

Module 16 : Tests et déploiement avec le Pi

TESTS DE CORRECTION DES ERREURS, TESTS UNITAIRES, TESTS
D'INTÉGRATION, PROFILAGE DU CODE, DÉPLOIEMENT D'UNE
APPLICATION PI, SÉCURITÉ D'UNE APPLICATION, GUIDE DE
FONCTIONNEMENT DE L'APPLICATION

A solid green horizontal bar at the bottom of the slide.

Tests et correction des erreurs

Le codage de tests unitaires est fortement suggéré lors de l'élaboration d'un programme informatique.

Python offre plusieurs modules pour répondre aux besoins du programmeur

- `pytest`, `unittest`, `doctest`

Python offre aussi le `REPL` qui permet d'inspecter son code interactivement et vérifier instantanément le contenu d'un objet (type et méthodes)

Avec le Raspberry Pi, les ressources sont souvent limitées. Pour mesurer et améliorer le temps d'exécution de notre projet GPIO, on peut profiler notre code et calculer le nombre de ressources nécessaires à l'exécution.

- Les modules `timeit` et `cProfile` permettent cette option.

Test d'intégration

Test unitaire

Test d'intégration

Test de validation

Le test d'intégration vérifie le bon fonctionnement d'une partie précise d'un programme

- Par exemple, un module.
- Il permet aussi de vérifier l'aspect fonctionnel, les performances et la fiabilité du programme.
- Il cible que les composants de votre application fonctionnent les uns avec les autres

En Python, on va utiliser le mot `assert` du module `pytest` pour faire un test d'intégration

- Soit l'exemple suivant pour montrer le fonctionnement de `assert`
- Si la condition n'est pas égale au mot demandé, il y aura une erreur

```
x = "hello"

#if condition returns True, then nothing happens:
assert x == "hello"

#if condition returns False, AssertionError is raised:
assert x == "goodbye"
```

```
Traceback (most recent call last):
  File "demo_ref_keyword_assert.py", line 5, in <module>
    assert x == "goodbye"
AssertionError
```

Source : https://www.w3schools.com/python/ref_keyword_assert.asp

Exemple de test d'intégration

Soit les deux fichiers suivants:

- Module pour calculer deux fonctions mathématiques
- Module pour tester les deux fonctions mathématiques.

Lorsque l'on veut tester nos deux fonctions, on utilise la ligne suivante:

- `assert rep_a == rep_b`

Voici le résultat

```
mpressmain12/exempleTestIntegration.py
Test 0 réussi
Traceback (most recent call last):
  File "c:\Users\sdeschenes\ProjetsPython\ExemplesSema
    test1_a_b()
  File "c:\Users\sdeschenes\ProjetsPython\ExemplesSema
    assert rep_a == rep_b
AssertionError
```

```
exempleModule.py
1 def a(x):
2     return x + 1
3
4 def b(x):
5     return x * 2
6
```

```
exempleTestIntegration.py
1 from exempleModule import a, b
2
3 def test0_a_b():
4     rep_a = a(7)
5     rep_b = b(4)
6     assert rep_a == rep_b
7     print("Test 0 réussi")
8
9 def test1_a_b():
10    rep_a = a(3)
11    rep_b = b(4)
12    assert rep_a == rep_b
13    print("Test 1 réussi")
14
15 test0_a_b()
16 test1_a_b()
```

Source: <https://mq-software-carpentry.github.io/python-testing/09-integration/>

Tests unitaires

Tests unitaires

Tests d'intégration

Tests de validation

Le test unitaire est une procédure vérifiant le bon fonctionnement d'une partie plus précise d'un programme.

- Une petite partie du programme. Par exemple: une méthode.

Avec Python, le cadre applicatif (*framework*) `unittest` permet de gérer l'automatisation des tests, le partage de code, l'agrégation de tests et l'indépendance des tests.

La conception d'un plan de test de base consiste à importer le module `unittest`.

- On crée une classe qui va hériter de `unittest.TestCase`.

Dans cette classe, on va créer plusieurs méthodes qui représentent plusieurs batteries de tests.

- Un test permet de vérifier qu'une méthode retourne une valeur spécifique.
- On utilise les assertions (méthodes) pour vérifier qu'une valeur est égale à Vrai (`True`) ou ne l'est pas, que deux objets sont égaux ou ne le sont pas, qu'une exception est renvoyée, etc.
- Types d'assertions: `assertTrue`, `assertFalse`, `assertEqual`, `assertNotEqual`, `assertRaise`, `assertNone`, `assertIsNone`

Par la suite, on exécute ses tests

- À l'exécution, un rapport est affiché : `OK` pour un test concluant et `FAIL` pour un test non concluant.

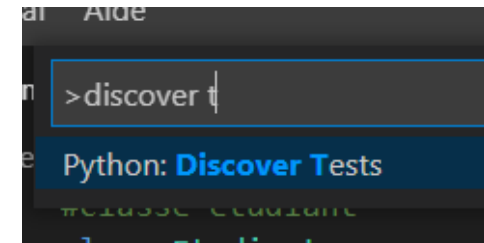
Configuration des tests dans VS Code

On va se créer un dossier Tests dans notre projet

- Ce dossier sera utilisé pour mettre nos tests

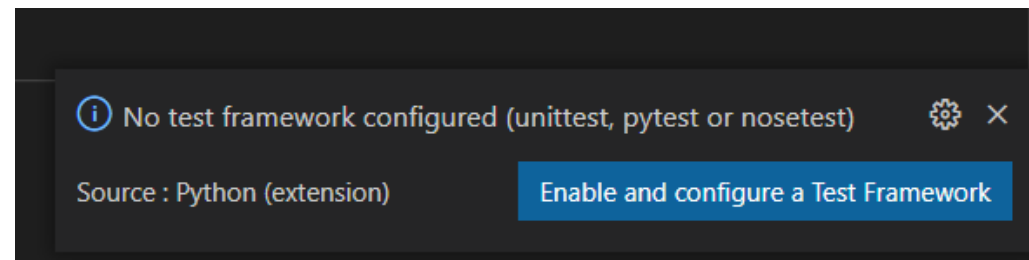
Ensuite, on appuie sur F1 ou CTRL+SHIFT+P

- Dans la barre de recherche, écrivez la commande Discover Tests



Par la suite, en bas à droite, une boîte de message devrait s'afficher pour la configuration du cadre applicatif (Framework).

- On doit appuyer sur le bouton Enable and configure a Test Framework



VS Code: configuration du cadre applicatif

Pour configurer le cadre applicatif

- Ensuite, on doit choisir notre cadre applicatif (framework) entre unittest, pytest et nose.
- Dans notre cas, on va choisir unittest

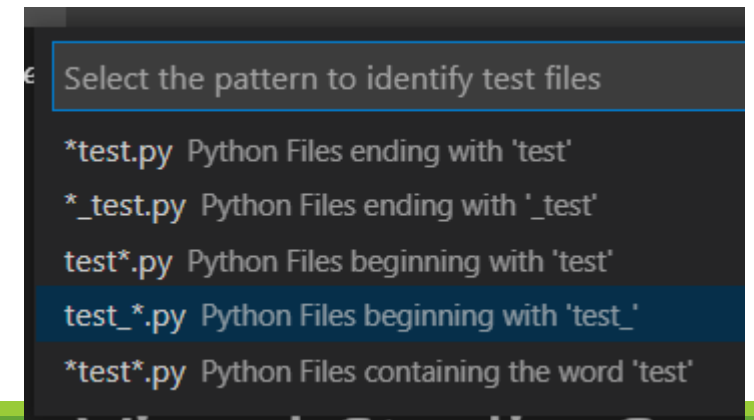
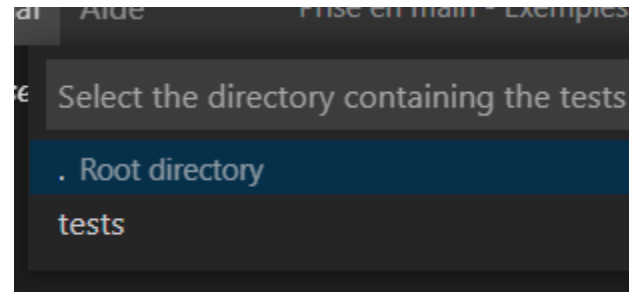
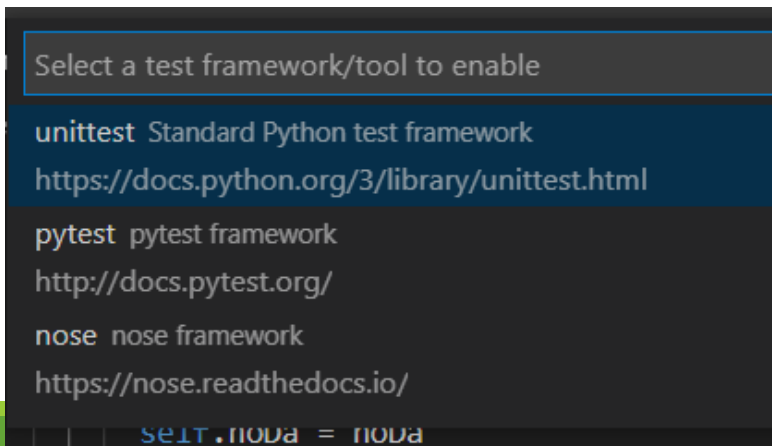
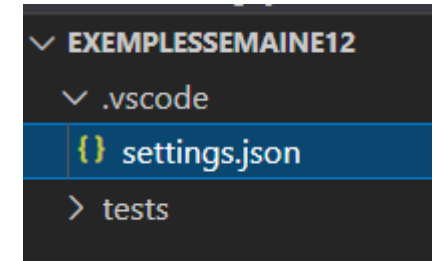
Ensuite, on choisit le dossier qui contiendra les tests (racine ou autre)

- On sélectionne notre dossier Tests

Par la suite, on doit choisir comment les fichiers de tests seront identifiés (pattern).

- J'ai choisi l'option 4 avec test_*.py au début du fichier.

Un dossier .vscode sera créé avec un fichier settings.json à l'intérieur. Ce fichier contient les paramètres des tests.



Cadre applicatif unittest

Résultats du test

Le test unitaire dans Python à trois résultats possibles:

Résultats possibles	Description
OK	Si tous les cas de tests sont réussis
Échec / Failure	Si l'un des cas de test a échoué et a déclenché une exception AssertionError
Erreur / Error	Si une exception autre que l'exception AssertionError est déclenché

Cadre applicatif unittest

fonctions de bases

Méthodes	Vérifie que...
<code>assertEqual(a,b)</code>	<code>a==b</code>
<code>assertNotEqual(a,b)</code>	<code>a!=b</code>
<code>assertTrue(x)</code>	<code>bool(x)</code> is True
<code>assertFalse(x)</code>	<code>bool(x)</code> is False
<code>assertIs(a,b)</code>	<code>a</code> is <code>b</code>
<code>assertIsNot(a,b)</code>	<code>a</code> is not <code>b</code>
<code>assertIsNone(x)</code>	<code>x</code> is none
<code>assertIsNotNone(x)</code>	<code>x</code> is not none
<code>assertIn(a,b)</code>	<code>a</code> in <code>b</code>
<code>assertNotIn(a,b)</code>	<code>a</code> not in <code>b</code>
<code>assertIsInstance(a,b)</code>	<code>isinstance(a,b)</code>
<code>assertNotIsInstance(a,b)</code>	<code>not isinstance(a,b)</code>

Cadre applicatif unittest

Exemple d'une classe test

Voici un exemple de base pour classe de tests unitaires.

Le nom du fichier est `test_exempleTest`.

- Il est important de respecter `test_*` dans le nom puisque nous avons configuré cette option dans VS Code auparavant.

On importe le module `unittest`

Dans la classe `Tester`, on ajoute `unittest.TestCase`

- Pour faire les tests, on déclare plusieurs méthodes.
 - Dans notre cas, nous avons `test_string` et `test_booleen`
 - On utilise l'attribut `assertEqual` pour tester nos deux valeurs

Pour exécuter nos tests, on doit ajouter `unittest.main()`

```
test_exempleTest.py
import unittest

class Tester(unittest.TestCase):
    def test_string(self):
        a = "Allo"
        b = "Allo"
        self.assertEqual(a, b)

    def test_booleen(self):
        a = True
        b = True
        self.assertEqual(a, b)

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

VS Code: exécution des tests unitaires

Pour exécuter nos tests unitaires, on doit mettre notre fichier de test dans le dossier test

Ensuite, on ouvre notre fichier de test et on appuie sur clic-droit de la souris > **Exécuter le fichier de test unitaire courant**

Les tests vont s'exécuter dans la sortie du programme

```
import unittest

class Tester(unittest.TestCase):
    def test_string(self):
        a = "Allo"
        b = "Allo"
        self.assertEqual(a, b)

    def test_booleen(self):
        a = True
        b = True
        self.assertEqual(a, b)

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

PROBLÈMES SORTIE CONSOLE DE DÉBOGAGE TERMINAL

test_valide (test_conversionTemperature.TestConversionTemperature):

Traceback (most recent call last):

File "c:\Users\sdeschenes\ProjetsPython\ExemplesSemaine1\test_conversionTemperature.py", line 10, in test_valide

self.assertEqual(valeurB, resultat)

AssertionError: -40 != -330.14222222222224 within 7 places

Run Code

Atteindre la définition

Atteindre les références

Aperçu

Find All References

Generate Docstring

Renommer le symbole

Modifier toutes les occurrences

Mettre en forme le document avec...

Mettre le document en forme

Remanier...

Action de la source

Couper

Copier

Coller

Run Current File in Interactive Window

Run From Line in Interactive Window

Run Selection/Line in Interactive Window

Run To Line in Interactive Window

Exécuter le fichier de test unitaire courant

PROBLÈMES SORTIE CONSOLE DE DÉBOGAGE TERMINAL



```
test_conversionTemperature.TestConversionTemperature.  
test_conversionTemperature.TestConversionTemperature.  
test_conversionTemperature.TestConversionTemperature.  
test_exemplePersonne.test.test_0_set_nom  
test_exemplePersonne.test.test_1_get_nom  
test_exempleTest.Tester.test_booleen  
test_exempleTest.Tester.test_string  
test_booleen (test_exempleTest.Tester) ... ok  
test_string (test_exempleTest.Tester) ... ok
```

Ran 2 tests in 0.000s

OK

VS Code: exécution des tests unitaires

Pour accéder à vos tests unitaires, dans la barre d'outils, il y a une nouvelle icône

- Vous aurez accès à tous vos tests créés pour votre programme
- C'est possible d'exécuter un seul test unitaire à la fois
- Le  permet de dire que le test a échoué
- Le  permet de dire que le test a réussi

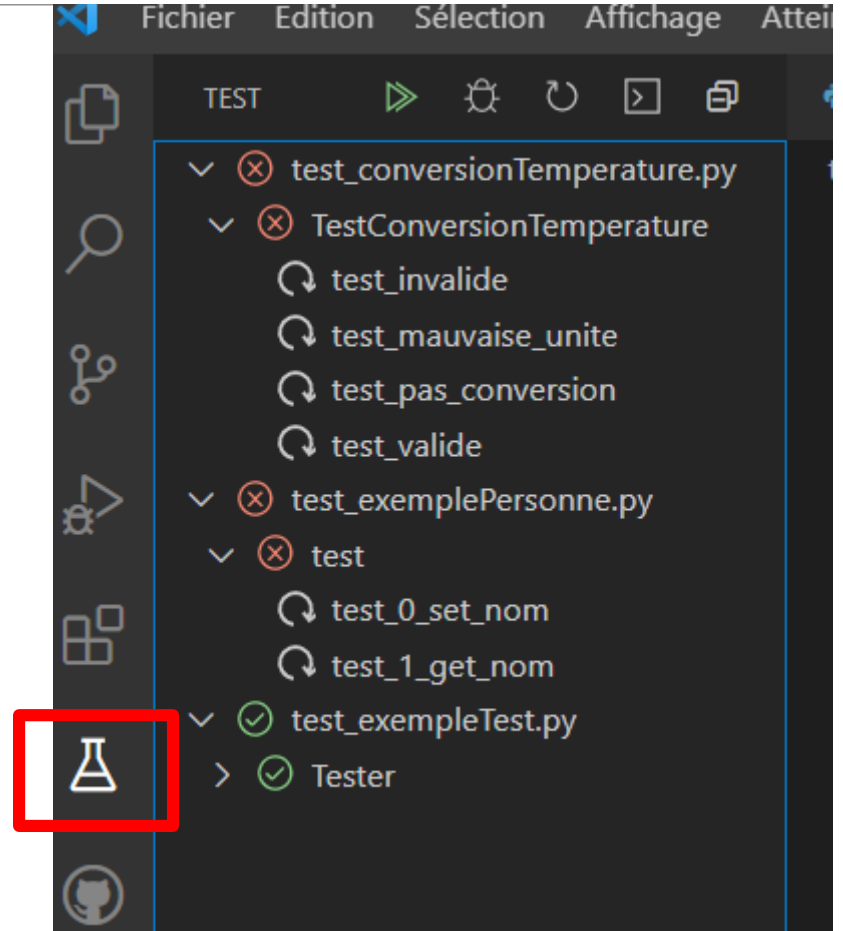
Ran 2 tests in 0.000s

OK

```
test_boolean (test_exampleTest.Tester) ... ok
test_string (test_exampleTest.Tester) ... ok
```

Ran 2 tests in 0.000s

OK



Cadre applicatif unittest

Exemple avec une classe Personne

Voici un autre exemple qui comprend le code de notre programme et une classe test

Notre programme est une classe `Personne` qui permet d'afficher le nom ou l'ID du nom.

```
personne.py
1 class Personne:
2     nom = []
3
4     def set_nom(self, nom):
5         self.nom.append(nom)
6         return len(self.nom) - 1
7
8     def get_nom(self, id_nom):
9         if id_nom >= len(self.nom):
10             return "Il n'y a pas d'utilisateur"
11         else:
12             return self.nom[id_nom]
13
14 if __name__ == "__main__":
15     unePersonne = Personne()
16     print("Utilisateur Simon est ajouté avec le ID ", unePersonne.set_nom("Simon"))
17     print("L'utilisateur associé à ID 0 est ", unePersonne.get_nom(0))
18
```

Exécution

```
Utilisateur Simon est ajouté avec le ID 0
L'utilisateur associé à ID 0 est Simon
```

```

test_exemplePersonne.py
import unittest
import personne as PersonneClasse

class test(unittest.TestCase):
    personne = PersonneClasse.Personne()
    personne_id = []
    personne_nom = []

    def test_0_set_nom(self):
        print("Démarrage test set_nom")
        for i in range(4):
            nom = "nom" + str(i) #initialisation du nom
            self.personne_nom.append(nom) #mettre le nom dans la liste
            id_nom = self.personne.set_nom(nom) #retourner le ID de la fonction
            self.assertIsNotNone(id_nom) #vérifier si le ID est null ou non
            self.personne_id.append(id_nom) #mettre le ID dans la liste
        print("longueur liste ID = ", len(self.personne_id))
        print(self.personne_id)
        print("longueur liste nom = ", len(self.personne_nom))
        print(self.personne_nom)
        print("fin test set_nom")

    def test_1_get_nom(self):
        print("Démarrage test get_nom")
        longueur = len(self.personne_id)
        print("longueur liste ID = ", longueur)
        print("longueur liste nom = ", len(self.personne_nom))

        for i in range(6):
            if i < longueur:
                #si deux noms non égaux, le test va échoué
                self.assertEqual(self.personne_nom[i], self.personne.get_nom(self.personne_id[i]))
            else:
                print("Pas de nom testé pour get_nom")
                self.assertEqual("Il n'y a pas d'utilisateur", self.personne.get_nom(i))
        print("fin test get_nom")

if __name__ == "__main__":
    unittest.main()

```

Cadre applicatif unittest

Exemple avec une classe Personne - suite

On importe les modules `unittest` et `personne`

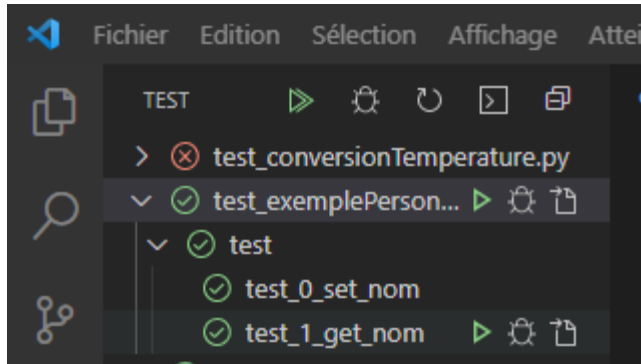
Dans la classe test:

- On créer une nouvelle personne vide
- Une méthode permet de tester le `set_nom` de la classe `personne`
 - Dans cette méthode, on va initialiser 4 noms et on va vérifier (tester) si le ID est `null` ou non
 - On affiche la liste des ID et la liste des noms
- Une méthode permet de tester le `get_nom` de la classe `personne`
 - On va tester si le nom de la liste et égal avec le nom de la personne et si le message est égal à celui de la classe `personne`.

Cadre applicatif unittest

Exemple avec une classe Personne - Résultats

Voici les résultats des tests précédents.



```
PROBLÈMES  SORTIE  CONSOLE DE DÉBOGAGE  TERMINAL
test_0_set_nom (test_exemplePersonne.test) ... Démarrage test set_nom
longueur liste ID = 4
[0, 1, 2, 3]
longueur liste nom = 4
['nom0', 'nom1', 'nom2', 'nom3']
fin test set_nom
ok
test_1_get_nom (test_exemplePersonne.test) ... Démarrage test get_nom
longueur liste ID = 4
longueur liste nom = 4
Pas de nom testé pour get_nom
Pas de nom testé pour get_nom
fin test get_nom
ok

-----
Ran 2 tests in 0.001s

OK
|
```

Cadre applicatif unittest

Tester une exception avec assertRaises

Pour vérifier si une exception spécifique est levée, on utilise `assertRaises()`

- Cela permet de lever une exception sans quitter l'exécution
- Le test réussit si l'exception est déclenchée
- Le test donne une erreur si une autre exception est déclenchée
- Le test échoue si aucune exception n'est déclenchée.

Deux façons d'utiliser

- En utilisant les arguments
 - `assertRaises(exception, méthode, *args, **keywords)`
- En utilisant le gestionnaire de contexte
 - `assertRaises(exception)`

```
import unittest

class TestAcceptRaises(unittest.TestCase):
    def test_split(self):
        s = 'Bonjour à tous'
        # vérifie si s.split échoue : séparateur pas une string
        with self.assertRaises(TypeError):
            s.split(1)

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

```
import unittest

class TestAcceptRaises(unittest.TestCase):
    # Retourne True si 100/0 déclenche une exception
    def test_division(self):
        with self.assertRaises(ZeroDivisionError):
            100 / 0

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



```
def conversionTemperature(valeur, uniteA, uniteB):
    # conversion et retourne la valeur de la temperature
    # de l'unité A à l'unité B
    #dictionnaire de conversion
    toK = {'K': lambda val: val,
           'C': lambda val: val + 273.15,
           'F': lambda val: (val+459.67)*5/9 }

    fromK = {'K': lambda val: val,
             'C': lambda val: val - 273.15,
             'F': lambda val: val*5/9 - 459.67 }

    #conversion de la température de uniteA en K
    try:
        temp = toK[uniteA](valeur)
    except KeyError:
        raise ValueError("unité de température non reconnue: {}".format(uniteA))

    if temp < 0:
        raise ValueError("Température invalide: {} {} est moins que 0K".format(valeur, uniteA))

    if uniteA == uniteB:
        #aucune conversion
        return valeur

    #conversion de K à unitéB et retourne la valeur
    try:
        return fromK[uniteB](temp)
    except KeyError:
        raise ValueError("unité de température non reconnue: {}".format(uniteB))
```

Cadre applicatif unittest

Exemple avec un module de conversion de température

Exemple avec les arguments

- Un module qui permet la conversion de la température (Celsius, Fahrenheit, Kelvin) entre deux valeurs

```

import conversionTemperature as c
import unittest

class TestConversionTemperature(unittest.TestCase):

    def test_invalide(self):
        #Pas de température en bas de 280C, ValueError
        self.assertRaises(ValueError, c.conversionTemperature, -280, 'C', 'F')

    def test_valide(self):
        #séries de tests de conversion de température
        test_cases = [((273.16, 'K'), (0.01, 'C')),
                        ((-40, 'C'), (-40, 'F')),
                        ((450, 'F'), (505.372222222222, 'K'))]

        for unTest in test_cases:
            ((valeurA, uniteA), (valeurB, uniteB)) = unTest
            resultat = c.conversionTemperature(valeurA, uniteA, uniteB)
            self.assertAlmostEqual(valeurB, resultat)

    def test_pas_conversion(self):
        #avoir uniteA et uniteB à la même température

        temp = 56.67
        resultat = c.conversionTemperature(temp, 'C', 'C')
        self.assertEqual(resultat, temp)

    def test_mauvaise_unite(self):
        #vérifier si les exception ValueError
        self.assertRaises(ValueError, c.conversionTemperature, 0, 'C', 'R')
        self.assertRaises(ValueError, c.conversionTemperature, 0, 'N', 'K')

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

```

Cadre applicatif unittest

Exemple avec un module de conversion de température

Voici le module de test avec 4 tests unitaires

- Méthode `test_invalide`
 - Cela permet de tester l'exception `ValueError` en bas de 280 C
- Méthode `test_valide`
 - Cela permet de tester une série de conversion de température
- Méthode `test_pas_conversion`
 - Cela permet de vérifier si la conversion fonctionne pour la même température
- Méthode `test_mauvaise_unite`
 - Cela permet de vérifier si l'exception `ValueError` fonctionne si c'est une mauvaise unité

Cadre applicatif unittest

Exemple avec un module de conversion de température - résultats

```
PROBLÈMES  SORTIE  CONSOLE DE DÉBOGAGE  TERMINAL
test_invalide (test_conversionTemperature.TestConversionTemperature) ... ok
test_mauvaise_unite (test_conversionTemperature.TestConversionTemperature) ... ok
test_pas_conversion (test_conversionTemperature.TestConversionTemperature) ... ok
test_valide (test_conversionTemperature.TestConversionTemperature) ... FAIL
NoneType: None

=====
FAIL: test_valide (test_conversionTemperature.TestConversionTemperature)
-----
Traceback (most recent call last):
  File "c:\Users\sdeschenes\ProjetsPython\ExemplesSemaine12\tests\test_conversionTemperature.py", line 19, in test_valide
    self.assertAlmostEqual(valeurB, resultat)
AssertionError: -40 != -330.14222222222224 within 7 places (290.14222222222224 difference)

-----
Ran 4 tests in 0.003s

FAILED (failures=1)
```

Profilage du code : `timeit`

Cette méthode permet de mesurer le temps d'exécution pour un code donné (`code snippet`)

- Python exécute l'instruction de code 1 million de fois et fournit le temps minimum pris à partir de l'ensemble de code donné
- Ça permet de vérifier les performances du code.

Syntaxe

- `timeit.timeit(stmt, setup, minuterie, nombre)`
 - `stmt` : code souhaité pour la mesure du temps d'exécution
 - `setup` : détails de configuration qui doivent être exécutés. On l'utilise pour importer les modules pour notre code.
 - `minuterie` : valeur de la minuterie. Il y a déjà une valeur par défaut.
 - `nombre` : le nombre de fois que le code va s'exécuter. Par défaut, c'est 1 million.

```

import timeit

# fonction recherche binaire
def rechercheBinaire(maListe, recherche):
    while len(maListe) > 0:
        milieu = (len(maListe))//2
        if maListe[milieu] == recherche:
            return True
        elif maListe[milieu] < recherche:
            maListe = maListe[:milieu]
        else:
            maListe = maListe[milieu + 1:]
    return False

# fonction recherche linéaire
def rechercheLineaire(maListe, recherche):
    for element in maListe:
        if element == recherche:
            return True
    return False

# calcul temps recherche binaire
def temps_binaire():
    SETUP_BIN = '''
from __main__ import rechercheBinaire
from random import randint'''

    TEST_BIN = '''
maliste = [x for x in range(10000)]
recherche = randint(0, len(maliste))
rechercheBinaire(maliste, recherche)'''

    # exécution de timeit pour binaire
    tempsBinaire = timeit.repeat(setup = SETUP_BIN, stmt = TEST_BIN,
                                repeat = 3, number = 10000)

    # affichage de l'exécution
    print('Temps recherche binaire: {}'.format(min(tempsBinaire)))

```

Profilage du code : timeit exemple

Comparaison du temps d'exécution entre la recherche binaire et la recherche linéaire

- Dans l'exemple, on répète 3 fois 10000 itérations.
- Suite du code à la diapositive suivante

La variable `SETUP_BIN` nous permet d'importer les modules dans le code que l'on veut tester

La variable `TEST_BIN` permet d'importer le code que l'on veut tester.

Profilage du code : timeit

exemple - suite

```
# calcul temps recherche linéaire
def temps_lineaire():
    SETUP_LIN = '''
from __main__ import rechercheLineaire
from random import randint'''

    TEST_LIN = '''
maliste = [x for x in range(10000)]
recherche = randint(0, len(maliste))
rechercheLineaire(maliste, recherche)'''

    # exécution de timeit pour linéaire
    tempsLineaire = timeit.repeat(setup = SETUP_LIN, stmt = TEST_LIN,
                                  repeat = 3, number = 10000)

    # affichage de l'exécution
    print('Temps recherche linéaire: {}'.format(min(tempsLineaire)))

if __name__ == "__main__":
    temps_binaire()
    temps_lineaire()
```

Suite du code

- Les variables `SETUP_LIN` et `TEST_LIN` font la même chose que les variables `SETUP_BIN` et `TEST_BIN`, mais pour la recherche linéaire

Voici le résultat:

- On constate que la recherche binaire est d'environ 2x plus optimale que la recherche linéaire

```
Temps recherche binaire: 4.852415499999999
Temps recherche linéaire: 8.3871594
PS C:\Users\sdeschenes\ProjetsPython\ExemplesSemaine12> █
```

```

exemplecProfile.py > fibo_sequence
1  import cProfile
2
3  def fibonacci(n):
4  # from http://en.literateprograms.org/Fibonacci\_numbers\_\(Python\)
5      if n == 0:
6          return 0
7      elif n == 1:
8          return 1
9      else:
10         return fibonacci(n-1) + fibonacci(n-2)
11
12 def fibo_sequence(n):
13     sequence = [ ]
14     if n > 0:
15         sequence.extend(fibo_sequence(n-1))
16     sequence.append(fibonacci(n))
17     return sequence
18
19 def main():
20     print("Résultats")
21     print("=====")
22     print(fibo_sequence(20))
23
24 cProfile.run("main()")

```

Profilage du code : cProfile

Pour vérifier le temps d'exécution, on peut aussi utiliser le module `cProfile`.

- Il permet de mesurer les ressources nécessaires à l'exécution (ressources mémoires et ressources du calcul)
- On utilise l'attribut `run()` pour exécuter du code dans l'argument.

Dans l'exemple, on affiche la suite de Fibonacci de façon récursive.

Profilage du code : cProfile

Résultat de l'exemple

Voici le résultat pour les 20 premiers nombres de Fibonacci:

- Il y a eu 57360 appels de fonctions en 0.07 secondes
- Il y a 57291 appels dans la méthode `fibonacci` et 21 appels dans la méthode `fibonacci_sequence`
- Le temps d'exécution est concentré dans la méthode `fibonacci`.

```
Résultats
=====
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765]
=====
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765]
57360 function calls (70 primitive calls) in 0.074 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
1      0.000   0.000   0.074    0.074 <string>:1(<module>)
57360 function calls (70 primitive calls) in 0.074 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
1      0.000   0.000   0.074    0.074 <string>:1(<module>)
21/1    0.000   0.000   0.073    0.073 exemplecProfile.py:12(fibonacci_sequence)
1      0.000   0.000   0.074    0.074 exemplecProfile.py:19(main)
57291/21 0.072   0.000   0.072    0.003 exemplecProfile.py:3(fibonacci)
1      0.000   0.000   0.074    0.074 {built-in method builtins.exec}
3      0.001   0.000   0.001    0.000 {built-in method builtins.print}
21     0.000   0.000   0.000    0.000 {method 'append' of 'list' objects}
1      0.000   0.000   0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
20     0.000   0.000   0.000    0.000 {method 'extend' of 'list' objects}
```



```

import cProfile

class memoize:
    # from http://avinashv.net/2008/04/python-decorators-syntactic-sugar/
    # https://www.python-course.eu/python3\_memoization.php
    def __init__(self, fonction):
        self.fonction = fonction
        self.memoized = {}

    def __call__(self, *args):
        try:
            return self.memoized[args]
        except KeyError:
            self.memoized[args] = self.fonction(*args)
            return self.memoized[args]

@memoize
def fibonacci(n):
    # from http://en.literateprograms.org/Fibonacci\_numbers\_\(Python\)
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n-1) + fibonacci(n-2)

```

Profilage du code : cProfile amélioration

Pour améliorer l'exemple précédent, on devrait ajouter une mémorisation automatique d'une fonction.

- On utilise la classe `memoize` et on l'ajoute à la méthode `fibonacci`
- Le reste du code est semblable à l'exemple précédent.

Profilage du code : cProfile

Amélioration - résultat

Voici le résultat amélioré pour les 20 premiers nombres de Fibonacci:

- Il y a eu 149 appels de fonctions en 0.001 seconde
- Il y a 21 appels dans la méthode `fibonacci` et 21 appels dans la méthode `fibonacci_sequence`
- Le temps d'exécution a beaucoup diminué en fonction de l'exemple précédent.

o

Résultats - memoize

=====

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765]

149 function calls (91 primitive calls) in 0.001 seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1	0.000	0.000	0.001	0.001	<string>:1(<module>)
21	0.000	0.000	0.000	0.000	exemplecProfile.py:16(fibonacci)
21/1	0.000	0.000	0.000	0.000	exemplecProfile.py:26(fibo_sequence)
1	0.000	0.000	0.001	0.001	exemplecProfile.py:33(main)
59/21	0.000	0.000	0.000	0.000	exemplecProfile.py:9(__call__)
1	0.000	0.000	0.001	0.001	{built-in method builtins.exec}
3	0.001	0.000	0.001	0.000	{built-in method builtins.print}
21	0.000	0.000	0.000	0.000	{method 'append' of 'list' objects}
1	0.000	0.000	0.000	0.000	{method 'disable' of '_lsprof.Profiler' objects}
20	0.000	0.000	0.000	0.000	{method 'extend' of 'list' objects}

Déploiement d'une application Pi

Pour la création d'un exécutable de votre programme, voici les commandes à faire dans le terminal:

- Installation de `pyinstaller`

```
pip3 install pyinstaller
```

- Création de l'exécutable

```
pyinstaller --onefile votrefichier.py
```

Ensuite, allez dans l'explorateur Windows dans le dossier où se situe votre programme `.py`

- L'exécutable se retrouve dans le dossier `dist`
- C'est possible de l'exécuter dans le terminal

Nom	
__pycache__	2
build	2
dist	2
exemplecProfile.py	2
exemplecProfile.spec	2

```
PS C:\Users\sdeschenes\ProjetsPython\ExemplesSemaine12\exempleExécution\dist> .\exemplecProfile
Résultats - memoize
=====
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765]
149 function calls (91 primitive calls) in 0.001 seconds

Ordered by: standard name

ncalls  tottime  percall  cumtime  percall filename:lineno(function)
1      0.000      0.000      0.001      0.001 <string>:1(<module>)
```

Sécurité de l'application – quelques bonnes pratiques

Voici quelques bonnes pratiques à utiliser pour le développement d'applications d'objets connectés avec Python

- Utilisation de la plus version récente de Python
- Utilisation d'un environnement virtuel isolé pour votre projet
 - En utilisant `virtualenv` ou `pipenv`
- Mettre le mode débogage à faux. Permet de cacher les possibles erreurs lors du déploiement de votre application en mode public.
 - `Debug = False` dans le fichier `settings.py`
- Installation de bibliothèques légitimes et qui ne contiennent pas de code malicieux
- Vérification des chemins d'importation (`import paths`) : absolue, relatif et implicite
 - Utilisation des chemins absolus
- Protection contre les injections SQL si vous utilisez les bases de données
- Utilisation de bandit qui permet de vérifier les vulnérabilités dans votre code.
 - <https://pypi.org/project/bandit/>
 - <https://github.com/PyCQA/bandit>

Références

- Python : Libérez le potentiel de votre Raspberry Pi, Cédric Lemaître, Éditions ENI, Août 2018, 245 pages
- Python et Raspberry Pi : Apprenez à développer sur votre nano-ordinateur (2e édition), Patrice Clément, Éditions ENI, Janvier 2019, 279 pages
- Tests dans VS Code
 - <https://code.visualstudio.com/docs/python/testing>
- Tests d'intégration
 - <https://realpython.com/python-testing/#unit-tests-vs-integration-tests>
 - <https://mq-software-carpentry.github.io/python-testing/09-integration/>
 - <https://www.tutorialspoint.com/pytest/index.htm>
- Tests unitaires:
 - <https://docs.python.org/fr/3.9/library/unittest.html>
 - <https://realpython.com/python-testing/#writing-your-first-test>
 - <https://code.visualstudio.com/docs/python/testing>
 - <https://kimsereylam.com/python/2020/10/16/setup-vscode-with-python-unittest.html>
 - <https://www.codingame.com/playgrounds/10614/python-unit-test-with-unittest>
 - <https://www.geeksforgeeks.org/test-if-a-function-throws-an-exception-in-python/>
 - <https://scipython.com/book/chapter-9-general-scientific-programming/examples/unit-test-example/>
- Timeit, cProfile, déploiement et sécurité
 - <https://www.geeksforgeeks.org/timeit-python-examples/>
 - <https://pymotw.com/2/profile/>
 - <https://datatofish.com/executable-pyinstaller/>
 - <https://blog.sqreen.com/top-10-python-security-best-practices/>