TDLOG séance 6 - Test

Xavier Clerc – xavier.clerc@enpc.fr

10 novembre 2025

Table des matières

1 Introduction

Lors des séances précédentes, nous avons vu comment écrire et architecturer un programme Python. Lors de cette séance, nous allons étudier les mécanismes de test qui permettent de s'assurer que le programme se comporte comme attendu. Les sections 2 à 6 présentent les principes à la base des tests qui sont, de fait, indépendants du langage utilisé. Les sections 7 et 8 proposent une approche du test également indépendante du langage utilisé. Enfin, les sections 9 et 10 montrent comment mettre en application les éléments précédents dans le cadre du langage Python : les modules et outils nécessaires au test de programmes Python sont présentés.

2 Questions générales

2.1 Pourquoi tester ?

Dans le cadre de l'enseignement, la question du test peut passer pour une question liée à l'évaluation. Tester un programme demandé en exercice à un étudiant serait pour l'enseignant une manière d'évaluer si le travail a été correctement effectué. Cependant, le test prend une importance cruciale dans le cadre de développements professionnels. En effet, le logiciel prend une place sans cesse croissante dans la vie quotidienne et des bugs dans celui-ci peuvent donc affecter tout un chacun dans ses activités. Une tendance lourde du logiciel (et de l'électronique lui servant de support) est de devenir *pervasif*, *ambient* ou encore *ubiquitaire*, c'est-à-dire présent à peu près partout et simultanément souvent invisible (fonctionnement d'appareils ménagers, de moyens de transport, *etc*).

Cette omniprésence du logiciel tend à produire une situation dans laquelle les bugs logiciels ne sont plus confinés à l'utilisation d'un ordinateur, mais affectent nos activités ordinaires. Classiquement, on distingue trois niveaux de désagrément liés aux bugs :

1. les pertes de temps (*p. ex.* réinitialisation d'un équipement) ou de confort (*p. ex.* mauvais fonctionnement d'un thermostat) ;
2. les pertes financières (*p. ex.* destruction d'un équipement) ;
3. les pertes humaines (*p. ex.* accident d'avion).

La première catégorie est en général une source de désagréments mineurs et a longtemps été considérée comme acceptable (voire inhérente à l'utilisation de l'électronique ou de l'informatique).

La seconde catégorie a toujours été prise plus au sérieux, notamment parce que les coûts induits peuvent être arbitrairement élevés. L'exemple classique est celui du premier tir de la fusée Ariane v : outre le coût (facilement chiffrable) de la perte de la cargaison, il faudrait également ajouter les coûts indirects par exemple en termes de réputation.

Enfin, la troisième catégorie, bien qu'extrêmement sensible, a elle aussi donné lieu à des accidents dramatiques. S'il n'existe pas (à ce jour et à notre connaissance) de situation de perte de vie provoquée par un bug dans un programme d'aéronautique ou de centrale nucléaire, il existe des exemples dans le domaine des voitures autopilotées et de la médecine. Par exemple, des bugs dans des appareillages de radiothérapie ont déjà provoqué des décès consécutifs à une irradiation excessive des patients traités.

Ce spectre des conséquences des bugs informatiques trace implicitement un autre spectre : celui de l'importance accordée aux tests lors d'un développement logiciel donné. Si l'objectif fondamental du test est toujours le même, les exigences sont évidemment très différentes selon les usages qui seront faits des logiciels.

2.2 Qu'attendre du test ?

Selon la catégorie de l'application développée, c'est à dire grossièrement les risques qu'elle fait peser sur l'utilisateur (de la perte de données à la perte de la vie), les objectifs et l'importance d'une politique de test sont tout à fait différents. Lorsque des vies humaines sont en jeu, on veut idéalement disposer d'une *preuve* que le programme va toujours se comporter comme attendu, ou à tout le moins avoir des défaillances non dramatiques.

Ainsi, dans le cadre de l'avionique, la politique de tests est entièrement spécifiée et normalisée (*p. ex.* DO-178), notamment par un grand nombre de documents qui doivent être rédigés pour prouver non pas le logiciel lui-même, mais que les efforts requis ont été effectués en matière de tests.

À l'autre extrémité du spectre, par exemple dans le cadre de jeux vidéos, on pourrait être tenté de considérer que le test est tout à fait accessoire et qu'il permet seulement d'élever un peu la qualité du logiciel produit.

Cela nous amène à considérer deux appréhensions du test qui peuvent être relativement indépendantes :

- la pratique du test au sein d'une structure afin d'atteindre les objectifs de qualité requis ;
- la pratique du test comme outil personnel afin d'écrire au plus vite un programme conforme aux attentes.

C'est pourquoi nous considérons que la maîtrise des questions relatives au test logiciel est importante, y compris dans le cadre de logiciels sans impact social fort. Nous défendons l'idée que la capacité à savoir tester une portion de logiciel écrite fait partie des qualités d'un bon développeur : il produit des programmes de meilleure qualité et est en meilleure situation de corriger des erreurs qui lui auraient échappé ou d'apporter des modifications à un programme déjà écrit.

2.3 Qu'est-ce que tester ?

Si l'on est convaincu de l'importance du test, quelle que soit la situation de développement, reste à définir exactement ce que l'on entend par *test*. Tester c'est *montrer qu'un programme donné se comporte comme spécifié*. Cette définition fait en général consensus, car elle masque pratiquement tous les problèmes soulevés par le test :

- qu'entend-on par *montrer* : par une démonstration ou par l'exemple ?
- qu'est-ce qu'une *spécification* : un descriptif en langue naturelle ou une formule dans un langage formel ?

Montrer

La première question trouve en général une réponse terminologique : quand on parle de *test* on exclut généralement la volonté de faire une démonstration, auquel cas on parlerait de *preuve* et non de *test* d'un logiciel.

La conséquence immédiate et élémentaire de cet état de fait est la suivante : le test permet de montrer l'existence de bugs, mais pas leur absence. Il s'agit d'une limite épistémologique bien connue des sciences expérimentales : une expérience peut produire un contre-exemple qui invalide une théorie, mais ne peut valider une théorie. Au mieux, une expérience dont l'issue est positive ne peut qu'améliorer la confiance que l'on a dans cette théorie.

De manière moins immédiate, on notera également que même dans les cas où l'on peut *prouver* qu'un logiciel est correct, il est néanmoins utile de le *tester*. On peut dire que l'échec du premier tir d'Ariane v n'était pas dû à un bug dans le programme, mais à un changement dans son environnement d'exécution (les caractéristiques de la nouvelle version de l'appareil ont produit des mesures hors des bornes attendues). Il s'agit d'une autre limite épistémologique : disposer d'une preuve ne garantit en rien que cette preuve s'applique effectivement aux objets du monde réel modélisés mathématiquement.

Spécifier

La seconde question trouve une réponse au cas par cas, mais la plupart du temps il s'agit d'un descriptif en langue naturelle. Bien entendu, c'est une source majeure de problèmes du fait des ambiguïtés des langues naturelles. À titre d'exemple, voici deux spécifications pour une même fonction $sqrt$, la première étant informelle et la seconde formelle :

— $sqrt(x)$ doit renvoyer la racine carrée de x ;
— pour $x \geq 0$ et $e > 0$ donnés, $sqrt$ doit être telle que $|sqrt(x)^2 - x| \leq e$.

Même sur cet exemple très simple, on peut observer que la spécification formelle est sensiblement plus précise que la spécification informelle : elle donne une caractérisation mathématique du résultat attendu (en tenant compte des problèmes d'arrondi) et donne clairement le domaine de définition de la fonction.

2.4 D'où viennent les bugs ?

Jusqu'ici, nous avons uniquement utilisé le terme *bug* pour qualifier les problèmes rencontrés, et c'est en général la pratique courante dans les équipes de développement. Cependant, dans la littérature sur les tests, les termes suivants sont utilisés pour qualifier plus précisément les problèmes rencontrés.

Une erreur est une action humaine à l'origine d'un défaut (*p. ex.* inattention, manque d'information).

Un défaut est un résultat présent dans le logiciel et issu d'une erreur (donc d'origine humaine).

Une défaillance est la réalisation effective d'un défaut (donc la manifestation observable de celui-ci).

Le terme générique *bug* recouvre ces trois aspects, c'est pourquoi ces éléments de lexique permettent d'identifier plus précisément les problèmes rencontrés. On notera que si la défaillance ne peut être constatée que sur un système s'exécutant, une erreur ou un défaut peuvent être consta-

tés sur la spécification et/ou le code source d'un logiciel. Donc en amont de l'observation d'un problème, soit pour un impact et un coût très inférieurs.

Ces termes précis mettent également l'accent sur le fait que la littérature sur le test logiciel définit clairement son périmètre d'étude : seuls les problèmes ayant une origine humaine sont concernés par le test. Ainsi, des problèmes observés dus à des rayons cosmiques ou des champs magnétiques ne rentrent pas en ligne de compte dans l'étude des causes des bugs. Bien entendu, cela n'empêche pas de les considérer par ailleurs lorsque le logiciel est amené à opérer alors que des vies sont en jeu (*p. ex.* avionique, nucléaire).

Enfin, dans le cadre de cette terminologie affinée, tester c'est identifier les défaillances. Une fois celles-ci identifiées, l'objectif est de déterminer l'origine des défauts (*p. ex.* problèmes de communication, omissions, interprétations erronées, *etc*). Pour remonter la chaîne de causalité, l'étape de débugage aura pour but de déterminer les sources des défauts c'est-à-dire les erreurs.

2.5 Quels sont les facteurs humains ?

Depuis le début de cette séance, nous n'avons pas encore évoqué les facteurs humains. Nous abordons ici brièvement les aspects psychologique et éthique du test : l'aspect psychologique traite à l'évidence de l'individu tandis que l'aspect éthique traite quant à lui du rapport de l'individu à la société.

L'aspect psychologique se manifeste dans le test logiciel, car il s'agit pour le programmeur de réfléchir au résultat de son activité afin d'y détecter de possibles erreurs. D'une certaine manière, c'est ce que l'on attend de toute personne ayant une formation scientifique et faisant preuve d'honnêteté intellectuelle. Étonnamment, dans le monde du logiciel, la pratique systématique du test ne va pas de soi, même si des progrès semblent avoir été faits ces dernières années. Il existe de nombreuses (pseudo-)raisons dépendantes du cadre de travail :

- le développeur a une activité constructive tandis que le testeur a une activité destructive (c'est évidemment faux, un scientifique rigoureux doit se faire l'*avocat du diable* avant de présenter ses travaux) ;
- le programme est la traduction d'une théorie prouvée correcte, on ne voit pas pourquoi tester une traduction (c'est évidemment faux, une erreur d'inattention étant vite arrivée) ;
- certaines organisations séparent les équipes de développement et de test, on ne voit pas pourquoi faire le travail d'un autre (c'est évidemment faux, le développeur est souvent plus à même de découvrir et corriger rapidement un bug) ;
- le temps manque pour effectuer les tests nécessaires (c'est une distorsion de la réalité, il faudra beaucoup plus de temps pour corriger une défaillance en production que pour repérer une erreur par le test).

L'aspect éthique, dont l'importance est bien entendu liée au niveau de désagrément que peut engendrer un bug, se manifeste de façon de plus en plus importante pour les raisons évoquées plus haut (informatique *ubiquitaire*). Cet aspect, originellement essentiellement cantonné à l'avionique, se propage à d'autres domaines où le logiciel prend pied (*p. ex.* automobile, appareils médicaux).

Le questionnement éthique d'un développeur travaillant dans ces domaines est donc d'estimer en conscience si lui et l'organisation dont il dépend ont effectué tous les tests nécessaires pour garantir la sécurité des utilisateurs. Ce questionnement doit se placer prioritairement à des questions d'intérêt personnel ou de contraintes hiérarchiques. L'ACM [1] a ainsi défini un code d'éthique

1. *Association for Computing Machinery*.

auquel devraient adhérer les personnes chargées de tester des logiciels critiques. L'extension du champ d'utilisation du logiciel et cette prise de conscience éthique pourraient développer des pratiques telles que le *whistleblowing* [2].

3 Types de tests

Nous présentons dans cette section une typologie classique des tests, indépendante de leur réalisation concrète. L'objectif d'une telle typologie est donc de classer les tests en catégories afin de délimiter les périmètres des diverses étapes ou parties d'une politique de tests.

En particulier, nous avons vu lors de la séance sur la gestion de projets que les différents types de tests ne sont pas conduits au même moment du développement d'un projet. La typologie permet alors d'estimer les tests à mener en fonction de l'état d'avancement du projet.

On distingue classiquement deux axes pour caractériser les tests :
- le niveau de granularité auquel ils se situent ;
- les caractéristiques qu'ils permettent de tester.

3.1 Niveaux de granularité

On distingue le plus souvent trois niveaux de granularité pour les tests, par granularité croissante on trouve :
- les tests *unitaires* ;
- les tests d'*intégration* ;
- les tests de *validation* (ou de *recette*).

Tests unitaires

Les tests unitaires sont ceux de plus faible granularité. Cela signifie qu'ils ont pour objet de tester les plus petites unités logiques du programme. Cette notion de plus petite unité logique est quelque peu variable : en fonction du langage principalement, en fonction du projet secondairement. La notion de plus petite entité dépend en effet du langage et des abstractions qu'il propose : dans un langage fonctionnel la plus petite entité est la fonction alors que dans un langage orienté objet la plus petite entité est la classe. La notion de plus petite entité dépend également du projet en ce sens que selon la complexité on pourra être tenté d'organiser les tests selon un grain un peu plus gros : par exemple les modules en Python.

Les tests unitaires permettent de s'assurer que les briques de base du logiciel, prises indépendamment les unes des autres fonctionnent correctement. Il s'agit donc de s'assurer que les fondations mêmes du logiciel en construction sont solides. D'une certaine manière, il s'agit de définir les lemmes de base qui vont être utilisés à divers endroits et dans différentes situations ; il est donc important de s'assurer non seulement de leur exactitude, mais également que l'on dispose de la généralité nécessaire.

L'expérience montre deux choses : d'une part, les tests unitaires sont le plus souvent négligés (au motif qu'il y a peu de probabilité de faire des erreurs sur les choses "simples"), et d'autre part, qu'ils sont très utiles pour diagnostiquer et corriger au plus tôt les erreurs (corriger une erreur relevée sur quelques dizaines de lignes est plus facile qu'une sur quelques milliers de lignes).

2. Lancement d'alerte par un individu lorsqu'une entreprise ou une institution enfreint des règles légales ou déontologiques.

Tests d'intégration

Les tests d'intégration sont de granularité intermédiaire : il ne s'agit ni du logiciel dans son ensemble ni des briques de base nécessaires à sa constitution. Cela signifie qu'ils ont pour but de tester des portions de l'architecture du logiciel en développement. Ils peuvent ainsi permettre d'identifier des erreurs de conception dans l'architecture de l'application.

Les tests d'intégration permettent de s'assurer que les briques de base se composent facilement et correctement. Par "facilement", il faut comprendre que les interfaces entre les différentes briques de base permettent de les composer sans avoir à écrire par exemple de code de conversion entre types de données. Par "correctement", il faut comprendre que les briques logicielles se comportent de la même manière isolées et composées ; ici, ce sont souvent les effets de bord qui posent problème (on pourra, par exemple, les limiter en ayant comme briques de base des éléments non mutables).

L'expérience montre qu'il faut éviter ce que l'on appelle le *big bang* de l'intégration, c'est-à-dire l'agrégation simultanée de toutes les pièces constituant le logiciel. En effet, même si les différentes pièces ont été testées individuellement, rien n'indique qu'elles vont se comporter correctement ensemble. Il convient donc d'effectuer les tests d'intégration au fur et à mesure, en composant les éléments déjà disponibles plutôt que de reporter l'intégration à plus tard.

Tests de validation

Les tests de validation, également appelés tests de recette, s'effectuent sur l'ensemble de l'application. Ils sont donc par définition les tests de plus forte granularité, indépendamment du langage ou du projet. On peut distinguer deux manières d'effectuer ces tests de validation : en interne ou avec la participation des utilisateurs.

Les tests de validation permettent de s'assurer que le logiciel produit correspond bien au cahier des charges, et convient aux utilisateurs. Le premier aspect peut s'effectuer par des tests menés en interne (soit par l'équipe de développement, soit par une équipe spécialisée). Le second aspect nécessite la participation active des utilisateurs (utilisateurs prospectifs, ou utilisateurs d'une version antérieure).

L'expérience montre que le cahier des charges (explicitement rédigé ou non) et les attentes des utilisateurs ne sont pas nécessairement alignés, même si le cahier des charges a été rédigé par ou avec l'aide des utilisateurs. Pour cette raison, il est le plus souvent intéressant de recueillir l'avis des futurs utilisateurs avec une version plus ou moins avancée du logiciel. C'est l'essence même des versions *alpha* et *bêta* : les utilisateurs savent qu'ils n'utilisent pas une version finalisée et peuvent faire des retours aux développeurs.

3.2 Caractéristiques testées

On distingue deux genres de caractéristiques concernant un logiciel :

— les caractéristiques *fonctionnelles* ;
— les caractéristiques *non fonctionnelles*.

Caractéristiques fonctionnelles

Les caractéristiques fonctionnelles concernent *ce que fait* le logiciel, donc les fonctionnalités qu'il propose. Nous nous trouvons ici dans la notion de *correction* d'un logiciel : produit-il toujours le résultat attendu ?

Les caractéristiques fonctionnelles sont généralement les plus faciles à tester car il est le plus souvent relativement aisé de caractériser ce qu'est un comportement correct du logiciel. Cependant, il faut être attentif au fait que cette caractérisation n'est pas nécessairement une fonction au sens mathématique :

- le comportement peut dépendre d'un *état* (*p. ex.* fichier, horloge) ;
- le comportement peut dépendre des interactions avec un utilisateur (*p. ex.* interface graphique) ;
- il peut y avoir plusieurs solutions à un problème donné, ou l'on peut se satisfaire d'une solution approximative (*p. ex.* calcul d'un itinéraire).

Caractéristiques non-fonctionnelles

Les caractéristiques non-fonctionnelles concernent *comment* le logiciel fait ce pour quoi il a été conçu, donc comment les fonctionnalités sont assurées. Nous nous trouvons ici en dehors de la notion de *correction* d'un logiciel : pourvu qu'il soit correct, est-il effectivement utilisable ?

Au sein des caractéristiques non-fonctionnelles, on trouve notamment :

- les performances : le logiciel délivre-t-il ses résultats en un temps acceptable ?
- la gestion des pannes : le logiciel bénéficie-t-il d'un système de redondance ou de reprise en cas d'échec qui évite de recommencer un calcul depuis le début ?
- la sécurité : le logiciel garantit-il la confidentialité des données voire l'intégrité physique des utilisateurs ?

Les exigences non-fonctionnelles ne sont pas toujours bien spécifiées (*p. ex.* si un utilisateur veut un temps de réponse "acceptable", s'agit-il d'une seconde ou d'un dixième de seconde ?) et ne sont pas nécessairement orthogonales aux exigences fonctionnelles (*p. ex.* dans un système de navigation en temps réel les performances sont liées à la fonctionnalité).

4 Natures de tests

Les techniques de tests sont dites en *boîte noire* si elles n'utilisent pas le code source (mais éventuellement seulement le code binaire) du logiciel. La partie testée est vue comme une boîte noire au sens où son contenu n'est pas examiné : on voit cette partie comme une abstraction pourvue d'entrées et de sorties. Le principe des tests en boîte noire est donc d'agir sur les entrées et de vérifier que les sorties sont correctes.

Les techniques de tests sont dites en *boîte blanche* si elles s'appuient sur le code source du logiciel. La partie testée est examinée pour déterminer de possibles erreurs en fonction de la manière dont elle est codée. Par exemple, si le code utilise un tableau et un indice, une technique en boîte blanche visera à vérifier que les valeurs de l'indice sont toujours dans les bornes du tableau.

En général, la littérature se limite à la dichotomie entre boîte blanche et boîte noire. Cependant, certains auteurs parlent également de *boîte grise* pour indiquer une connaissance partielle du

code source (*p. ex.* l'algorithme seul ou les structures de données seules, sans leurs réalisations concrètes).

5 Techniques de tests

5.1 Techniques de base

Assertions

Nous avons utilisé, lors de précédentes séances, l'instruction `assert` qui permet de s'assurer qu'une propriété donnée est vraie en un point d'un programme. Le test par assertion fonctionne selon la même logique à ceci près que :

— dans le cadre du test, on ne veut pas s'arrêter à la première assertion violée, il est en effet utile d'avoir une vue d'ensemble des erreurs révélées par les tests ;

— l'utilisation de l'instruction `assert` a lieu pendant une quelconque exécution du programme alors que les assertions dont nous parlons ici ne sont utilisées que lors des tests.

Le test par assertions revient à effectuer un certain nombre de calculs et à ensuite vérifier que le résultat correspond à ce qui est attendu. Par exemple, le code Code 5.1 montre une fonction testant la fonction *factorielle*. On énumère un certain nombre de cas pour lesquels on connaît les valeurs d'entrée et de sortie attendues.

Code 5.1 – Tests par assertions.

```
1 def test_fact() -> None:
2   assert factorielle(0) == 1
3   assert factorielle(2) == 2
4   assert factorielle(5) == 120
```

De manière plus générale, les tests par assertions (souvent utilisés pour les tests unitaires) reposent sur les étapes suivantes :

1. mise en place du test (*setup*), *p. ex.* création des objets nécessaires ;
2. calculs relatifs au test ;
3. comparaisons des résultats effectifs aux résultats attendus ;
4. nettoyage des éléments créés à l'étape 1 (*teardown*).

Énumérations

Les tests par énumérations peuvent être vus comme une généralisation des tests par assertion, ou comme une version pragmatique et drastiquement limitée du *model checking*. Le *model checking* consiste à explorer l'ensemble des états possibles d'un logiciel. Cependant, le cardinal de cet ensemble est le plus souvent gigantesque, ce qui limite le *model checking* à des sous-portions de logiciels ou à des logiciels critiques (*p. ex.* en avionique).

Le test par énumérations revient souvent en pratique à effectuer un test par assertions sur une plage de données. Si cela est matériellement possible, on peut tester l'ensemble des entrées possibles d'une fonction (*p. ex.* fonction de quatre paramètres de type booléen) ; sinon, on se restreint à des intervalles significatifs (soit du point de vue de l'utilisateur dans le cadre de tests de type boîte noire, soit du point de vue de l'implémentation de la fonction dans le cadre de tests de type boîte blanche).

Alea

Il peut sembler paradoxal d'introduire de l'*alea* dans une procédure qui revient à s'assurer qu'un programme écrit correspond à ce qui est attendu. Cependant, en pratique, les tests aléatoires ont montré quelque intérêt : par exemple, ils sont capables d'exhiber des contre-exemples pour lesquels on n'aurait pas pensé à écrire de test.

En effet, le risque est grand lorsque l'on teste une fonction que l'on soit partiellement aveuglé :
- par l'implémentation que l'on vient d'en faire ;
- par un type d'erreur auquel on est particulièrement sensible ;
- psychologiquement, en écrivant plutôt des tests qui confortent le code plus que des tests qui le *challengent* ;
- *etc*

Pour toutes ces raisons, il peut être utile **en complément** de tests par assertions ou énumérations, d'effectuer des tests aléatoires en misant sur la probabilité qu'une valeur à laquelle on n'a pas pensé va être générée et produire un contre-exemple. Les tests aléatoires sont également utiles pour construire des données d'entrée de grande taille, là où il serait très fastidieux de le faire manuellement.

Bien entendu, les tests aléatoires souffrent du problème majeur qu'ils ne sont pas reproductibles. Si on les utilise, il faut donc être particulièrement attentif à bien stocker le contre-exemple trouvé (ou la graine du générateur pseudo-aléatoire). Contre-exemple que l'on peut ensuite ajouter aux tests effectués par assertions.

5.2 Techniques fondées sur les propriétés des éléments à tester

Conditions aux limites

L'expérience montre que de nombreuses erreurs de programmation se nichent dans les conditions aux limites des éléments à tester, le plus souvent parce que le programmeur s'est concentré sur le cas général au détriment des cas particuliers.

L'exemple classique de condition aux limites concerne la taille des éléments manipulés. Ainsi, pour tous les types de données où cela aura un sens, on vérifiera ce qui se passe si l'élément est vide : liste, tableau, chaîne de caractères, *etc*

Un autre exemple concerne les valeurs d'entrée d'une fonction, qu'elles aient été fournies par un utilisateur ou une autre partie du programme. Il est important de vérifier que le programme se comporte correctement (y compris par la levée d'une exception) si ces données n'ont pas le format attendu.

Composition à l'identité

Dans l'écriture de logiciels, et plus encore dans l'écriture de bibliothèques, il est fréquent que l'on dispose d'un ensemble de fonctionnalités travaillant sur les mêmes concepts. Une manière très simple de tester la robustesse de plusieurs fonctions est de les tester simultanément. Le cas d'école est celui d'une librairie disposant d'une fonction f et d'une fonction g effectuant le calcul inverse de f. Dans ce cas, on peut tirer parti du faire que $f \circ g = g \circ f = Id$ pour créer un ensemble de tests.

On peut, au choix, procéder par assertion, énumération ou aléa puisque l'on sait que la composition des deux fonctions est l'identité. De tels tests sont donc triviaux à écrire : puisque

$\forall x, f(g(x)) = x$ et $\forall y, g(f(y)) = y$, on peut choisir librement une liste de x_i et de y_i. Un avantage de cette technique de test est qu'elle permet de détecter des erreurs arbitrairement complexes tout en regardant une fonction comme une boîte noire. Plus encore que dans les boîtes noires *classiques*, on n'a même pas ici à s'interroger sur les valeurs d'entrée et de sortie (il faut tout de même porter attention aux domaines de définition des fonctions).

Bien entendu, le fait que l'on teste la composition de plusieurs fonctions signifie que le test, si l'on veut être rigoureux, porte sur la propriété qui lie les deux fonctions et non sur les fonctions elles-mêmes. Cette méthode de test de permet en aucun cas de garantir que f et g sont correctes ; seulement qu'une partie de leur spécification l'est. En effet, à l'extrême, pour satisfaire ce genre de test il suffit de poser $f = g = Id$.

Enfin, on notera que si l'on n'a pas toujours de fonction inverse g pour une fonction f donnée, il existe de nombreux cas dans lesquels la fonction f est difficile à écrire tandis que la fonction g est triviale à écrire. Dans cette optique, il peut être utile de coder g uniquement à des fins de tests.

Involutions, points fixes, oracles, *etc*

Dans la même veine que le test par composition à l'identité, on peut mettre en place de nombreuses autres techniques de tests fondées sur des propriétés mathématiques. Par exemple, si l'on n'a pas de fonction inverse et/ou que celle-ci est trop difficile à écrire, on peut se rabattre sur d'autres propriétés mathématiques utiles dans le cadre du test.

Ainsi, si une fonction est involutive, on peut utiliser la technique de test précédente en se fondant sur le fait que $\forall x, f(f(x)) = f(x)$ [1]. Plus généralement, on peut utiliser la propriété qu'une fonction donnée possède un point fixe. Il est également possible de combiner plusieurs propriétés ; par exemple on peut avoir deux fonctions f et g dont la composition n'est pas l'identité, mais est involutive, *etc*

Enfin, une technique commune est le recours à un *oracle*. Un oracle est essentiellement un prédicat, c'est à dire une fonction qui retourne *vrai* ou *faux*. L'utilisation d'un oracle pour les tests peut se faire de plusieurs manières :

— on dispose d'une implémentation de référence ;
— on sait tester facilement si un résultat est acceptable.

Le premier cas se produit lorsque l'on cherche par exemple à écrire une fonction optimisée d'une fonction qui soit existe déjà, soit pour laquelle on peut écrire à des fins de test une version naïve. Il est alors facile de vérifier que pour toute valeur, les résultats donnés par les deux fonctions sont égaux.

Le second cas se produit lorsque la programmation d'une fonction est complexe, mais le fait de vérifier si le résultat renvoyé est correct est beaucoup plus simple à programmer. Par exemple, il est plus facile de vérifier qu'une liste est triée que d'écrire une fonction de tri. Donc, si l'on doit programmer une fonction f de tri, il peut être utile pour la tester de programmer en plus une fonction g vérifiant qu'une liste est triée. On peut alors se fonder sur la propriété $\forall x, g(f(x)) = vrai$ pour écrire facilement des tests, y compris aléatoires.

1. En outre, dans ce cas on est certain de ne bien tester que la fonction f.

6 Couverture de code

6.1 Généralités

Nous avons jusqu'ici présenté diverses catégories de tests et quelques manières de concevoir des tests. Si l'on a commencé à écrire des tests, on peut se poser la question de savoir ce que l'on a effectivement testé, d'une part. Et savoir quand il n'est plus nécessaire d'ajouter des tests, d'autre part.

La notion de couverture de code permet de donner des éléments de réponse à ces deux questions. La couverture de code consiste à déterminer, pour un ensemble de tests donné, quelles sont les portions du code qui ont été exécutées. L'idée sous-jacente est que plus les tests ont *exercé* une partie importante du code, plus on peut avoir confiance dans la correction du programme.

Cette notion est très importante, car elle permet d'identifier sans ambiguïté les parties du programme qui n'ont jamais été exécutées lors des tests. En cela, elle indique clairement le travail de test restant à faire : il faut produire de nouveaux tests dont l'exécution *exercera* de nouvelles parties du code.

Cependant, il est également très important de ne pas se méprendre sur l'indication que donne la couverture de code, et en particulier le pourcentage de code couvert (qui est parfois utilisé comme métrique de qualité par un encadrement hiérarchique non-technique). Nous allons voir au travers de trois types de couvertures pourquoi il ne s'agit que d'une indication partielle.

6.2 Couverture d'instruction

Le type de plus simple de couverture est la couverture d'instruction. Elle répond à la question "une instruction donnée a-t-elle été exécutée lors de l'exécution d'un ensemble de tests ?". Le code Code 6.1 montre une fonction Python pour laquelle chaque instruction exécutée par le jeu de test "1, 2, 4" est marquée par une étoile.

Code 6.1 – Couverture d'instruction.

```
1 (*) def f(x):
2 (*)   if x <= 1:
3 (*)     return 1
4 (*)   elif ((x % 2) == 0) or (x > 100):
5 (*)     return 2
6 ( )   else:
7 ( )     return 0
```

La couverture d'instruction marque bien qu'aucune des valeurs de test ne passe dans la branche *else* de l'instruction conditionnelle. Cependant, l'information de couverture n'est que partielle, car elle nous masque que l'on est passé par la branche *elif* qu'à la faveur d'entrées paires et jamais par une entrée supérieure à 100.

On peut donc être amené à penser qu'il faut rajouter un test (*p. ex.* 5) alors qu'il en faudrait deux (*p. ex.* 5 et 150) pour couvrir tous les cas.

6.3 Couverture de condition

Pour résoudre le problème précédent, un type de couverture un peu plus élaboré a été mis en place : la couverture de condition. Cette fois-ci, on dispose de deux étiquettes possibles par instruction : un symbole plus pour indiquer que l'instruction a été totalement couverte et un symbole moins pour indiquer qu'elle n'a été que partiellement couverte. Si l'on reprend l'exemple précédent, on obtient le code Code 6.2 qui nous montre bien qu'il manque deux tests.

Code 6.2 – Couverture de condition.

```
(+) def f(x):
(+)   if x <= 1:
(+)     return 1
(-)   elif ((x % 2) == 0) or (x > 100):
(+)     return 2
( )   else:
( )     return 0
```

Cependant, ce type de couverture n'est pas non plus suffisant, comme le montre le code Code 6.3. En effet, si l'on teste la fonction avec les valeurs `(-1, -1)` et `(1, 1)` on obtient une couverture totale. Or, celle-ci est également trompeuse, car :

— dans le premier cas, on est passé par les branches `if` puis `else` ;
— dans le second cas, on est passé par les branches `else` puis `if`.

Dans aucun cas de test nous n'avons parcouru les combinaisons `if` / `if` ou `else` / `else`.

Code 6.3 – Limite de la couverture de condition.

```
def f(x, y):
  res = 0
  if x < 0:
    res += 1
  else:
    res -= 1
  if y > 0:
    res //= y
  else:
    res *= y
  return res
```

6.4 Couverture de chemin

Pour résoudre le problème précédent, un type de couverture encore plus élaboré a été mis au point : la couverture de chemins. Il s'agit ici de marquer tous les chemins parcourus lors des tests par rapport à tous les chemins possibles.

Cette notion de couverture est donc encore plus fine que la précédente, mais pose un problème de représentation. Il faut extraire du code l'ensemble des chemins possibles pour pouvoir les exposer au développeur, en marquant les chemins *exercés* comme le montre la figure Figure 6.1 pour l'exemple précédent.

Sur cette figure, les rectangles représentent les différents blocs d'instructions et les losanges, les instructions de choix. Les flèches pleines montrent les enchaînements possibles entre blocs tandis que les flèches pointillées montrent les quatre chemins possibles.

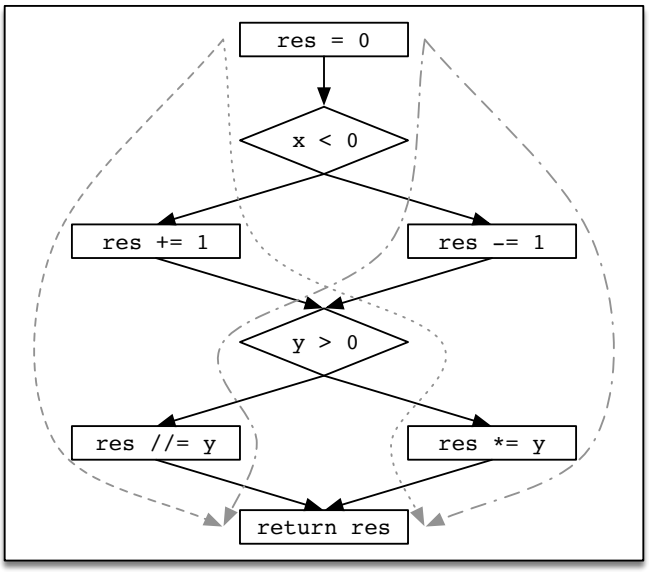

FIGURE 6.1 – Couverture de chemins.

Cependant, outre le problème de représentation, cette notion de couverture n'est, elle non plus pas suffisante. Pour s'en convaincre, il suffit de penser à un code contenant une boucle. La couverture de chemins peut montrer que le corps de cette boucle est soit jamais parcouru soit parcouru au moins une fois. Quid alors d'une erreur qui se produirait par exemple après n parcours de la boucle ?

La conclusion qu'il faut retenir est que la notion de couverture de code est très utile pour identifier les parties non testées, mais que les parties marquées comme testées ne le sont pas forcément de manière satisfaisante. C'est donc un problème de *faux positif* dont il faut avoir conscience lors de l'utilisation de cet outil.

7 Pratiques associées au test

7.1 Problématique de non-régression

Une problématique importante des logiciels dont le développement s'étale dans le temps est celle de la non-régression. Une régression, comme son nom l'indique, est la disparition (totale ou partielle selon la sévérité du bug) d'une fonctionnalité dans une version ultérieure, qui peut être la réapparition d'un bug déjà corrigé.

Or, le risque est grand, avec le temps, que des modifications apportées à une partie du logiciel en affectent une autre et entraînent une régression. Cela peut se produire quelles que soient les modifications apportées : correction d'un (autre) bug, ajout d'une fonctionnalité, changement d'architecture du logiciel, *etc*

Dans cette optique, la question de la non-régression consiste à disposer de suffisamment de tests afin d'être assuré que tant que les tests montrent que la nouvelle version du logiciel fonctionne, on n'effectue pas de régression par rapport à la version précédente. Une pratique commune pour se

prémunir des régressions consiste par exemple à systématiquement ajouter un test à l'ensemble de tests existant lorsque l'on corrige un bug. Ainsi, si celui-ci *réapparaît*, la régression sera diagnostiquée par l'ensemble de tests augmenté.

7.2 *Test-driven development*

Le *Test-First* et plus encore le *Test-Driven Development* (ou TDD) sont des méthodes de développement qui renversent radicalement les étapes du développement logiciel telles qu'on les conçoit classiquement. En effet, classiquement, on commence par écrire un programme puis on teste sa correction.

Le TDD propose tout simplement d'inverser ces étapes : on commence par écrire un ensemble de tests qui permet d'assurer la correction du programme à écrire, puis on écrit le programme. Ensuite, on corrige le programme jusqu'à ce qu'il se conforme à tous les tests et on a ainsi *par construction* (selon les promoteurs du TDD) un programme correct.

Dans les détails, le TDD est une démarche hautement itérative : on écrit un test puis la portion de code nécessaire à la satisfaction de ce test. Ensuite, on répète cette petite opération jusqu'à disposer du programme complet. On construit ainsi de manière incrémentale le programme.

Cette vision extrême du test n'est à l'évidence pas une réponse universelle, même si elle donne de bons résultats en pratique. En effet, les concepteurs du TDD ont proposé cette méthode car pour eux, si l'on procède dans l'ordre classique, on est (volontairement ou involontairement) *guidé* dans les tests que l'on met en place par le code que l'on a écrit (*p. ex.* on ne pense à tester que les conditions aux limites que l'on a prises en compte lors de l'implémentation). En d'autres termes, on risque d'être victime du biais de confirmation.

Cet argument est tout à fait juste et fait peser une légitime suspicion sur les tests écrits après-coup. En revanche, la conclusion menant au TDD ne semble pas tout à fait exempte de reproches : n'est-on pas maintenant influencé dans la manière d'écrire le programme par les tests que l'on a mis en place ? L'argument des partisans du TDD est généralement que l'écriture de tests étant beaucoup plus facile que l'écriture de programme, le risque est moindre dans ce sens que dans l'autre. Cependant, aucune étude définitive ne permet d'en attester.

8 Automatisation des tests

Par automatisation des tests, nous entendons le fait de pouvoir déclencher de manière simple (par une commande ou un clic) le processus comportant les éléments suivants :

- exécution de tous les tests (même si certains échouent), on parlera de *suite de tests* pour les désigner dans leur ensemble ;
- pour chaque test, comparaison *automatique* du résultat du programme à un résultat de référence ;
- production *automatique* d'un rapport de synthèse indiquant le nombre de tests exécutés avec les nombres de résultats corrects et erronés.

Il est très important que l'ensemble du processus soit automatique : il n'est pas pensable de comparer les résultats effectifs et attendus *à l'œil* ou de devoir lire un rapport de tests de plusieurs pages pour connaître le nombre de tests qui ont échoué.

Le développeur doit être en mesure de lancer et d'analyser les résultats en une fraction de seconde. Seuls les tests peuvent être consommateurs de temps, mais il doit s'agir de temps-machine

et non de temps-développeur.

Même si la charge de l'exécution des tests revient entièrement à la machine, la durée effective peut avoir son importance. C'est pourquoi nous distinguons deux cas :

— la suite de tests pour le développement ;
— la suite de tests pour la non-régression.

8.1 Pour le développement

Il est souhaitable que le développeur puisse exécuter les tests lorsqu'il effectue une modification afin de valider celle-ci avant de passer à une autre tâche. Dans cette optique, la durée d'exécution des tests peut devenir un problème :

— d'organisation : sur quelle tâche le développeur travaille-t-il pendant qu'il attend le résultat des tests ;
— de concentration : s'il doit attendre trop longtemps avant d'avoir le résultat de la confrontation entre ses idées et le résultat des tests, le développeur risque de ne plus avoir en tête les tenants et aboutissants de la portion de programme écrite.

En conséquence, on considère que la durée maximale d'exécution des tests pendant le développement doit être au plus de quelques minutes et idéalement quelques secondes. Si la durée est supérieure, il est nécessaire de mettre en place un système qui permette au développeur de n'exécuter qu'une sous-partie de l'ensemble des tests (la sous-partie qu'il estime avoir pu impacter par ses modifications). Bien entendu, l'exécution d'une sous-partie doit conserver les propriétés d'automatisation évoquées.

8.2 Pour la non-régression

Les tests de non-régression doivent couvrir l'ensemble des fonctionnalités clefs, ainsi que tous les bugs qui ont été corrigés au fil de la vie du logiciel. La suite de tests peut alors devenir considérable et nécessiter un temps d'exécution important. Cependant, ce n'est pas ici un problème, car la suite de tests ne sera pas exécutée après chaque modification du programme.

Typiquement, les tests de non-régression sont exécutés après une modification majeure (*p. ex.* après une *pull request*) et/ou automatiquement chaque nuit. Ils sont en outre, bien entendu, exécutés avant toute livraison du logiciel à ses utilisateurs.

9 Tests en Python

9.1 Tests unitaires par `doctest`

La manière la plus simple d'ajouter des tests à une fonction Python est d'utiliser le mécanisme dit de *doctest* [1] qui permet de les inclure dans la documentation de la fonction. Pour ce faire, il suffit d'ajouter dans le commentaire de la fonction la liste des cas de tests avec les valeurs qui seraient affichées par l'interpréteur Python en cas d'exécution de ces cas de tests.

Par exemple, le code Code 9.1 montre comment tester la fonction *Fibonacci* par le biais de *doctest*. Reste à exécuter les tests insérés au sein de la documentation. Cela se fait typiquement à l'aide des quelques lignes présentées au code Code 9.2. Le module `doctest`, et plus précisément sa fonction `testmod` permet d'exécuter tous les tests du module courant. L'idiome

1. Référence complète à l'adresse http://docs.python.org/library/doctest.html

`__name__ == "__main__"` permet de savoir si le module courant est le module principal. Le module principal est celui dont le nom est passé à l'interpréteur pour exécution et reçoit pour nom `"__main__"` tandis que tous les autres modules reçoivent un nom qui correspond au fichier dans lequel ils sont déclarés.

Code 9.1 – Cas de tests insérés dans la documentation.

```
def fibonacci(n):
  """ Computes the n-th Fibonacci number.
  >>> fibonacci(0)
  1
  >>> fibonacci(1)
  1
  >>> fibonacci(5)
  8
  """
  ...
```

Code 9.2 – Lancement des tests ssi il s'agit du module principal.

```
if __name__ == "__main__":
  import doctest
  doctest.testmod()
```

9.2 Tests par assertion avec `pytest`

Un outil de test populaire pour Python est `pytest`[2]. Lorsque l'on exécute `pytest` dans un répertoire, l'outil va chercher les fichiers nommés `test_xyz.py` (qui contiennent typiquement les tests pour le module défini dans le fichier `xyz.py`) et exécuter les cas de tests qui s'y trouvent. Les cas de tests sont simplement des fonctions dont le nom commence par `test_`.

Dans un cas de test, on utilise simplement l'instruction `assert` pour vérifier si une valeur est telle qu'attendue. Par exemple, le code Code 9.3 vérifie que la fonction `f` du module `m` renvoie la valeur `5` lorsqu'elle reçoit la valeur `3`.

Code 9.3 – Exemple de cas de tests.

```
import m

def test_f() -> None:
  assert m.f(3) == 5
```

Lorsque l'outil est exécuté, il produit un rapport de tests semblable à Code 9.4 lorsqu'il n'y a pas d'erreur ; le rapport contient la liste des fichiers exécutés, et pour chaque fichier le nombre de cas de tests (nombre de points sur la ligne). En cas d'erreur, le rapport est augmenté des descriptions des erreurs rencontrées, comme le montre Code 9.5.

2. https://www.pytest.org/

Code 9.4 – Exemple de rapport de test (succès).

```
========================================= test session starts
platform darwin - Python 3.10.4, pytest-7.1.2, pluggy-1.0.0
collected 4 item

test_m.py .                                                        [100%]
test_n.py .......                                                  [100%]
...

========================================== 8 passed in 0.01s
```

Code 9.5 – Exemple de rapport de test (échec).

```
========================================= test session starts
platform darwin - Python 3.10.4, pytest-7.1.2, pluggy-1.0.0
collected 4 item

test_m.py F                                                        [100%]
test_n.py .......                                                  [100%]
...

================================================ FAILURES
_____________________________________________ test_f

    def test_f():
>     assert m.f(3) == 5
E     assert 4 == 5
E      +  where 4 = <function f at 0x7fec6fdd5240>(3)
E      +    where <function f at 0x7fec6fdd5240> = m.f

test_m.py:4: AssertionError
====================================== short test summary info
FAILED test_m.py::test_f - assert 4 == 5
========================================== 1 failed in 0.10s
```

Il est important de noter que `pytest` *détourne* le fonctionnement normal de l'instruction `assert`. En particulier, l'outil ne s'arrête pas à la première assertion fausse, mais continue son exécution pour trouver toutes les assertions fausses dans les cas de tests. Cela est très utile en pratique, car cela signifie que l'utilisateur obtient un rapport sur l'état de l'ensemble des cas de tests plutôt que seulement sur la première erreur.

9.3 Tests aléatoires par `hypothesis`

Notion de décorateur

Un décorateur est une construction syntaxique qui permet de décorer (ou qualifier) une définition de classe, de fonction ou de méthode. Un décorateur se trouve sur une ligne propre,

précédant immédiatement l'élément à décorer ; il est introduit par le caractère `@` et peut recevoir des paramètres (le plus souvent sous forme nommée). Le code Code 9.6 montre un exemple d'utilisation de décorateur. On dit que la fonction `func` est annotée (ou décorée) par le décorateur `mydecorator` avec les paramètres `p1` et `p2`.

Code 9.6 – Exemple d'utillisation de décorateur.

```
@mydecorator(p1=1, p2="two")
def func(x: ..., y: ...) -> ...:
  ...
```

Un décorateur est utilisé pour modifier le comportement de l'élément annoté selon les paramètres qu'il a reçus. Il existe par exemple des décorateurs pour déclarer une méthode statique, ou pour générer automatiquement un affichage (ou un événement de log) à chaque appel de fonction.

Module `hypothesis`

Nous allons utiliser le décorateur `given` du module `hypothesis`[3]. Ce décorateur permet de générer automatiquement des valeurs d'entrées pour des types prédéfinis. Le code Code 9.7 montre comment l'utiliser pour générer automatiquement des couples ⟨ entier, flottant ⟩ pour une fonction donnée.

Code 9.7 – Exemple d'utillisation de décorateur `hypothesis`.

```
from hypothesis import given, settings
import hypothesis.strategies as strats

@given(x = strats.integers(), y = strats.floats())
@settings(max_examples = 100)
def func(x: int, y: int) -> None:
  ...

func()
```

Dans cet exemple, le décorateur a transformé la fonction `func` qui prenait deux paramètres en une fonction de même nom mais ne prenant aucun paramètre : lorsqu'elle sera appelée, elle se comportera comme si on appelait 100 fois la fonction `func` originale avec deux paramètres (un entier et un flottant) générés aléatoirement.

Il est possible de générer de cette manière des entiers (`integers`), des flottants (`floats`), des chaînes de caractères (`text`), mais également des listes (`lists`), n-uplets (`tuples`), *etc* Il est également possible de borner les éléments générés, par exemple :

- par `min_value` et `max_value` pour `integers` et `floats` ;
- par `min_size` et `max_size` pour `lists`.

Ce mécanisme permet donc de facilement automatiser l'appel à une fonction avec des paramètres aléatoires. À charge à la fonction appelée d'utiliser les appels comme autant de cas de tests. Le code Code 9.8 montre comment définir une fonction `test_f` pour tester la fonction `f`.

3. Documentation disponible à l'adresse https://hypothesis.readthedocs.io

Code 9.8 – Exemple de test par `hypothesis`.

```python
from hypothesis import given, settings
import hypothesis.strategies as strats
import unittest

@given(x = strats.integers(), y = strats.floats())
@settings(max_examples = 100)
def test_f(x: int, y: int) -> None:
  assert f(x, y) == ...
```

10 Couverture de code en Python

L'outil `coverage` [1] permet de vérifier très facilement la couverture d'un programme Python. Cet outil s'exécute en ligne de commande et le test de couverture se fait en deux étapes :

- lancement du programme par `coverage run prog.py` en lieu et place de `python prog.py` (le programme exécuté doit lancer des tests *exerçant* un maximum de fonctionnalités) ;
- consultation de la couverture produite par l'exécution *via* `coverage report` (affichage sur la console) ou `coverage html` (résultats générés au format HTML dans le sous-répertoire `htmlcov`).

D'autres options sont disponibles pour des usages plus avancés (exécuter `coverage help` pour plus d'informations), mais celles présentées ci-dessus permettent déjà de poser un diagnostic sur la qualité d'une suite de tests.

Le rapport produit par `coverage report` est semblable à celui montré par le code Code 10.1, et ne donne donc qu'une métrique par module. En revanche, l'utilisation de `coverage html` produit des fichiers HTML qui permettent de distinguer les lignes exécutées de celles qui ne l'ont jamais été, comme nous l'avons fait à la section 5.

Code 10.1 – Exemple de rapport sommaire de couverture par `coverage`.

```
Name           Stmts   Miss  Cover
-----------
dir/submod        56     24    57%
module            38      0   100%
-----------
TOTAL             94     24    74%
```

Comme on le voit sur le rapport Code 10.1, la couverture est une couverture d'instruction (*cf.* *Stmts* comme abréviation de *Statements*). En ce sens, `coverage` est un outil basique, mais déjà très utile pour s'assurer que l'on n'a pas oublié de tester tout un pan de code.

1. Disponible à l'adresse http://pypi.python.org/pypi/coverage

Annexes

À retenir

D'un point de vue terminologique, il est important de connaître les différentes classifications possibles des tests :

- niveaux de granularité (unitaires, d'intégration, de validation) ;
- caractéristiques testées (fonctionnelles et non-fonctionnelles) ;
- natures (boîtes blanches et noires).

D'un point de vue pratique, il faut connaître quelques techniques de bases (par assertions, par énumérations et par alea) ainsi que les librairies Python associées.

La couverture de code est un indicateur très utile, dont il faut aussi connaître les limites. L'automatisation des tests semble indispensable pour des développements non triviaux.

Références

Voici la liste des ouvrages qui ont servi à bâtir cette séance. Ils couvrent l'ensemble du spectre de la littérature sur les tests : de la théorie à la pratique, en passant par les aspects certification et normalisation.

- "Les tests logiciels - fondamentaux" (Bernard Homès", Hermes Sciences) propose une approche du test du point de vue de la certification et de la normalisation.
- "Le test des logiciels" (Spyros Xanthakis, Pascal Régnier et Constantin Karapoulios, Hermes Sciences) propose une approche académique des différentes facettes du test.
- "Debug It !" (Paul Butcher, The Pragmatic Programmers) propose une approche complète du cycle développement/test/débugage de manière indépendante du langage.
- "Pragmatic Unit Testing with JUnit" (Andrew Hunt et David Thomas, The Pragmatic Programmers) propose des techniques et *check-lists* pour l'écriture de tests en Java, les principes sous-jacents étant valables dans tous les langages.
- "Test-Driven Development By Example" (Kent Beck, Addison-Wesley) propose une approche (*dogmatique*) du TDD illustrée par du code Java, mais qui s'applique à tout langage disposant d'un *framework* de type xUnit.