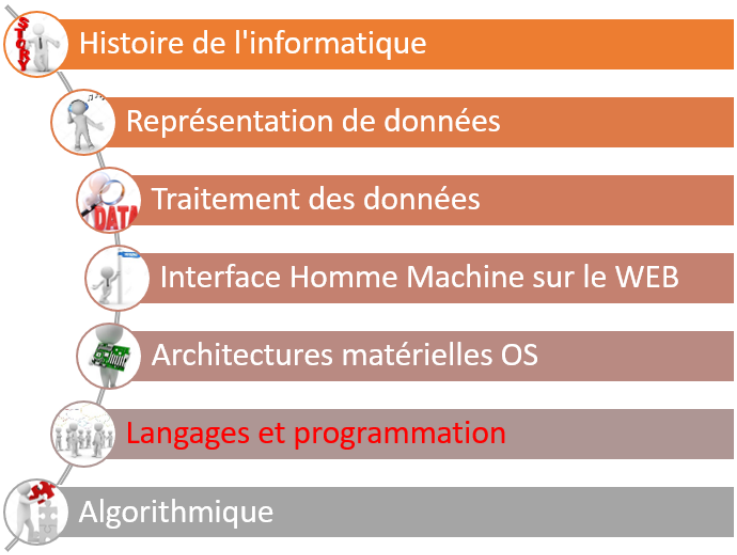

Table des matières

1 Tester une fonction avec Unittest

1.1 Un premier exemple résolu

1.1.1 Jeu de tests

Commençons par illustrer les notions de tests et de spécifications d'une fonction par le biais de l'exemple décrit ci-dessous :

On souhaite coder une fonction dont le but est de retourner la liste contenant tous les nombres entiers pairs contenus dans une liste de nombres entiers naturels entrée en paramètre de cette fonction.
Nous enregistrerons donc un fichier `premier_exemple_resolu.py` contenant la fonction `tri_pairs()`. Mais avant de coder cette fonction (ce qui n'est pas très difficile, d'ailleurs vous l'avez codée en exercice en début d'année lors d'un travail sur les parcours de listes) nous allons réfléchir aux résultats que l'on peut attendre d'une telle fonction (en essayant de lister tous les cas de figure possibles) :

- cas "zéro" pour une liste vide, la fonction retournera une liste vide ;
- pour une liste = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10], la fonction renverra [2, 4, 6, 8, 10] ;
- pour une liste désordonnée [10, 93, 24, 55, 26, 77, 38, 9, 130], la fonction renverra [10, 24, 26, 38, 130] ;
- pour une liste ne contenant que des impairs [1, 3, 5, 7, 9], la fonction retournera une liste vide.

Vous noterez que les résultats attendus en sortie sont obtenus a priori, sans avoir fait travailler la fonction !

Ci-dessous nous avons codé un script de tests pour vérifier que notre fonction "fait" bien ce que l'on attend d'elle :

```
1  from premier_exemple_resolu import tri_pairs
2
3  import unittest
4
5  class test_tri_pairs(unittest.TestCase):
6
7      def test_ok_vide(self):
8          self.assertEqual( tri_pairs([]), [], "(Attendu : [])")
9      def test_ok_pairs(self):
10         self.assertEqual( tri_pairs([1,2,3,4,5,6,7,8,9,10]),  [2,4,6,8,10], "(Attendu :   [2,4,6,8,10])")
11
12     def test_ok_desordonnee(self):
13         self.assertEqual( tri_pairs([10,93,24,55,26,77,38,9,130]), [10,24,26,38,130], "(Attendu :
       [10,24,26,38,130])")
14
15     def test_ok_impairs(self):
16         self.assertEqual( tri_pairs([1,3,5,7,9]), [], "(Attendu : [])")
17
18 if __name__ == '__main__':
19     unittest.main()
```

 test_premier_exemple_resolu.py

Quelques explications :

Ce script commence par importer la fonction `tri_pairs()` (que nous coderons ensuite) à partir du fichier `premier_exemple_resolu.py` et le module `unittest`.
On créé alors une classe de tests de cette fonction héritant de `unittest.TestCase`, puis on définit un test dans une méthode dont le nom commence par test pour chacun des 4 cas identifiés tout à l'heure. Ces 4 tests utilisent la méthode `assetEqual(a, b)` qui vérifie que les deux variables a et b sont égales.
Les deux dernières lignes du script nous permettent d'exécuter le test.

1.1.2 La fonction à tester

Voyons maintenant la fonction `tri_pairs()`. (On supposera ici que le paramètre en entrée `liste_origine` est bien une liste d'entiers naturels) :

```
def tri_pairs(liste_origine):
    """
    retourne la liste contenant tous les nombres entiers pairs
    contenus dans une liste de nombres entiers naturels entrée
    en paramètre

    Arguments :
    entree : liste d'entiers naturels

    Sortie :
    sortie : liste d'entiers naturels
    """

    return [nb for nb in liste_origine if nb%2==0]
```

 premier_exemple_resolu.py

1.1.3 Docstring et spécification d'une fonction

On remarquera la création d'une docstring :

- placée juste après la signature de la fonction entre deux paires de : """
- on y décrit le fonctionnement de la fonction : ce commentaire qu'on a ajouté décrit ce que fait la fonction, ainsi que les paramètres qu'il faut lui fournir. Il est donc complet et permet d'utiliser la fonction comme il faut.
- on y définit les spécifications de la fonction :
  - les pré-conditions portant sur les arguments en entrée : ce sont toutes les conditions qui doivent être satisfaites avant de pouvoir appeler la fonction, que ce soit sur des variables globales ou sur ses paramètres ;
  - les post-conditions sur les variables en sortie : ce sont toutes les conditions qui seront satisfaites après appel de la fonction, si les pré- conditions étaient satisfaites, que ce soit sur des variables globales ou sur l'éventuelle valeur renvoyée.
- cela permet à un utilisateur qui lit le code de comprendre le codage de la fonction
- cela permet à un utilisateur du code, qui ne le lit pas, d'avoir accès à la documentation par le biais de la fonction `help()` affichant cette documentation dans un `shell`.

Travail à faire :
Enregistrer les deux fichiers `test_premier_exemple_resolu.py` et `premier_exemple_resolu.py` dans un même répertoire et lancer le script `test_premier_exemple_resolu.py`. Vérifier que le jeu de tests ne détecte aucune erreur.

1.2 Un deuxième exemple

On rappelle qu'un anagramme est un mot formé en changeant l'ordre des lettres d'un autre mot, par exemple "rage" est un anagramme de "gare".

On souhaite écrire une fonction qui prend deux chaînes de caractères en paramètres et qui retourne `True`, si la deuxième chaîne est un anagramme de la première (et réciproquement), `False`, sinon.

1.2.1 Jeu de tests

Voici une série de tests qui ont été écrits pour vérifier le fonctionnement d'une telle fonction :

- on teste si deux anagrammes sont reconnus ;
- on teste si deux mots qui ne sont pas des anagrammes sont bien identifiés comme tels ;
- on teste si deux mots qui ont le même nombre de lettres, et sont écrits avec les mêmes caractères mais ne sont pas des anagrammes sont bien identifiés comme tels.

Les tests utilisent les deux méthodes `assertFalse(x)` et `asserTrue(x)` qui vérifient respectivement, comme leurs noms l'indiquent, que l'assertion `x` est respectivement `False` ou `True`.

```
from anagramme_1 import anagramme

import unittest

class test_anagramme(unittest.TestCase):

    def test_ok(self):
        self.assertTrue(anagramme('aasdfr','farsda'))    # Attendu :   True

    def test_not_ok_1(self):
        self.assertFalse( anagramme('aasdfr','ffarsda')) # Attendu : False

    def test_not_ok2(self):
        self.assertFalse( anagramme('hjgulj','lujhgh')) # Attendu : False


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

test_anagramme_1.py

1.2.2 Une première fonction `anagramme()`

Voici une première fonction `anagramme()`. Cette fonction est mal codée :

```
def anagramme(chaine1,chaine2):
    if len(chaine1) == len(chaine2):
         for car in chaine1:
            if car not in chaine2:
                return False
         for car in chaine2:
              if car not in chaine1:
                return False
         return True
    else:
        return False
```

anagramme_1.py

Travail à faire :
Faire fonctionner le jeu de tests précédent et préciser l'erreur de codage qui a été commise dans cette première fonction `anagramme()`.

1.2.3 Une deuxième fonction `anagramme()`

Voici une deuxième fonction `anagramme()` enregistrée dans le fichier `anagramme_2.py`. Cette fonction est correcte mais n'est ni documentée ni spécifiée :

```
def anagramme(chaine1, chaine2):

    if len(chaine1)==len(chaine2):
        for car in chaine1:
            if car not in chaine2:
                return False
            else:
                if chaine2.index(car) == len(chaine2):
                    chaine2 = chaine2[:len(chaine2)-1]
                else:
                    chaine2 = chaine2[:chaine2.index(car)] + chaine2[chaine2.index(car)+1:]
        return True
    else:
        return False
```

 anagramme_2.py

Travail à faire :
Documenter et spécifier cette fonction (vous veillerez notamment à comprendre pourquoi cette fonction corrige l'erreur précédente).
De plus pour que la fonction puisse travailler correctement il faut que `chaine1` et `chaine2` soient des chaînes de caractères. On pourra aussi s'assurer que ces deux chaînes ne sont pas vides, sinon aucun intérêt de faire tourner notre fonction.
Pour vérifier cela avant l'exécution de la fonction, nous allons utiliser des assertions. C'est un moyen simple de s'assurer qu'une condition est remplie avant de continuer l'exécution de la fonction.
La syntaxe est simple : `assert essai`
Si `essai` renvoie `True` , l'exécution se poursuit, sinon une exception `AssertionError` est levée.

Travail à faire :
Après la docstring rédigée, insérer deux assertions pour vérifier que les deux entrées sont des chaînes de caractères d'une part et qu'aucune n'est vide d'autre part.
Nous verrons plus tard que l'on peut aussi gérer les exceptions avec un bloc `try: except:`

Travail à faire :
Reprendre le jeu de tests précédent `test_anagramme_1`, modifier le pour l'adapter à cette nouvelle fonction `angramme()`, vous définirez en particulier deux nouvelles fonctions de tests, utilisant la méthode `assertRaises()` qui permet de vérifier que la fonction lève les exceptions attendues. La syntaxe étant : `with self.assertRaises(AssertionError): fonction à tester avec certains paramètres....`
Vous enregistrerez ce nouveau test dans un fichier `test_anagramme_2`.
Faire fonctionner ce nouveau jeu de test.

1.3 Liste des principales méthodes unittest :

Voici les principales méthodes que l'on peut utiliser dans `unittest` :

Méthode	Explications
assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertTrue(x)	x is True
assertFalse(x)	x is False
assertIs(a, b)	a is b
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None
assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b
assertRaises(exception, fonction, *args, **kwargs)	Vérifie que la fonction lève l'exception attendue.

2 Assertions

2.1 Programmation défensive

L'instruction `assert` existe dans presque tous les langages de programmation. Cela aide à détecter les problèmes au début de votre programme, où la cause est claire, plutôt que plus tard comme effet secondaire d'une autre opération. La syntaxe est simple :

```
assert condition
```

Vous dites au programme de tester cette condition et de déclencher immédiatement une erreur si la condition est fausse. Cela interrompt le programme.

Les assertions peuvent inclure un message facultatif :

```
assert condition, "Ooops, assertion échouée"
```

`assert` est une instruction, pas besoin de parenthèses comme pour une fonction.

Exemple :
Dans l'exemple suivant, la fonction `find` vérifie si le tableau n'est pas vide, puis recherche l'élément `x`.

```
def find(tab, x):
    assert len(tab) > 0, "Tableau est vide"
    if x in tab:
        return True
    return False
```

Ainsi le code suivant :

```
tab =[]
find(tab, 1)
```

renvoie :

```
Traceback (most recent call last):
  File "C:\Users\LAMBERT\Desktop\test.py", line 8, in <module>
    print(find(tab, 1))
  File "C:\Users\LAMBERT\Desktop\test.py", line 2, in find
    assert len(tab) > 0, "Tableau est vide"
AssertionError: Tableau est vide
```

2.2 Générer des tests aléatoires

On peut à l'aide de l'instruction `assert` générer des tests aléatoirement.
Dans le script ci-après on considère une fonction `tableau_croissant(n)` qui génère et renvoie un tableau aléatoire dont les éléments sont des entiers naturels rangés dans l'ordre croissant.
Pour tester cette fonction :

- On définit une fonction `est_croissant(tab)` chargée de vérifier que les éléments du tableau `tab` sont bien rangés dans l'ordre croissant ;
- On applique cette vérification à une série de tableaux générés aléatoirement par `tableau_croissant(n)`. Dans l'exemple on teste dix tableaux de chacune des tailles de 0 à 19.

```python
from random import randint

def tableau_croissant(n):
  """
  Génère un tableau aléatoire dont les éléments
  sont des entiers naturels rangés dans l'odre croissant
  Entrée : un entier naturel n
  Sortie : un tableau de taille n, rangé dans l'ordre croissant
  """
  tab = [randint(0, 10)]*n
  for i in range(1, n):
    tab[i] = tab[i - 1] + randint(0, 10)
  return tab


if __name__ == '__main__':
  def est_croissant(tab):
    """
    Teste si les élts de tab sont rangés dans l'ordre croissant
    Entrée : tab une liste de nombres
    Sortie : un booléen
    """
    for i in range(len(tab) - 1):
      if tab[i] > tab[i + 1]:
        return False
    return True

  for n in range(20):
    for _ in range(10):
      tab = tableau_croissant(n)
      assert est_croissant(tab)
```

 est_croissant.py

3 Exceptions

3.1 Signaler une erreur avec une exception

Lors de l'exécution d'un programme, l'interprète Python détecte un problème empêchant la réalisation complète du programme, en programmation on appelle ces erreurs des exceptions.

```
>>> tab = [1, 2, 3, 4, 5, 6]
>>> tab[7]
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    tab[7]
IndexError: list index out of range
>>> 1/0
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    1/0
ZeroDivisionError: division by zero
>>> int('ABC')
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    int('ABC')
ValueError: invalid literal for int() with base 10: 'ABC'
```

Voici la liste des principales exceptions en Python :

exception	contexte
NameError	accès à une variable inexistante
IndexError	accès à un indice invalide d'un tableau
KeyError	accès à un clé inexistante d'un dictionnaire
ZeroDivisionError	division par zéro
TypeError	opération appliquée à des valeurs incompatibles
ValueError	paramètres inadaptés passé à une fonction

3.2 Rattraper une exception

Intéressons-nous au script suivant :

```
nombre = float(input('Entrer un nombre : '))
inverse = 1.0/nombre
print("Inverse de", nombre, " : ", inverse)
```

 inverse.py

Quand vous entrez un nombre, tout se déroule normalement, mais on peut avoir des problèmes :

```
>>>
 RESTART: C:/Users/gtram/Documents/Lycee/COURS/1ere_NSI/20_21/13_Test_et_debbugage/inverse.py
Entrer un nombre : 10
Inverse de 10.0 : 0.1
>>>
 RESTART: C:/Users/gtram/Documents/Lycee/COURS/1ere_NSI/20_21/13_Test_et_debbugage/inverse.py
Entrer un nombre : bonjour
Traceback (most recent call last):
  File "C:/Users/gtram/Documents/Lycee/COURS/1ere_NSI/20_21/13_Test_et_debbugage/inverse.py", line 2, in
    <module>
    nombre = float(chaine)
ValueError: could not convert string to float: 'bonjour'
>>>
 RESTART: C:/Users/gtram/Documents/Lycee/COURS/1ere_NSI/20_21/13_Test_et_debbugage/inverse.py
Entrer un nombre : 0
Traceback (most recent call last):
  File "C:/Users/gtram/Documents/Lycee/COURS/1ere_NSI/20_21/13_Test_et_debbugage/inverse.py", line 3, in
    <module>
    inverse = 1.0/nombre
ZeroDivisionError: float division by zero
```

Python a détecté une erreur : une exception est levée. Ici nous avons une exception de type `ZeroDivisionError` (division par 0) et une exception de type `ValueError`. Une exception arrête l'exécution normale d'un programme.
Heureusement, il est possible de gérer les exceptions pour éviter l'arrêt brutal du programme. Pour cela, on utilise conjointement les instructions `try` et `except`. L'instruction `else` est optionnelle :

```
try:
    nombre = float(input('Entrer un nombre : '))
    inverse = 1.0/nombre
except:
    # ce bloc est exécuté si une exception est levée dans le bloc try
    print("Erreur !")
else:
    # on arrive ici si aucune exception n'est levée dans le bloc try
    print("L'inverse de", nombre, "est :", inverse)
```

On peut distinguer les différents types d'exceptions :

```
try:
    nombre = float(input('Entrer un nombre : '))
    inverse = 1.0/nombre
except ValueError:
    # ce bloc est exécuté si une exception est levée dans le bloc try
    print("Erreur !")
except ZeroDivisionError:
    # ce bloc est exécuté si une exception de type ZeroDivisionError est levée dans le bloc try
    print("Division par zéro !")
else:
    # on arrive ici si aucune exception n'est levée dans le bloc try
    print("L'inverse de", nombre, "est :", inverse)
```

Si l'utilisateur entre autre chose qu'un nombre, le programme va générer une erreur et s'arrêter. Pour éviter cela, il faut remplacer la ligne

```
nombre = float(input('Entrer un nombre : '))
```

par :

```
while True:
    try:
        nombre = float(input('Entrer un nombre : '))
        inverse = 1.0/nombre
        break
    except ValueError: print("Réponse non valide. Réessayer !")
    except ZeroDivisionError: print("Division par zéro !")
```

C'est une boucle infinie (`while True`) dont le seul moyen de sortir (`break`) est d'entrer un nombre. Tant que l'exception `ValueError` est levée, un message d'erreur créé par l'utilisateur est affiché à l'écran et le programme repose la question.

4 Exercices

Exercice n° 1.

Depuis l'ajustement du calendrier grégorien, une année est bissextile (et a 366 jours) :
- si l'année est divisible par 4 et non divisible par 100, ou
- si l'année est divisible par 400.

Sinon, l'année n'est pas bissextile (et a 365 jours).

On souhaite créer une fonction *annee_bissextile()* qui prend en paramètre un nombre entier naturel et renvoie *True* si l'année est bissextile et *False* sinon.

1. Travail sur papier :
   (a) On souhaite commencer par écrire un test de la fonction.
   Après avoir relu le rappel ci-dessus proposé un jeu de tests de la fonction écrit en langage naturel.
   (b) Pour la fonction *annee_bissextile()*, quelles sont les variables en entrée et en sortie ? De quel(s) type(s) sont-elles ? Quelles sont les spécifications pour cette fonction ?
   (c) Écrire en langage naturel un algorithme possible pour la fonction *annee_bissextile()*.
2. Coder le script de test proposé et la fonction en python.
   Appliquer le test, corriger les éventuelles erreurs...

Exercice n° 2.

Dans cet exercice on munit le plan d'un repère orthonormé $\left(O\,;\,\vec{i},\,\vec{j}\right)$. Tout point du plan sera alors défini par ses coordonnées à l'aide d'un tuple.

1. Écrire une fonction *distance()* qui prend en paramètres deux points et renvoie la distance séparant ces deux points.

   Dans la suite de l'exercice, on considère une liste, *liste_origine*, de points du plan (définis chacun par son tuple de coordonnées) et A un point quelconque.
   On souhaite écrire une fonction, *tri_distance_croissante()*, qui prend en paramètres *liste_origine* et A et qui renvoie une liste de ces mêmes points rangés dans l'ordre croissant des distances séparant chacun de ces points du point A.
   Ainsi si l'on tape :

```
1    liste_origine = [(2, 3), (3, 1), (8, 3), (7, 4), (7, 5), (2, 6), (1, 3), (5, 8), (12, 43), (7,
     2)]
2    print(tri_distance_croissante(liste_origine, (0, 0)))
3
```

   on obtient :

```
1    [(3, 1), (1, 3), (2, 3), (2, 6), (7, 2), (7, 4), (8, 3), (7, 5), (5, 8), (12, 43)]
2
```

2. Travail sur papier
   (a) Écrire un test (en langage naturel) de cette fonction. Vous veillerez à penser à tester tous les cas possibles (liste ordonnée, liste désordonnée, points équidistants...)
   (b) Spécifier la fonction *tri_distance_croissante()*. Vous penserez à insérer des assertions pour vérifier que les entrées sont correctes avant d'exécuter la fonction. Modifier en conséquence le test précédent.
3. Coder le script de test proposé et la fonction en python.
   Appliquer le test, corriger les éventuelles erreurs...

Exercice n° 3.

On souhaite créer une fonction `plus_grand()` qui prend en paramètre trois nombres entiers naturels et qui renvoie le plus grand des trois.

1. Écrire un jeu de tests de la fonction *plus_grand()*.
2. On considère les deux fonctions ci-dessous :

```
def plus_grand_v1(a, b, c):
  if a > b:
    if a > c:
      return a
    else:
      return c
  else:
    if b > c:
      return b
    else:
      return c

def plus_grand_v2(a, b, c):
  if a > b and a > c:
    return a
  elif b > a and b > c:
      return b
  else:
      return c
```

 plus_grand.py

Utiliser votre test pour déterminer laquelle de ces deux fonctions est fausse et corriger l'erreur.

Exercice n° 4.

1. L'objectif est de tester une fonction qui prend en paramètre un entier naturel et qui renvoie `True` si ce nombre est premier et `False` sinon.
   Écrire le jeu de tests correspondant.
2. On considère la fonction ci-dessous :

```
def premier(n):
  from math import sqrt

  """
  Entrée : n un entier naturel
  Sortie : Booléen

  Renvoie True si l'entier naturel n est premier et False sinon
  """
  if n == 0 or n == 1:
    return False
  # On teste les divisuerus potentiels jusqu'à racine de n.
  N = int(sqrt(n))
  for i in range(2, N):
    if n %i ==0 :
      return False
  return True
```

 premier.py

Utiliser votre test pour déterminer l'erreur. La corriger.

Exercice n° 5.

1. Écrire une fonction `tableau_aléatoire(n)` prenant en paramètre un entier `n` et qui renvoie un tableau de taille `n` dont les éléments sont des entiers compris entre `1` et `n - 1`.
2. Écrire une fonction `doublon(tab)` prenant en paramètre un tableau `tab` et renvoyant un élément qui apparait au moins deux fois dans `tab` (il en existe au moins un) et `None` sinon.
3. Écrire une fonction `vérifie_doublon(tab, r)` prenant un paramètres un nombre entier `r` et un tableau `tab`, et qui renvoie `True` si `tab` contient au moins deux fois la valeur `r`.

4. En prenant modèle sur le paragraphe 2.2, générer un test aléatoire de votre fonction `tableau_aléatoire(n)`.

Exercice n° 6.

1. Par deux élèves, réaliser le docstring d'une fonction d'échanger (le but est d'échanger deux valeurs dans un tableau)
2. Eleve 1 réalisera le code de la fonction échanger.
3. Eleve 2 réalisera le code des tests.

Exercice n° 7.

1. Par deux élèves, réaliser le docstring d'une fonction maximum d'un tableau(sans utiliser la fonction max !) ((le but est de trouver le maximum et son indice dans un tableau)
2. Eleve 1 réalisera le code de la fonction échanger.
3. Eleve 2 réalisera le code des tests.

Exercice n° 8.

1. Par deux élèves, réaliser le docstring d'une fonction minimum d'un tableau(sans utiliser la fonction min !) ((le but est de trouver le minimum et son indice dans un tableau)
2. Eleve 1 réalisera le code de la fonction échanger.
3. Eleve 2 réalisera le code des tests.