Lab - Create a Python Unit Test

Objectives

Part 1: Launch the DEVASC VM

Part 2: Explore Options in the unittest Framework

Part 3: Test a Python Function with unittest

Background / Scenario

Unit tests examine independent units of code, like functions, classes, modules, and libraries. There are many reasons for writing a script using Python's **unittest** library. One obvious reason is that if you find an issue in isolated code by deliberate testing, you know that the problem is in the function or other unit under test. The problem is not in the larger application that may call this function. You will also know exactly what triggered the error because you wrote the unit test that exposed the issue. Bugs found this way are usually quick and easy to fix, and fixes made at this detailed level are less likely to cause unanticipated side effects later on in other code that relies on the tested code.

You can run unit tests manually if the code is small, but typically unit tests should be automated. When writing a unit test, think about the following:

- The unit test should be simple and easy to implement.
- The unit test should be well documented, so it's easy to figure out how to run the test, even after several years.
- Consider the test methods and inputs from every angle.
- Test results should be consistent. This is important for test automation.
- Test code should work independently of code being tested. If you write tests that need to change program state, capture state before changing it, and change it back after the test runs.
- When a test fails, results should be easy to read and clearly point out what is expected and where the issues are.

In this lab, you will explore the **unittest** framework and use **unittest** to test a function.

Required Resources

- 1 PC with operating system of your choice
- Virtual Box or VMWare
- DEVASC Virtual Machine

Instructions

Part 1: Launch the DEVASC VM

If you have not already completed the **Lab - Install the Virtual Machine Lab Environment**, do so now. If you have already completed that lab, launch the DEVASC VM now.

© 2020 - 2020 Cisco and/or its affiliates. All rights reserved. Cisco Public

Part 2: Explore Options in the unittest Framework

Python provides a Unit Testing Framework (called **unittest**) as part of the Python standard library. If you are not familiar with this framework, study the "Python unittest Framework" to familiarize yourself. Search for it on the internet to find the documentation at python.org. You will need that knowledge or documentation to answer questions in this part.

What **unittest** class do you use to create an individual unit of testing?

A test runner is responsible for executing tests and providing you with results. A test runner can be a graphical interface but, in this lab, you will use the command line to run tests.

How does the test runner know which methods are a test?

What command will list all of the command line options for **unittest** shown in the following output?

```
devasc@labvm:~/labs/devnet-src$
<output omitted>
optional arguments:
  -h, --help           show this help message and exit
  -v, --verbose        Verbose output
  -q, --quiet          Quiet output
  --locals             Show local variables in tracebacks
  -f, --failfast       Stop on first fail or error
  -c, --catch          Catch Ctrl-C and display results so far
  -b, --buffer         Buffer stdout and stderr during tests
  -k TESTNAMEPATTERNS  Only run tests which match the given substring

Examples:
  python3 -m unittest test_module               - run tests from test_module
  python3 -m unittest module.TestClass          - run tests from module.TestClass
  python3 -m unittest module.Class.test_method  - run specified test method
  python3 -m unittest path/to/test_file.py      - run tests from test_file.py
<output omitted>
For test discovery all test modules must be importable from the top level
directory of the project.
devasc@labvm:~/labs/devnet-src$
```

Part 3: Test a Python Function with unittest

In this part, you will use **unittest** to test a function that performs a recursive search of a JSON object. The function returns values tagged with a given key. Programmers often need to perform this kind of operation on JSON objects returned by API calls.

This test will use three files as summarized in the following table:

File	Description
recursive_json_search.py	This script will include the **json_search()** function we want to test.
test_data.py	This is the data the **json_search()** function is searching.

© 2020 - 2020 Cisco and/or its affiliates. All rights reserved. Cisco Public

File	Description
test_json_search.py	This is the file you will create to test the **json_search()** function in the recursive_json_search.py script.

Step 1: Review the test_data.py file.

Open the **~/labs/devnet-src/unittest/test_data.py** file and examine its contents. This JSON data is typical of data returned by a call to Cisco's DNA Center API. The sample data is sufficiently complex to be a good test. For example, it has **dict** and **list** types interleaved.

```
devasc@labvm:~/labs/devnet-src$ more unittest/test_data.py
key1 = "issueSummary"
key2 = "XY&^$#*@!1234%^&"

data = {
  "id": "AWcvsjx864kVeDHDi2gB",
  "instanceId": "E-NETWORK-EVENT-AWcvsjx864kVeDHDi2gB-1542693469197",
  "category": "Warn",
  "status": "NEW",
  "timestamp": 1542693469197,
  "severity": "P1",
  "domain": "Availability",
  "source": "DNAC",
  "priority": "P1",
  "type": "Network",
  "title": "Device unreachable",
  "description": "This network device leaf2.abc.inc is unreachable from controll
er. The device role is ACCESS.",
  "actualServiceId": "10.10.20.82",
  "assignedTo": "",
  "enrichmentInfo": {
    "issueDetails": {
      "issue": [
        {
--More--(12%)
```

Step 2: Create the json_search() function that you will be testing.

Our function should expect a key and a JSON object as input parameters, and return a list of matched key/value pairs. Here is the current version of the function that needs to be tested to see if it is working as intended. The purpose of this function is to import the test data first. Then it searches for data that matches the key variables in the **test_data.py** file. If it finds a match, it will append the matched data to a list. The **print()** function at the end prints the contents for the list for the first variable **key1 = "issueSummary"**.

```
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}
```

© 2020 - 2020 Cisco and/or its affiliates. All rights reserved. Cisco Public

```
                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 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))
```

a. Open the **~/labs/devnet-src/unittest/recursive_json_search.py** file.

b. Copy the code above into the file and save it.

   Note: If you are viewing this lab as a PDF file, then you may need to edit the line breaks to make the code valid. Note that the inline comments should not break to the next line. When pasted into the **recursive_json_search.py**, there should be 21 lines of code inclusive of the open comment **# Fill the Python code in this file**.

c. Run the code. 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.

```
devasc@labvm:~/labs/devnet-src/unittest$ python3 recursive_json_search.py
[]
devasc@labvm:~/labs/devnet-src/unittest$
```

d. But how do you know that the **json_search()** function is working as intended? You could open the **test_data.py** file and search for the key “issueSummary”, as shown below. If you did, you would indeed find that there is an issue. This is a small data set and a relatively simple recursive search. However, production data and code is rarely this simple. Therefore, testing code is vital to quickly finding and fixing errors in your code.

```
<output omitted>
      "issue": [
        {
          "issueId": "AWcvsjx864kVeDHDi2gB",
          "issueSource": "Cisco DNA",
          "issueCategory": "Availability",
          "issueName": "snmp_device_down",
          "issueDescription": "This network device leaf2.abc.inc is unreachable from
controller. The device role is ACCESS.",
          "issueEntity": "network_device",
          "issueEntityValue": "10.10.20.82",
          "issueSeverity": "HIGH",
          "issuePriority": "",
          "issueSummary": "Network Device 10.10.20.82 Is Unreachable From Controller",
          "issueTimestamp": 1542693469197,
          "suggestedActions": [
            {
```

© 2020 - 2020 Cisco and/or its affiliates. All rights reserved. Cisco Public

```
<output omitted>
```

Step 3: Create some unit tests that will test if the function is working as intended.

a. Open the **~ labs/devnet-src/unittest/test_json_search.py** file.

b. In the first line of the script after the comment, import the **unittest** library.

```
import unittest
```

c. Add lines to import the function you are testing as well as the JSON test data the function uses.

```
from recursive_json_search import *
from test_data import *
```

d. Now add the following **json_search_test** class code to the **test_json_search.py** file. The code creates the subclass **TestCase** of the **unittest** framework. The class defines some test methods to be used on the **json_search()** function in the **recursive_json_search.py** script. Notice that each test method begins with **test_**, enabling the **unittest** framework to discover them automatically. Add the following lines to the bottom of your ~labs/devnet-src/unittest/test_json_search.py file:

```
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)
```

In the **unittest** code, you are using three methods to test the search function:

1) Given an existing key in the JSON object, see if the testing code can find such a key.

2) Given a non-existent key in the JSON object, see if the testing code confirms that no key can be found.

3) Check if our function returns a list, as it should always do.

To create these tests, the script uses some of the built-in assert methods in the **unittest** TestCase class to check for conditions. The **assertTrue(x)** method checks if a condition is true and **assertIsInstance(a, b)** checks if **a** is an instance of the **b** type. The type used here is **list**.

Also, notice that the comments for each method are specified with the triple single quote (`'''`). This is required if you want the test to output a description of the test method when it runs. Using the single hash symbol (**#**) for the comment would not print out the description of a failed test.

e. For the last part of the script, add the **unittest.main()** method. This enables running **unittest** from the command line. The purpose of **if __name__ == '__main__'** is to make sure that the **unittest.main()** method runs only if the script is run directly. If the script is imported into another program, **unittest.main()** will not run. For example, you might use a different test runner than **unittest** to run this test.

```
if __name__ == '__main__':
    unittest.main()
```

© 2020 - 2020 Cisco and/or its affiliates. All rights reserved. Cisco Public

Step 4: Run the test to see the initial results.

a. Run the test script in its current state 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.

```
devasc@labvm:~/labs/devnet-src/unittest$ python3 test_json_search.py
 []
.F.
======================================================================
FAIL: test_search_found (__main__.json_search_test)
key should be found, return list should not be empty
----------------------------------------------------------------------
Traceback (most recent call last):
  File "test_json_search.py", line 11, in test_search_found
    self.assertTrue([]!=json_search(key1,data))
AssertionError: False is not true

----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures=1)
devasc@labvm:~/labs/devnet-src/unittest$
```

b. 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 **test_json_search.py** script. You can see that your test method **test_search_found** is failing.

Note: Python does not necessarily run your tests in order. Tests are run in alphabetical order based on the test method names.

```
devasc@labvm:~/labs/devnet-src/unittest$ python3 -m unittest -v
test_json_search
[]
test_is_a_list (test_json_search.json_search_test)
Should return a list ... ok
test_search_found (test_json_search.json_search_test)
key should be found, return list should not be empty ... FAIL
test_search_not_found (test_json_search.json_search_test)
key should not be found, should return an empty list ... ok

======================================================================
FAIL: test_search_found (test_json_search.json_search_test)
key should be found, return list should not be empty
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/devasc/labs/devent-src/unittest/test_json_search.py", line 11, in
test_search_found
    self.assertTrue([]!=json_search(key1,data))
AssertionError: False is not true

----------------------------------------------------------------------
Ran 3 tests in 0.001s
```

© 2020 - 2020 Cisco and/or its affiliates. All rights reserved. Cisco Public

```
FAILED (failures=1)
devasc@labvm:~/labs/devnet-src/unittest$
```

Step 5: Investigate and correct the first error in the recursive_json_search.py script.

The assertion, **key should be found, return list should not be empty ... FAIL**, indicates the key was not found. Why? If we look at the text of our recursive function, we see that the statement **ret_val=[ ]** is being repeatedly executed, each time the function is called. This causes the function to empty the list and lose accumulated results from the **ret_val.append(temp)** statement, which is adding to the list created by **ret_val=[ ]**.

```
def json_search(key,input_object):
    ret_val=[]
    if isinstance(input_object, dict):
        for k, v in input_object.items():
            if k == key:
                temp={k:v}
                ret_val.append(temp)
```

a. Move the **ret_val=[ ]** out of our function in **recursive_json_search.py** so that the iteration does not overwrite the accumulated list each time.

```
ret_val=[]
def json_search(key,input_object):
```

b. Save and run the script. You should get the following output which verifies that you resolved the issue. The list is no longer empty after the script runs.

```
devasc@labvm:~/labs/devnet-src/unittest$ python3 recursive_json_search.py
[{'issueSummary': 'Network Device 10.10.20.82 Is Unreachable From Controller'}]
devasc@labvm:~/labs/devnet-src/unittest$
```

Step 6: Run the test again to see if all errors in the script are now fixed.

a. You got some output last time you ran **recursive_json_search.py**, you cannot yet be sure you resolved all the errors in the script? Run **unittest** again without the **-v** option to see if **test_json_search** returns any errors. Typically, you do not want to use **-v** option to minimize console output and make tests run faster. At the start of the log you can see **..F**, meaning that the third test failed. Also notice that the list is still printing out. You can stop this behavior by removing the **print()** function in the **resursive_json_search.py** script. But that is not necessary for your purposes in this lab.

```
devasc@labvm:~/labs/devnet-src/unittest$ python3 -m unittest test_json_search
[{'issueSummary': 'Network Device 10.10.20.82 Is Unreachable From Controller'}]
..F
======================================================================
FAIL: test_search_not_found (test_json_search.json_search_test)
key should not be found, should return an empty list
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/home/devasc/labs/devnet-src/unittest/test_json_search.py", line 14, in
test_search_not_found
    self.assertTrue([]==json_search(key2,data))
AssertionError: False is not true
```

© 2020 - 2020 Cisco and/or its affiliates. All rights reserved. Cisco Public

www.netacad.com

```
----------------------------------------------------------------------
Ran 3 tests in 0.001s

FAILED (failures=1)
devasc@labvm:~/labs/devnet-src/unittest$
```

b. Open the **test_data.py** file and search for **issueSummary**, which is the value for key1. You should find it twice, but only once in the JSON **data** object. But if you search for the value of key2, which is **XY&^$#*@!1234%^&**, you will only find it at the top where it is defined because it is not in the **data** JSON object. The third test is checking to make sure it is not there. The third test comment states **key should not be found, should return an empty list**. However, the function returned a non-empty list.

Step 7: Investigate and correct the second error in the recursive_json_search.py script.

a. Review the **recursive_json_search.py** code again. Notice that the **ret_val** is now a global variable after you fixed it in the previous step. This means that its value is preserved across multiple invocations of the **json_search()** function. This is a good example of why it's bad practice to use global variables within functions.

b. To resolve this issue, wrap the **json_search()** function with an outer function. Delete your existing **json_search()** function and replace with the refactored one below: (It won't hurt to call the function twice but it's not best practice to repeat a function.)

```
from test_data import *
def json_search(key,input_object):
    """
    Search a key from JSON object, get nothing back if 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 actually 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))
```

© 2020 - 2020 Cisco and/or its affiliates. All rights reserved. Cisco Public

c. Save the file and run **unittest** on the directory. You do not need the name of the file. This is because the **unittest** Test Discovery feature will run any local file it finds whose name begins with test. You should get the following output. Notice that all tests now pass and the list for the "issueSummary" key is populated. You can safely delete the **print()** function as it would not normally be used when this test is aggregated with other tests for a larger test run.

```
devasc@labvm:~/labs/devnet-src/unittest$ python3 -m unittest
[{'issueSummary': 'Network Device 10.10.20.82 Is Unreachable From Controller'}]
...
----------------------------------------------------------------------
Ran 3 tests in 0.001s

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

© 2020 - 2020 Cisco and/or its affiliates. All rights reserved. Cisco Public

www.netacad.com