Chapitre 2 : Récursivité

La récursivité est une méthode de programmation essentielle dans le paradigme fonctionnel, utilisé de plus en plus dans les langages modernes (Kotlin par exemple). Elle permet de programmer de manière "naturelle" la réponse à un problème en évitant la notion de variables mutables.

I. Introduction aux piles d'exécution

1) Notion de piles

❐ — Exercice 1 —

Vous trouverez sur bouillotvincent.github.io le programme ci-dessous contenant deux fonctions.

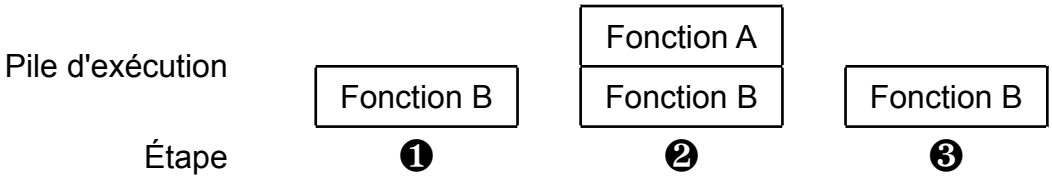
```
def fonctionA():
    print("Début fonctionA")
    for i in range(5): print(f"fonctionA {i}")
    print ("Fin fonctionA")

def fonctionB():
    print("Début fonctionB")
    i=0
    while i<5:
        if i==3:
            fonctionA()
            print("Retour à la fonctionB")
        print(f"fonctionB {i}")
        i = i + 1
    print ("Fin fonctionB")

fonctionB()
```

- Analysez le programme en prédisant son fonctionnement.
- Copiez le programme sur pythontutor.com et testez-le étape par étape : que fait la fonction B quand la fonction A est en train d'être exécutée ?

Explications:



Il existe plusieurs modules Python permettant d'avoir accès à la pile d'exécution. La bibliothèque **`inspect`** est particulièrement utile : l'instruction **`inspect.stack()`** renvoie une liste contenant la pile d'exécution complète.

❐ — Exercice **2** —

Dans cet exercice, j'ai créé un fichier appelé draw_stack.py qui contient une classe **`dessine_pile`** permettant de représenter l'évolution de la pile. Nous utiliserons ce fichier comme un module en l'important grâce à l'instruction : **`from draw_stack import ….`**

Nous allons utiliser le modules **`draw_stack`** et le module **`inspect`** pour comprendre exactement le fonctionnement de la pile d'exécution.

- Ouvrir draw_stack.py et étudier la docstring afin de savoir comme utiliser cette bibliothèque.
- En introduisant les lignes adéquates dans le code de *l'exercice 1*, représentez précisément l'évolution de la pile d'exécution pour les fonctions A et B.
- À votre avis, que représente la ligne commençant par 0x ?

2) Notion de récursivité

Dans la section précédente, une fonction B appelait une fonction A. Une fonction peut bien sûr s'appeler elle-même : dans ce cas, on parle de fonction récursive.

Définition :
Une fonction récursive est une fonction qui **s'appelle elle-même** dans sa définition.

❐ — Exercice **3** —

- Que va faire le programme suivant ?

```
def fct():
    print(f"Je m'appelle moi-même !")
    fct()
fct()
```

- Lorsque vous lancez ce programme, que constatez-vous ?
- Expliquez la réponse à votre question précédente en utilisant une représentation en terme de piles comme dans la section précédente.

Explications :

II. Utilisation des algorithmes récursifs

1) Initiation à la récursivité : premier problème

Pour définir la puissance n-ième d'un nombre, on a l'habitude d'écrire la formule :

$$x^n = x \times x \times \ldots \times x \times x$$

Cette formule semble simple et facile à lire : x est simplement répétée n fois.

Exercice :

1) Écrire une fonction Python qui renvoie la valeur de x^n sans recourir à la récursivité et **sans utiliser l'opérateur puissance ****. x et n seront les arguments de la fonction.
2) Cette fonction est-elle directement liée à la formule ci-dessus ?

La fonction que vous avez écrite demande l'utilisation d'une variable intermédiaire, ce qui est étrange compte tenu du fait que la formule initiale n'en comporte pas.

Une autre manière d'aborder cette question est de réécrire notre problème mathématique sous la formule d'une suite :

$$\text{puissance}(x, n) = \begin{cases} 1 & \text{si } n = 0 \\ x \times \text{puissance}(x, n-1) & \text{si } n > 0 \end{cases}$$

Remarque : On a ici **défini** la fonction puissance à partir de d'elle-même et d'une condition initiale.

Exercice :

Démontrez que la suite ci-dessus revient bien à la définition initiale de x^n. Un raisonnement par récurrence est demandée.

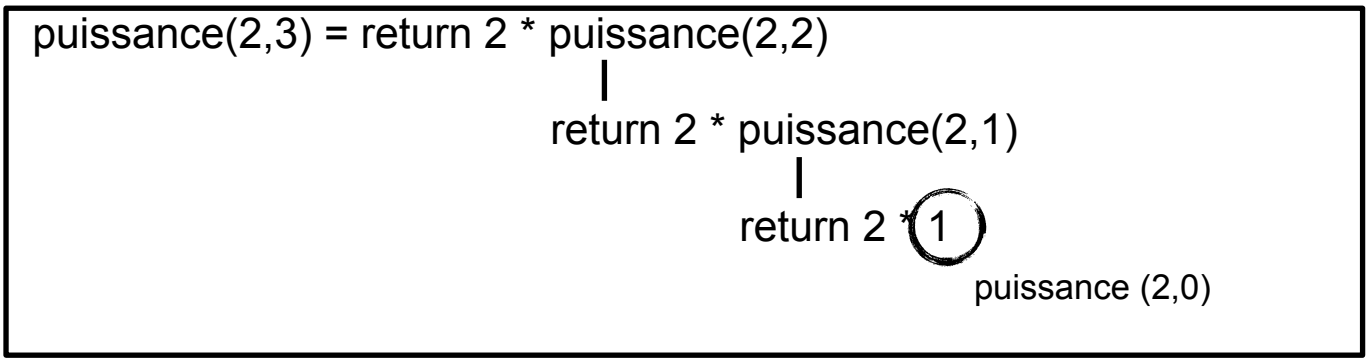
Propriété :

Une structure informatique récursive est toujours mathématiquement associée à la notion de définition par récurrence.

Exercice 4

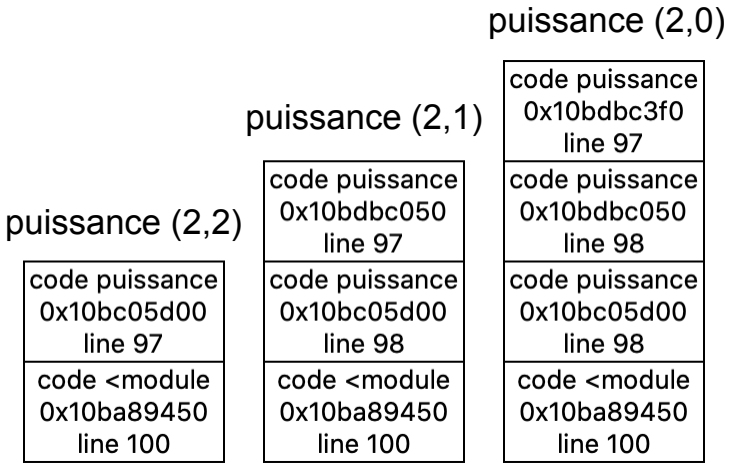
- Programmer la fonction puissance de manière récursive à partir de la définition ci-dessus. Les deux arguments x et n seront conservés.
- Tester votre fonction pour x=2 et n=3 puis n=10. On observera ce qu'il se passe en copiant/collant notre fonction sur PythonTutor.com

La fonction puissance, représentée dans **un arbre d'appels**, fonctionne comme suit :



En terme de pile d'exécution (voir schéma ci-contre), l'appel à puissance(2,0) va être au sommet de la pile, suivi par puissance(2,1) suivi par puissance(2,2) etc.

Remarque : dans l'exemple précédent, le dépilage n'est pas accessible car return est une instruction bloquante.



❐ — Exercice **5** —

Nous allons étudier le calcul de la factorielle grâce à une fonction récursive.

D'après Wikipédia : "En mathématiques, la factorielle d'un entier naturel n est le produit des nombres entiers strictement positifs inférieurs ou égaux à n".

Par exemple : la factorielle de 3 est : 3 x 2 x 1 = 6 ; la factorielle de 4 est 4 x 3 x 2 x 1 = 24 ; la factorielle de 5 est 5 x 4 x 3 x 2 x 1 = 120 ...

Si on note la factorielle de n par n!, on a :

- 0! = 1 (par définition)
- Pour tout entier n > 0, n! = n x (n – 1)!

Vous allez utiliser cette définition de la factorielle pour définir une fonction récursive appelée fact. Cette fonction prendra un nombre n en argument en renverra la factorielle de ce nombre. Tester votre fonction pour n=5 puis n=10. Observez le résultat sur PythonTutor.com.

2) Double récursion

En mathématiques, une suite définie par récurrence est une suite définie par son premier terme et par une relation de récurrence, qui définit chaque terme à partir du précédent ou **des précédents** lorsqu'ils existent. Dans le cadre d'une double récursion, on s'intéresse aux deux termes précédents.

Prenons l'exemple de la suite de Fibonacci qui est définie par :

$u_0 = 0,\ u_1 = 1$

et par la relation de récurrence suivante avec n entier et n > 1 : $u_n = u_{n-1} + u_{n-2}$

Ce qui nous donne pour les 6 premiers termes de la suite de Fibonacci :

$u_0 = 0$

$u_1 = 1$

$u_2 = u_1 + u_0 = 1 + 0 = 1$

$u_3 = u_2 + u_1 = 1 + 1 = 2$

$u_4 = u_3 + u_2 = 2 + 1 = 3$

$u_5 = u_4 + u_3 = 3 + 2 = 5$

❐ — Exercice **6**—

En vous aidant de ce qui a été fait pour la fonction fact, écrivez une fonction récursive **fib** qui donnera le n-ième terme de la suite de Fibonacci.

Cette fonction prendra en argument l'entier n.

3) Erreurs et bonnes pratiques

Règles importantes :
Lorsque l'on définit une structure récursive, il faut s'assurer :

- que la récursion va se terminer en prenant garde à définir un (ou plusieurs) cas terminal.
- que toutes les valeurs utilisées par une fonction soient dans le domaine de définition de la fonction
- qu'il y ait bien une définition pour toutes les valeurs du domaine.

Exemple :
Expliquez pourquoi les fonctions récursives ci-dessous sont incorrectes :

- $f(n) = \begin{cases} 1 \text{ si } n = 0 \\ n + f(n+1) \text{ si } n > 0 \end{cases}$
- $g(n) = \begin{cases} 1 \text{ si } n = 0 \\ n + f(n-3) \text{ si } n > 0 \end{cases}$ avec n entier naturel
- $h(n) = \begin{cases} 1 \text{ si } n = 1 \\ n + h(n-1) \text{ si } n > 1 \end{cases}$

Principe :
Une fois que l'on a suivi les trois règles ci-dessus, lorsque l'on écrit une fonction récursive, il faut toujours supposer que les appels récursifs produisent les bons résultats, sans chercher à construire de tête l'arbre des appels.
Il faut également éviter de penser en mutabilité de variables : on travaille sur les **fonctions.**

4) Optimisation de fonctions récursives

Il est possible d'améliorer de manière très importante les performances d'une fonction récursive en sauvegardant les résultats des appels précédents (programmation dynamique — voir chapitre ???) ou en définissant notre fonction récursive de manière plus astucieuse.

Exemple : fonction puissance

1) Justifiez la définition ci-dessous :

$$\text{puissance}(x,n) = \begin{cases} ???? & \text{si } ???? \\ ???? & \text{si } ???? \\ \text{puissance}(x, n/2) & \text{si } n > 1 \text{ et n est pair} \\ x \times \text{puissance}(x, (n-1)/2) & \text{si } n > 1 \text{ et n est impair} \end{cases}$$

2) Quelles sont les conditions initiales qui doivent être spécifiées ?
3) Programmer cette fonction puissanceRapide de manière récursive en Python. L'argument sera deux entiers x et n.

4) Combien d'appels récursifs sont effectués pour calculer 7^{28} ? Dans la version naïve, combien d'appels récursifs étaient effectués ? Expliquez.