Numérique et sciences informatiques
Classe de terminale

I Programmation et algorithmes
A1 Techniques de programmation
1. Programmation récursive

I Les principes de la programmation d'une fonction récursive

I.1 Un premier exemple de programmation récursive

a) La fonction factorielle en version itérative

• Soit n un entier positif. On veut programmer la fonction qui renvoie la **factorielle de n**, c'est-à-dire :

* le nombre 1 si n est égal à 0 ;
* si n est supérieur ou égal à 1, le produit des n premiers entiers consécutifs c'est à dire

$$1 \times 2 \times \ldots \times n$$

En mathématiques, la factorielle de n est notée $n!$.
On a par exemple :

$$4! = 1 \times 2 \times 3 \times 4 = 24$$

• On peut écrire une première version « itérative » de cette fonction en utilisant une boucle :

```
def factorielle_it(n) :
    """renvoie la factorielle de n.
       On suppose que n est un entier positif"""
    f = 1
    for k in range(1, n+1) :
        f = f*k
    return f
```

• L'appel de fonction :

```
print(factorielle_it(10))
```

donne le résultat suivant :

```
3628800
```

b) La fonction factorielle en version récursive

• Pour programmer une deuxième version de la fonction factorielle, nous allons utiliser les deux propriétés suivantes :

* on sait que $\text{factorielle}(0) = 1$ et $\text{factorielle}(1) = 1$;
* si l'entier n est supérieur ou égal à 2 (ou même à 1), on peut toujours utiliser l'égalité suivante :

$$\text{factorielle}(n) = n \times \text{factorielle}(n-1)$$

• De plus, on peut remarquer ces deux propriétés sont suffisantes pour calculer la factorielle de tout entier naturel car cela revient à utiliser le même raisonnement par lequel on définit une suite par récurrence.

• Une version « récursive » de la fonction factorielle est donc la suivante :

```
def factorielle_rec(n) :
    """renvoie la factorielle de n.
       On suppose que n est un entier positif"""
    if n == 0 or n==1 :
        return 1
    return n* factorielle_rec(n-1)
```

L'appel de la fonction

```
print(factorielle_rec(10))
```

donne le même résultat que pour la version itérative :

```
3628800
```

c) L'exécution du code de la version récursive

• Pour comprendre le code de la version récursive, il faut suivre pas à pas son exécution par exemple pour un appel de fonction `factorielle_rec(3)`
Nous allons représenter cette exécution par un « arbre d'appels » :

```
factorielle_rec (3)
        return 3* factorielle_rec(2)
                  return 2* factorielle_rec (1)
                                return 1
```

* Lors de l'appel `factorielle_rec(3)`, l'instruction `return 3* factorielle_rec(2)` va être exécutée puisque l'argument est différent de 0 et de 1.
* L'interpréteur va donc exécuter l'appel de fonction `factorielle_rec(2)` qui va à son tour lancer l'exécution de l'instruction `return 2* factorielle_rec(1)`.
* Enfin, l'appel de fonction `factorielle_rec(1)` va renvoyer 1 puisque cette fois la condition `n == 0 or n==1` sera évaluée à `True`.
* L'expression `factorielle_rec (2)` va donc renvoyer 2×1 donc 2.
* Puis l'expression `factorielle_rec (3)` va renvoyer 3×2 donc 6.

• Pour l'appel de fonction `factorielle_rec(10)`, on suivrait le même processus mais avec davantage d'étapes :
* l'appel `factorielle_rec(10)` lancera l'instruction `return 10* factorielle_rec (9);`
…
* l'appel `factorielle_rec(1)` lancera l'instruction `return 1`.

En remontant les appels de fonctions, `factorielle_rec(10)` renverra bien le résultat :

$$10 \times 9 \times \ldots \times 1$$

soit 3628800.

• On voit donc que l'exécution de la version récursive équivaut à l'exécution d'une boucle.

I.2 La notion de fonction récursive

• Le code de la fonction `factorielle_rec` contient un appel de la fonction `factorielle_rec` elle-même.

• On appelle **fonction récursive** une fonction qui s'appelle elle-même dans son code. Ce procédé est autorisé en Python comme dans la plupart des langages de programmation.

• On appelle **appel récursif** un appel de fonction à l'intérieur du code de cette même fonction.

• Une **fonction récursive** est une fonction dont l'implémentation utilise un ou plusieurs appel(s) récursif(s).

• La notion de fonction récursive concerne donc *l'implémentation* de la fonction et non pas sa spécification ou son *interface*. Comme nous venons de le voir, il est possible de programmer de deux façons différentes, itérative ou récursive, une « même » fonction, c'est-à-dire une fonction définie par ses entrées et sa sortie.

I.3 Conditions de validité d'une programmation récursive

• Une programmation récursive doit être écrite avec beaucoup de soin pour qu'elle réalise le résultat souhaité. Pour que le code d'une fonction récursive soit correct, il faut qu'il vérifie les conditions suivantes.

(1) Il faut qu'il comporte au moins un « **cas de base** », c'est-à-dire une valeur de l'argument pour laquelle la fonction renvoie le résultat *sans appel récursif*. Ceci correspond dans notre exemple, aux cas où l'argument n est égal à 0 ou à 1. Ces cas de base sont à distinguer de ceux qui conduisent à un appel récursif, dit « **cas récursifs** ».

(2) Il faut que chaque appel récursif utilise une valeur d'argument différente de l'argument initial. En effet, si cet appel porte sur la même valeur, il est certain que l'exécution ne pourra pas aboutir. Dans notre exemple, l'appel de fonction `factRec(n)` engendre l'appel récursif `factRec(n-1)`.

(3) Il faut que les appels récursifs successifs *à partir de n'importe quelle valeur de l'argument* conduisent en un nombre fini d'étapes à un appel sur une valeur correspondant à un cas de base. Dans notre exemple, en partant d'un cas récursif (un entier supérieur ou égal à 2), on est certain d'arriver par décrémentation successive à l'appel `factRec(1)`.

I.4 Un style de programmation gourmand en mémoire

• La programmation récursive est souvent élégante car elle permet souvent de réduire le nombre de lignes de code en évitant notamment d'écrire des structures itératives (boucles `for` ou `while`).

• En revanche, il est important de comprendre que ce style de programmation conduit souvent à beaucoup utiliser la mémoire de l'ordinateur. En effet, pour l'exécution de nombreux appels de fonction emboités conduit à la mémorisation du contexte de chaque appel dans une structure de données appelée la **pile d'exécution** (en anglais *call stack*).

• Cette pile d'exécution possède une limite de capacité. En Python, cette limite est fixé par défaut à 1000 appels récursifs emboités.
On pourra par exemple vérifier que l'appel de fonction `factRec(993)` donne un résultat (très grand) mais que l'appel `factRec(994)` conduit à l'affichage du message d'erreur suivant :

```
RecursionError: maximum recursion depth exceeded in
comparison
```

• Il est possible de repousser cette limite, par exemple en la fixant à 2000 appels récursifs emboîtés, en utilisant la fonction `setrecursionlimit` du module `sys`. Il suffit pour cela de placer le code suivant au début de son programme :

```
import sys
sys.setrecursionlimit(2000)
…
```

II Utilisations de la programmation récursive dans différents contextes

II.1 Les suites définies par récurrence

• La définition d'une suite par récurrence peut se traduire de façon quasi-automatique par une programmation récursive.

• Considérons une suite (u_n) est définie par récurrence de la façon suivante

$$u_0 = a$$

et pour tout entier naturel n $u_{n+1} = f(u_n)$

où a est un nombre réel et f est une fonction telle que $a \in D_f$ et pour tout $x \in D_f$, $f(x) \in D_f$.

On peut alors programmer de façon récursive une fonction `U(n)` qui renvoie la valeur de u_n pour tout entier naturel `n` donné en argument en traduisant directement la définition par récurrence de la suite. Une telle fonction écrite en Python donnerait :

```
def f(x) :
 ...

def U(n):
    if n == 0 :
        return a
    return f(U(n-1))
```

• Par exemple, soit (u_n) est la suite définie par

$$u_0 = -5$$
$$u_{n+1} = -3{u_n}^2 + 7u_n + 2$$

La fonction `U(n)` peut alors être définie de la façon suivante :

```
def g(x) :
    return -3*x**2 + 7*x + 2

def U(n):
    if n == 0 :
        return -5
    return g(U(n-1))
```

II.2 Utilisation des tranches dans un tableau ou une chaine de caractère

• Nous avons vu que le langage Python nous permet de manipuler des « tranches » (*slices* en anglais) de tableaux ou de chaînes de caractères.

* Si `T` est un tableau (donc un objet de type **`list`**) l'expression `T[i :j]` renvoie un tableau dont les éléments sont formés des éléments du tableau situés entre l'indice `i` et l'indice `j - 1`.

* De même, si `texte` est une chaîne de caractères (donc un objet de type **`str`**) l'expression `texte[i : j]` renvoie une chaîne de caractères formée des caractères situés entre l'indice `i` et l'indice `j - 1`.

• On peut exploiter cette possibilité offerte par Python pour écrire des programmes récursifs manipulant des tableaux ou des chaînes de caractères. Par exemple écrivons de façon récursive une fonction `palindrome(texte)` qui renvoie True si le texte est un palindrome, c'est-à-dire un texte qui est identique qu'on le lise de gauche à droite ou de droite à gauche :

```
def palindrome(texte) :
    lg = len(texte)
    if lg < 2 :
        return True
    return texte[0] == texte[lg-1] \
                and palindrome(texte[1:lg-1])
```

II.3 L'algorithme d'Euclide

L'algorithme d'Euclide qui permet de déterminer le PGCD (plus grand commun diviseur) de deux entiers naturels a et b consiste à effectuer les opérations suivantes (en supposant ici que l'on a $a \geq b$) :

* Si b = 0 alors PGCD(a, b) = a

* Sinon PGCD(a, b) = PGCD(b, r) où r est le reste de la division euclidienne de a par b.

Par définition du reste, on aura alors $r < b < a$. Donc la substitution répétée du couple $(a; b)$ par le couple $(b; r)$ conduira nécessairement en un nombre fini d'étapes à la situation $r = 0$.

• Cet algorithme se traduit très naturellement par le programme récursif suivant :

```
def PGCD_rec(a, b) :
    """ a est supérieur ou égal à b"""
    if b == 0 :
        return a
    return PGCD_rec(b, a%b)

def PGCD(a, b) :
    if b > a :
        a, b = b, a
    PGCD_rec(a, b)
```

II.4 Dessiner des fractales

• Prenons l'exemple d'une fonction `koch(n, l)` qui permet de dessiner une courbe de Koch (voir la définition sur *wikipedia*) d'ordre n et de longueur l à l'aide de la bibliothèque `Turtle` :

```
from turtle import *
speed(0)

def koch(n, l):
    if n == 0 :
        forward(l)
    else :
        koch(n-1, l/3)
        left(60)
        koch(n-1, l/3)
        right(120)
        koch(n-1, l/3)
        left(60)
        koch(n-1, l/3)
```

• On voit ici que cette fonction comporte quatre appels récursifs