



















Tests unitaires avec Pytest

Tests unitaires avec Python

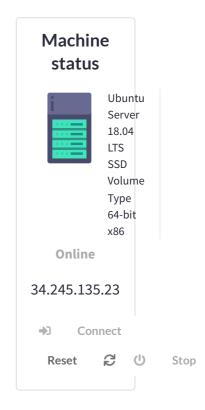
1. Qu'est-ce que le test unitaire? A. Principe

Le test unitaire est une méthode agile de travail qui consiste à tester des petites parties d'un code, que l'on appelle unités, de manière isolée. Les unités que l'on teste sont le plus souvent des fonctions et des classes mais la procédure de test peut s'appliquer sur des modules entiers.

L'objectif des tests unitaires est donc de s'assurer que chaque élément constitutif du code est en bonne santé. Dans la pratique, on donne des entrées à l'unité et on vérifie que la sortie produite par l'unité correspond bien à ce qu'elle est censée renvoyer.

B. Test unitaire vs Test d'intégration

Alors que le test unitaire vérifie que toutes les unités du code fonctionnent indépendamment, le test d'intégration s'assure lui qu'elles fonctionnent ensemble. Les tests d'intégration sont axés sur des cas d'usage réels. Ils font ainsi souvent appel à des données externes comme des databases ou des serveurs web.



Δ



Robin









Imaginez que l'on teste le fonctionnement d'un phare sur une voiture. Le test d'intégration vérifie que le phare s'allume lorsqu'on appuie sur le bon bouton. Les tests unitaires, s'assureront eux du bon fonctionnement de chaque élément du phare pris séparément (fonctionnement du bouton, de la batterie, des câbles, des ampoules...)

Exemple : La fonction total effectue la somme des élements d'une liste.

```
1 def total(liste):
2    """ renvoie la somme des élé
3
4    result : float = 0.0
5
6    for item in liste:
7     result += item
8
9    return (result)
```

Imaginez plusieurs tests que vous pouriez écrire dans votre console afin de vous assurer que du bon fonctionnement de la fonction total :

Show / Hide solution

Les tests unitaires sont souvent négligés par les développeurs peu méticuleux qui le considèrent comme une perte de temps, étant donné que les bugs seront de toutes manières décelés lors des tests d'intégration. C'est bien évidemment faux et nous allons voir pourquoi à travers tous les avantages qui découlent des tests unitaires.

2. Avantages et limites du test unitaire

A. Les avantages

Voici une liste non exhaustive des avantages du test unitaire qui en font un outil indispensable de la boite à outil d'un bon développeur :

 Gain de temps. Il arrive que certaines erreurs très basiques deviennent difficiles à identifier lors de la phase de test d'intégration, du fait des nombreuses couches de code qui s'accumulent. Ces erreurs se décèlent en revanche très simplement, très rapidement et très tôt dans la construction du code grâce aux tests unitaires.















- verifier que la fonction a toujours le comportement attendu en effectuant le test unitaire de cette fonction.
- Amélioration de la qualité du code. Une approche parfois pertinente dans la manière de coder consiste à coder les tests unitaires avant même de coder les unités en question. Cela permet de se forcer à réfléchir à toutes les éventualités auxquelles devra faire face notre unité. En pensant mieux la manière dont on va coder l'unité, cela la rend en général plus claire et robuste par la suite. On appelle cette approche le TDD (test driven development).
- Aide à la compréhension du code. Les tests unitaires sont aussi utilisés par les développeurs comme des documentations explicatives de chaque partie du code. En effet, il est très simple de se rendre compte du comportement attendu d'une fonction en lisant préalablement le test unitaire qui lui est associé.

B. Les limites

- Il est toutefois impossible de tester l'infinité des éventualités auxquelles devra faire face l'unité. Passer le test unitaire sans accroc n'est donc pas non plus gage total de bon fonctionnement.
- Les tests unitaires ne peuvent, par construction, pas tester l'interaction entre les unités.

Les tests que nous avons écrits précédemment ont le désavantage d'être rébarbatifs à devoir réécrire à chaque fois que l'on veut tester sa fonction. De plus, chaque coopérateur testera la fonction de son côté sans qu'il n'y ait d'uniformisation entre tous les développeurs.

Pour remédier à cela, nous allons structurer un peu notre projet.

- Dans le dossier maison de la machine virtuelle, créez un fichier python qui vous appelerez code1.py et qui contient la fonction total.
- Créez maintenant un fichier python code1 test.py. Dans ce fichier, créez une fonction testtotal qui regroupe les tests précédent et renvoie le tuple de booléens correspondant aux résultats des différents tests. Avoutez un print pour que l'exécution de ce code renvoie automatiquement le tuple.

Show / Hide solution

 Dans votre terminal, exécutez maintenant ce fichier code1_test.py (\$ python3 code1_test.py)

Le désavantage de cette méthode est qu'elle ne permet pas de se rendre compte rapidement pourquoi un des tests est faux. Il est aussi difficile de vérifier qu'une fonction renvoie un procédure de test.















3. L'automated testing avec pytest A. Qu'est-ce que l'automated testing?

Pour automatiser les tests unitaires, il existe des frameworks qui vont grandement nous faciliter la tâche. Le développeur doit paramétrer les critères des tests qu'il souhaite effectuer, puis le framework s'occupe d'effectuer les tests automatiquement et de fournir des rapports d'erreur détaillés. Le framework de base de l'automated testing sur python est unittest. Nous allons dans ce cours apprendre à utiliser pytest qui est légèrement plus intuitif.

B. Présentation de pytest

Imaginons que nous souhaitions tester les fonctions d'un fichier: code1.py La bonne démarche à adopter est de créer dans le même dossier, un autre fichier python que l'on appellera code1_test.py

Le wrokflow à adopter est le suivant :

- 1. Ecriture d'une fonction total dans le fichier code1.py
- Création d'un fichier code1_test.py dans lequel on importe la fonction souhaitée
- 3. Ecriture d'une fonction test_total. Les fonctions tests doivent systématiquement commencer par test_. C'est une convention de pytest qui doit être respectée sans quoi pytest n'exécute pas le test.
- 4. Il suffit ensuite d'exécuter le fichier code1_test.py avec le module pytest dans le terminal (nous détaillerons la procédure plus bas).

Cette fois dans notre fonction test, nous n'allons plus utiliser des bouléens. Pour vérifier que la fonction renvoie le bon résultat lorsqu'on lui donne certains arguments, on utilisera la méthode assert de la manière suivante : assert fonction(arguments) == résultat attendu. De cette manière, pytest nous fournira un rapport détaillé lorsque la condition n'est pas vérifiée.

Après avoir importé la fonction total de code1, réécrivez une fonction test_total qui reprend les 5 tests précédents mais cette fois-ci à l'aide de assert.

```
1 from code1 import total
2
3
4 def test_total():
5  #Les use cases:
```













```
"""1 - 1 = 0"""
9
10
        assert total([1,-1]) == 0
11
        """-1 -1 = -2"""
12
        assert total([-1,-1]) == -2
13
14
        #Les edge cases :
15
16
        """La somme doit être égal à
        assert(total([1.0])) == 1.0
17
18
        """In commo d'una licta vida
```

Nous allons maintenant avoir besoin du package pytest. Effectuez donc un pip3 install pytest dans votre terminal

```
1 pip3 install pytest
2
```

Maintenant pour afficher le rapport de pytest, il vous suffit d'exécuter python3 -m pytest code1_test.py

```
1 python3 -m pytest code1_test.py
2
```

Le point vert correspond à la bonne exécution d'un test (il y en a un seul car nous avons implémenté une seule fonction de la forme test_).

Si on change la première vérification en : assert total([1, 2, 3]) == 5, on obtient l'erreur suivante :



Le point vert s'est transformé en F rouge car une des vérification du test n'a pas été vérifiée. L'erreur renvoyée est une erreur d'assertion et Pytest nous indique même précisément à quel endroit elle se trouve.

C. Les effets secondaires (side effects)

Il arrive parfois que l'exécution d'une unité de code va modifier des éléments de l'environnement. Par exemple, un attribut d'une classe ou la valeur d'une variable pourraient être modifiés. Nous appelons cela des sides effets ou effets

















D. Gérer les messages d'erreur

Parfois il peut être intéressant de vérifier qu'une fonction renvoie un message d'erreur (ex : TypeError) lorsqu'un certain type d'argument lui est donné.

Notre fonction total est construite de manière à fonctionner sur des intérables (listes, tuples, dictionnaires). Si nous lui passons un int en argument ou une chaîne de caractères, elle renverra un message d'erreur de type TypeError. Ce comportement est celui attendu car nous avons voulu créer une fonction qui fait la somme d'éléments d'un itérable seulement. Imaginons pour notre exercice que nous ne voulons absolument pas que notre fonction total renvoie 1 lorsqu'on l'appelle avec comme argument l'int 1 (au lieu de la liste [1]). Nous voulons alors vérifier que notre fonction renvoie une TypeError lorsqu'on ne lui passe pas le bon type d'objet en argument.

Proposez une fonction test pour vérifier que la fonction total renvoie une erreur lorsqu'on lui soumet un int. La commande with pytest.raises(TypeError) permet de s'assurer qu'une erreur est renvoyée par ce qui suit cette commande. Pensez à importer le module pytest dans votre fichier de test avant de pouvoir utiliser pytest.raises(TypeError).

Show / Hide solution

Imaginons que nous sommes un deuxième ingénieur et que nous voulons modifier la fonction total. Nous ajoutons la possibilité à cette fonction de traiter les int.

```
1
    def total(liste):
        """ renvoie la somme des élém
 2
 3
        if type(liste) == int :
 4
 5
            return (liste)
 6
 7
        result : float = 0.0
 8
 9
        for item in liste:
            result += item
10
11
        return (result)
12
```

Relancez le test après avoir modifié la fonction total



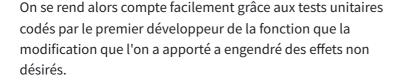












C. Le widget pytest

Il est aussi possible d'intégrer pytest sur votre ide sous la forme d'un widget. Cela permet d'exécuter les tests unitaires sans avoir à passer par le terminal à chaque fois :

Voici un exemple du rendu de ce plugin sur spyder :



4. Exercice d'application

Notre objectif dans cet exercice est de créer une classe Wallet qui possède une méthode d'ajout d'argent (add*cash) et une méthode de retrait (spend*cash).

Dans un fichier wallet.py, créez une classe Wallet qui :

- Accepte un apport d'argent initial et le stocke dans l'argument balance (= 0 si l'apport initial n'est pas précisé)
- Possède une méthode pour ajouter de l'argent add_cash
- Possède une méthode pour retirer de l'argent spend_cash. Cette méthode vérifie préalablement que le solde est suffisant et renvoie une exception InsufficientAmount le cas contraire.

Show / Hide solution

Dans un autre fichier python wallet_test.py, nous allons maintenant écrire nos tests unitaires. Pour cela il faut importer les fonctions que l'on veut tester ainsi que le module pytest (pour pouvoir tester l'exception InsuffisentAmount).

```
1 from wallet import Wallet, Insuffic:
2 import pytest
3
```









- un portefeuille créé avec une balance initiale de 100 a bien une balance de 100
- un portefeuille créé avec une balance initiale de 10 auguel on ajoute 90 a une balance de 100
- un portefeuille créé avec une balance initiale de 20 auguel on ôte 10 a une balance de 10
- un portefeuille qui essaie de dépenser plus que sa balance va provoquer une erreur InsufficientAmount

Show / Hide solution

Affichez le rapport pytest

5. Les Fixtures

Dans la correction précédente, il peut être parfois redondant d'avoir à créer un nouvel objet Wallet() pour chaque test que l'on veut effectuer. Pour remédier à cela, nous allons utiliser les **fixtures**. Les fixtures sont des fonctions générées grâce au décorateur @pytest.fixture. Il suffit ensuite de passer ces fonctions en argument de nos fonctions tests.

Dans votre fichier de tests, créez un fonction empty_wallet() qui instancie un nouveau portefeuille vide. Pensez bien à utiliser le décorateur @pytest.fixture en amont.

Faites de même avec une fonction wallet() qui instancie un portefeuille contenant le valeur de votre choix.

Show / Hide solution

Pour utiliser les **fictures**, on peut simplement appeler les fonctions dans les tests: par exemple, notre premier test deviendrait:

```
1 def test_default_initial_amount(emp1
2    assert empty_wallet.balance == 0
3
```

Recréez les tests pécédents en utilisant ces **fixtures**. Cela vous permet de ne plus avoir besoin de créer un nouveau Wallet() pour chaque test.

Show / Hide solution

L'avantage de cette méthode est que l'objet wallet est instancié de nouveau à chaque fonction test.





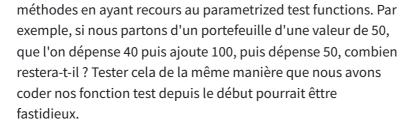












Nous pouvons donc tester plusieurs scénarios dans une seule fonction à l'aide du décorateur @pytest.mark.parametrize dans lequel nous pouvons préciser le noms des arguments qui seront passer dans la fonction de test, ainsi qu'une liste d'arguments correspondants.

Cette fonction test part d'un portefeuillede valeur 30, ajoute 10, retire 20 puis vérifie le résultat. Elle fait ensuite pareil avec un portefeuille de 20, auquel elle ajoute 2 puis retire 18.

Remarque : il est maintenant tout à fait possible de combiner les fixtures et les parametrized test functions

7. Qu'est-ce qui caractérise un bon ou un mauvais test unitaire ?

A. Les bon tests

Un bon test unitaire est un test qui est :

- Facile à écrire
- Facile à lire et comprendre
- Fiable
- Rapide
- Ne nécessite aucune intégration

B. Les mauvais tests

En réalité, à part si vous vous trompez dans la valeur du résultat attendu que vous vérifiez, il n'y a pas vraiment de mauvais test. En revanche, il y a des mauvaises manières de coder une unité qui font que le test unitaire ne sera pas vraiment un test unitaire. Cela peut être le cas lorsqu'une fonction crée beaucoup de variables locales. Si par exemple, imaginons une fonction qui instancie dans son corps une variable heure correspondant à l'heure au moment où la fonction est appelée.







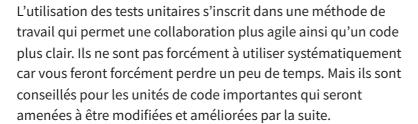
Robin BIRON





Conclusion







Validate