I Fonctions récursives

1. Notion de récursivité

Nous allons nous intéresser à un exemple simple : créer une fonction prenant en argument un entier n et qui affiche la valeur $n!$ de factorielle(n). Il est bien sûr très facile de programmer cette fonction avec une boucle (programmation itérative) :

```
def fact_iter(n):
    s = 1
    for i in range(2,n+1):
        s = s*i
    return s
```

On peut remarquer que cette fonction `fact_iter` renvoie 1 si n est un entier $\leqslant 0$ car la boucle `for` n'est alors pas exécutée.
Cette fonction étant construite, on peut calculer $n!$ d'une autre façon en utilisant la relation de récurrence $n! = n(n-1)!$:

```
def fact_iter_2(n):
    return n*fact_iter(n-1)
```

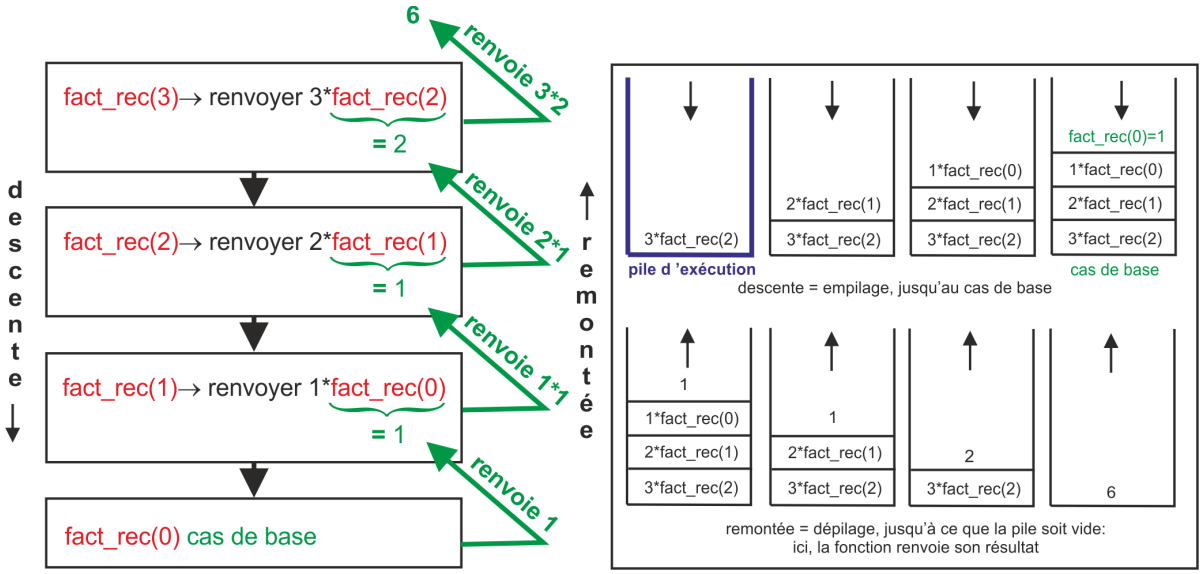
Cette deuxième fonction produit le même effet que la première mais n'apporte rien par rapport à la précédente.
Pour remplacer la programmation itérative, on peut utiliser un mode de programmation, dit récursif, dans lequel la fonction s'appelle elle-même :

```
def fact_rec(n):
    if n==0:
        return 1
    else:
        return n*fact_rec(n-1)
```

Définition : Une **fonction récursive** est une fonction qui s'appelle elle-même, au moins une fois, au cours de son exécution. Pour assurer la terminaison d'une telle fonction, elle doit comporter au moins un **cas de base**.

2. Récursivité et pile d'exécution

On s'intéresse ici à ce qui se passe dans l'ordinateur lors de l'appel d'une fonction récursive ; on prend l'exemple de `fact_rec(n)` qui calcule $n!$ pour illustrer ce qui se passe. Si on cherche à calculer `fact_rec(3)`, ie $3! = 6$, on utilise une **pile d'exécution** dont les modifications seront les suivantes :



Pour évaluer `fact_rec(n)`, dans le cas où $n \geqslant 1$, on doit renvoyer $n \times$ `fact_rec(n-1)` mais cette valeur ne peut pas être renvoyée tant que la valeur de `fact_rec(n-1)` n'est pas connue. L'ordinateur va donc différer le calcul en le gardant en mémoire dans la pile d'exécution, et va exécuter `fact_rec(n-1)` : on place $(n-1) \times$ `fact_rec(n-2)` dans la pile d'exécution en attendant d'évaluer `fact_rec(n-2)`,... La pile d'exécution va ainsi augmenter au fur et à mesure du calcul (on parle de descente du schéma) jusqu'à tomber sur le cas de base $n = 0$, où 1 est renvoyé. Cette valeur est « dépilée » et on peut exécuter le calcul du dessous puis le dépiler. On recommence jusqu'à ce que la pile soit vide ; c'est ce que l'on appelle la remontée.
Il est à noter qu'à chaque appel récursif, n prend une valeur différente (c'est une variable locale).
Le cas de base est bien sûr essentiel. Sans lui l'algorithme ne termine pas.

Ce mode de programmation, très naturel, a donc comme inconvénient d'utiliser beaucoup de place en mémoire (forte complexité spatiale). En comparaison, le code `fact_iter` utilise nettement moins de place en mémoire.
Il faut donc faire attention lors de la programmation d'une fonction récursive de ne pas faire exploser la pile d'exécution : sous PYTHON, la taille de cette pile est limitée à 1000 (la taille de la pile d'exécution et son remplissage dépend en fait du langage de programmation, mais peut être augmentée, avec modération). Par exemple l'appel de `fact_rec(988)` nécessite de créer une pile de plus grande taille que 1000 et retourne le message d'erreur « *RecursionError : maximum recursion depth exceeded in comparison* » ; de plus ce message n'apparaît que quand la pile devient trop grande donc après avoir déjà empilé les 1000 premiers éléments (ce qui peut être long dans des cas plus complexes).

3. Spécificités des fonctions récursives

— Contrairement à ce que l'on recommande de faire dans le cas des fonctions itératives, c'est-à-dire n'utiliser un `return` qu'à la fin de la fonction, il est souvent indispensable de placer des `return` en fin des blocs d'instructions pour une fonction récursive. Par exemple, le code suivant est incorrect :

```
def fact_rec(n):
    if n==0:
        return 1
    else:
        print(n)
        n*fact_rec(n-1)
```

comme on peut le constater :

```
>>> fact_rec(3)
3
2
1
Traceback (most recent call last):
  in fact_rec
    n*fact_rec(n-1)
TypeError: unsupported operand type(s) for *: 'int' and 'NoneType'
```

Il n'y a pas de problème lors de la descente comme le prouve la bonne exécution des commandes `print`, mais lors de la remontée, `fact_rect(1)` ne renvoie rien ce qui provoque une erreur dans l'exécution de `fact_rect(2)`.

— La connaissance du processus de descente et de montée est nécessaire pour bien afficher les résultats voulus. Par exemple :

```
def fact_rec(n):
    if n==0 :
        return 1
    else :
        print('--'*n+'> appel de fact',n-1)
        A=n*fact_rec(n-1)
        print('--'*n+'> retour de fact',n-1)
        return A
```

Le premier `print` est exécuté lors de la descente (pour n décroissant) car son exécution ne nécessite que la connaissance de la variable locale n. Le second `print` est mis en attente comme toutes les instructions qui suivent l'appel récursif `A=n*fact_rec(n-1)`. Ce n'est donc que lors de la remontée (pour n croissant) qu'il est exécuté :

```
>>> fact_rec(3)
------> appel de fact 2
----> appel de fact 1
--> appel de fact 0
--> retour de fact 0
----> retour de fact 1
------> retour de fact 2
6
```

Si on avait utilisé ce code :

```
def fact_rec(n):
    if n==0 :
        return 1
    else :
        print('--'*n+'> appel de fact',n-1)
        return n*fact_rec(n-1)
        print('--'*n+'> retour de fact',n-1)
```

Le return placé trop tôt dans le bloc `else` interrompt la suite d'instructions, et le deuxième `print` n'est jamais exécuté

```
>>> fact_rec(3)
------> appel de fact 2
----> appel de fact 1
--> appel de fact 0
6
```

— La programmation récursive est adaptée à des problèmes « qui s'appellent eux-mêmes », c'est-à-dire à des problèmes qui font intervenir des relations de récurrence. Elle est donc alors relativement intuitive et une version itérative de ces algorithmes peut être beaucoup plus longue et délicate à programmer. Le prix à payer est une complexité spatiale accrue, qui interdit l'utilisation d'algorithmes récursifs appliqués à de trop grandes structures de données. Même pour des petites structures, la pile d'exécution peut exploser dans le cas de la récursivité multiple, comme on le verra plus loin.

4. Un autre exemple simple

La programmation récursive est en particulier très adaptée pour l'étude de suites définies par récurrence par une relation de la forme $u_0 \in \mathbb{R}$ donné et $u_{n+1} = f(n, u_n)$ pour $n \geqslant 0$. Par exemple, les différentes fonctions qui suivent calculent x^n, c'est-à-dire le $n^{\text{ème}}$ terme de la suite géométrique définie par $u_0 = 1$ et $u_{n+1} = x \times u_n$.

```
def geom1(x,n) :
    res = 1
    for _ in range(n) :
        res *= x
    return res
```

Une programmation récursive effectue la boucle dans l'autre sens : pour calculer $u_n = x \times u_{n-1}$, on calcule u_{n-1}, qui nécessite le calcul de u_{n-2}, ...jusqu'à tomber sur le cas de base $u_0 = 1$. Une programmation itérative serait plutôt :

```
def geom2(x,n) :
    res = 1
    while n > 0 :
        res *= x
        n -= 1
    return res
```

Une programmation récursive de cette suite serait :

```
def geom3(x,n) :
    if n == 0 : # le cas de base
```

```
        return 1
    else :
        return x * geom3(x,n-1)
```

On peut aussi calculer x^n en suivant la remarque suivante : $x^n = \begin{cases} \left(x^{n/2}\right)^2 & \text{si } n \text{ est pair} \\ x \times \left(x^{(n-1)/2}\right)^2 & \text{si } n \text{ est impair} \end{cases}$

Une autre programmation récursive serait :

```
def geom4(x,n) :
    if n == 0 : # le cas de base
        return 1
    elif n%2 == 0 :
        return geom4(x,n//2)**2
    else :
        return x*geom4(x,n//2)**2
```

Exercice : comparer l'efficacité des fonctions `geom3` et `geom4` en comptant le nombre d'appels faits à la fonction pour calculer x^n.

5. Exercice : palindrome

Une chaîne de caractères est un palindrome si elle peut se lire indifféremment dans les deux sens (sans tenir compte des majuscules, des espaces et des symboles de ponctuation). Pour tester si `chaine` est un palindrome, on peut procéder de la façon suivante :

- Une chaîne vide ou à une seul caractère est un palindrome.
- Si la chaîne contient plus de deux caractères, on note `debut` le premier caractère de `chaine` et `fin` le dernier caractère de `chaine` ; si `debut` et `fin` sont des lettres, on vérifie si `debut` et `fin` sont égaux et dans ce cas, on teste si la chaîne restante est un palindrome. Si `debut` n'est pas une lettre, on teste si la chaîne obtenue en enlevant `debut` est un palindrome ; on procède de même dans le cas où `fin` n'est pas une lettre.

Proposer un code PYTHON récursif testant si une chaîne de caractères est un palindrome. En donner la version itérative la plus proche.

La méthode `chaine.lower()` met la chaîne en minuscule et la méthode `elt.isalpha()` teste si `elt` est une lettre.

II Différents types de récursivité

1. Récursivité simple

Une fonction récursive simple est une fonction qui dans son exécution fait un seul appel à elle même ; c'est le cas des différentes fonctions rencontrées dans la partie I.

On peut en donner un autre exemple : le calcul du pgcd de deux entiers n et p. Ce calcul repose sur l'égalité suivante (algorithme d'Euclide) : $\text{pgcd}(n,p) = \text{pgcd}(p,r)$ où r est le reste de la division euclidienne de n par p. En effet, comme $n = pq + r$ avec $0 \leqslant r < p$, tout diviseur commun de p et r divise n et inversement, tout diviseur commun de n et p divise aussi r. Le cas de base correspond au cas où $r = 0$ (comme le second argument est une suite strictement décroissante d'entiers, on finira par tomber sur $r = 0$) : $\text{pgcd}(p,0) = p$. On peut donc coder cet algorithme de la façon suivante :

```
def pgcd_rec(n,p) :
    if p == 0 : # le cas de base
        return n
    else :
        return pgcd_rec(p,n%p) # appel à la fonction elle-même
```

La récursivité simple est un mode de programmation très naturel pour toutes les « suites » du type u_0 donné et $u_{n+1} = f(n,u_n)$ (u_n peut représenter un objet plus complexe qu'un simple vecteur, f représente alors les « transformations » à effectuer pour déterminer u_{n+1} à partir de u_n).

2. Récursivité multiple

Une fonction récursive peut également faire plusieurs appels à elle-même au cours de son exécution, on parle alors de récursivité multiple.

Un premier exemple est le calcul des termes de la suite de Fibonacci définie par :

$$f_0 = f_1 = 1 \quad \text{et} \quad f_n = f_{n-1} + f_{n-2} \text{ pour } n \geqslant 2$$

On peut déterminer la valeur de f_n par le code suivant :

```
def fibo_rec(n) :
    if n in [0,1] : # le cas de base
        return 1
    else :
        return fibo_rec(n-1) + fibo_rec(n-2) # double appel récursif
```

La récursivité multiple est très naturelle pour les « suite » du type u_0 donné et $u_{n+1} = f_n(u_n, u_{n-1}, \dots, u_{n-k_n})$ où $1 \leqslant k_n \leqslant n$. Le défaut de cette programmation est que le calcul de u_n, qui nécessite à priori d'avoir calculé u_{n-1} avant, se fait sans mémoriser la valeur de u_{n-1} ; ce terme sera donc calculé deux fois.
Pour illustrer ce problème de complexité, on peut examiner le code suivant qui calcule le $n^{\text{ème}}$ terme de la suite (u_n) définie par :

$$u_0 = 1 \quad \text{et} \quad u_n = \sum_{k=0}^{n-1} \binom{n}{k} u_k \text{ pour } n \geqslant 1$$

```
def b(k,n) :
    r = 1
    for j in range(1,k+1) :
        r = r * (n-j+1) // j
    return r

def suite_rec(n) :
    if n == 0 : # le cas de base
        return 1
    else :
        S = 0
        for k in range(n) :
            S += b(k,n)*suite_rec(k)
        return S
```

Si on note α_n le nombre d'appels à la fonction nécessaires pour calculer u_n, on a $\alpha_0 = 1$ et $\alpha_n = \sum_{k=0}^{n-1} \alpha_k = 2\alpha_{n-1}$ donc $\alpha_n = 2^n$.
Pour contourner ce problème, on peut transformer ces exemples en récursivités simples :

— Pour la suite de Fibonacci, si on pose $F_n = (f_{n+1}, f_n)$, on a $F_n = (f_n + f_{n-1}, f_n)$ donc on peut calculer F_n par une récurrence simple :

```
def fibo_rec_simple(n) :
    if n == 0 :
        return (1,1)
    else :
        a,b = fibo_rec_simple(n-1)
        return (a+b,a)

def fibo(n) :
    return fibo_rec_simple(n)[1]
```

— Pour la suite (u_n) :

```
def suite_rec_simple(n) :
    if n == 0 :
        return [1]
    else :
        L = suite_rec_simple(n-1)
        S = 0
        for k in range(n) :
            S += b(k,n)*L[k]
        L.append(S)
        return L

def suite(n) :
    return suite_rec_simple(n)[-1]
```

On transforme la récursivité multiple en une récursivité simple mais en manipulant des objets plus complexes.

3. Récursivités croisées

Des fonctions récursives croisées sont des fonctions qui font chacune appel à une autre fonction : on peut par exemple considérer les suites (u_n) et (v_n) définies par :

$$u_0 = 0, v_0 = 1 \quad \text{et} \quad \begin{cases} u_n = 1 + u_{n-1} + v_{n-1} \\ v_n = u_{n-1} - (n-1)v_{n-1} \end{cases}$$

Une façon de les programmer est la suivante :

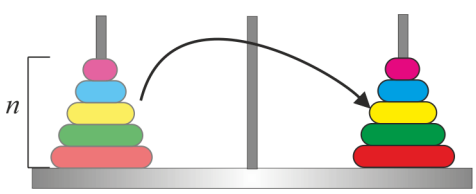
```
def U_rec(n) :
    if n == 0 :
        return 0
    else :
        return 1 + U_rec(n-1) + V_rec(n-1)

def V_rec(n) :
    if n == 0 :
        return 1
    else :
        return U_rec(n-1) - (n-1)*V_rec(n-1)
```

On peut à nouveau facilement remplacer cette récursivité croisée par une récursivité simple (avec un objet plus complexe, le couple (u_n, v_n)) :

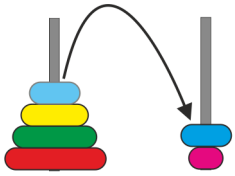
```
def couple_rec(n) :
    if n == 0 :
        return (0,1)
    else :
        a,b = couple_rec(n-1)
        return (1+a+b,a-(n-1)*b)
```

III Tours de Hanoï

On s'intéresse au jeu suivant : on dispose de n disques de tailles distinctes empilés sur une tige a et on souhaite les déplacer sur une tige c en utilisant une troisième tige b intermédiaire, en respectant les règles suivantes :



Hanoi(n,tour,a,b,c)


déplacement interdit

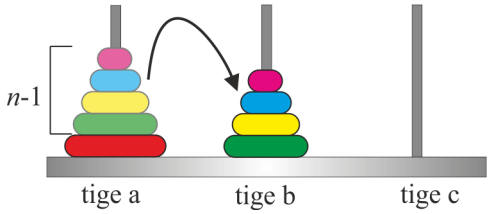
— On ne peut déplacer qu'un disque à la fois.
— On ne peut poser sur un disque qu'un disque de diamètre plus petit.

On suppose que cette condition est remplie à l'état initial : les disques forment au départ une pyramide sur la tige a. La situation initiale, dans le cas de 5 disques, sera représentée par la liste $[[5,4,3,2,1],[\,],[\,]]$ appelée tour : les disques sont représentés par des entiers et chaque tige par un numéro (0 pour a, 1 pour b, 2 pour c). Avec ces notations, la position finale est $[[\,],[\,],[5,4,3,2,1]]$ (on ordonne les listes dans cet ordre car il est plus facile de manipuler les derniers éléments d'une liste avec les méthodes **pop** et **append**.

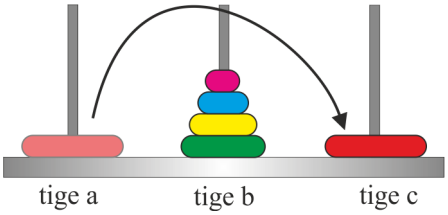
1. Définir la fonction `tour_init(n)` qui crée la position initiale avec n disques.

Le but de cet exercice est de créer une fonction qui donne la suite des manipulations à effectuer pour déplacer les disques. La démarche à suivre est la suivante :

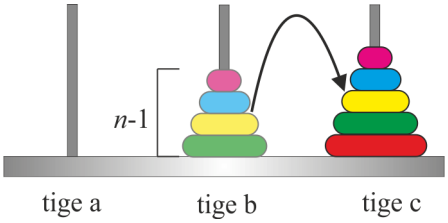
— Pour un seul disque, il suffit de le déplacer de la tige a vers la tige c.
— Si on suppose être capable de déplacer une pyramide de $n-1$ disques d'une tige vers une autre, il suffit pour déplacer une pyramide de n disques de la tige a vers la tige c de déplacer la pyramide contenant les $n-1$ premiers disques de la tige a vers la tige b (sans toucher au disque n du bas de la pyramide) : on arrive à la situation $[[n],[n-1,\dots,2,1],[\,]]$.



Puis on déplace le disque n de la tige a vers la tige c : situation $[[\,],[n-1,\dots,2,1],[n]]$.



Enfin, on redéplace la pyramide du milieu de la tige b vers la tige c pour aboutir à $[[\,],[\,],[n,n-1,\dots,2,1]]$.



On se rend alors compte que pour programmer cette fonction, on a besoin de savoir déplacer une pyramide d'une tige quelconque vers une autre tige quelconque (et pas seulement de la première vers la dernière). On va donc définir une fonction `Hanoi(n,tour,a,b,c)` qui modifie la situation initiale définie par la liste `tour` en déplaçant la pyramide supérieure de n disques de la tour a vers la tour c.

2. On suppose définie la fonction `Hanoi`. Donner à l'aide de la fonction `Hanoi` et des méthodes **pop** et **append** les instructions permettant de réaliser les opérations du schéma ci-dessus.

3. Écrire la fonction récursive `Hanoi` qui affiche toutes les étapes du jeu.

IV Correction de l'exercice sur les palindromes

Version récursive :

```
def palindrome_rec(texte):
    if len(texte)<=1:
        return True
    else:
        debut,fin=texte[0].lower(),texte[-1].lower()
        if debut.isalpha() and fin.isalpha():
            if debut==fin:
                return palindrome_rec(texte[1:len(texte)-1])
            else:
                return False
        elif not debut.isalpha():
            return palindrome_rec(texte[1:])
        else:
            return palindrome_rec(texte[0:len(texte)-1])
```

Version itérative (en s'autorisant des `return` au sein de la fonction pour avoir la programmation la plus proche de la version récursive) :

```
def palindrome_iter(texte):
    while len(texte)>1:
        debut,fin=texte[0].lower(),texte[-1].lower()
        if debut.isalpha() and fin.isalpha():
            if debut==fin:
                texte=texte[1:len(texte)-1]
            else:
                return False
        elif not debut.isalpha():
            texte=texte[1:]
        else:
            texte=texte[0:len(texte)-1]
    return True
```