Chapitre 9 : Fonctions récursives

1. Notion de récursivité

1.1.Introduction-Une nouvelle façon d'écrire un script

Comme premier exemple, considérons une fonction `affiche1` permettant d'afficher les entiers naturels compris entre `0` et un entier `n` passé en argument.
Proposer un premier script pour la fonction `affiche1` utilisant un boucle non-conditionnelle puis un second avec une boucle conditionnelle.

On considère maintenant une autre version nommée `affiche2` qui présente la particularité de s'appeler elle-même.

```
def affiche2(k):
    if k<10:
        print(k)
        affiche2(k+1)
```

Si nous écrivons l'instruction `affiche2(0)`, la fonction `affiche2` est appelée avec le paramètre `0` qui vérifie le test `k<10`. Après l'affichage de `0`, la fonction `affiche2` est à nouveau appelée mais avec le paramètre `1`, puis après l'affichage de `1`, elle est à nouveau appelée avec le paramètre `2`, et ceci continue avec le paramètre `k` et l'appel qui suit tant que `k<10`.
Remarquons que dans ce code, nous n'avons ni boucle non-conditionnelle ni boucle conditionnelle.

Examinons un deuxième exemple un peu plus complexe : on considère une fonction qui dessine des carrés dont les dimensions sont contenus dans une liste. Pour cela, nous utiliserons le module `Turtle`.

Proposer une fonction `carre` qui trace le carré de côté `d` passé en argument.

Écrire une fonction `dessine1` appelant la fonction `carre` de manière répétitive à l'aide d'une boucle et qui trace des carrés dont les longueurs des côtés sont contenus dans la liste `liste` en effectuant entre chaque carré une rotation vers la gauche d'un angle `angle`, `liste` et `angle` étant passés en argument.

On peut tester le programme avec les éléments suivants :

- `lst=[100,90,80,70,60]`
- `a=20`

On peut une fois de plus proposer une deuxième version `dessine2` pour laquelle la fonction s'appelle elle-même.

```
def dessine2(liste,i,angle):
    if i>=0 and i<len(liste):
        left(angle)
        carre(liste[i])
        dessine2(liste,i+1,angle)
```

Si nous écrivons l'instruction `dessine2(lst,0,a)`, la fonction `dessine2` est appelée avec le paramètre `0` qui vérifie les tests `i>=0` et `i<len(lst)`. Après le dessin du premier carré de côté `lst[0]`, la fonction `dessine2` est à nouveau appelée mais avec le paramètre `1`, puis après le tracé du carré de côté `lst[1]`, elle est à nouveau appelée avec le paramètre `2`, et ceci continue avec le paramètre `i` et l'appel qui suit tant que `i<len(lst)`.
Remarquons qu'une fois de plus, dans ce code, nous n'avons ni boucle non-conditionnelle ni boucle conditionnelle.

1.2.Définition

Une fonction est dite récursive si elle s'appelle elle-même. On parle alors d'appel récursif.

Dans les chapitres précédents, les programmes ont été écrits avec des boucles et des affectations. C'est un style de programmation qui est :
- Impératif : les séquences d'instructions sont exécutées les unes après les autres
- Itératif : des instructions sont exécutées dans des boucles `while` ou `for`.

Comme cela est visible sur les exemples précédents, une fonction qui s'appelle elle-même permet de remplacer une boucle. On parle alors de programme récursif par opposition à itératif.

1.3.Construction d'une fonction récursive

Pour illustrer la façon dont on construit une fonction récursive, appuyons nous sur la fonction `produit` ci-dessous ayant pour objet le calcul du produit de deux entiers naturels `n` et `p` sans faire appel à l'opération multiplication de python.

```
def produit(n,p):
    if p==0:
        return(0)
    else:
        return(n+produit(n,p-1))
```

Supposons que nous exécutions `produit(4,3)` :
- > le calcul de `4+produit(4,2)` est en attente de `produit(4,2)`
- >> le calcul de `4+produit(4,1)` est en attente de `produit(4,1)`
- >>> le calcul de `4+produit(4,0)` est en attente de `produit(4,0)`
- >>>> la valeur de `produit(4,0)` est obtenue est vaut `0`
- >>> la valeur de `4+produit(4,0)` est calculée est vaut `4`
- >> la valeur de `4+produit(4,1)` est calculée est vaut `8`
- > la valeur de `4+produit(4,2)` est obtenue est vaut `12`

La valeur retournée est `12` (le produit de `4` par `3`).

On peut effectuer plusieurs remarques en examinant cette fonction :
- Le test est une condition d'arrêt. Il peut y avoir plusieurs conditions. Au moins une condition est nécessaire afin que le programme ne boucle pas indéfiniment.
- Les valeurs passées en paramètres dans les appels récursifs constituent une suite de nombres qui décroit vers la valeur d'arrêt. Dans l'exemple, la valeur n-1 passée en paramètre permet de faire décroître la valeur du paramètre. Pour que le programme ne boucle pas indéfiniment, il est impératif que le ou les paramètres atteignent une valeur d'arrêt après un nombre fini d'appels.

Nous pouvons en déduire quelques généralités sur les fonctions récursives :
- Une fonction récursive doit contenir une ou des conditions d'arrêt. Sinon le programme boucle indéfiniment.
- Les valeurs passées en paramètres dans les appels récursifs doivent être différents. Sinon la fonction s'exécute à chaque appel de manière identique et continue de s'exécuter indéfiniment.
- Après un nombre fini d'appels, la ou les valeurs passées en paramètres doivent permettre de satisfaire la condition d'arrêt.

L'exemple de la fonction produit nous permet de déduire que les instructions :

```
Si condition:
      appel récursif
```

Et

```
Tant que condition:
      instructions
```

sont équivalentes.
Mais avec la fonction produit, des calculs ne peuvent pas être effectués avant d'obtenir les valeurs renvoyées par les appels récursifs. La mémoire de l'ordinateur est alors sollicitée pour stocker des instructions en attente. On parle dans ce cas de récursivité profonde. Nous allons voir que ceci peut parfois poser problème et nécessiter quelques modifications.

2. Quelques exemples classiques

2.1.La fonction factorielle

À l'aide d'une boucle non-conditionnelle, écrire une fonction `factorielle1` permettant la calcul de `n!`, `n` entier naturel non nul étant passé en argument.

En remarquant que la définition de la fonction factorielle permet d'écrire $0!=1!=1$ et $n!=n\times(n-1)!$, proposer une fonction récursive `factorielle2` effectuant le calcul de `n!`.

Dans cette fonction récursive, la dernière instruction est une multiplication par n qui ne peut être effectuée qu'après avoir obtenu la valeur renvoyée par l'appel récursif. Les différentes multiplications sont donc mises en attente comme les additions dans la fonction `produit`.
La fonction peut ainsi être modifiée en utilisant un paramètre supplémentaire, un accumulateur, par lequel des informations sont passées dans les appels récursifs. On munit ainsi la fonction d'un peu de mémoire. C'est l'accumulateur qui est renvoyé lorsque la condition des appels récursifs est satisfaite. Il doit contenir le résultat.

```
def factorielle3(n,acc):
    if n>0:
        return(factorielle3(n-1,acc*n))
    else:
        return(acc)
```

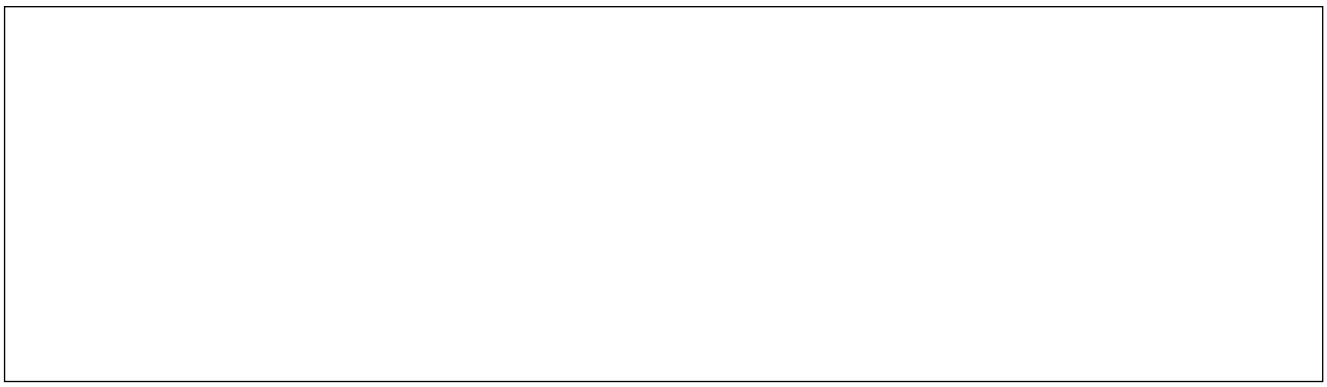
Cette fonction renvoie la valeur de `acc*n!` Donc `factorielle3(n,1)` a pour valeur `n!` . Il n'y a alors aucune opération en attente hormis les appels récursifs.

2.2.La suite de Fibonacci

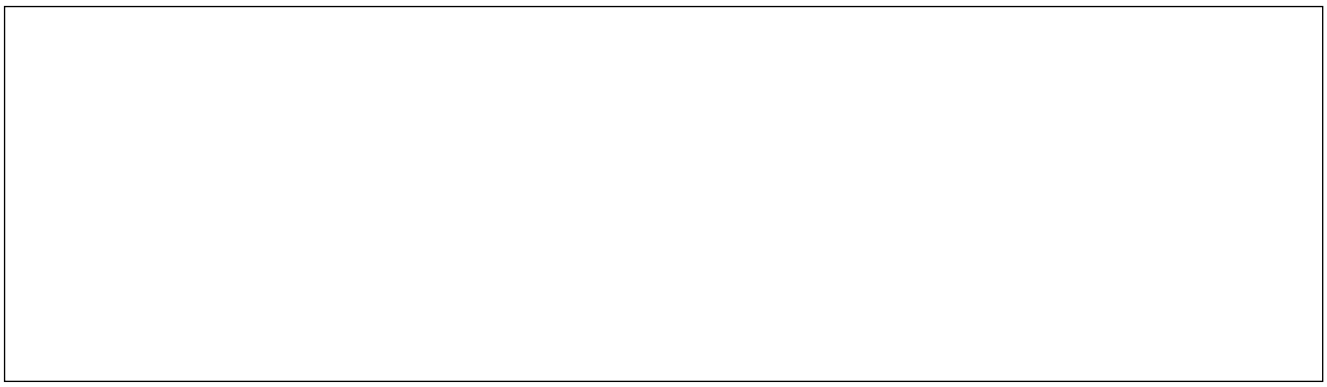
La suite de Fibonacci est définie de la manière suivante :

$$\forall n > 1,\ u_n = u_{n-1} + u_{n-2}\ ;\ u_0 = 0\ \text{et}\ u_1 = 1$$

Le terme de rang n peut se calculer avec $n-2$ additions. On peut donc écrire une fonction `fibo1` utilisant une boucle `for` ou une boucle `while`, qui prend en paramètre un entier n et renvoie le terme u_n .



En traduisant la formule de calcul, écrire une fonction récursive `fibo2`.



On peut remarquer que cette version récursive est moins efficace que la version itérative. En effet, le nombre d'appels récursifs est presque multiplié par deux chaque fois qu'on augmente n

d'une unité. Il suffit de constater que $2^{30} \simeq 10^9$, pour conclure qu'avec des valeurs supérieures à 30, le calcul devient lent, et sûrement trop long pour des valeurs supérieures à 40.

Il est ainsi possible de construire une nouvelle fonction en utilisant tout comme dans le cas de la fonction calculant $n!$, non pas un mais deux accumulateurs.

```
def fibo3(n,a,b):
    if n==0:
        return(a)
    else:
        return(fibo3(n-1,b,a+b))
```

Pour calculer u_6, il faudra appeler `fibo3(6,0,1)`.