TD 2 — Introduction aux tests unitaires

Introduction

L'objectif est de se sensibiliser à la création de *tests unitaires*, qui sont une façon de tester les codes écrits afin de vérifier qu'ils sont exempts de bugs.

Ces tests sont extrêmement importants pour avoir confiance dans la correction d'un programme. Un programme est souvent composé d'un très grand nombre de fonctions qui interagissent entre elles. La moindre erreur dans une de ces fonctions peut avoir des conséquences catastrophiques sur le bon fonctionnement du programme. Les erreurs peuvent être de plusieurs types.

Erreurs de langage Le code n'est pas correctement écrit et contient des erreurs (de syntaxe, de type, des variables non initialisées, etc.). Le code refuse alors de s'exécuter ou de compiler[1]. Ces erreurs sont souvent plus faciles à détecter et ne sont pas liée aux tests unitaires.

Erreurs d'exécution Le code s'exécute, mais plante pendant l'exécution dans certains cas d'usage (erreurs de segmentation en C, exceptions en Python, etc.). Ces erreurs sont plus difficiles à détecter, car elles peuvent ne pas se produire dans tous les cas d'usage. Pour les détecter et les corriger, il est important de prévoir des tests dans des situations aussi variées que possible.

Erreurs fonctionnelles Le programme s'exécute sans erreurs, mais ne produit pas le résultat attendu. Ce sont les erreurs les plus difficiles à identifier et à corriger, car les cas d'usage sont potentiellement infinis et il n'est pas possible de les couvrir de façon exhaustive. L'utilisation de tests automatisée est indispensable dans ce cas, car ils permettent d'exécuter le code dans de très nombreux cas d'usage, donc de couvrir un large panel de scénarios.

Il existe de nombreuses stratégies pour tester des programmes de façon exhaustive. Dans le cadre de cette ressource, nous n'entrerons pas dans tous ces détails, l'objectif ici est de faire une première sensibilisation à l'importance des tests et d'encourager les étudiants à écrire des tests qui accompagnent leurs programmes.

Souvent, les programmeurs·ses débutant·es écrivent leurs tests en exécutant manuellement leurs fonctions avec quelques valeurs d'entrée et observant qu'elles fonctionnent correctement. D'autres ajoutent des appels à `print` dans leur code à certains endroits. Ces approches peuvent aider à détecter des problèmes simples, mais elles sont à éviter en pratique. Leur inconvénient majeur est qu'elles forcent le programmeur à passer beaucoup de temps pour tester son programme. Et comme on manque souvent de temps, les tests sont rarement, exécutés et le code risque fort de ne pas être correct.

Une meilleure approche est de s'appuyer sur un *framework* de test qui permet d'automatiser les tests. En utilisant un de ces frameworks, il suffit de taper une commande pour vérifier que l'ensemble des tests du programme continuent à s'exécuter correctement. L'apprentissage d'un tel framework de test prend un peu de temps, mais il offre l'avantage de pouvoir facilement éviter les régressions. Une régression est une modification mineure du code qui provoque une erreur subtile qui n'est pas

1. Si c'est un langage compilé.

détectée avant que le programme ne soit finalisé.

Bien utilisé en combinaison avec un logiciel de versions (tel que Git), les tests unitaires permettent de garantir une bonne qualité logiciel. En tout état de cause, il est indispensable de garder en tête le principe essentiel suivant :

Principe fondamental de la qualité de développement

> Un code non testé est un code faux par principe.

Dans ce TD, nous allons utiliser Python comme langage support pour automatiser des tests. Les principes et concepts transposent dans tous les langages de programmation.

Tests unitaires avec Python

Il est également possible d'implémenter des tests unitaires avec des instructions natives du langage Python, notamment `assert`. Lorsque l'on souhaite effectuer des tests plus avancés, ou sur des projets de plus grande taille, utilise un framework Python adapté. Nous allons nous intéresser à `unittest`, dont la documentation est accessible ici : https://docs.python.org/3/library/unittest.html. Il existe des alternatives, notamment `pytest`.

Tests unitaires sans framework

L'idée consiste à faire correspondre à une fonction `maFonction()`, une autre fonction de test associée, nommée par exemple `test_maFonction()`, qui exécutera `maFonction())` avec l'utilisation du mot-clé `assert`. Supposons que l'on ait à coder une fonction `estPair(n)` qui indique si un nombre, passé en argument, est un nombre pair. La fonction renvoie alors la valeur `True`, sinon elle renvoie `False`. La définition de la fonction est la suivante :

exemples.py

```
def estPair(n):
    """
    @pre: n nombre entier
    @post: True si n est pair, False sinon
    return (n%2) == 0
```

On crée alors un fichier `tests_exemples.py`, et on y intègre le code suivant :

tests_exemples.py

```
def test_estPair():
    assert estPair(2) == True, 'Erreur dans la fonction estPair()'

## Exécution du test :
test_estPair ()
```

Comme on sait *à l'avance* que la fonction doit renvoyer `True` lorsque le paramètre est égal à 2, on peut comparer la valeur attendue avec la valeur effectivement retournée. Si les deux résultats sont incohérent, on sait immédiatement que la fonction contient un bug.

Il est évidemment possible de réaliser plusieurs tests :

tests_exemples.py

```
def test_estPair():
    assert estPair(2) == True, 'Erreur dans la fonction estPair()'
    assert estPair(1) == False, 'Erreur dans la fonction estPair()'
    assert estPair(21) == False, 'Erreur dans la fonction estPair()'
    assert estPair(100) == True, 'Erreur dans la fonction estPair()'
    # d'autres assertions peuvent suivre...

## Exécution du test :
test_estPair()
```

Le mot-clé **assert** permet donc de vérifier qu'une certaine assertion[2] est vraie et de lever une exception[3] si ce n'est pas le cas.

On n'oublie pas la programmation modulaire

Les principes généraux de la programmation modulaire ne doivent pas être obliés, d'autant qu'ils constituent un enjeu central dans la qualité de développement.

À ce titre, il est d'usage de *séparer* le code source « fonctionnel » de la partie tests unitaire. Si l'on travaille dans un fichier `exemples.py`, on place les tests unitaires dans un fichier `exemples_tests.py`. Ainsi, le code participant au fonctionnement de l'application, qui sera un jour déployé en production, est *indépendant* des tests, qui ne font pas à proprement parler de l'application.

Utilisation du framework `unittest`

Tests unitaires et programmation orientée objet

Ce paragraphe utilise des notions élémentaires de programmation orientée objet en Python, parmi lesquelles les classes et l'héritage.

Ces concepts vont être étudiés prochainement dans le cadre de la ressource R2.01. En attendant, la connaissance approfondie de ces notions n'est pas requise pour comprendre ce qui suit.

Dans un premier temps, nous utiliserons `unittest` de façon à vérifier que l'implémentation d'une fonction écrite en Python est correcte. On se base sur l'exemple très simple de la fonction valeur absolue, dont on rappelle la définition : pour tout nombre réel x, la valeur absolue de x est égale à x si $x \geq 0$ et à $-x$ sinon.

2. Au sens mathématique du terme.
3. En informatique, les exceptions sont une façon de gérer les erreurs. Cette notion sera abordée ultérieurement.

exemples.py

```
def val_abs(x):
    """
    @pre: i est un entier
    @post: retourne la valeur absolue de l'entier a
    """
    if i>0:
        return x
    else:
        return -x
```

Dans le fichier `tests_exemples.py`, on intègre le code suivant :

tests_exemples.py

```
import unittest
import exemples
class TestAbs(unittest.TestCase): # On crée une classe (cf. R2.01)
    """
    Classe de test permettant de valider le bon fonctionnement de val_abs
    """

    def test_val_abs(self):
        """
        @pre: -
        @post: vérifie le fonctionnement correct de la fonction my_abs
        """
        self.assertEqual(val_abs(1), 1)
        self.assertEqual(val_abs(0), 0)
        self.assertEqual(val_abs(-1), 1)

unittest.main() # Le sens précis de cette instruction sera étudié en R2.01
```

Analysons l'utilisation de `unittest`. Tout d'abord (première ligne du script), il faut importer le module unittest de façon à pouvoir utiliser ses fonctionnalités.
Avec unittest, un test unitaire s'écrit en étendant la classe `unittest.TestCase` et en ajoutant les méthodes permettant de tester les nouvelles fonctions. Dans le cas du calcul de la valeur absolue, nous avons ajouté la fonction `test_val_abs()`. Celle-ci vérifie que la fonction `my_abs()` retourne le résultat attendu pour trois valeurs différentes de son argument :

- `val_abs(0)` retourne `0` ;
- `val_abs(1)` retourne `1` ;
- `val_abs(-1)` retourne `1`.

Nous avons utilisé ici la méthode `assertEqual(a, b)` qui vérifie que `a==b`. Le framework `unittest` fournit plusieurs autres méthodes utiles :

`assertEqual(a, b)`	Vérifie que `a==b`
`assertNotEqual(a, b)`	Vérifie que `a!=b`
`assertTrue(x)`	Vérifie que le booléen `x` est vrai
`assertFalse(x)`	Vérifie que le booléen `x` est faux
`assertIs(a, b)`	Vérifie que `a` et `b` font référence au même objet
`assertIsNot(a, b)`	Vérifie que `a` et `b` ne font pas référence au même objet
`assertIsNone(a)`	Vérifie que la valeur `a` est `None`
`assertIsNotNone(a)`	Vérifie que la valeur `a` n'est pas `None`
`assertIn(a, b)`	Vérifie que `a` appartient à la liste `b`
`assertIn(a, b)`	Vérifie que `a` n'appartient pas à la liste `b`
`assertInstance(a, b)`	Vérifie que `a` est une instance de la classe `b`
`assertInstance(a, b)`	Vérifie que `a` n'est pas une instance de la classe `b`

Ces méthodes lèvent une exception lorsque l'une des vérifications échoue. Les exceptions sont remontées par `unittest` et un rapport d'erreur est affiché dans le terminal.

Exercices d'application

Travail à rendre

Vous devrez déposer sur EPREL deux fichiers sources : `r203_td2_<nomDeFamille>.py` et les tests associés, dans le fichier `r203_td2_<nomDeFamille>_tests.py`. On créera un dépôt Git à la racine du projet pour faciliter le suivi des versions [a].

a. Même si le dépôt ne sera pas à rendre.

Calcul de la médiane

Exercice 1. On s'intéresse à une fonction qui calcule la médiane de trois nombres. Des étudiant·es ont proposé trois versions différentes de solutions, que l'on donne en annexe page 7.

1) Testez les trois versions proposées pour différents jeux de données. Quelle(s) version(s) vous semble(nt) correcte(s) ?
2) Utilisez `unittest` pour implémenter des tests unitaires. On réalisera (au moins) dix tests pour chaque version de la fonction.

Exercice 2. La solution de test unitaire développée dans l'exercice précédent n'est pas totalement satisfaisante. En effet, la fiabilité de ce test est fortement rédyute en raison du nombre limité d'assertions de vérifications et surtout du choix des jeux de valeurs utilisés pour produire le test.

Proposer une réécriture du test de la médiane sachant que la valeur médiane d'un ensemble de trois valeurs stockées dans une liste est, une fois la liste triée, la valeur du milieu de cette liste. Ainsi, par exemple la liste de trois valeurs `L=[10, 1, 6]` a pour valeur médiane `6` puisque cette valeur se retrouve à l'index `1` de la liste (milieu d'une liste de 3 éléments) une fois triée.

On pourra générer cinquante tests pour chaque version, avec des données tirées aléatoirement.

Plus grand diviseur

Définition (Plus grand diviseur)**.** Soit $n \geq 2$ un nombre entier. On appelle *plus grand diviseur* de n le plus grand entier $d < n$, tel que le reste dans la division euclidienne de n par d est nul.

En particulier, avec cette définition, les entiers naturels 0 et 1 n'ont pas de plus grand diviseur. Une fonction qui calcule le plus grand diviseur d'un entier devrait donc renvoyer `None` lorsque l'entier est égal à `0` ou `1`. Dans l'annexe page 8, on propose quatre fonctions qui implémentent un calcul de plus grand diviseur.

Exercice 3. Intégrer le code des quatre fonctions proposées dans le fichier `r203_td2_<nomDeFamille>.py` et implémenter des tests unitaires pour ces fonctions dans le fichier `r203_td2_<nomDeFamille>_tests.py`. On veillera à tester tous les cas d'usage.

Annexes

Médiane de trois nombres

r203_td2.py

```python
# Version 1
def mediane1(a, b, c):
    if (a<= c and c<=b) or (b<=c and c<=a):
        mediane=c
    elif (b<=a and a<=c) or (c<=a and a<=b):
        mediane=a
    else:
        mediane=b
    return mediane

# Version 2
def mediane2(a, b, c):
    if a>=b and a<=c:
        median=a
    elif b>=a and b<=c:
        mediane=c
    elif c>=a and c<=b:
        mediane=c
    return mediane

# Version 3
def mediane3(a, b, c):
    mediane = a
    if a < b < c and c < b < a :
        mediane = b
    if b < c < a and a < c < b :
        mediane = c
    return mediane
```

Plus grand diviseur

r203_td2.py

```python
# Version 1
def pgd1(a):
    """
    Exemple de code version 1
    """
    l = []
    if a == 0:
        return None
    for i in range(1, a):
        while i <= a-1:
            if a%i == 0:
                l.append(i)
    l.reverse()
    return l[0]

# Version 2
def pgd2(a):
    """
    Exemple de code version 2
    """
    l = []
    for i in range(a+1):
        if a <= 2:
            return None
        elif i >= 1 and a%i == 0 and i < a:
            l.append(i)
    return max(l)

# Version 3
def pgd3(a):
    """
    Exemple de code version 3
    """
    if a == 0 or a == 1:
        return None
    else:
        l = []
        i = 1
        while i < a:
            if a%i == 0:
                l.append(i)
            i += 1
        return l[a]

# Version 4
def gcd4(a):
    """
    Exemple de code version 3
    """
    if a <= 1:
        return None
    for div in range(a-1, 1, -1):
        if a%div == 0:
            return div
    return 1
```