Sous-programmes

Objectifs

— Savoir écrire un sous-programme
— Comprendre les paramètres en Python
— Savoir écrire des sous-programmes itératifs et récursifs

1 Vocabulaire

Exercice 1 : Vocabulaire
On considère la fonction suivante.

```
1  def a(b, c):
2      '''
3      Bla bla bla...
4      '''
5      d = b + c
6      return d
7
8  e = 7
9  a(1, e)
```

Indiquer (on donnera les numéros de ligne et/ou les éléments concernés) ce qui est :

1. signature d'une fonction
2. spécification (ou interface) d'une fonction
3. implantation (ou corps) d'une fonction
4. paramètre formel
5. paramètre effectif
6. variable locale
7. variable globale

2 Une première fonction

Exercice 2 : Permuter deux éléments d'une liste
On souhaite disposer d'un sous-programme qui permute deux éléments d'une liste, ces deux

éléments étant identifiés par leurs indices. On utilisera les fichiers suivants : `permuter.py` et `test_permuter.py`.
2.1. Donner deux exemples d'utilisation de ce sous-programmes en indiquant les données manipulées par ce sous-programme.
2.2. Écrire la spécification (l'interface) de ce sous-programme.
2.3. Écrire un programme de test (avec `pytest`) de ce sous-programme. Exécuter le programme de test qui devrait révéler des erreurs puisque le sous-programme précédent n'a pas été implanté.
2.4. Écrire l'implantation (le corps) de ce sous-programme.

3 Spécification de sous-programmes

Exercice 3 : Spécifier des sous-programmes
Pour chacun des énoncés suivants, donner la spécification du sous-programme correspondant. On écrira les spécifications dans le fichier `signatures.py`.

1. Calculer la puissance entière d'un réel
2. Connaître le nombre de jours d'un mois.
3. Saisir un entier au clavier.
4. Obtenir le quotient et le reste d'une division entière
5. Savoir si une année est bissextile
6. Saisir un entier compris entre une borne inférieure et une borne supérieure. Avant chaque demande à l'utilisateur, une consigne lui est affichée pour lui expliquer ce qui est attendu.
7. Classer une liste d'élèves dans l'ordre décroissant de leur moyenne.

4 Signature d'un sous-programme

Exercice 4 Définissons la signature de différentes fonctions.
4.1. Donner une signature possible pour la fonction `f` sachant que les appels suivants sont valides :

```
f(1)
f(2, 3)
f(m = 5, c = 6)
f(7, m = 8)
```

et que les appels suivants sont refusés :

```
f()
f(9, 10, 11)
```

On complètera le fichier `test_f.py`.
4.2. Donner la signature et le code d'une fonction `produit` qui permet de faire le produit d'un nombre quelconque de paramètres. Voici quelques exemples d'utilisation :

```
assert produit(5) == 5
assert produit(2, 5) == 10
assert produit() == 1
assert produit(1, 2, 3, 4, 5, 6) == 720
```

On complètera le fichier `produit.py` et on le testera avec `test_produit.py`.

4.3. Définir une signature pour la fonction g sachant que les appels suivants sont possibles :

```
g(a=1, b=1)
g(2, a=2)
g(a=3)
```

et que les appels suivants sont interdits :

```
g(10, 10)
g()
```

On complètera le fichier `test_g.py`.

4.4. Définir la signature de `sp` pour que les tests de `test_sp_signature.py` passent.

5 Comprendre la récursivité

Exercice 5 : Comprendre la récursivité

Une fonction récursive est une fonction dont l'implantation contient un appel à elle-même. Un exemple classique est la factorielle définie en mathématiques de la manière suivante :

$$n! = \begin{cases} 1 & \text{si } n = 0 \text{ ou } n = 1 \\ n \times (n-1)! & \text{sinon} \end{cases}$$

On peut en déduire le code suivant.

```
def fact(n):
    ''' Factorielle d'un entier n positif...'''
    if n <= 1:
        return 1
    else:
        return n * fact(n - 1)

if __name__ == "__main__":
    print('4! =', fact(4))
```

5.1. Indiquer les différents appels qui ont lieu quand on demande à calculer `fact(4)`.

5.2. Exécuter le programme précédent sous Python tutor :

`http://www.pythontutor.com/visualize.html#mode=edit`.

5.3. Est-ce que Python peut calculer `fact(1000)` ?

Pour exécuter ce programme, taper, depuis le dossier qui contient le fichier `fact.py` :

```
python fact.py
```

5.4. Rappeler ce qu'il est conseillé de faire lorsque l'on définit une fonction récursive, en particulier, pour garantir sa terminaison ?

6 Itératif et récursif : la fonction puissance

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

Exercice 6 : 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.
6.1. Écrire le code de la fonction `puissance_positive_iterative` dans le module `puissance.py` qui calcule la puissance entière d'un nombre avec comme précondition que l'exposant est positif. On écrira le code de **manière itérative**, donc en utilisant une répétition.
6.2. Tester avec le fichier `test_puissance_positive_iterative.py`.

Exercice 7 : Exposants négatifs
Prennons maintenant en compte le cas général où l'exposant peut être négatif.
7.1. Caractériser le domaine de définition de la fonction puissance.
7.2. Écrire le code de la fonction `puissance_iterative`. On précisera ses préconditions et on les instrumentera en utilisant `assert`. On utilisera la fonction précédente (exercice 6).
7.3. Tester avec le programme de test `test_puissance_iterative.py`.

Exercice 8 : Puissance récursive
On se propose maintenant d'écrire une version récursive de la puissance.
8.1. Écrire le code de la fonction `puissance_recursive`.
8.2. La tester en utilisant le programme `test_puissance_recursive.py`.

Exercice 9 : 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 $9 = 3^2$.
9.1. *Version récursive.* Écrire une version récursive (`puissance_recursive_mieux` dans `puissance.py`) et la tester (`test_puissance_recursive_mieux.py`).
9.2. *Version itérative.* Écrire une version itérative de la puissance (`puissance_positive_iterative_mieux` dans `puissance.py`) exploitant la remarque ci-dessus et la tester (`test_puissance_iterative_mieux.py`).

7 Récursivité

Exercice 10 : Les tours de Hanoï
C'est Lucas, mathématicien français, qui sous le pseudonyme de Claus a inventé ce jeu.

Il se présente sous la forme d'un support en bois sur lequel sont plantées trois tiges A, B et C. Sur ces trois tiges peuvent être enfilés des disques de diamètres différents (8 dans la version

originale, mais N de manière générale). Dans la configuration initiale (figure 1), les disques sont empilés par ordre de taille décroissante sur la tige A.

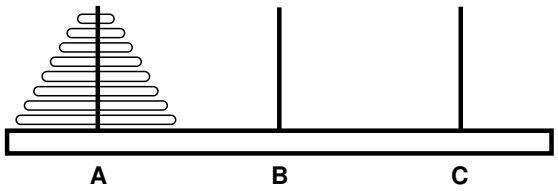

FIGURE 1 – Configuration initiale du jeu des tours de Hanoï

Le but est de déplacer tous les disques de la tige A vers la tige C sachant qu'on ne peut déplacer qu'un seul disque à la fois et qu'on ne peut pas le poser sur un disque plus petit que lui.

Ainsi, dans la configuration initiale, les deux seuls déplacements possibles sont `A -> B` et `A -> C`. On remarquera qu'un déplacement est complètement déterminé par la donnée de la tige de départ et de la tige d'arrivée, car on ne peut déplacer qu'un disque à la fois : celui qui se trouve en haut de la tige d'origine et qui sera placé au sommet de la tige destination.

Écrire un programme (fichier `hanoi.py`) qui donne la solution de ce jeu (les déplacements à effectuer). On commencera par appliquer un raisonnement par récurrence sur le nombre de disques N à déplacer :

1. Donner la solution pour $N = 0$?
2. Donner la solution pour $N = 1$?
3. Donner la solution pour $N = 2$?
4. Supposons que l'on sait résoudre un problème de Hanoï pour $N - 1$ disques ($N > 1$). Montrer que l'on sait résoudre un problème de Hanoï de taille N.

On en déduira alors :

1. la spécification du sous-programme qui modélise le problème des tours de Hanoï,
2. l'implantation de ce sous-programme.

8 Vers du fonctionnel

Exercice 11 : Vers une approche fonctionnelle : map

Dans cet exercice, on souhaite produire une nouvelle liste à partir d'une liste existante. Après avoir expérimenté sur quelques cas particuliers, nous mettons en œuvre une solution plus générale.

On utilisera les fichiers `map.py` et `test_map.py`.

11.1. 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. Appliquée sur la liste `[5, 2, 3, 0, 2]`, elle retournera donc `[25, 4, 9, 0, 4]`.

11.2. 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. Appliquée sur la liste `[5, 2, 3, 0, 2]`, elle retournera donc `[2, 1, 1, 0, 1]`|.

11.3. 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. Appliquée sur la liste `[5, 2, 3, 0, 2]`, elle retournera donc `[False, True, False, True, True]`.

11.4. On constate que la structure du code des fonctions précédentes est la même. Proposer une fonction, nommée **`map`**, qui permet de généraliser les fonctions précédentes et d'autres qui seraient sur le même modèle comme par exemple produire la liste des factorielles des nombres d'une liste. Le principe est de représenter par un ou plusieurs paramètres les parties variables des premières fonctions écrites.