Sous-programmes

Objectifs

— Savoir écrire un sous-programme
— Comprendre les paramètres en Python
— Savoir écrire des sous-programmes itératifs et récursifs
— Tester et exploiter les outils de couverture de test

1 Comprendre les sous-programmes et leur test

Exercice 1 : Pourquoi et comment tester

Le langage Python s'appuie sur un typage dynamique. Ceci signifie que quand un programme Python est chargé par l'interpréteur, aucune vérification du programme n'est faite à part le respect de la syntaxe Python. Il est donc d'autant plus important de tester son programme et de s'assurer que l'ensemble de ses instructions ont été exécutées au moins une fois. Ceci ne garantira pas que le programme est correct mais permettra de détecter des erreurs !

Considérons la fonction du fichier `minimum.py` qui calcule le minimum de ses deux paramètres. Elle contient manifestement une erreur que nous ne corrigerons pas maintenant.

Le fichier `test_minimum.py` est un programme de test pytest de la fonction minimum. pytest considère les fichiers dont le nom commence par `test_` comme des programmes de test et, dans ces fichiers, les fonction dont le nom commence par `test` sont des fonctions de test. Les tests s'appuient sur l'instruction `assert` qui lève une exception si l'expression booléenne est fausse et ne fait rien sinon.

1.1. *Exécution des tests.* Exécuter le programme de test avec pytest. Par exemple :

```
pytest test_minimum.py
```

Est-ce que le test réussi ?

1.2. *Couverture des instructions.* Un outil de couverture de code permet de recenser les instructions qui ont été exécutées et celles qui ne l'ont pas été. C'est donc un outil qui nous donne des indications pour identifier de nouveaux tests à faire en s'appuyant sur la structure du programme. On parle de test en boîte blanche.

Pour exécuter les tests, on peut faire :

```
coverage run -m py.test test_minimum.py
```

Les résultats sont conservées dans un fichier caché (.coverage sous Linux). Pour les exploiter, on peut faire :

```
coverage html
```

Les résultats sont engendrés au format HTML dans le dossier `htmlcov`. Il suffit alors d'ouvrir le fichier `htmlcov/index.html` avec un navigateur. On peut alors cliquer sur la ligne `minimum.py` pour consulter les résultats concernant ce module.

Consulter les résultats.

1.3. *Couverture des enchaînements.* En plus des instructions, `coverage` peut aussi analyser les enchaînements. Par exemple, pour **`if`** (même sans **`else`**), il faudrait faire un test pour lequel la condition est vraie et l'autre où elle est fausse. Ainsi, on teste les deux enchaînements possibles après l'évaluation de la condition du **`if`**.

Pour demander aussi l'analyse des enchaînements, il faut relancer ajouter l'option `--branch` :

```
coverage run --branch -m py.test test_minimum.py
```

Bien sûr, il faut ré-engendrer les résultats :

```
coverage html
```

Consulter les résultats et constater que la couverture du **`if`** est marquée partielle. Comment interpréter les numéros affichés à droite ?

1.4. *Exploiter les résultats de couverture.* Compléter les tests pour atteindre un taux de couverture de 100% des instructions et branchements. Que se passe-t-il ? Que faire ?

1.5. *Complétude.* Est-ce qu'avoir un test de couverture de 100% des instructions et enchaînements sur des tests qui réussissent est suffisant pour en déduire que le programme est correct ?

2 Quelques fonctions

Exercice 2 : Quelques statistiques

Le point de départ est la fonction `statistiques` fournie (statistiques.py) et son programme de test `test_statistiques.py` que l'on veut compléter. En plus de la moyenne, du min et du max, on souhaite aussi obtenir le nombre d'occurrences du min et le nombre d'occurrences du max.

2.1. Compléter la spécification (et que la spécification) de la fonction `statistiques`.

2.2. Corriger les tests. Les tests doivent donc échouer.

2.3. Corriger le corps (l'implantation) de la fonction `statistiques` et la tester. Compléter les tests si un taux de couverture de 100atteint.

Exercice 3 Écrire une fonction robuste appelée `input_int` qui permet de lire au clavier un entier. Elle prend en paramètre le message à afficher avant de solliciter l'utilisateur. Si aucun message n'est fourni, rien n'est affiché. On peut préciser les bornes dans lequel doit se situer l'entier grâce à deux options min et max. L'entier doit être entre ces deux bornes, bornes comprises.

Voici quelques exemples d'utilisation :

```
1 n = input_int('Un entier quelconque :')
2 a = input_int()
3 mois = input_int('Numéro de mois :', min=1, max=12)
4 positif = input_int('Un entier positif :', min=0)
5 negatif = input_int('Un entier strictement négatif :', max=-1)
6 erreur = input_int('impossible !', min=1, max=0)      # provoque ValueError
7 erreur = input_int('Mois :', 1, 12)                   # provoque TypeError
```

3 Autour de la fonction puissance

L'objectif de ces exercices est de proposer plusieurs implantations de la fonction puissance et de les tester.

Exercice 4 : Puissance entière avec exposant positif
Intéressons nous d'abord à la puissance entière d'un nombre quand l'exposant est positif. Par exemple, si le nombre est 4 et l'exposant est 3, la puissance est 64 (4 * 4 * 4). On utilisera la convention généralement admise que 0^0 vaut 1.
4.1. Dans un module `puissance.py`, définir cette fonction puissance en utilisant une répétition. On l'appellera `puissance_iterative`.
4.2. Dans un fichier `test_puissance.py`, écrire une fonction de test de la puissance.
4.3. Utiliser l'outil `coverage` pour évaluer le taux de couverture des tests de la fonction puissance en terme d'instructions et de branches. Compléter éventuellement les tests pour obtenir 100% de couverture sur ces deux critères.

Exercice 5 : Extension aux exposants négatif
Il s'agit de modifier la fonction précédente pour qu'elle fonctionne quelque que soit l'exposant entier considéré, y compris négatif.
5.1. Indiquer dans quels cas on ne peut pas calculer la puissance entière d'un nombre.
5.2. Modifier le code de la fonction `puissance_iterative` pour prendre en compte les exposants négatifs. On lèvera l'exception `ValueError` si le calcul ne peut pas être réalisé.
5.3. Exécuter le programme de test pour vérifier qu'aucune régression n'a été introduite.
5.4. Vérifier le taux de couverture des tests puis compléter les tests pour atteindre un taux de couverture de 100% en instructions et en branches.

Exercice 6 : Puissance récursive
On se propose maintenant d'écrire une version récursive de la puissance.
6.1. Dans le module `puissance.py`, écrire une version récursive de la fonction puissance que l'on notera `puissance_recursive`.
6.2. Cette nouvelle fonction devrait réussir tous les tests de la précédente. On pourrait copier/-coller le programme de test de la première et remplacer les occurrences de `puissance_iterative` par `puissance_recursive`. Ceci n'est pas satisfaisant car si on corrige un test ou on ajoute un nouveau test dans l'une des deux fonctions de test, il faudra faire la même modification sur l'autre fonction de test.

La solution que nous allons choisir ici est de paramétrer la fonction de test par la fonction puissance à utiliser. On renommera cette fonction en `executer_test_puissance` pour ne pas que pytest la considère comme une fonction de test. On pourra alors écrire deux fonctions de test `test_puissance_iterative` et `test_puissance_recursive` qui appellent la fonction `executer_test_puissance` en lui passant respectivement en paramètre `puissance_iterative` et `puissance_recursive`.

Faire ces modifications et tester les deux fonctions qui calculent la puissance. On complètera les tests pour avoir 100% de couverture pour les instructions et les branches.

Exercice 7 : Amélioration de la puissance
On peut améliorer le calcul de la puissance en remarquant que :

$$x^n = \begin{cases} (x^2)^p & \text{si } n = 2p \\ (x^2)^p \times x & \text{si } n = 2p+1 \end{cases}$$

Ainsi, pour calculer 3^5, on peut faire $3 * 9 * 9$ avec bien sûr $9 = 3^2$.
7.1. *Version récursive.* Écrire et tester une version récursive de la puissance exploitant la remarque ci-dessus. On l'appellere `puissance_mieux_recursive`.
7.2. *Version itérative.* Écrire et tester une version itérative de la puissance exploitant la remarque ci-dessus. On l'appellere `puissance_mieux_iteraive`.
Attention : Dans les cas, on doit avoir un taux de couverture des tests des 100% en instructions et branches.

Exercice 8 : Programme interactif qui calcule la puissance
Écrire un programme Python (`ihm_puissance.py`) qui utilise le module `puissance.py` pour calculer la puissance entière d'un nombre. Le nombre et l'exposant seront saisis au clavier.

On veillera à ce que le calcul se fasse sur des entiers si l'argument fourni sur la ligne de commande est un entier.

Exercice 9 : Programme en ligne de commande qui calcule la puissance
Même question que l'exercice précédent mais le programme récupère les informations de la ligne de commande et s'appelle `cmd_puissance.py`. On utilisera le module `sys` et sa variable `argv`.

4 Vers du fonctionnel

Exercice 10 Dans cet exercice, on souhaite produire une nouvelle liste à partir d'une liste existante. Après avoir écrits quelques cas particuliers, on met en œuvre une solution plus générale.
10.1. Écrire une fonction qui, pour une liste d'entiers donnée, retourne une nouvelle liste qui contient tous les éléments de la première mis au carré.
10.2. Écrire une fonction qui, pour une liste d'entiers donnée, retourne une nouvelle liste qui contient tous les éléments de la première divisés (division entière) par 2.
10.3. Écrire une fonction qui, pour une liste d'entiers donnée, retourne une nouvelle liste qui contient vrai si l'élément correspondant de la première liste est pair, faux sinon.
10.4. On constate que la structure du code des fonctions précédentes est la même. Proposer une fonction qui permettent de généraliser les fonctions précédentes. On pourra l'appeler **`map`**.