Práctica de Laboratorio 3 - Crear una prueba unitaria de Python

Objetivos

Parte 1: Iniciar la máquina virtual (Virtual Machine) de DEVASC.

Parte 2: Explorar las opciones en el unittest del framework.

Parte 3: Probar una función de Python con unittest.

Aspectos básicos/Situación.

Los unittest examinan unidades independientes de código, como funciones, clases, módulos y librerías. Hay muchas razones para escribir un script usando la librería **unittest** de Python. Una razón obvia es que si encuentra un problema en el código aislado mediante pruebas deliberadas, sabrás que el problema está en la función u otra unidad bajo prueba. El problema no está en la aplicación más grande que pueda llamar a esta función. También sabrá exactamente qué desencadenó el error porque el unittest escrito por el usuario expondrá el problema. Los errores encontrados de esta manera suelen ser rápidos y fáciles de corregir, y las correcciones hechas en este nivel detallado tienen menos probabilidades de causar efectos secundarios imprevistos más adelante en otro código que se basa en el código probado.

Puede ejecutar unittest manualmente si el código es pequeño, pero normalmente las unittest deben ser automatizadas. Al escribir una unittest, piense en lo siguiente:

- El unittest debe ser simple y fácil de implementar.
- El unittest debe estar bien documentado, por lo que es fácil averiguar cómo ejecutar la prueba, incluso después de varios años.
- Considerar los métodos de prueba y las entradas desde todos los ángulos.
- Los resultados de las pruebas deben ser consistentes. Esto es importante para la automatización de pruebas.
- El código de prueba debe funcionar independientemente del código que se esté probando. Si escribe pruebas que necesitan cambiar el estado del programa, capture el estado antes de cambiarlo, y vuelva a cambiarlo después de que se ejecute la prueba.
- Cuando una prueba falla, los resultados deben ser fáciles de leer y señalar claramente lo que se espera y dónde están los problemas.

En este laboratorio, explorará el **unittest** framework y usará **unittest** para probar una función.

Recursos necesarios

- Una computadora con el sistema operativo de su elección.
- VirtualBox o VMware.
- Máquina virtual de (Virtual Machine) DEVASC.

Instrucciones

Parte 1: Inicie la máquina virtual (Virtual Machine) de DEVASC.

Si aún no ha completado el **Laboratorio: Instalar el Entorno de Laboratorio de Máquina Virtual**, hágalo ahora. Si ya ha completado ese laboratorio, ejecute la máquina virtual (Virtual Machine) de DEVASC ahora.

Parte 2: Explore las opciones en el unittest del framework

Python proporciona un framework de unittest (llamado **unittest**) como parte de la librería estándar de Python. Si no está familiarizado con este framework, estudie el "Python Unittest Framework" para familiarizarse. Busque en Internet para encontrar la documentación en python.org. Necesitará ese conocimiento o documentación para responder a las preguntas de esta parte.

¿Qué clase de **unittest** usa para crear una unidad individual de pruebas?

El módulo unittest proporciona una clase base, TestCase, que se puede utilizar para crear nuevos casos de prueba.

Un corredor de pruebas (test runner) es responsable de ejecutar las pruebas y proporcionarle los resultados. Un corredor de pruebas (test runner) puede ser una interfaz gráfica pero, en este laboratorio, usará la línea de comandos para ejecutar las pruebas.

¿Cómo sabe el corredor de pruebas (test runner) qué métodos son una prueba?

Métodos cuyos nombres comienzan con las letras test_ informa al corredor de pruebas (test runner) acerca de qué métodos son pruebas.

¿Qué comando enumerará todas las opciones de línea de comandos para **unittest** que se muestran en la siguiente salida?

```
devasc @labvm: ~/labs/devnet-src$ python3 -m unittest -h
<output omitted>
Argumentos opcionales:
  -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

Ejemplos:
  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
Directorio del proyecto.
devasc@labvm:~/labs/devnet-src$
```

The command is python3 -m unittest -h

Parte 3: Probar una función Python con unittest

En esta parte, usará **unittest para probar** una función que realiza una búsqueda recursiva de un objeto JSON. La función devuelve valores etiquetados con una clave dada. Los programadores a menudo necesitan realizar este tipo de operación en objetos JSON devueltos por llamadas API.

Esta prueba utilizará tres archivos como se resume en la siguiente tabla:

Archivo	Descripción
recursive_json_search.py	Este script incluirá la función **json_search ()** que queremos probar.
test_data.py	Estos son los datos que la función **json_search ()** está buscando.
test_json_search.py	Este es el archivo que creará para probar la función **json_search ()** en el script recursive_json_search.py.

Paso 1: Revise el archivo test_data.py.

Abra el archivo **~/labs/devnet-src/unittest/test_data.py** y examine su contenido. Estos datos JSON son típicos de los datos devueltos por una llamada a la API de DNA Center de Cisco. Los datos de la muestra son lo suficientemente complejos como para ser una buena prueba. Por ejemplo, tiene tipos de **dict** y **lista** intercalados.

```
devasc @labvm: ~/labs/devnet-src$ más 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%)
```

Paso 2: Cree la función json_search () que va a probar.

Nuestra función debe esperar una clave y un objeto JSON como parámetros de entrada, y devolver una lista de pares clave/valor coincidentes. Aquí está la versión actual de la función que necesita ser probada para ver si funciona como estaba previsto. El propósito de esta función es importar primero los datos de prueba. Luego busca datos que coincidan con las variables clave en el archivo **test_data.py**. Si encuentra una coincidencia, agregará los datos coincidentes a una lista. La función **print ()** al final imprime el contenido de la lista para la primera 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 == clave: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)
            json_search(key,v)
                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)
    retorno ret_val
print (json_search ("IssueSummary", data))
```

a. Abra el archivo **~/labs/devnet-src/unittest/recursive_json_search.py**.

b. Copie el código anterior en el archivo y guárdelo.

Nota: Si está viendo este laboratorio como un archivo PDF, puede que tenga que editar los saltos de línea para que el código sea válido. Tenga en cuenta que los comentarios en línea no deben pasar a la siguiente línea. Cuando se pega en el **recursive_json_search.py**, debe haber 21 líneas de código incluyendo el comentario abierto **# Rellene el código Python en este archivo**.

c. Ejecute el código. No debe obtener errores y salida de **[]** indicando una lista vacía. Si la función **json_search ()** se codificó correctamente (lo cual no es), esto le indicaría que no hay datos con la clave "IssueSummary" reportados por los datos JSON devueltos por la API de Cisco DNA Center. En otras palabras, no hay cuestiones que informar.

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

d. Pero, ¿cómo saber que la función **json_search ()** funciona como se esperaba? Puede abrir el archivo **test_data.py** y buscar la clave "IssueSummary", como se muestra a continuación. Si usted lo hizo, descubrirá que hay un problema. Este es un pequeño conjunto de datos y una búsqueda recursiva

relativamente simple. Sin embargo, los datos de producción y el código rara vez son tan simples. Por lo tanto, probar código es vital para encontrar y corregir rápidamente errores en su código.

```
<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": [
            {
<output omitted>
```

Paso 3: Cree algunas pruebas unitarias que probarán si la función trabaja según lo previsto.

a. Abra el archivo **~ labs/devnet-src/unittest/test_json_search.py**.

b. En la primera línea del script después del comentario, importe la librería **unittest**.

```
import unittest
```

c. Agregue líneas para importar la función que está probando, así como los datos de prueba JSON que utiliza la función.

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

d. Ahora agregue el siguiente código de clase **json_search_test** al archivo **test_json_search.py**. El código crea la subclase **TestCase** del **unittest frmawork**. La clase define algunos métodos de prueba que se utilizarán en la función **json_search ()** en el script **recursive_json_search.py**. Observe que cada método de prueba comienza con **test_**, lo que permite que el **unittest framework** los descubra automáticamente. Agregue las siguientes líneas al final de su archivo ~labs/devnet-src/unittest/test_json_search.py:

```
class json_search_test(unittest.TestCase):
    "'módulo de prueba para probar la función de búsqueda en
`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)
```

En el código **unittest**, está utilizando tres métodos para probar la función de búsqueda:

1) Dada una clave existente en el objeto JSON, vea si el código de prueba puede encontrar dicha clave.
2) Dada una clave inexistente en el objeto JSON, vea si el código de prueba confirma que no se puede encontrar ninguna clave.
3) Compruebe si nuestra función devuelve una lista, como siempre debería hacer.

Para crear estas pruebas, el script utiliza algunos de los métodos de aserción incorporados en la clase **unittest** TestCase para comprobar las condiciones. El método **assertTrue (x)** comprueba si una condición es verdadera y **assertIsInstance (a, b)** comprueba si a es **una** instancia del tipo **b**. El tipo utilizado aquí es la **lista**.

Además, observe que los comentarios para cada método se especifican con la comilla simple triple (**"**). Esto es necesario si desea que la prueba muestre una descripción del método de prueba cuando se ejecuta. Usar el símbolo hash único (**#**) para el comentario no imprimiría la descripción de una prueba fallida.

e. Para la última parte del script, agregue el método **unittest.main ()**. Esto permite ejecutar **unittest** desde la línea de comandos. El propósito de **if __name__ == '__main__'** es asegurarse de que el método **unittest.main ()** se ejecuta sólo si el script se ejecuta directamente. Si el script se importa a otro programa, **unittest.main ()** no se ejecutará. Por ejemplo, puede utilizar un corredor de prueba diferente a **unittest** para ejecutar esta prueba.

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

Paso 4: Ejecute la prueba para ver los resultados iniciales.

a. Ejecute el script de prueba en su estado actual para ver qué resultados devuelve actualmente. Primero, verá la lista vacía. En segundo lugar, verá la **.F.** resaltada en la salida. Un punto (.) significa una prueba superada y una F significa una prueba fallida. Por lo tanto, la primera prueba pasó, la segunda prueba falló y la tercera prueba pasó.

b.

```
devasc @labvm: ~/labs/devnet-src/unittest$ python3 test_json_search.py
 []
.F.
======================================================================
FAIL: test_search_found (__main__.json_search_test)
Se debe encontrar, la lista de retorno no debe estar vacía
----------------------------------------------------------------------
Traceback (última llamada más reciente):
  File "test_json_search.py", line 11, in test_search_found
    self.assertTrue([]!=json_search(key1,data))
AssertionError: False is not true

----------------------------------------------------------------------
```

```
Realicé 3 pruebas en 0.001 s

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

c. Para enumerar cada prueba y sus resultados, ejecute el script bajo el **unittest** con la opción verbose (**-v**). Observe que no necesita la extensión.py para el script **test_json_search.py**. Puede ver que su método de prueba **test_search_found** está fallando.

Nota: Python no necesariamente ejecuta sus pruebas en orden. Las pruebas se ejecutan en orden alfabético basado en los nombres de los métodos de prueba.

```
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

======================================================================
FAILE: test_search_found (test_json_search.json_search_test)
key should be found, return list should not be empty
----------------------------------------------------------------------
Traceback (última llamada más reciente):
  Archivo "/home/devasc/labs/devent-src/unittest/test_json_search.py", línea 11, en
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$
```

Paso 5: Investigar y corregir el primer error en el script recursive_json_search.py.

La aserción, **la clave debe ser encontrada, la lista de retorno no debe estar vacía... FAIL**, indica que no se encontró la clave. ¿Por qué? Si miramos el texto de nuestra función recursiva, vemos que la sentencia **ret_val= []** se está ejecutando repetidamente, cada vez que se llama a la función. Esto hace que la función vacíe la lista y pierda los resultados acumulados de la instrucción **ret_val.append (temp)**, que se agrega a la lista creada por **ret_val= []**.

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

```
    if isinstance (input_object, dict):
        para k, v en input_object.items ():
            if k == clave:if k == key:
                temp={k:v}
                ret_val.append(temp)
```

a. Mueva el **ret_val= []** fuera de nuestra función en **recursive_json_search.py** para que la iteración no sobrescriba la lista acumulada cada vez.

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

b. Guarde y ejecute el script. Debería obtener el siguiente resultado que verifica que resolvió el problema. La lista ya no está vacía después de que se ejecute el script.

```
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$
```

Paso 6: Ejecutar la prueba de nuevo para ver si todos los errores del script están ahora solucionados.

a. ¿Obtuvo algún resultado la última vez que ejecutó **recursive_json_search.py**, aún no puede estar seguro de haber resuelto todos los errores en el script? Ejecute **unittest** de nuevo sin la opción **-v** para ver si **test_json_search** devuelve algún error. Normalmente, no se elige utilizar la opción **-v** para minimizar la salida de la consola y hacer que las pruebas se ejecuten más rápido. Al inicio del registro se puede ver**.. F**, lo que significa que la tercera prueba falló. También tenga en cuenta que la lista todavía se está imprimiendo. Puede detener este comportamiento eliminando la función **print ()** en el script **resursive_json_search.py**. Pero eso no es necesario para tus propósitos en este laboratorio.

```
devasc @labvm: ~/labs/devnet-src/unittest$ python3 -m unittest
test_json_search
[{'issueSummary': 'Network Device 10.10.20.82 Is Unreachable From Controller'}]
.. F
======================================================================
FALL: test_search_not_found (test_json_search.json_search_test)
La clave no debe ser encontrada, debe devolver una lista vacía
----------------------------------------------------------------------
Traceback (última llamada más reciente):
  Archivo "/home/devasc/labs/devnet-src/unittest/test_json_search.py", línea 14, en
test_search_not_found
    self.assertTrue([]==json_search(key2,data))
AssertionError: False is not true

----------------------------------------------------------------------
Realicé 3 pruebas en 0.001 s

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

b. Abra el archivo **test_data.py** y busque **IssueSummary**, que es el valor de la key1. Debe encontrarlo dos veces, pero solo una vez en el objeto de **data** JSON. Pero si busca el valor de key2, que es **XY&^$#*@!1234% ^&**, solo lo encontrará en la parte superior donde está definido porque no está en el

objeto JSON de **los datos** . La tercera prueba es verificar para asegurarse de que no está allí. El tercer comentario de prueba, indica que los estados deben indicar que la **clave no debe ser encontrada, debe devolver una lista vacía**. Sin embargo, la función devolvió una lista no vacía.

Paso 7: Investigue y corrija el segundo error en el script recursive_json_search.py.

a. Revise el código **recursive_json_search.py** de nuevo. Observe que la **ret_val** es ahora una variable global después de corregirla en el paso anterior. Esto significa que su valor se conserva a través de múltiples invocaciones de la función **json_search ()**. Este es un buen ejemplo de por qué es una mala práctica usar variables globales dentro de las funciones.

b. Para resolver este problema, envuelva la función **json_search ()** con una función externa. Elimine su función **json_search ()** existente y reemplace con la refactorneada a continuación: (No hará daño llamar a la función dos veces, pero no es la mejor práctica repetir una función).

```
from test_data import *
def json_search(clave, input_object):
    """
    Buscar una clave del objeto JSON, no obtener nada si la clave no se encuentra
    key: "keyword" a buscar, distingue entre mayúsculas y minúsculas
    input_object: objeto JSON a analizar, test_data.py en este caso
    inner_function () está haciendo la búsqueda recursiva
    devolver una lista de par clave: valor
    """
    ret_val= []
    def inner_function(clave, input_object):
        if isinstance(input_object, dict): # Iterate dictionary
            for k, v in input_object.items(): # searching key in the dict
                if k == clave:if k == key:
                    temp={k:v}
                    ret_val.append(temp)
                if isinstance(v, dict): # the value is another dict so repeat
                    función inner_function (tecla, 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: # Itere una lista porque algunas API devuelven un objeto JSON en una
lista
            for val in input_object:
                if not isinstance(val, (str,int)):
                    inner_function(key,val)
    inner_function(key,input_object)
    retorno ret_val
print (json_search ("IssueSummary", data))
```

c. Guarde el archivo y ejecute **unittest** en el directorio. No necesita el nombre del archivo. Esto se debe a que la característica de **unittest** de detección ejecutará cualquier archivo local que encuentre cuyo nombre comience con la prueba. Debería ver el siguiente resultado: Observe que todas las pruebas pasan ahora y la lista de la clave "IssueSummary" se rellena. Puede eliminar de forma segura la función

print () ya que normalmente no se usaría cuando esta prueba se agrega con otras pruebas para una ejecución de prueba más grande.

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

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