Séquences, sous-programmes, récusivité

Objectifs

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

1 Les séquences

Exercice 1 : Le type list

Indiquer l'effet des instructions du programme suivant :

```
1  s = []
2  s.append(2)              # s ?
3  s.insert(0, 4)           # s ?
4  s.insert(20, 7)          # s ?
5  s[1] = 'd'               # s ?
6  s[2] /= s[2]             # s ?
7  s.count(1)               # ?
8  s[0], s[1] = s[1], s[0] # s ?
9
10 p, _, d = s              # p ? _ ? d ?
11 premier, *suite = s      # premier ? suite ?
12
13 b = [False, True]
14 s.extend(b)              # s ?
15
16 x = s.pop(1)             # x ? s ?
17
18 s2 = [2, 3, 5]
19 i, s2[i], x = s2         # i ? s2 ? x ?
20
21 s.append(s2)             # s ?
22 s2.append(s)             # s2 ? s ?
23
24 t = tuple(b)             # t ?
25 s = list('Fin.')         # s ? s2 ?
```

2 Signature d'un sous-programme

Exercice 2 Définissons la signature de différentes fonctions.
2.1. 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`.
2.2. Donner la signature et le code d'une fonction `produit` qui permet de faire la somme 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`.
2.3. 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`.

3 Comprendre la récursivité

Exercice 3 : 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.

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

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

3.1. Indiquer les différents appels qui ont lieu quand on demande à calculer `fact(4)`.
3.2. Exécuter le programme précédent sous Python tutor :

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

3.3. 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
```

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

4 Itératif et récursif : la fonction puissance

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

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. 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.
4.2. Tester avec le fichier `test_puissance_positive_iterative.py`.

Exercice 5 : Exposants négatifs
Prennons maintenant en compte le cas général où l'exposant peut être négatif.
5.1. Caractériser le domaine de définition de la fonction puissance.
5.2. Écrire le code de la fonction `puissance_iterative`. Cette fonction lèvera l'exception `ValueError` si le calcul ne peut pas être réalisé. On pourra utiliser la fonction précédente (exercice 4).
5.3. Tester avec le programme de test `test_puissance_iterative.py`.

Exercice 6 : Puissance récursive
On se propose maintenant d'écrire une version récursive de la puissance.
6.1. Écrire le code de la fonction `puissance_recursive`.

6.2. La tester en utilisant le programme `test_puissance_recursive.py`.

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 une version récursive (`puissance_recursive_mieux` dans `puissance.py`) et la tester (`test_puissance_recursive_mieux.py`).
7.2. *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`).

5 Encore de la récursivité

Exercice 8 : 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 plantés 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.

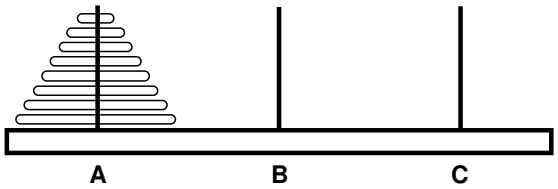

FIGURE 1 – Configuration initiale du jeu des tours de Hanoï

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 $N = 1$?
2. Donner la solution pour $N = 2$?

3. 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.

Exercice 9 : Tri fusion

Le tri fusion est un algorithme de tri efficace (complexité en $n\log(n)$) Il s'appuie sur le principe *diviser pour régner* qui consiste à 1) diviser : découper le problème initial en sous-problèmes, 2) régner : résoudre les sous-problèmes (généralement récusivement) et 3) combiner : calculer une solution au problème initial à partir des solutions des sous-problèmes. Ce tri est stable : si deux éléments sont égaux, le tri préserve leurs position relatives.

Le principe du tri fusion est le suivant. On découpe en deux tableaux de taille à peu près égales (par exemple la première moitié et la deuxième moitié du tableau). On trie chacun des deux tableaux. On produit alors le tableau résultat par fusion (d'où le nom du tri) des deux sous-tableaux. On remarque que cette fusion se fait en temps linéaire car il suffit de considérer à chaque fois le premier élément de chaque sous-tableau. Le plus petit des deux est supprimé du sous-tableau et ajouté au tableau fusionné.

Bien sûr un tableau de taille inférieure ou égale à 1 est déjà trié.

Le tri fusion crée un nouveau tableau qui contient les mêmes éléments que le tableau initial mais dans l'ordre croissant.

9.1. Dérouler le tri par fusion sur le tableau [7, 1, 3, 7, 4]

9.2. Écrire et tester l'algorithme du tri fusion. On complètera le fichier `tri_fusion.py` que l'on testera avec le fichier `test_tri_fusion.py`.

6 Pour aller plus loin...

Exercice 10 : IHM pour calculer la puissance

On souhaite proposer une IHM à l'utilisateur pour calculer la puissance entière d'un réel.

On pourra utiliser la méthode `isdecimal()` de **`str`** pour savoir si tous les caractères d'une chaîne sont bien des chiffres. Dans l'affirmative, on peut la transformer en entier grâce à la fonction prédéfinie `int()`.

10.1. Écrire un programme Python (`puissance_ihm.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'utilisateur a fourni un entier pour le nombre. On pourra tester avec 5^{442} et 5.0^{442}.

10.2. Écrire un programme (`puissance_menu.py`) qui propose le menu suivant à l'utilisateur.

```
----------------------------
1. Choisir un nombre x
2. Choisir un exposant n
```

```
3. Calculer la puissance x^n
0. Quitter
-----------------------------
Votre choix : _
```

Bien sûr, on doit avoir choisi un nombre et un exposant avant de pouvoir demander le calcul de la puissance. Il faut aussi que le calcul soit possible.

10.3. Écrire un programme (`puissance_cmd.py`) pour exploiter les arguments de la ligne de commande. Par exemple pour calculer 3^2, l'utilisateur pourra faire :

```
python puissance_cmd.py 3 2
```

Le principe est le suivant. Si l'utilisateur a fourni des arguments sur la lignes de commande, on vérifie qu'il y en a bien 2. Le premier correspond au nombre, le second à la puissance. On peut alors faire automatiquement le calcul.

On utilisera le module `sys` et sa variable `argv` qui donne accès aux arguments de la ligne de commande sous la forme d'une liste de chaînes de caractères. Attention, il y a un premier argument qui correspond au nom du programme exécuté.