Travaux pratiques - Créer un test unitaire Python

# Objectifs

Partie 1 : Lancer la DEVASC VM

Partie 2 : Explorer les options dans le cadre unittest

Partie 3 : Tester une fonction Python avec unittest

# Contexte/scénario

Les tests unitaires examinent des unités de code indépendantes, telles que des fonctions, des classes, des modules et des bibliothèques. Il y a plusieurs raisons d'écrire un script en utilisant la bibliothèque **unittest** de Python. Une raison évidente est que si vous trouvez un problème dans le code isolé par des tests délibérés, vous savez que le problème est dans la fonction ou une autre unité testée. Le problème n'est pas dans l'application plus grande qui peut appeler cette fonction. Vous saurez également exactement ce qui a déclenché l'erreur car vous avez écrit le test unitaire qui a exposé le problème. Les bogues trouvés de cette façon sont généralement rapides et faciles à corriger, et les correctifs faits à ce niveau détaillé sont moins susceptibles de provoquer des effets secondaires imprévus plus tard dans un autre code qui repose sur le code testé.

Vous pouvez exécuter des tests unitaires manuellement si le code est petit, mais les tests unitaires doivent généralement être automatisés. Lorsque vous écrivez un test unitaire, pensez à ce qui suit :

* Le test unitaire doit être simple et facile à mettre en œuvre.
* Le test unitaire doit être bien documenté, il est donc facile de comprendre comment exécuter le test, même après plusieurs années.
* Considérez les méthodes d'essai et les entrées sous tous les angles.
* Les résultats des essais doivent être cohérents. Ceci est important pour l'automatisation des tests.
* Le code de test doit fonctionner indépendamment du code en cours de test. Si vous écrivez des tests qui doivent changer l'état du programme, capturez l'état avant de le modifier, puis modifiez-le après l'exécution du test.
* Lorsqu'un test échoue, les résultats devraient être faciles à lire et indiquer clairement ce qui est attendu et où se trouvent les problèmes.

Dans ce TP, vous allez explorer le framework **unittest** et utiliser **unittest** pour tester une fonction.

# Ressources requises

* 1 PC avec système d'exploitation de votre choix
* Boîte virtuelle ou VMWare
* Machine virtuelle DEVASC

# Instructions

## Lancer la machine virtuelle DEVASC

Si vous n'avez pas encore terminé **TP - Installer l'environnement de laboratoire de machines virtuelles**, faites-le maintenant. Si vous avez déjà terminé ce TP, lancez la machine virtuelle DEVASC maintenant.

## Explorer les options dans le cadre unittest

Python fournit un cadre de test unitaire (appelé **unittest**) dans le cadre de la bibliothèque standard Python. Si vous n'êtes pas familier avec ce framework, étudiez le « Python unittest Framework » pour vous familiariser. Cherchez-le sur Internet pour trouver la documentation sur python.org. Vous aurez besoin de ces connaissances ou de ces documents pour répondre aux questions de cette partie.

#### Question

Quelle classe **unittest** utilisez-vous pour créer une unité de test individuelle ?

Saisissez vos réponses ici

Un testeur (test runner) est responsable de l'exécution des tests et de vous fournir des résultats. Un testeur peut être une interface graphique mais, dans ce laboratoire, vous utiliserez la ligne de commande pour exécuter des tests.

#### Questions :

Comment le testeur sait-il quelles méthodes sont un test ?

Saisissez vos réponses ici

Quelle commande listera toutes les options de ligne de commande pour **unittest** montrées dans la sortie suivante ?

devasc@labvm:~/labs/devnet-src$ **python3 -m unittest -h**

<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$

Saisissez vos réponses ici

## Tester une fonction Python avec unittest

Dans cette partie, vous allez utiliser **unittest** pour tester une fonction qui effectue une recherche récursive d'un objet JSON. La fonction renvoie des valeurs marquées avec une clé donnée. Les programmeurs ont souvent besoin d'effectuer ce type d'opération sur des objets JSON retournés par des appels d'API.

Ce test utilisera trois fichiers tels que résumés dans le tableau suivant :

| Fichier | Description |
| --- | --- |
| recursive\_json\_search.py | Ce script inclura la fonction **json\_search ()** que nous voulons tester. |
| test\_data.py | Il s'agit des données que la fonction **json\_search ()** recherche. |
| test\_json\_search.py | C'est le fichier que vous allez créer pour tester la fonction **json\_search ()** dans le script recursive\_json\_search.py. |

### Passez en revue le fichier test\_data.py.

Ouvrez le fichier **~/labs/devnet-src/unittest/test\_data.py** et examinez son contenu. Ces données JSON sont typiques des données renvoyées par un appel à l'API DNA Center de Cisco. Les données de l'échantillon sont suffisamment complexes pour être un bon test. Par exemple, il a des types **dict** et **list** entrelacés.

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%)

### Créez la fonction json\_search () que vous allez tester.

Notre fonction devrait s'attendre à une clé et un objet JSON en tant que paramètres d'entrée, et renvoyer une liste de paires clé/valeur correspondantes. Voici la version actuelle de la fonction qui doit être testée pour voir si elle fonctionne comme prévu. Le but de cette fonction est d'importer les données de test en premier. Ensuite, il recherche les données qui correspondent aux variables clés dans le fichier **test\_data.py**. S'il trouve une correspondance, il ajoutera les données correspondantes à une liste. La fonction **print ()** à la fin imprime le contenu de la liste de la première 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}

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))

* + - 1. Ouvrez le fichier **~/labs/devnet-src/unittest/recursive\_json\_search.py**.
      2. Copiez le code ci-dessus dans le fichier et enregistrez-le.

**Remarque**: Si vous affichez ce laboratoire sous la forme d'un fichier PDF, vous devrez peut-être modifier les sauts de ligne pour rendre le code valide. Notez que les commentaires en ligne ne doivent pas passer à la ligne suivante. Lorsqu'il est collé dans le **fichier recursive\_json\_search.py**, il devrait y avoir 21 lignes de code incluant le commentaire ouvert **# Fill the Python code in this file**.

* + - 1. Exécutez le code. Vous ne devriez obtenir aucune erreur et la sortie de **[ ]** indiquant une liste vide. Si la fonction **json\_search ()** a été codée correctement (ce qui n'est pas le cas), cela vous indiquerait qu'il n'y a pas de données avec la clé "issueSummary" signalée par les données JSON renvoyées par l'API Cisco DNA Center. En d'autres termes, il n'y a pas de questions à signaler.

devasc@labvm:~/labs/devnet-src/unittest$ **python3 recursive\_json\_search.py**

[]

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

* + - 1. Mais comment savez-vous que la fonction **json\_search ()** fonctionne comme prévu ? Vous pouvez ouvrir le fichier **test\_data.py** et rechercher la clé “issueSummary”, comme indiqué ci-dessous. Si c'était le cas, vous constateriez qu'il y a un problème. Il s'agit d'un petit ensemble de données et d'une recherche récursive relativement simple. Cependant, les données de production et le code sont rarement aussi simples. Par conséquent, tester le code est essentiel pour trouver et corriger rapidement les erreurs dans votre 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": [

{

<output omitted>

### Créez des tests unitaires qui vérifieront si la fonction fonctionne comme prévu.

* + - 1. Ouvrez le fichier **~ labs/devnet-src/unittest/test\_json\_search.py**.
      2. Dans la première ligne du script après le commentaire, importez la bibliothèque **unittest**.

importer unittest

* + - 1. Ajoutez des lignes pour importer la fonction que vous testez ainsi que les données de test JSON que la fonction utilise.

from recursive\_json\_search import \*

from test\_data import \*

* + - 1. Maintenant, ajoutez le code de classe **json\_search\_test** suivant au fichier **test\_json\_search.py**. Le code crée la sous-classe **TestCase** de l'infrastructure **unittest**. La classe définit certaines méthodes de test à utiliser sur la fonction **json\_search ()** dans le script **recursive\_json\_search.py**. Notez que chaque méthode de test commence par **test\_**, ce qui permet au framework **unittest** de les découvrir automatiquement. Ajoutez les lignes suivantes au bas de votre fichier ~labs/devnet-src/unittest/test\_json\_search.py :

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)

Dans le code **unittest**, vous utilisez trois méthodes pour tester la fonction de recherche :

* + - * 1. Étant donné une clé existante dans l'objet JSON, voyez si le code de test peut trouver une telle clé.
        2. Étant donné une clé inexistante dans l'objet JSON, voyez si le code de test confirme qu'aucune clé ne peut être trouvée.
        3. Vérifiez si notre fonction renvoie une liste, comme il devrait toujours le faire.

Pour créer ces tests, le script utilise certaines des méthodes d'assertion intégrées dans la classe **unittest** TestCase pour vérifier les conditions. La méthode **assertTrue (x)** vérifie si une condition est vraie et **assertisInstance (a, b)** vérifie si **a** est une instance du type **b.** Le type utilisé ici est **liste**.

Notez également que les commentaires pour chaque méthode sont spécifiés avec le triple guillemet simple (**'''**). Ceci est requis si vous voulez que le test affiche une description de la méthode de test lors de son exécution. L'utilisation du symbole de hachage unique (**#**) pour le commentaire n'afficherait pas la description d'un test échoué.

* + - 1. Pour la dernière partie du script, ajoutez la méthode **unittest.main ()**. Cela permet d'exécuter **unittest** à partir de la ligne de commande. Le but de **if \_\_name\_\_ == '\_\_main\_\_'** est de s'assurer que la méthode **unittest.main ()** ne s'exécute que si le script est exécuté directement. Si le script est importé dans un autre programme, **unittest.main ()** ne s'exécutera pas. Par exemple, vous pouvez utiliser un autre coureur de test que **unittest** pour exécuter ce test.

if \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

### Exécutez le test pour voir les résultats initiaux.

* + - 1. Exécutez le script de test dans son état actuel pour voir les résultats qu'il renvoie actuellement. Tout d'abord, vous voyez la liste vide. Deuxièmement, vous voyez le **.F.** mis en surbrillance dans la sortie. Un point (.) signifie un test réussi et un F signifie un test échoué. Par conséquent, le premier test a réussi, le second a échoué et le troisième a réussi.

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$

* + - 1. Pour répertorier chaque test et ses résultats, exécutez à nouveau le script sous **unittest** avec l'option verbose (**-v**). Notez que vous n'avez pas besoin de l'extension .py pour le script **test\_json\_search.py**. Vous pouvez voir que votre méthode de test **test\_search\_found** échoue.

**Remarque** : Python n'exécute pas nécessairement vos tests dans l'ordre. Les tests sont exécutés dans l'ordre alphabétique en fonction des noms des méthodes de test.

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

FAILED (failures=1)

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

### Examinez et corrigez la première erreur dans le script recursive\_json\_search.py.

L'assertion, **la clé doit être trouvée, la liste de retour ne doit pas être vide ... FAIL**, indique que la clé n'a pas été trouvée. Pourquoi ? Si nous regardons le texte de notre fonction récursive, nous voyons que l'instruction **ret\_val= []** est exécutée à plusieurs reprises, chaque fois que la fonction est appelée. Cela provoque la fonction de vider la liste et de perdre les résultats accumulés à partir de l'instruction **ret\_val.append (temp)**, qui ajoute à la liste créée par **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)

* + - 1. Déplacez le **ret\_val= [ ]** hors de notre fonction dans **recursive\_json\_search.py** afin que l'itération n'écrase pas la liste accumulée à chaque fois.

ret\_val= []

def json\_search(key,input\_object):

* + - 1. Enregistrez et exécutez le script. Vous devriez obtenir la sortie suivante qui vérifie que vous avez résolu le problème. La liste n'est plus vide après l'exécution du 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$

### Exécutez à nouveau le test pour voir si toutes les erreurs dans le script sont maintenant corrigées.

* + - 1. Vous avez obtenu une sortie la dernière fois que vous avez exécuté **recursive\_json\_search.py**, vous ne pouvez pas encore être sûr d'avoir résolu toutes les erreurs dans le script ? Exécutez à nouveau **unittest** sans l'option **-v** pour voir si **test\_json\_search** renvoie des erreurs. En règle générale, vous ne voulez pas utiliser l'option **-v** pour réduire la sortie de la console et rendre les tests plus rapides. Au début du journal, vous pouvez voir**.. F**, ce qui signifie que le troisième test a échoué. Notez également que la liste est toujours en cours d'impression. Vous pouvez arrêter ce comportement en supprimant la fonction **print ()** dans le script **resursive\_json\_search.py**. Mais ce n'est pas nécessaire à vos fins dans ce labo.

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

----------------------------------------------------------------------

Ran 3 tests in 0.001s

FAILED (failures=1)

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

* + - 1. Ouvrez le fichier **test\_data.py** et recherchez **issueSummary**, qui est la valeur de key1. Vous devriez le trouver deux fois, mais une seule fois dans l'objet de **données** JSON. Mais si vous recherchez la valeur de key2, qui est **XY&^$#\*@!1234% ^&**, vous ne le trouverez qu'en haut où il est défini car il ne se trouve pas dans l'objet JSON de **données** . Le troisième test consiste à vérifier qu'il n'est pas là. Le troisième commentaire de test indique que la **clé ne doit pas être trouvée, devrait retourner une liste vide**. Cependant, la fonction a renvoyé une liste non vide.

### Examinez et corrigez la deuxième erreur dans le script recursive\_json\_search.py.

* + - 1. Examinez à nouveau le code **recursive\_json\_search.py**. Notez que le **ret\_val** est maintenant une variable globale après l'avoir corrigé à l'étape précédente. Cela signifie que sa valeur est préservée à travers plusieurs invocations de la fonction **json\_search ()**. C'est un bon exemple de la raison pour laquelle il est une mauvaise pratique d'utiliser des variables globales dans les fonctions.
      2. Pour résoudre ce problème, enveloppez la fonction **json\_search ()** avec une fonction externe. Supprimez votre fonction **json\_search ()** existante et remplacez-la par la fonction refactored ci-dessous: (Cela ne fera pas de mal d'appeler la fonction deux fois mais il n'est pas recommandé de répéter une fonction.)

à partir de test\_data import \*

def json\_search(clé, 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))

* + - 1. Enregistrez le fichier et exécutez **unittest** sur le répertoire. Vous n'avez pas besoin du nom du fichier. En effet, la fonctionnalité de découverte de **test unittest** exécutera tout fichier local qu'elle trouve dont le nom commence par test. Vous devriez obtenir la sortie suivante. Notez que tous les tests réussissent maintenant et que la liste de la clé "issueSummary" est renseignée. Vous pouvez supprimer en toute sécurité la fonction **print ()** car elle ne serait pas normalement utilisée lorsque ce test est agrégé avec d'autres tests pour une plus grande exécution de test.

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$

Fin du document