Status of the Path Request

Refer to **Section 5** of the code. The path trace request that we submitted in the previous step only begins the path trace process and returns the flow analysis ID. The path trace can take time to execute depending on the size of the network and other factors.

In order to display the results of the path trace we have to wait for the process to complete before we can receive the results. To do this, we establish a **while** loop that will repeatedly request the status of the trace. The loop will execute until the API returns a status of ‘COMPLETED.’ The code within the loop repeatedly examines the status key in the data returned by the API, and acts accordingly. The status of the request can be COMPLETED, FAILED, or INPROGRESS. If the status is COMPLETED, the program moves on to the next section of code, which will display the results of the path trace. If it is FAILED, a message is displayed and the program terminates. In addition, a counter is set that is used to exit the loop if a predetermined number of iterations is exceeded. This is created to avoid an endless loop. Because a one-second timer is set within the loop to give the APIC-EM time to complete the path trace, the counter variable roughly corresponds to the amount of time allowed for the loop to complete. If it takes too long for the path trace to complete, then an error may exist. The threshold value for the counter can be increased in the code according to network conditions.

**Note:** Not all devices with IP addresses can successfully function as endpoints for a path trace. If your path trace fails, try using only host addresses as endpoints.

* + - 1. The URL for requests to the **\flow-analysis** end point can include the flow analysis id, as shown in the Swagger documentation for Flow Analysis endpoint. Since multiple path traces could be occurring simultaneously, this allows us to receive data about a specific request. Construct a new variable, called **check\_url** that contains the URL to the flow analysis endpoint, created in Section 1, followed by a back slash and the flow analysis id that was received from the API and put into the variable in the previous step.
      2. The JSON that is returned by the Path Trace API is "deep." That is, it can contain four or more levels of objects, some with multiple arrays. Luckily, the status value that we need to evaluate in order to determine the status of our request is only three levels in. Look at the example below:



The screen shot shows the high-level structure of the JSON with details of the **request** object expanded. We are interested in the **status** object shown. If we move down the tree from the highest level object, we could express the path to the status object as *response->request->status*. Assume that we will put the response into a Python dictionary variable called **response\_json**, as shown in the code. We want to extract the value of the status object from **response\_json** and put it into a variable called **status**. Construct the line of code that will accomplish this task below, and enter the line into your program. Consult previous examples and check with your neighbor if you want to verify your answer first.

status = \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

status = response\_json["response"]["request"]["status"]

* + - 1. Save the code and copy and paste everything that has been completed up to this point into a new Python file and run it. Before running it, what do you expect to see displayed on the screen as the code runs?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

In addition to what has already be displayed, the status of the request, INPROGRESS should repeatedly be displayed until the request is completed, fails, or exceeds the permitted number of iterations.

* + - 1. Leave the Python shell open.
    1. Display Results of Path Trace

Refer to **Section 6** of the code. If all has gone right you have successfully posted a path trace request by supplying source and destination IP addresses that come from lists of devices that are on the network. You will have received a flow analysis ID and watched the status of the path trace continuously update until the trace is complete. However, nothing has yet been displayed regarding the results of the trace itself.

We know that the JSON that is returned for path trace can be complex. An example of the full JSON is available in a text file called **json\_data.json** that is included with the files for this workshop. This JSON data poses several challenges.

* The JSON is long with many nested objects and arrays.
* The information returned for each device on the path varies, with some objects being present for some devices but not for others.
  + - 1. Open the **path\_trace\_json.json** file in a text editor, select the entire contents of the file, and paste it into [JSON Viewer](https://codebeautify.org/jsonviewer).
      2. Generate a tree view of the JSON and answer the following questions.
         1. The top level objects in the JSON are **"response":** and **"version":** (The **object** and **array** levels appear by default in all trees.) What four objects make up the next level of objects under **"response":**?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

The next level of objects is **"request":**, **"lastUpdate":**, **"networkElementsInfo":**, and **"detailedStatus":**.

* + - * 1. How many objects are there in the array under **"networkElementsInfo":** and what does each object represent?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

In the example file there are twelve objects in the array. Each object contains keys for different features of the devices returned by the path trace.

* + - * 1. Look at the keys for the devices on the path. What keys are found in some objects but not others?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Some objects lack keys like **"name":** and **engressInterface":**, for example. Other keys are also missing.

* + - 1. In IDLE, create a new file, and save the file in the folder that contains the files for this workshop. Name the file as you wish. In this file, please do the following:
         1. Import the **json** library.
         2. Open the **path\_trace\_json.json** file, convert it to Python objects, and assign the result to a variable called **json\_data**. Follow the pattern below:

**variable** = json.load(open("**filename.extension**"))

* + - * 1. Save and run the program. The program should run and provide you with a Python shell. You will be working in this Python shell. The **json\_data** variable will be available in the shell. If you close the shell, you must re-run the code to continue.
      1. Display the contents of **json\_data** in the shell with **print()**. This is what the imported and converted JSON looks like to Python.

We will now access different JSON objects to get the information we need.

* + - 1. Display the contents of the **"request":** object that is under the **"response":** key. Follow the pattern below:

>>> print(json\_data['object1']['object2'])

* + - 1. Now follow the same pattern, but print the source and destination IP addresses that were posted to the API. This requires references to another level of keys under the **"request"** key. Check the JSON structure in the JSON Viewer for the names of the keys. You will use this code in a program later.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

>>> print(json\_data['response']['request']['sourceIP'])

>>> print(json\_data['response']['request']['destIP'])

* + - 1. The data that we need to display for the results of the path trace is in the **networkElementsInfo** JSON object. This object contains an array, or list of objects, that represent all of the objects on the path from source to destination. We want to extract values from each object and display them to user after the path trace has successfully completed. Since this object is an array, in Python we need to refer to a combination of dictionary keys and list elements in order to get the information we need for each device.

Create a variable called **networkElementsInfo** and assign it the contents of the **networkElementsInfo** object. Display the contents of this variable in the shell.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

>>> networkElementsInfo = json\_data['response']['networkElementsInfo']

>>> **print(networkElementsInfo)**

This variable contains a list that contains dictionaries for each device. To verify this, use the Python **len()** function with this variable as its argument to display the number of elements in the list. This should match the number of elements shown in the JSON viewer.

* + - 1. Return to the path trace code file. Based on what you did above, supply values for the **path\_source**, **path\_dest**, and **networkElementsInfo** variables. Leave the shell window open.
      2. Return to the shell. You will continue to explore accessing values in the JSON data.
         1. You can address slices of the list that represent a single device or multiple devices. Let's display the information for the first device by displaying the first element of the **devices** variable. Remember that the first element of a list is at position 0. Repeat this for other elements in the list of devices. Follow this pattern:

list\_variable[*# of element***]**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

>>> networkElementsInfo[0]

* + - * 1. Now, address individual keys in the device list elements. This requires you to add a reference to the key for the information that you want to see. Try the "id" key of the first element. Follow this pattern:

list\_variable[**# of element**][**"key"**]

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

>>> networkElementsInfo[0]["id"]

**Note:** Some objects lack some keys like **"name"**, and **"egressInterface",** for example. Other keys are also present for some devices, but not for others.

What happens if you attempt to access a key that is not present for the device? For example, try to get the value of the "name" key for the first device. If this code was running in a program, what would happen?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

It generates an error. If this was running in code, the program would crash.

* + - 1. Experiment with accessing other keys for the devices. For example, enter an expression that displays the **"name"** value for the **"physicalInterface"** key of the **"ingressInterface"** key of one of the switches or routers in the list of devices.

To do this, inspect the JSON data either in the data file or JSON viewer. Find a switch or router in the list by looking at the "type" attribute. Note the position of the device in the list. Now, access the slice of the device variable for that position. Then, add keys for the nested dictionaries down to the "name" attribute for the physical ingress interface as shown below:

>>> devices[9]['ingressInterface']['physicalInterface']['name']

* + - 1. The last section of the **path\_trace.py** code creates a list of the information that we want to display for each device present in the path trace. This list is passed to the **tabulate()** function, along with a list of the column headings to use for each element in the list. Look at the code and answer the following questions.
         1. What values are extracted from the JSON and printed for each device in the path?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

This can be found in in a number places in the code. The variables that are assigned to the device variable, for example, are an ordinal number for the device, the name for the device, IP address, and ingress and egress interfaces. This can also be seen in the list of headings supplied to the tabulate() function.

* + - * 1. What is the purpose of the **for… in…** loop?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

The loop cycles through each element in the list of devices in the networkElementsInfo key. Within the loop, values are assigned to variables for value to be reported for the device. This list of values is appended to the all\_devices variable which is passed to tabulate() and printed.

* + - * 1. Why are so many **if** conditions required?

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Because of the variability in the structure of the objects that represent each device, the if conditions are used to capture missing keys and assign generic values for the corresponding variables. Without this, the program would crash and the list of devices would never be displayed.

* + 1. Summary

Now you should be able to run the entire path trace application. If you have been unable to complete all activities in this workshop, please inspect and run the various solution files that are distributed with this workshop.

1. For Additional Study

For an additional challenge, try the following activities. We encourage you to start a GitHub project and share your work with others. Get a few email addresses from other workshop participants, and work together to accomplish these challenge activities.

* + - 1. Format the output of the path trace for Graphziv. Instead of printing a table, print out a list of devices that is formatted for Graphviz. Run the file and paste the output into the online Graphviz viewer at <http://www.webgraphviz.com/>. Check the Graphviz documentation for formatting information.
      2. Find a partner. Create a new GitHub project and work together, either in class or remotely, to further modularize the code. For example, create generic functions that execute **requests.post()** and **request.get()** methods and return the results of the post to the calling code. This will require passing parameters to the functions. These functions can be saved in the **apic\_em\_functions.py** file and called from the **04\_path\_trace.py** file.
      3. Create a function that validates IP addresses that have been input by users.
      4. Create code that checks if an IP address input by a user is present in the host and device inventory.