TP n° 9 – Annotations, assertions, jeu de tests

1 Introduction

On commence par rappeler qu'en programmation, plusieurs aspects sont en jeux :

— ce que l'on calcule (quoi) ;
— la manière de le calculer (comment) ;
— la raison pour laquelle c'est correct (pourquoi).

Le programme n'est que le « comment », c'est-à-dire la manière dont la fonction remplit son contrat et rien d'autre. On appelle **contrat d'une fonction** la description précise, en français, de ce que fait la fonction. Il nécessite :

— d'être introduit par un commentaire en début de programme qui précise ce qui est calculé, la nature des arguments donnés et des résultats. C'est ce que l'on peut retrouver avec la fonction `help`, par exemple :

```
>>> help(abs)
Help on built-in function abs in module builtins:

abs(x, /)
    Return the absolute value of the argument.
```

— une pré-condition qui doit être vraie à l'appel du programme ;
— une post-condition qui garantit sa correction.

Le contrat indique le « quoi » (ce que fait la fonction), mais pas le « comment ». Le « quoi » correspond à une spécification, décrite dans un cahier des charges. Le « pourquoi » est un ensemble d'expressions mathématiques qui justifie ce qui est calculé, souvent le point de départ d'un algorithme. Il n'a de sens que si la pré-condition est vérifiée. Le « comment » correspond à l'algorithme mis en œuvre pour réaliser le calcul. C'est sa correction partielle (post-condition vérifiée) et sa terminaison qui garantit que le programme est correct.

1.1 Annotations, documentation d'une fonction

L'aide des fonctions python doit être rédigée avec `docstring` (voir la PEP 257 de 2001) sous la forme :

```
def fonction(arg1, arg2, argopt1=val1, ...):
    """ Description de la fonction, en français dans notre contexte,
    possiblement sur plusieurs lignes...

    Arguments :

        arg1 (type) : description
        arg2 (type) : description


    Argument(s) optionnel(s) :

        argopt1 (type) : description

    Résultat(s) :

        (type) : description

    """
```

On peut afficher cette aide, soit en tapant

```
>>> help(fonction)
```

soit

```
>>> print(fonction.__doc__)
```

On rappelle les principaux types :

```
>>> [type(e) for e in [1, 2.2, [1,2.2], (1,2), 2*"cou"]]
[<class 'int'>, <class 'float'>, <class 'list'>, <class 'tuple'>,
<class 'str'>]
```

On peut vérifier le type d'une variable avec :

```
>>> type(2*"cou")==str
True
```

qui est équivalent à

```
>>> isinstance(2*"cou", str)
True
```

Par exemple, pour s'assurer que `x` est un nombre, il faudrait que `isinstance(x, (int,float))` soit vrai.

1.2 Assertions, pré et post-conditions

Le contrat décrit ce que fait la fonction « dans le cas général », mais aussi dans les cas particuliers et donc la façon dont les erreurs sont gérées. Il décrit également les pré et post-conditions.

Une pré-condition est une propriété portant sur les données du problème qui est censée être vraie avant d'appeler la fonction. Son résultat est un booléen. Si la pré-condition n'est pas vérifiée, le comportement de la fonction n'est pas spécifié. Par principe, une pré-condition est censée être vérifiée avant d'appeler la fonction par le code appelant – et non pas dans la fonction elle-même. En pratique, notamment en phase de développement (donc avant de livrer son programme), il peut être utile de s'assurer que la pré-condition est vérifiée à chaque appel. Cela ralentit considérablement le programme mais permet de détecter les erreurs au plus vite. En python, la vérification peut être réalisée avec le mot-clé `assert` suivi d'un tuple constitué d'un test et d'un message d'erreur s'il n'est pas vérifié ; par exemple pour s'assurer que l'argument d'une fonction est un nombre, on écrira :

```
def f(x):
    assert isinstance(x, (int,float)), "x doit être un nombre"
    ...
```

Une post-condition est une propriété portant sur les données et le résultat du problème qui est censée être vraie après l'appel de la fonction. Son résultat est un booléen. Par exemple, si une fonction doit renvoyer un réel positif, on peut tester le résultat `r` avec :

```
def f(x):
    assert isinstance(x, (int,float)), "x doit être un nombre"
    # ...
    # calcul de r
    # ...
    assert (isinstance(r, (int,float)) and r>=0), \
        "le résultat doit être un nombre positif"
    return(r)
```

Le résultat `r` ne sera renvoyé que si c'est un nombre positif.

Exercice 9.1
Définir une fonction `f`, implémentation de la fonction $x \mapsto x^2$, avec son aide rédigée avec le `docstring` ainsi que la pré-condition et la post-consition avec `assert`.

1.3 Jeux de tests, correction

Dans une approche rigoureuse de la programmation, on se donne une **spécification** écrite en langage mathématique et on souhaite une **preuve** formelle que le programme vérifie cette spécification, qu'il soit **correct**.

En pratique, on réalise souvent un nombre de tests fini visant à s'assurer du comportement d'un programme dans différents cas. L'étude de chaque cas est appelé un **test unitaire**. On écrit un test pour confronter une réalisation à sa spécification. Le test définit un critère d'arrêt (état ou sorties à l'issue de l'exécution) et permet de statuer sur le succès ou sur l'échec d'une vérification. Grâce à la spécification, on est en mesure de faire correspondre un état d'entrée donné à un résultat ou à une sortie. Le test permet de vérifier que la relation d'entrée/sortie donnée par la spécification est bel et bien réalisée.

On définit généralement 4 phases dans l'exécution d'un test unitaire :

Initialisation définition d'un environnement de test complètement reproductible, idéalement un nouveau *shell*, c'est-à-dire avec un répertoire de travail `dir()` vierge.

Exercice le module à tester est exécuté.

Vérification Comparaison des résultats obtenus avec un vecteur de résultats défini. Ces tests définissent le résultat du test (succès ou échec).

Désactivation Désinstallation pour retrouver l'état initial du système, dans le but de ne pas polluer les tests suivants. Tous les tests doivent être indépendants et reproductibles unitairement (quand exécutés seuls).

Il existe un certain nombre de modules qui permettent de faire des tests unitaires en python (comme `unittest` ou `pytest`) mais leur utilisation est hors programme. Dans notre contexte, on se limitera à l'utilisation d'instructions `try...except...` : l'instruction `try` permet d'essayer d'appliquer une fonction et la clause `except` permet de gérer les exceptions, c'est-à-dire les erreurs détectées lors de l'exécution. On ne distinguera que les exceptions venant d'assertions que nous avons définies dans la fonction testée. Pour ce faire, on utilisera le mot-clé `AssertionError`. Pour forcer la poursuite de l'exécution du code dans une boucle malgré cette exception, on pourra utiliser le mot-clé `continue`. Dans ce cas, on passera immédiatement à la valeur suivante de la boucle.

Exercice 9.2
Expliquer ce que permettent de faire les deux codes suivants :

```
for x in [1, 2.3, "a", [1,2], 4.5]:
    try:
        y = f(x)
    except AssertionError as msg:
        print(x,msg)
    print(y)
```

```
for x in [1, 2.3, "a", [1,2], 4.5]:
    try:
        y = f(x)
    except AssertionError as msg:
        print(x,msg)
        continue
    print(y)
```

Comparer en particulier les sorties. En déduire l'intérêt de l'instruction `continue`.

Ainsi, afin de poursuivre l'exécution de tests après l'échec d'un test avec une clause `assert`, on veillera à utiliser un structure de la forme :

```
try:
    # evaluation de yt, à comparer à y
    assert yt==y, "résultat faux"
    print("test réussi avec ...")
except:
    print("test NON réussi avec ...")
```

Il reste à fournir pour chaque test un environnement d'évaluation neutre. On sauvegardera donc avant chaque test l'état du répertoire `dir()` pour le remettre à l'identique après exécution. En pratique, on peut constituer une liste `dirinit` avec les noms des éléments initialement présents dans le `dir()` de façon à purger en fin de test tous les éléments du `dir()` sauf ceux de `dirinit` ; on pensera juste à ajouter la chaîne `"dirinit"` pour protéger la liste `dirinit` elle-même de toute suppression après le test.

Finalement, pour tester une fonction `f` avec des arguments dans une liste `X` et des résultats connus dans une liste `Y` (de même longueur que `X`), à des choses du genre :

```
# on sauvegarde le répertoire dir()
dirinit = dir()+['dirinit']
for x,y in zip(X,Y):
    try:
        # on essaie de calculer f(x)
        try:
            yt=f(x)
        except AssertionError as msg:
            print("*** PB :", msg)
            continue # pour passer si l'exception vient de f
        assert yt==y, "résultat faux"
        print("test réussi avec ", x)
    except:
        print("test NON réussi avec ", x)
    finally:
        # on purge le répertoire dir()
        for e in dir():
            if e not in dirinit:
                del(e)
```

On veillera que les test réalisés permettent de partitionner les domaines d'entrée et de tester les limites.

Exercice 9.3 (Implémentation du logarithme népérien)
On souhaite implémenter une fonction `ln` qui permet de calculer le logarithme népérien d'un réel positif x avec une précision ε donnée.

1. Décrire les spécifications mathématiques de la fonction ln. En déduire une aide rédigée avec le `docstring` ainsi que les pré-conditions avec `assert`.
2. En utilisant un développement limité de $\ln(1+x)$ au voisinage de $x = 0$, déterminer une approximation à ε près de $\ln(x)$ dans le cas $x \in [1; 2[$.
3. Sachant que $2\ln(\sqrt{x}) = \ln(x)$, proposer une méthode permettant de calculer, pour tout $x \geqslant 2$, $\ln(x)$ par appels récursifs, en s'appuyant notamment sur l'expression précédente.
4. Sachant que $\ln(x) = -\ln(1/x)$, proposer une implémentation de la fonction ln, pour tout $x > 0$. On veillera à minimiser le nombre de calculs, notamment les puissances de x.
5. Définir un certain nombre de tests unitaires permettant de vérifier le comportement de la fonction. On pourra librement utiliser la fonction `log` de la bibliothèque `math` pour vérifier les calculs.

Exercice 9.4 (Évaluation de polynômes – schéma de Horner)
On s'intéresse au problème d'évaluation de polynômes de la forme

$$P(X) = c_0 + c_1\,X + c_2\,X^2 + \cdots + c_n\,X^n = \sum_{i=0}^{n} c_i\,X^i$$

connaissant ses coefficients $(c_i)_{0\leqslant i\leqslant n}$ en un réel $x \in \mathbb{R}$. Pour avoir moins de multiplications à faire, le mathématicien britannique William George Horner (1786-1837) a eu l'idée de les écrire sous la forme

$$P(X) = c_0 + X\,(c_1 + X\,(\ldots(c_{n-2} + X\,(c_{n-1} + X\,(c_n)))\ldots))$$

Dans ce qui suit, on souhaite définir une fonction `Horner` qui prend comme arguments un réel $x \in \mathbb{R}$ et une séquence (liste ou tuple) de $n+1$ coefficients réels $(c_i)_{0\leqslant i\leqslant n}$.

1. Décrire les spécifications mathématiques de la fonction `Horner`. En déduire une aide rédigée avec le `docstring` ainsi que les pré-conditions avec `assert`.
2. Déterminer un variant et un invariant permettant de définir une version récursive de la fonction `Horner`. Proposer une implémentation.
3. Définir un certain nombre de tests unitaires permettant de vérifier le comportement de la fonction. On pourra librement utiliser la fonction `polyval` de la bibliothèque `numpy.polynomial.polynomial` sous la forme :

   ```
   import numpy.polynomial.polynomial as npp
   npp.polyval(x, C)
   ```

   pour vérifier les calculs.

Exercice 9.5 (Exponentiation rapide, arbre des puissances de Knuth)
On étudie le problème du calcul de x^n, étant donnés $x \in \mathbb{R}$ et $n \geqslant 1$ un entier positif. On pose $y_0 = x$ et on décompose

$$y_k = y_i \, y_j, \qquad 0 \leqslant i, j \leqslant n-1$$

Le but est d'atteindre $y_k = x^n$ le plus vite possible. La méthode naïve que nous venons d'utiliser pour le calcul des polynômes peut être définies par la relation de récurrence

$$\forall k \in [\![1, n-1]\!], \quad y_k = y_0 \times y_{k-1}$$

où $y_{n-1} = x^n$ et qui nécessite donc $n-1$ multiplications. Pour améliorer cet algorithme, on peut exploiter la propriété d'associativité de la loi $\times$, telle que :

$$\forall n \geqslant 2, \quad x^n = \begin{cases} x^{2k} = x^k \times x^k & \text{si } n = 2k \text{ (pair)} \\ x^{2k+1} = x \times x^k \times x^k & \text{sinon.} \end{cases}$$

aussi appelée méthode chinoise. Ainsi, on calcule d'abord x^k puis on en fait le produit, si besoin avec un produit avec x en plus dans le cas impair.

1. Déterminer un variant et un invariant permettant de définir une version récursive du calcul de x^n.
2. Vérifier la correction de votre algorithme et évaluer sa complexité.
3. Définir une fonction `puissance` qui prend comme arguments un réel $x \in \mathbb{R}$ et un entier naturel positif n et qui renvoie x^n, incluant son aide rédigée avec le `docstring` ainsi que les pré-conditions avec `assert`.
4. Définir un certain nombre de tests unitaires permettant de vérifier le comportement de la fonction. On pourra librement utiliser la fonction `pow` de la bibliothèque standard sous la forme :

   ```
   puissance(x,n)==pow(x,n)
   ```

   pour vérifier les calculs.

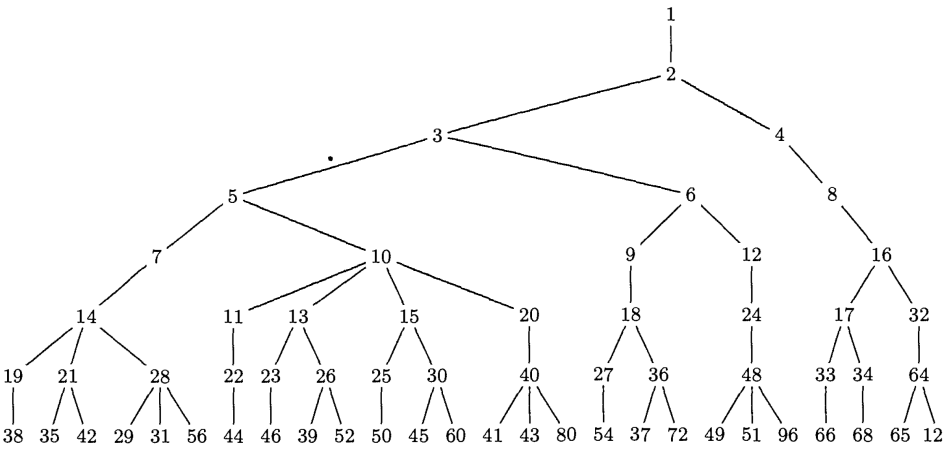
Une autre méthode est d'utiliser une décomposition en facteurs premiers de $n = pq$, $p \leqslant q$ de sorte que

$$x^n = (x^p)^q$$

Si n est premier, on calcule x^{n-1} que l'on multiplie par x.

5. Définir une fonction `factorisation` qui prend comme argument un entier positif n et qui renvoie une liste avec son plus petit facteur premier p et son quotient $q = n/p$ si n n'est pas premier, avec n sinon.
6. Définir une fonction `puissance_facteur` qui prend comme arguments un réel $x \in \mathbb{R}$ et un entier naturel positif n et qui renvoie x^n avec la méthode des facteurs. Tester votre implémentation.

Une autre façon de calculer x^n est d'exploiter l'arbre des puissances de Knuth



D.E. Knuth, *The Art of Computer Programming : Semi-Numerical Algorithms (vol 2)*, page 464.

Par exemple, pour calculer x^{29}, on calcule successivement x^2, x^3, x^5, x^7, x^{14}, x^{28} et x^{29}.

Supposant connus k niveaux, pour construire le niveau $(k+1)$ de cet arbre, il s'agit de calculer, de gauche à droite :

$$n+1,\ n+a_1,\ n+a_2, \dots n+a_{k-1}$$

où les $a_1, a_2, \dots, a_{k-1}$ est le chemin depuis la racine de l'arbre (1) jusqu'à n. On n'ajoutera pas un nœud si celui-ci est déjà présent dans l'arbre. Pour stocker les informations de l'arbre, on pourra utiliser un dictionnaire de la forme :

```
N = {1:None,2:1,3:2,4:2}
```

avec

```
>>> N.keys()
dict_keys([1, 2, 3, 4])
>>> N.values()
dict_values([None, 1, 2, 2])
>>> N.get(2)
1
>>>N.get(12)==None
True
```

où `N.get(12)` ne renvoie rien puisque 12 n'est pas une clé.

7. Définir une fonction `chemin` qui prend comme arguments un dictionnaire `N` et une clé `i` et qui renvoie la liste des clés parcourues en partant de la racine 1 pour aller à la feuille i (inclue).
8. Définir une fonction `niveau` qui prend comme arguments un dictionnaire `N` et une clé `i` et qui renvoie l'entier correspondant au niveau de la feuille i dans l'arbre.
9. Définir une fonction `knuth` qui prend comme argument un nombre de niveaux `z` et de façon optionnelle un dictionnaire `N` correspondant à l'arbre déjà construit et qui renvoi le dictionnaire de l'arbre de Knuth avec z niveaux.
10. Définir une fonction `knuth_exp` qui prend comme argument un entier positif `n` et renvoie la liste des puissances à calculer ; par exemple :

    ```
    >>> knuth_exp(29)
    [1, 2, 3, 5, 7, 14, 28, 29]
    ```

11. Définition une fonction `combinaison` qui prend comme argument une liste `L`, sous-liste d'une liste de puissances de Knuth, et qui renvoie une combinaison possible ; par exemple :

    ```
    >>> combinaisons([1, 2, 3, 5], 7)
    (5, 2)
    ```

12. Définir une fonction `puissance_knuth` qui prend comme arguments un réel $x \in \mathbb{R}$ et un entier naturel positif n et qui renvoie x^n avec la méthode des facteurs. Tester votre implémentation.

* *
*