MINISTÈRE
DE L'ÉDUCATION
NATIONALE,
DE LA JEUNESSE
ET DES SPORTS

Liberté
Égalité
Fraternité

VOIE GÉNÉRALE

2DE | 1RE | TLE

Numérique et sciences informatiques

ENSEIGNEMENT
SPÉCIALITÉ

ÉCRITURE DE TESTS

SOMMAIRE

Introduction 2
Vocabulaire de base 2
Concevoir des tests « boîte noire » 3
Concevoir des tests « boîte blanche » 4
Écrire des tests en Python 5
Génération de cas de test : étude de cas 6
Génération d'un jeu de tests « boîte noire » 7
Génération d'un jeu de tests « boîte blanche » 11
Le Test Driven Development 12
Principe général 12
Bibliographie 12
En guise de conclusion 12
Annexe B - Couverture des tests avec coverage 14
Au mois de janvier, nous sommes en hiver 18
Au mois d'avril, nous sommes au printemps 18
Le 25 mars, nous sommes au printemps 19
Le 10 mars, nous sommes en hiver 19
Amélioration de la structure du code 20
Amélioration de la structure du code 20
Refactoring des tests 21
Au mois de mai, nous sommes au printemps. 22
Début juin, nous sommes au printemps 22
Fin juin, nous sommes en été 23
Trop de « nombres magiques » disséminés dans le fichier 24
Et ensuite... 25

Introduction

Les bugs entraînent chaque année des millions d'euros de perte pour les entreprises et les États mais parfois également à l'origine de morts civiles ou militaires (voir http://tisserant.org/cours/qualite-logiciel/qualite_logiciel.html par exemple). L'une des solutions les plus utilisées pour éviter de livrer un programme buggué passe par la réalisation de tests. Pendant quelques décennies, les tests de programme étaient écrits (quand ils étaient écrits!) de manière presque aléatoires, et n'apportaient ainsi pas une grande garantie sur les programmes. Mais tout cela a bien changé. Le développement d'outils permettant d'exécuter les tests automatiquement a notamment rendu les tests bien plus utiles. En effet, pour que les tests soient exécutés aussi souvent que possible, il faut que cela soit faisable automatiquement. En Python, il existe plusieurs outils permettant d'exécuter des tests automatiquement. On peut notamment citer *unittest*, *pytest* et *doctest*.

Vocabulaire de base

On peut classer les tests selon différents critères :

- le niveau des tests (tests unitaires, tests d'intégration, tests de recette) ;
- le processus de conception des tests (tests *boîte blanche*, tests *boîte noire)* ;
- le sujet du test (tests fonctionnels, tests de montée en charge, tests d'utilisabilité, etc.).

Voici quelques définitions essentielles concernant les tests.

On appelle **cas de test** un triplet (descriptif, données d'entrée, résultat attendu) précisant, pour des données précises, le résultat attendu de la partie du programme que l'on veut tester.

On appelle **jeu de tests** un ensemble de cas de test destinés à valider une partie précise du fonctionnement d'un programme.

Le terme *test* peut se référer suivant les circonstances à un cas de test, à un jeu de tests, ou au processus de test en général.

Un **test unitaire** est un test concernant une petite unité d'un programme ; typiquement, une fonction.

Un **test « boîte noire»** est un test qui est conçu à partir des données d'entrées potentielles, indépendamment du code écrit. Un test « boîte noire » peut donc être écrit avant le code, ou s'il est écrit après, il doit être écrit par quelqu'un qui ne connaît pas le code.

Exemple : une fonction doit générer l'en-tête d'une lettre. Pour cela, elle prend en paramètre un objet représentant le destinataire de la lettre (prénom, nom, sexe). Sans connaître le code de la fonction, on peut déjà envisager 2 cas de test, un pour un homme et un pour une femme, et vérifier que dans le premier cas l'en-tête commence par « Cher » alors qu'il commence par « Chère » dans le second cas.

Un **test « boîte blanche »** est un test qui est conçu à partir du programme. Le but de ce type de test est de tester les différents cas prévus par le programme.

Exemple « Soit la fonction suivante » :

```
def f(n) :
  if n % 2 == 0 :
    return «pair»
  else :
    return «impair»
```

On envisage alors un cas de test avec `n` pair pour tester la branche « alors » du `if`, et un cas de test avec `n` impair pour tester la branche « sinon ».

Un glossaire un peu plus complet sur les tests est donné en annexe.

> **Dans le cadre de NSI, on peut se contenter d'aborder les tests fonctionnels unitaires, sans donner de définition formelle des tests « boîte noire » et « boîte blanche ».**

Il est important de comprendre qu'à moins d'être exhaustif, ce qui n'est que très rarement possible, **un test ne garantit jamais la correction d'un programme**. Cependant, en écrivant pour chaque fonction un jeu de tests boîte noire pertinent, on augmente la confiance que l'on peut avoir dans le programme et on limite le risque de bug. Il ne faut pas oublier que **si un cas de test est correct**, une réussite du test n'est pas significative, alors un échec établit de manière certaine la présence d'un bug.

Concevoir des tests « boîte noire »

Pour concevoir un jeu de tests « boîte noire » pertinent, on essaie en général de concevoir plusieurs cas de test, représentatifs de chacune des *classes d'équivalence*[1] des données d'entrée, en rajoutant des tests spécifiques aux limites des classes d'équivalence. En général, on considère non seulement des données valides, mais également des données invalides.

Inconvénient des tests « boîte noire » : on risque de ne pas passer par des parties du programme traitant des cas particuliers n'apparaissant pas au premier abord à partir des données d'entrée.

> **Dans le cadre du cours de NSI, l'approche informelle des tests boîte noire donnée ci-dessus est amplement suffisante.**

1. On peut définir de façon informelle une classe d'équivalence pour un paramètre d'entrée comme un ensemble de valeurs de ce paramètre donnant lieu à un comportement identique.

Concevoir des tests « boîte blanche »

Pour concevoir un jeu de tests « boîte blanche », la « **couverture des instructions** » (pour chaque instruction, avoir au moins un cas de test qui amène à l'exécuter) n'est pas satisfaisante. Prenons par exemple la fonction suivante :

```
def heure_suivante(heure):
  heure += 2
  if heure ≥ 24 :
    heure = 0
  return heure
```

Un cas de test avec (« changement de jour », heure = 23, résultat = 0) assure une couverture des instructions. Par ailleurs, la fonction `heure_suivante` le vérifie. Pourtant, la fonction est bien évidemment fausse (on ne teste pas ce qui se passe si la condition du « if » n'est pas vérifiée). Aussi, on cherche en général à assurer au moins une **couverture des décisions** : lorsqu'il y a une conditionnelle, écrire un test pour chacune des branches.

Cependant, on préfère en général tester chacune des raisons ayant amené à prendre chacune des branches. Ainsi, dans la fonction suivante :

```
def f(a,b) :
  if a > 2 or (a < 0 and b == 4) :
    resultat = 1
  else :
    resultat = 0
  return resultat
```

Il faut des cas de test pour chacune des situations suivantes (ne pas oublier que l'évaluation des expressions booléennes est paresseuse) :

- branche « alors » :
  - a > 2
  - a ≤ 2 ET a < 0 ET b = 4
- branche « sinon » :
  - a ≤ 2 ET a ≥ 0
  - a ≤ 2 ET a < 0 ET b ≠ 4

On appelle ce type de couverture : **couverture des conditions multiples**.

Inconvénient des tests « boîte blanche » : la réflexion est biaisée par le code écrit, et on risque de ne pas générer de cas de test pour les cas non traités par le programme.

> Dans le cadre de NSI, on peut se contenter de la version « couverture des décisions » des tests « boîte blanche ».

Écrire des tests en Python

Il existe principalement 3 outils pour écrire et exécuter des tests en Python :

- *unittest* est l'outil de base, intégré à Python ;
- *doctest* est un outil externe qui permet d'exécuter des tests inclus dans la documentation des fonctions. Les tests servent alors également de documentation de la fonction ;
- *pytest* est un outil externe qui permet d'exécuter des tests écrits dans des fichiers dédiés. C'est l'outil le plus utilisé, car le plus complet.

À des fins pédagogiques, *doctest* est très intéressant. Cependant, il alourdit les fichiers sources. Aussi, dans ce document, nous avons choisi de présenter *pytest*.

Lorsqu'on utilise *pytest*, un cas de test est implanté par une fonction dont le nom commence par `test_` qui doit lever un `AssertionError` si le test est raté. Les cas de test sont regroupés dans des fichiers python spécifiques dont le nom commence également par `test_`.

Pour exécuter les tests, soit on exécute la commande *pytest* en passant en paramètre le nom du fichier de test, soit on exécute la commande *pytest* sans argument. Dans ce dernier cas, les tests de tous les fichiers python commençant par `test_` dans le répertoire courant sont exécutés.

Un cas de test se présente en général sous la forme suivante :

```
def test_descriptif_du_test() :
        attendu = resultat_attendu
        effectif = calcul_du_resultat_par_le_programme
        assert attendu == effectif
```

Ainsi, si le résultat effectif correspond au résultat attendu, rien ne se passe ; *pytest* comptabilise le test parmi les tests réussis. Sinon, une `AssertionError` est levée. Celle-ci est « attrapée » par *pytest* qui comptabilise ce test parmi les tests ratés.

Après avoir exécuté tous les tests, *pytest* génère un rapport indiquant les éventuels tests ratés.

Voici un premier exemple de rapport lorsque tout réussit :

```
collected 24 items

test_boite_blanche.py ....                                              [ 16%]
test_boite_noire.py ....................                                [100%]

============================== 24 passed in 0.05s ==============================
```

Et voici un exemple de rapport généré avec 2 tests ratés :

```
collected 44 items

test_autre.py F.......F...........                                       [ 45%]
test_boite_blanche.py ....                                               [ 54%]
test_boite_noire.py ..................                                   [100%]

==================================== FAILURES ====================================
_____________________________________ test_errone_1 _____________________________________

    def test_errone_1():
        with pytest.raises(AssertionError):
>           saison(2, 12)
E           Failed: DID NOT RAISE <class 'AssertionError'>

test_autre.py:6: Failed
_____________________________________ test_errone_2 _____________________________________

    def test_errone_2():
        attendu = «automne»
        effectif = saison(1,12)
>       assert attendu == effectif
E       AssertionError: assert 'automne' == 'hiver'
E         - automne
E         + hiver

test_autre.py:39: AssertionError
============================= 2 failed, 42 passed in 0.11s ==============================
```

Comme on peut le voir, dans un première partie, pour chaque fichier de tests, les tests réussis sont indiqués par un « . », les tests ratés par un « F ».

Dans une deuxième partie, les tests ratés sont détaillés.

Enfin, la dernière liste donne un bilan global sur l'exécution de tous les tests.

Génération de cas de test : étude de cas

L'étude de cas présentée ici permet de montrer la mise en œuvre des tests « boîte noire » et « boîte blanche ». Les solutions présentées vont au-delà de ce qui est exigible en NSI, mais elles permettent d'avoir plus de recul sur la génération de jeux de tests.

On souhaite écrire une fonction indiquant la saison en fonction du jour (numéro compris entre 1 à 31) et du mois (numéro compris entre 1 et 12). Pour simplifier, on admet que les changements de saison ont toujours lieu le 21.

Par ailleurs, pour les données invalides, on suppose qu'on attend une `AssertionError`, ce qui oblige à formaliser les préconditions par des `assert`. Mais ce n'est pas forcément le cas. On peut souhaiter des corrections automatiques, l'affichage d'un message d'avertissement, etc.

Génération d'un jeu de tests « boîte noire »

Pour le mois, on distingue les classes d'équivalence suivantes :

- classes « invalides »
  - mois n'est pas nombre
  - mois n'est pas un entier
  - mois est inférieur à 1
  - mois est supérieur à 12
- classes « valides » :
  - 1 ≤ mois ≤ 2
  - mois = 3
  - 4 ≤ mois ≤ 5
  - mois = 6
  - 7 ≤ mois ≤ 8
  - mois = 9
  - 10 ≤ mois ≤ 11
  - mois = 12

Pour le jour, on distingue les classes d'équivalence suivantes :

- classes invalides
  - jour n'est pas un nombre
  - jour n'est pas un entier
  - jour est inférieur à 1
  - jour est supérieur au nombre de jours dans le mois
- classes valides
  - pour les mois 1, 2, 4, 5, 7, 8, 10, 11 : une seule classe 1..nb_jours(mois)
  - pour les mois 3, 6, 9, 12 : 2 classes : 1..21 et 21..nb_jours(mois) pour tenir compte du changement de saison le 21 du mois

Pour chaque classe invalide de chaque paramètre, on conçoit un cas de test avec des classes valides quelconques pour tous les autres paramètres. On peut donc commencer par générer les cas de test suivants :

Descriptif	Mois	Jour	Résultat attendu
Mois n'est pas un nombre	« toto »	12	AssertionError
Mois n'est pas un entier	3.14	22	AssertionError
Mois est inférieur à 1	-5	7	AssertionError
Mois est supérieur à 12	2618	30	AssertionError
Jour n'est pas un nombre	2	« quatre »	AssertionError
Jour n'est pas un entier	8	2.71828	AssertionError
Jour est inférieur à 1	12	0	AssertionError
Jour supérieur au nb de jours dans le mois	2	30	AssertionError

Ensuite, on génère des cas de test pour les classes valides, en essayant de couvrir plus de classes valides non encore vues. On peut alors générer les cas de test suivants :

Descriptif	Mois	Jour	Résultat attendu
Plein hiver	1	12	« hiver »
Fin d'hiver	3	10	« hiver »
Début de printemps	3	24	« printemps »
Printemps	5	14	« printemps »
Fin de printemps	6	20	« printemps »
Début d'été	6	23	« été »
Plein été	7	28	« été »
Fin d'été	9	4	« été »
Début d'automne	9	21	« automne »
Plein automne	11	11	« automne »
Fin d'automne	12	18	« automne »
Début d'hiver	12	25	« hiver »

On note qu'on essaie également de prendre des valeurs différentes autant que possible.

Si on utilise pytest pour les tests unitaires, on peut alors traduire ce jeu de tests ainsi :

```
from saison import saison
import pytest

def test_mois_pas_nombre():
    with pytest.raises(AssertionError):
        saison(«toto», 12)

def test_mois_pas_entier():
    with pytest.raises(AssertionError):
        saison(3.14, 22)

def test_mois_inferieur_1():
    with pytest.raises(AssertionError):
        saison(-5, 7)

def test_mois_superieur_12():
    with pytest.raises(AssertionError):
        saison(2618, 30)

def test_jour_pas_nombre():
    with pytest.raises(AssertionError):
        saison(2, «quatre»)
```

```
def test_jour_pas_entier():
    with pytest.raises(AssertionError):
        saison(8, 2.71828)

def test_jour_inferieur_1():
    with pytest.raises(AssertionError):
        saison(12, 0)

def test_jour_trop_grand():
    with pytest.raises(AssertionError):
        saison(2, 30)

def test_plein_hiver():
    attendu = «hiver»
    effectif = saison(1,12)
    assert attendu == effectif

def test_fin_hiver():
    attendu = «hiver»
    effectif = saison(3,10)
    assert attendu == effectif

def test_debut_printemps():
    attendu = «printemps»
    effectif = saison(3,24)
    assert attendu == effectif

def test_plein_printemps():
    attendu = «printemps»
    effectif = saison(5,14)
    assert attendu == effectif

def test_fin_printemps():
    attendu = «printemps»
    effectif = saison(6,20)
    assert attendu == effectif

def test_debut_ete():
    attendu = «été»
    effectif = saison(6,23)
    assert attendu == effectif

def test_plein_ete():
    attendu = «été»
    effectif = saison(7,28)
    assert attendu == effectif

def test_fin_ete():
    attendu = «été»
    effectif = saison(9,4)
    assert attendu == effectif
```

```
def test_debut_automne():
    attendu = «automne»
    effectif = saison(9,21)
    assert attendu == effectif

def test_plein_automne():
    attendu = «automne»
    effectif = saison(11,11)
    assert attendu == effectif

def test_fin_automne():
    attendu = «automne»
    effectif = saison(12,18)
    assert attendu == effectif

def test_debut_hiver():
    attendu = «hiver»
    effectif = saison(12,25)
    assert attendu == effectif
```

Pour renforcer les tests aux endroits critiques, on peut éclater les classes « aux valeurs frontières ». Par exemple pour les jours, les 4 classes :

- [-∞;0] ;
- [1..20] ;
- [21..nb_jours(mois)] ;
- [nb_jours(mois)+1..+∞].

peuvent être transformées en 10 classes ainsi :

- [-∞;-1] ;
- [0] ;
- [1] ;
- [2..19] ;
- [20] ;
- [21] ;
- [22..nb_jours(mois)-1] ;
- [nb_jours(mois)] ;
- [nb_jours(mois)+1] ;
- [nb_jours[mois]+2..+∞].

Ensuite, on génère les cas de test comme dans le cas précédent. On obtient ainsi un jeu de test renforcé autour des valeurs frontières, niche privilégiée des bugs.

Génération d'un jeu de tests « boîte blanche »

Pour écrire des tests « boîte blanche », nous devons connaître le code de la fonction à tester. Nous supposons donc que la solution fournie est la suivante :

```
nb_jours = [31,29,31,30,31,30,31,31,30,31,30,31]

def saison(mois, jour):
    assert isinstance(mois, int) and not isinstance(mois, bool)
    assert isinstance(jour, int) and not isinstance(jour, bool)
    assert mois >= 1 and mois <= 12
    assert jour>= 1 and jour <= nb_jours[mois - 1]

    if (mois == 12 and jour >= 21) or mois == 1 or mois == 2 or (mois == 3
and jour < 21):
        return «hiver»
    if (mois == 3 and jour >= 21) or mois == 4 or mois == 5 or (mois == 6
and jour < 21):
        return «printemps»
    if (mois == 6 and jour >= 21) or mois == 7 or mois == 8 or (mois == 9
and jour < 21):
        return «été»
    if (mois == 9 and jour >= 21) or mois == 10 or mois == 11 or (mois ==
12 and jour < 21):
        return «automne»
```

Pour vérifier les différentes branches, nous pouvons alors proposer les cas de test suivants :

```
def test_printemps():
    attendu = «printemps»
    effectif = saison(3, 22)
    assert attendu == effectif
def test_ete():
    attendu = «ete»
    effectif = saison(8, 21)
    assert attendu == effectif
def test_automne():
    attendu = «automne»
    effectif = saison(12, 6)
    assert attendu == effectif
def test_hiver():
    attendu = «hiver»
    effectif = saison(12, 24)
    assert attendu == effectif
```

Par ailleurs, pour vérifier la couverture du code effectuée par les tests que l'on génère (hors programme), on peut utiliser un outil tel que *coverage*. Un exemple d'utilisation de *coverage* sur cette étude de cas est présenté dans l'annexe B. Cet exemple permet d'ailleurs de comprendre pourquoi les cas de test présentés ci-dessus ne permettent pas d'obtenir une couverture des décisions.

Le Test Driven Development

Principe général

Le *Test Driven Development*, ou TDD, est une méthode de conception qui suit la démarche résumée sur le schéma suivant.

Bien que le TDD soit hors programme, nous le présentons succinctement ici, car il s'agit d'une démarche qui prend de plus en plus d'importance. Une présentation avec une mise en œuvre en *live coding*[2] au cours d'une séance est l'occasion de bien mettre en avant l'écriture et l'intérêt des tests.

Lorsqu'on met en œuvre le TDD, il est important de comprendre les points essentiels suivants :

- « Développer le code répondant au nouveau cas de test » implique de développer le code en question en **maximisant la somme de travail non fait** : il ne faut notamment pas anticiper sur les prochains cas de tests. Ainsi, si l'on doit développer une fonction qui donne la représentation binaire d'un nombre entier, et si le premier cas de test précise que la représentation en base 2 de 4, c'est « 100 », alors le code développé pour répondre au premier cas de test doit être simplement `return «100»` ;
- la phase *Améliorer le code* (ou *refactoring*) est indispensable si l'on veut éviter d'obtenir à la fin un code très mal structuré et illisible, ressemblant à une usine à gaz avec de multiples conditions imbriquées les unes dans les autres ;
- comme, à chaque itération, tous les tests sont ré-exécutés, on garantit la non-régression.

Une étude de cas présentant un début de mise en œuvre du TDD est proposée dans l'annexe C.

Bibliographie

The Art of Software Testing, Glenford J. Meyers *et al.*, https://docs.google.com/viewer?a=v&pid=sites&srcid=ZGVmYXVsdGRvbWFpbnxtcnNoZWhyaWNvbXxne-DoyN2YyYjJlNWEyZmY1M2Q2

En guise de conclusion

Les tests constituent une part importante d'un développement logiciel et permettent souvent d'éviter de perdre beaucoup de temps en débuggage. Les projets constituent des contextes particulièrement adaptés à leur mise en œuvre. Notamment, en ce qui concerne les tests « boîte noire », on peut tout à fait envisager que l'élève qui développe la partie A conçoive les tests de la partie B et inversement.

Toutefois, pour que la conception de tests sur des projets soit efficace et pertinente, il est nécessaire de la pratiquer d'abord sur de petits exemples afin que la conception et l'écriture des cas de test deviennent quasi automatiques.

2. Le *live coding* est une technique pédagogique dans laquelle l'enseignant (mais il peut s'agir aussi des élèves) code en direct, en projetant son écran tout en expliquant ce qu'il fait.

Annexe A - Glossaire

Cas de test : on appelle *cas de test* un triplet (descriptif, données d'entrée, résultat attendu) précisant, pour des données précises, le résultat attendu de la partie du programme que l'on veut tester.

Jeu de tests : on appelle *jeu de tests* un ensemble de cas de test destinés à valider une partie précise du fonctionnement d'un programme.

Refactoring : amélioration de la qualité du code d'un programme sans en changer la fonctionnalité.

Taux de couverture : pourcentage du code par lequel passe un jeu de tests. Il existe plusieurs types de couvertures, la couverture des instructions, la couverture des décisions, la couverture des conditions, etc.

Test boîte blanche : un test boîte blanche est un test conçu en connaissant le code à tester, et dont le but est en général de maximiser le *taux de couverture* du code.

Test boîte noire : un test boîte noire est un test conçu à partir des données d'entrées potentielles, indépendamment du code écrit.

Test fonctionnel : test visant à déterminer la correction de l'état final (ou de la valeur de retour) par rapport à l'état initial (ou aux données d'entrée).

Test d'intégration : un test d'intégration est un test destiné à vérifier que 2 parties d'un programme, développées a priori indépendamment l'une de l'autre, fonctionnent correctement lorsqu'elles sont mises ensemble.

Test de montée en charge : test visant à vérifier si un serveur continue à bien fonctionner avec un grand nombre de sollicitations simultanées.

Test de recette : un test de recette est un test destiné à vérifier une fonctionnalité générale d'un logiciel dans son ensemble.

Test unitaire : un test unitaire est un test destiné à tester une petite partie (comme une fonction, une méthode, une classe) d'un programme indépendamment des autres parties du programme.

Test d'utilisabilité : test destiné à évaluer l'ergonomie d'un programme.

Annexe B - Couverture des tests avec coverage

Coverage est un outil qui permet d'analyser la couverture d'un code Python.

Pour installer cet outil :

- si vous avez installé python avec *anaconda* : `conda install coverage ;`
- si vous avez installé python avec *pip* : `pip install coverage.`

Coverage permet de calculer deux types de couverture :

- la couverture des instructions ;
- la couverture des décisions.

La couverture des conditions multiples n'est malheureusement pas analysée par *coverage*.

Pour calculer la couverture des instructions à partir d'un fichier de test pytest :

```
coverage run -m pytest fichier_de_test.py
```

Pour calculer la couverture des décisions :

```
coverage run --branch -m pytest fichier_de_test.py
```

Les résultats du calcul de la couverture sont stockés dans une base *sqlite3* appelée `.coverage`. Ils peuvent être affichés :

- de façon résumée dans le terminal : `coverage report -m`
- de façon plus détaillée dans une page web : `coverage html; firefox htmlcov/index.html`

Écrivons un premier cas de test, pour passer par la branche renvoyant «printemps» :

```
def test_printemps():
    attendu = «printemps»
    effectif = saison(3, 22)
    assert attendu == effectif
```

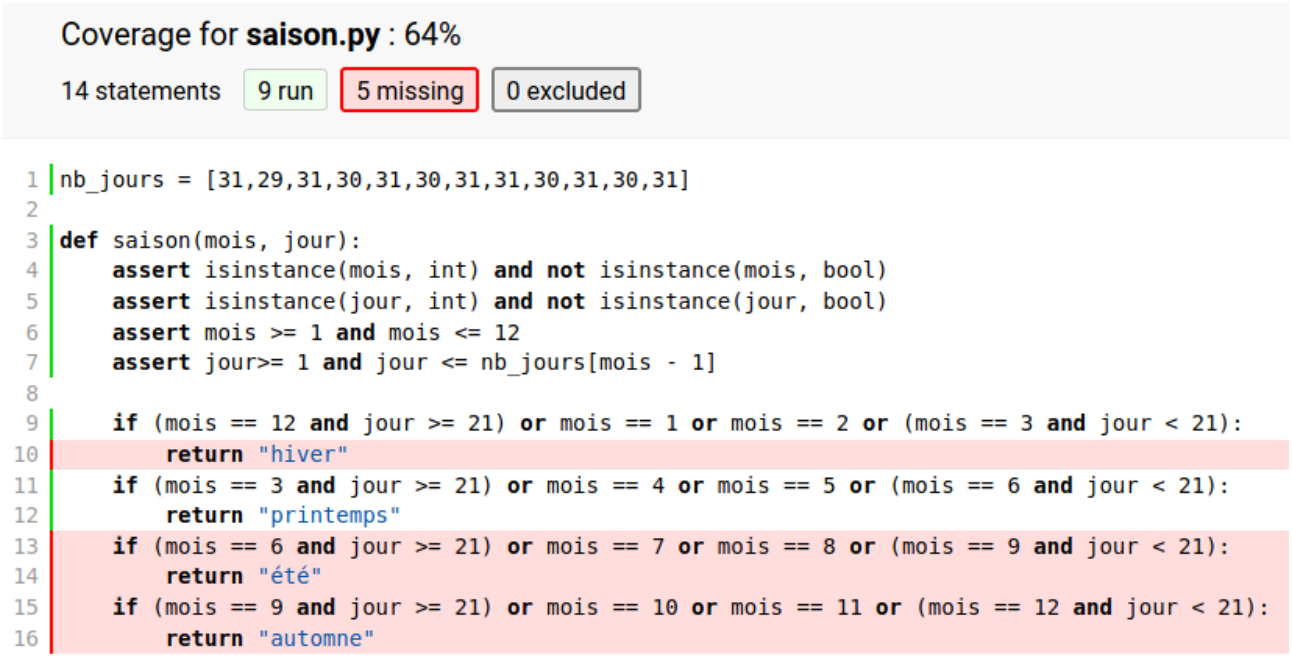
Couverture des instructions

Si on demande à *coverage* d'analyser la couverture des instructions, le rapport de test résumé est le suivant :

```
Name                      Stmts   Miss  Cover   Missing
-------------------------------------------------------
saison.py                    14      5    64%   10, 13-16
test_boite_blanche.py         7      0   100%
-------------------------------------------------------
TOTAL                        21      5    76%
```

Le taux de couverture de la classe de test n'est pas particulièrement significatif, mais il permet de vérifier malgré tout que tous les tests ont bien été exécutés.

Via le rapport HTML détaillé, on a accès à la page suivante :



Comme on peut le voir, il s'agit bien d'une couverture des instructions :

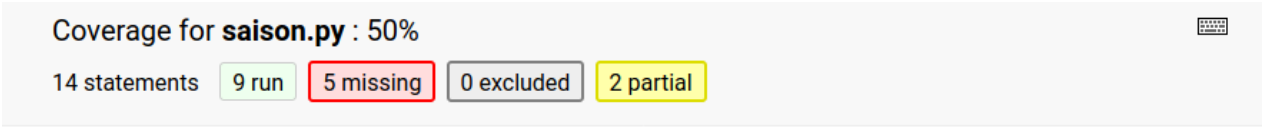
- on est passé par la ligne 9 ; elle est donc en vert ;
- comme la condition de la ligne 9 était fausse, on n'est pas passé par la ligne 10 ;
- on est passé par la ligne 11 ; elle est donc en vert ;
- comme la condition de la ligne 11 était vraie, on est passé par la ligne 12. Avec le return, les lignes suivantes n'ont pas été exécutées ;
- en revanche, tous les « assert » précédents ont également été exécutés ; ils sont donc en vert.

Couverture des décisions

Si on demande à *coverage* une analyse de la couverture des décisions, on obtient le rapport résumé suivant :

```
Name                    Stmts   Miss Branch BrPart  Cover   Missing
-------------------------------------------------------------------
saison.py                  14      5      8      2    50%   9->10, 10, 11->13, 13-16
test_boite_blanche.py       7      0      0      0   100%
-------------------------------------------------------------------
TOTAL                      21      5      8      2    62%
```

On constate un taux de couverture plus faible. Le rapport HTML détaillé permet cette fois-ci d'accéder aux informations suivantes :




```
1  nb_jours = [31,29,31,30,31,30,31,31,30,31,30,31]
2
3  def saison(mois, jour):
4      assert isinstance(mois, int) and not isinstance(mois, bool)
5      assert isinstance(jour, int) and not isinstance(jour, bool)
6      assert mois >= 1 and mois <= 12
7      assert jour>= 1 and jour <= nb_jours[mois - 1]
8
9      if (mois == 12 and jour >= 21) or mois == 1 or mois == 2 or (mois == 3 and jour < 21):    9 ↛ 10
10         return "hiver"
11     if (mois == 3 and jour >= 21) or mois == 4 or mois == 5 or (mois == 6 and jour < 21):    11 ↛ 13
12         return "printemps"
13     if (mois == 6 and jour >= 21) or mois == 7 or mois == 8 or (mois == 9 and jour < 21):
14         return "été"
15     if (mois == 9 and jour >= 21) or mois == 10 or mois == 11 or (mois == 12 and jour < 21):
16         return "automne"
```

Les lignes 9 et 11 sont maintenant en jaune car une seule des 2 branches a été « couverte ». Les indications à droite signalent que le passage de la ligne 9 à la ligne 10 n'a pas été effectué et qu'il en est de même pour le passage de la ligne 11 à la ligne 13.

On constate cependant bien qu'il s'agit d'une couverture des décisions et non des conditions multiples : le fait d'avoir choisi la branche « printemps » une fois suffit.

Pour assurer la couverture des décisions, on peut donc se contenter du jeu de tests suivants :

Le rapport détaillé de couverture du code a alors la forme suivante :

```
def test_printemps():
    attendu = «printemps»
    effectif = saison(3, 22)
    assert attendu == effectif

def test_hiver():
    attendu = «hiver»
    effectif = saison(2, 10)
    assert attendu == effectif

def test_ete():
    attendu = «été»
    effectif = saison(6,23)
    assert attendu == effectif

def test_automne():
    attendu = «automne»
    effectif = saison(11,11)
    assert attendu == effectif
```

Coverage for **saison.py** : 95%

14 statements | 14 run | 0 missing | 0 excluded | 1 partial

```
1  nb_jours = [31,29,31,30,31,30,31,31,30,31,30,31]
2
3  def saison(mois, jour):
4      assert isinstance(mois, int) and not isinstance(mois, bool)
5      assert isinstance(jour, int) and not isinstance(jour, bool)
6      assert mois >= 1 and mois <= 12
7      assert jour>= 1 and jour <= nb_jours[mois - 1]
8
9      if (mois == 12 and jour >= 21) or mois == 1 or mois == 2 or (mois == 3 and jour < 21):
10         return "hiver"
11     if (mois == 3 and jour >= 21) or mois == 4 or mois == 5 or (mois == 6 and jour < 21):
12         return "printemps"
13     if (mois == 6 and jour >= 21) or mois == 7 or mois == 8 or (mois == 9 and jour < 21):
14         return "été"
15     if (mois == 9 and jour >= 21) or mois == 10 or mois == 11 or (mois == 12 and jour < 21):
16         return "automne"
```

Le taux de couverture de 95 % peut surprendre. En regardant le rapport détaillé, on constate que le dernier test est a priori inutile : si on n'est ni en hiver, ni au printemps, ni en été, c'est qu'on est en automne. Ainsi, la branche `else` du dernier `if` ne peut pas être testée. On peut donc simplifier le code en supprimant le dernier `if`.

On note par ailleurs que *coverage* ne considère pas le fait qu'un `assert` lève ou pas une exception comme 2 branches différentes. Aussi, la couverture des décisions avec *coverage* n'incite pas à tester les cas « illégaux ».

Annexe C - TDD – Étude de cas

Début de mise en œuvre

On reprend ici l'étude de cas précédente. On commence par écrire le fichier `saison.py` avec une fonction `saison` en ne faisant rien :

```
def saison(mois, jour):
    pass
```

On va maintenant ajouter progressivement des cas de test.

Au mois de janvier, nous sommes en hiver

Code ajouté au fichier `test_tdd.py` :

```
from saison_tdd import saison
import pytest

def test_janvier():
    attendu = «hiver»
    effectif = saison(1, 12)
    assert attendu == effectif
```

Les tests échouent. On corrige le fichier `saison.py` ainsi :

```
def saison(mois, jour):
    return «hiver»
```

Les tests réussissent.

Au mois d'avril, nous sommes au printemps

Code ajouté au fichier `test_tdd.py` :

```
def test_avril():
    attendu = «printemps»
    effectif = saison(4, 20)
    assert attendu == effectif
```

Les tests échouent. On corrige le fichier `saison.py` ainsi :

```
def saison(mois, jour):
    if mois == 4:
        return «printemps»
    else:
        return «hiver»
```

Les tests réussissent.

Le 25 mars, nous sommes au printemps

Code ajouté au fichier `test_tdd.py` :

```
def test_fin_mars():
    attendu = «printemps»
    effectif = saison(3, 25)
    assert attendu == effectif
```

Les tests échouent. On corrige le fichier `saison.py` ainsi :

```
def saison(mois, jour):
    if mois == 3 or mois == 4:
        return «printemps»
    else:
        return «hiver»
```

Les tests réussissent.

Le 10 mars, nous sommes en hiver

Code ajouté au fichier `test_tdd.py` :

```
def test_debut_mars():
    attendu = «hiver»
    effectif = saison(3, 10)
    assert attendu == effectif
```

Les tests échouent. On corrige le fichier `saison.py` ainsi :

```
def saison(mois, jour):
    if (mois == 3 and jour >= 21) or mois == 4:
        return «printemps»
    else:
        return «hiver»
```

Les tests réussissent.

Amélioration de la structure du code

La condition dans le `if` de la fonction `saison` est bien compliquée. On l'isole dans une méthode, en modifiant ainsi le fichier `saison.py` :

```
def est_printemps(mois, jour):
    fin_mars = mois == 3 and jour >= 21
    avril = mois == 4
    return fin_mars or avril

def saison(mois, jour):
    if est_printemps(mois, jour):
        return «printemps»
    else:
        return «hiver»
```

Les tests réussissent. Le *refactoring* est validé.

Amélioration de la structure du code

Plutôt que de passer systématiquement les 2 paramètres mois et jour, on peut passer par une classe. On modifie donc ainsi le fichier `saison.py` :

```
class Date:
    def __init__(self, mois, jour):
        self.mois = mois
        self.jour = jour

    def est_printemps(self):
        fin_mars = self.mois == 3 and self.jour >= 21
        avril = self.mois == 4
        return fin_mars or avril

    def saison(self):
        if self.est_printemps():
            return «printemps»
        else:
            return «hiver»

def saison(mois, jour):
    date = Date(mois, jour)
    return date.saison()
```

Les tests réussissent. Le *refactoring* est donc valide.

Refactoring des tests

La structure du code a été modifiée avec l'introduction de nouvelles méthodes. Il faut corriger la structure des tests pour coller à la nouvelle structure du code. Le fichier de test prend alors la forme suivante :

```
from saison_tdd import saison, Date
import pytest

def test_janvier():
    date = Date(1,12)
    attendu = «hiver»
    effectif = date.saison()
    assert attendu == effectif

def test_avril():
    date = Date(4,20)
    attendu = «printemps»
    effectif = date.saison()
    assert attendu == effectif

def test_fin_mars():
    date = Date(3, 25)
    attendu = «printemps»
    effectif = date.saison()
    assert attendu == effectif

def test_debut_mars():
    date = Date(3,10)
    attendu = «hiver»
    effectif = date.saison()
    assert attendu == effectif

def test_avril_est_printemps():
    date = Date(4, 20)
    assert date.est_printemps()

def test_fin_mars_est_printemps():
    date = Date(3, 25)
    assert date.est_printemps()
```

Les tests réussissent.

Au mois de mai, nous sommes au printemps.

Code ajouté au fichier `test_tdd.py` :

```
def test_mai_est_printemps():
    date = Date(5, 14)
    assert date.est_printemps()
```

Les tests échouent. On corrige le fichier `saison.py` ainsi :

```
    def est_printemps(self):
        fin_mars = self.mois == 3 and self.jour >= 21
        avril = self.mois == 4
        mai = self.mois == 5
        return fin_mars or avril or mai
```

Les tests réussissent.

Début juin, nous sommes au printemps

Code ajouté au fichier `test_tdd.py` :

```
def test_debut_juin_est_printemps():
    date = Date(6, 6)
    assert date.est_printemps()
```

Les tests échouent. On corrige le fichier `saison.py` ainsi :

```
    def est_printemps(self):
        fin_mars = self.mois == 3 and self.jour >= 21
        avril = self.mois == 4
        mai = self.mois == 5
        juin = self.mois == 6
        return fin_mars or avril or mai or juin
```

Les tests réussissent.

Fin juin, nous sommes en été

Code ajouté au fichier `test_tdd.py` :

```
def test_fin_juin():
    date = Date(6, 23)
    attendu = «été»
    effectif = date.saison()
    assert attendu == effectif
```

Les tests échouent. On corrige le fichier `saison.py` ainsi :

```
class Date:
    def __init__(self, mois, jour):
        self.mois = mois
        self.jour = jour

    def est_printemps(self):
        fin_mars = self.mois == 3 and self.jour >= 21
        avril = self.mois == 4
        mai = self.mois == 5
        juin = self.mois == 6 and self.jour < 21
        return fin_mars or avril or mai or juin

    def est_ete(self):
        return self.mois == 6 and self.jour >= 21

    def saison(self):
        if self.est_printemps():
            return «printemps»
        elif self.est_ete():
            return "été"
        else:
            return «hiver»
```

Les tests réussissent.

Trop de « nombres magiques » disséminés dans le fichier

On *refactorise* ainsi le fichier `saison.py` :

```
JANVIER = 1
FEVRIER = 2
MARS = 3
AVRIL = 4
MAI = 5
JUIN = 6
JUILLET = 7
AOUT = 8
SEPTEMBRE = 9
OCTOBRE = 10
NOVEMBRE = 11
DECEMBRE = 12
CHANGEMENT = 21
class Date:
    def __init__(self, mois, jour):
        self.mois = mois
        self.jour = jour

    def est_printemps(self):
        fin_mars = self.mois == MARS and self.jour >= CHANGEMENT
        avril = self.mois == AVRIL
        mai = self.mois == MAI
        juin = self.mois == JUIN and self.jour < CHANGEMENT
        return fin_mars or avril or mai or juin

    def est_ete(self):
        return self.mois == JUIN and self.jour >= CHANGEMENT

    def saison(self):
        if self.est_printemps():
            return «printemps»
        elif self.est_ete():
            return «été»
        else:
            return «hiver»

def saison(mois, jour):
    date = Date(mois, jour)
    return date.saison()
```

Les tests réussissent. Le refactoring est validé.

Et ensuite…

En continuant le processus, on va également être amené à définir la méthode `est_automne`, corriger la méthode `est_ete`, etc . Il faut également rajouter des cas de tests pour les données non valides, ce qui amène à ajouter des `assert` pour matérialiser les préconditions des méthodes.

On constate qu'en appliquant le TDD :

- on obtient un code mieux structuré que ce qu'on aurait probablement fait sans ;
- le module est équipé d'une collection de tests qui permettent d'assurer la non-régression lors des prochaines évolutions du module ;
- le quatrième test écrit dans la première version, et révélé inutile par la couverture de code sur les tests boîte blanche, n'est pas écrit.

On obtient ainsi un code bien plus sûr et de meilleure qualité, mais après un travail qui peut sembler relativement long et fastidieux. Cependant le temps passé à concevoir un code ainsi est en général bien inférieur au temps pris à corriger les éventuels bugs laissés en procédant autrement, et peut coûter bien moins cher que les éventuelles conséquences des bugs.