## Lab 07:  Create a Python Unit Test

### Case Study

TechSphere Networks, a growing network infrastructure enterprise, encountered issues managing the increasing complexity of its network device data. Their team manually tracked information like routers, switches, and appliances using plain text files. As device deployments expanded across different locations, inconsistencies, data duplication, and formatting errors became common. These manual methods created inefficiencies, made data verification difficult, and delayed important tasks. The organization recognized the need for a more structured and automated approach to manage and validate its growing datasets effectively.

### Business Challenge

TechSphere Networks faced operational bottlenecks due to their reliance on manually maintained device logs. The manual system resulted in duplicated entries and errors, and it demanded technical expertise that only a few staff members possessed. As a result, updates were infrequent, collaboration across departments was limited, and data accuracy suffered. With an increasing need to maintain device records cleanly and consistently, TechSphere sought a reliable solution that could reduce errors, standardize formatting, and enable even non-technical users to contribute to data management efficiently.

### Solution

To resolve these issues, TechSphere implemented a Python-based solution that combined structured data (in JSON format) with a recursive search function tested using the unittest framework. The lab-guided approach involved creating a recursive function to extract key-value pairs from nested JSON data, common in network automation.PIs. The solution was tested thoroughly using unit tests to validate whether the function worked as expected, detected valid keys, ignored invalid ones, and consistently returned correct outputs. By automating validation and structuring the data access logic, TechSphere improved data reliability, reduced human errors, and enabled wider staff participation in maintaining device records, laying the foundation for scalable automation in their IT processes.

Follow these steps to complete the lab:

1. Explore Options in the unittest Framework
2. Test a Python Function with unittest

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **// Explore Options in the unittest Framework**  1. Launch the DEVASC VM.  2. Double-click on the terminal icon to open it.    3. Python’s unittest framework, part of its standard library, allows developers to build and run tests efficiently. To define an individual unit test, the framework provides the TestCase class, which acts as the foundation for creating test cases by subclassing it.  To identify test methods within a test class, the framework employs a naming convention that automatically recognizes any method beginning with 'test\_' as a test method by the test runner.  The test runner, which executes test cases and reports results, can be invoked from the command line. To view all available command-line options supported by unittest, use the command **python3 -m unittest -h.**  .  **//Test a Python Function with unittest**  1. To test a function that performs a recursive search through a JSON object. The goal of the function is to return values associated with a specified key, a common task when dealing with JSON responses from APIs. To carry out this test, you will work with three files: **recursive\_json\_search.py**, which contains the **json\_search()** function to be tested; **test\_data.py**, which provides the sample **JSON data**; and **test\_json\_search.py**, where the actual unit tests will be written.  2. Review the contents of the test\_data.py file located in ~/labs/devnet-src/unittest/ using command **more unittest/test\_data.py.** This file contains structured JSON data that closely resembles responses from Cisco’s DNA Center API. It includes a mix of dictionaries and lists, making it an ideal candidate for testing recursive search functionality. The complexity of this data allows for a realistic validation of the function’s ability to handle nested structures and various data types.    3. To create the **json\_search()** function that will be tested, start by designing it to accept two input parameters: a **key** to search for and a **JSON object** to search within. The function will perform a recursive traversal through the JSON structure and collect all key-value pairs that match the provided key. First, the test data is imported from the **test\_data.py** file. The function then checks whether the input is a dictionary or a list. If it encounters the specified key, it appends the key and its corresponding value as a dictionary to the results list. If the value itself is a dictionary or list, the function recursively searches through those as well. At the end of the script, a **print()** statement displays the results for the sample key **key1 = "issueSummary"** to verify that the function is returning the expected output.  To implement this, open the **recursive\_json\_search.py** file located in **~/labs/devnet-src/unittest/,** paste the provided function code into the file, and save it. Ensure that line breaks and inline comments are formatted correctly to maintain valid Python syntax. This function serves as the foundation for your upcoming unit tests.   |  | | --- | | from test\_data import \*  def json\_search(key,input\_object):  ret\_val=[]  if isinstance(input\_object, dict): # Iterate dictionary  for k, v in input\_object.items(): # searching key in the dict  if k == key:  temp={k:v}  ret\_val.append(temp)  if isinstance(v, dict): # the value is another dict so repeat  json\_search(key,v)  elif isinstance(v, list): # it's a list  for item in v:  if not isinstance(item, (str,int)): # if dict or list repeat  json\_search(key,item)  else: # Iterate over a list because some APIs return a JSON object in a list  For val in input\_object:  if not isinstance(val, (str,int)):  json\_search(key,val)  else: # Iterate a list because some APIs return a JSON object in a list  for val in input\_object:  if not isinstance(val, (str,int)):  json\_search(key,val)  return ret\_val  print(json\_search("issueSummary",data)) |     4. In terminal, Run the code using the command **python3 recursive\_json\_search.py.** You should get no errors and output of **[ ]** indicating an empty list. If the **json\_search()** function was coded correctly (which it is not), this would tell you that there is no data with the “issueSummary” key reported by JSON data returned by the Cisco DNA Center API. In other words, there are no issues to report.    5. To determine whether the json\_search() function is working correctly, you need to compare its output against known values manually verified in the test data. For example, by inspecting the **test\_data.py** file, you can visually confirm that the key "issueSummary" exists and that its value is:  **"Network Device 10.10.20.82 Is Unreachable From Controller"**  This known value becomes the expected result in your unit test. When you run the unit test and the json\_search() function returns the same value for the "issueSummary" key, you can be confident that the function behaves correctly for that case.  While this dataset is small and straightforward, real-world JSON data from production environments can be deeply nested and difficult to verify by hand. That's why automated unit tests are critical; they quickly alert you to bugs or unintended changes in behavior, especially when working with complex or frequently changing data.  Open **test\_data.py** and search for "issueSummary" using the command **more test\_data.py.**    6. To create a unit test for the json\_search() function, begin by opening the test\_json\_search.py file located in the **~/labs/devnet-src/unittest/** directory. In the first line after any initial comments, import the unittest library using **import unittest**, which provides the framework for writing and running tests.  7. Next, import the function and data you will be testing. To do this, add the following lines to the top of the script: **from recursive\_json\_search import \*** and **from test\_data import \*.** These lines import the json\_search() function and the JSON test data, including the keys defined in the **test\_data.py** file.  8. After the imports, define a test class that inherits from **unittest.TestCase**. Name the class **json\_search\_test**, and include three test methods within it. Each method name must begin with **test\_** so that the **unittest** framework can automatically detect them. The first method, **test\_search\_found**, checks that a known key exists in the data and that the result is not an empty list. The second method, **test\_search\_not\_found**, checks that a non-existent key returns an empty list. The third method, **test\_is\_a\_list**, ensures that the return type of the function is always a list. Each method includes a descriptive docstring enclosed in triple single quotes ('''), which helps display readable output during test execution.  Finally, add a conditional block at the end of the script to ensure the test runs only when the script is executed directly. This is done by including:   |  | | --- | | import unittest  from recursive\_json\_search import \*  from test\_data import \*  class json\_search\_test(unittest.TestCase):  '''test module to test search function in `recursive\_json\_search.py'''  def test\_search\_found(self):  '''key should be found, return list should not be empty'''  self.assertTrue([]!=json\_search(key1,data))  def test\_search\_not\_found(self):  '''key should not be found, should return an empty list'''  self.assertTrue([]==json\_search(key2,data))  def test\_is\_a\_list(self):  '''Should return a list'''  self.assertIsInstance(json\_search(key1,data),list)  if \_\_name\_\_ == '\_\_main\_\_':  unittest.main() |     9. Now in the terminal, run the test script in its current state using command **python3 test\_json\_search.py** to see what results it currently returns. First, you see the empty list. Second, you see the **. F.** highlighted in the output. A period (.) means a test passed, and an F means a test failed. Therefore, the first test passed, the second test failed, and the third test passed.    10. To list each test and its results, run the script again under unittest with the verbose (-v) option. Notice that you do not need the .py extension for the p**y** script. You can see that your test method **test\_search\_found,**is failing.    11. To fix the first error **key should be found, return list should not be empty … FAIL,** in the **recursive\_json\_search.py** script, open the file and examine the json\_search() function. The failure occurs because the line **ret\_val = []** is placed inside the function, which causes the list to reset every time the function is called recursively. As a result, previously collected matches are lost during recursion, and the final result appears empty. To resolve this, modify the function to accept **ret\_val** as an optional parameter with a default value of None. Inside the function, check if ret\_val is None, and if so, initialize it to an empty list. This ensures that only the first call starts with an empty list and all recursive calls reuse the same list to accumulate results. The corrected function should look like this: def json\_search(key, input\_object, ret\_val=None): followed by if ret\_val is None: ret\_val = [], and then include the recursive logic for handling dictionaries and lists. After saving the file, run the script using python3 recursive\_json\_search.py, and you should see a non-empty list returned, confirming that the key was successfully found and the issue has been resolved.   |  | | --- | | from test\_data import \*  ret\_val=[]  def json\_search(key,input\_object):  if isinstance(input\_object, dict): # Iterate dictionary  for k, v in input\_object.items(): # searching key in the dict  if k == key:  temp={k:v}  ret\_val.append(temp)  if isinstance(v, dict): # the value is another dict so repeat  json\_search(key,v)  elif isinstance(v, list): # it's a list  for item in v:  if not isinstance(item, (str,int)): # if dict or list repeat  json\_search(key,item)  else: # Iterate a list because some APIs return a JSON object in a list  for val in input\_object:  if not isinstance(val, (str,int)):  json\_search(key,val)  return ret\_val  print(json\_search("issueSummary",data)) |     12.  Save and run the script in terminal using command **python3 recursive\_json\_search.py**. You should get the following output, which verifies that you resolved the issue. The list is no longer empty after the script runs.    13. To ensure that all issues in the recursive\_json\_search.py script are resolved, you should run the unit tests again without using the **-v** (verbose) option. This helps minimize console output, allowing you to focus on test results more efficiently. Open your terminal and navigate to the **~/labs/devnet-src/unittest/** directory, then run the command python3 **test\_json\_search.py**. Observe the output at the beginning of the test log—if you see **..F**, it means two tests passed, and one failed. The **"F"** indicates a failure, most likely in the test that checks whether the function returns a list, which may point to a remaining issue in your function’s return structure. Additionally, if you still see output being printed to the screen (like a list of results), it is likely due to a lingering **print()** statement inside recursive\_json\_search.py. While this is not critical for completing the lab, you can remove or comment out the print() statement to clean up the console output during testing.    14. To correct the second error in the recursive\_json\_search.py script, you need to address the issue of using a global variable. In the previous step, the **ret\_val** list was moved outside the function to preserve values during recursion, but this unintentionally made it a global variable, causing results to persist across multiple function calls. This is bad practice because global variables can introduce bugs and unexpected behavior in larger applications. To fix this, you should refactor the function by encapsulating the recursive logic inside an inner function, which allows ret\_val to remain local to the main function while still being shared during recursion. To implement this, first delete the old **json\_search()** function, then replace it with the new version provided. This new function defines a local list **ret\_val = []** and a nested **inner\_function()** that performs the recursive search. The recursion is handled safely within the inner function, and ret\_val is only accessible within the outer json\_search() call. After defining the new function, you can test it by calling **print(json\_search("issueSummary", data))** to confirm it returns the expected result. This structure avoids global variables and follows good coding practices for recursion.   |  | | --- | | from test\_data import \*  def json\_search(key,input\_object):      """      Search a key from a JSON object, get nothing back if the key is not found      key: "keyword" to be searched, case sensitive      input\_object: JSON object to be parsed, test\_data.py in this case      inner\_function() is doing the recursive search      Return a list of key: value pair      """      ret\_val=[]      def inner\_function(key,input\_object):          if isinstance(input\_object, dict): # Iterate dictionary              for k, v in input\_object.items(): # searching key in the dict                  if k == key:                      temp={k:v}                      ret\_val.append(temp)                  if isinstance(v, dict): # the value is another dict so repeat                      inner\_function(key,v)                  elif isinstance(v, list):                      for item in v:                          if not isinstance(item, (str,int)): # if dict or list repeat                              inner\_function(key,item)          else: # Iterate a list because some APIs return JSON object in a list              for val in input\_object:                  if not isinstance(val, (str,int)):                      inner\_function(key,val)      inner\_function(key,input\_object)      return ret\_val  print(json\_search("issueSummary",data)) |     15. After saving the updated recursive\_json\_search.py file with the corrected json\_search() function that avoids using global variables, you can run the unit tests using Python’s unittest test discovery feature. To do this, open your terminal, navigate to the **~/labs/devnet-src/unittest**/ directory, and run the command **python3 -m unittest**. There is no need to specify the test file name because unittest automatically detects and runs all test files in the directory that start with test. If everything is set up correctly, you should see output indicating that all tests have passed, typically shown as three dots (...) followed by a message like **“Ran 3 tests in 0.001s”** and “**OK**.” This confirms that the function now behaves as expected and successfully returns the correct value for the "issueSummary" key. Since the **print()** function in recursive\_json\_search.py was used only for debugging and is not necessary for unit testing, you can now safely remove or comment it out to keep your output clean and professional during future test runs. |