2.1 Qu'est ce que la récursivité ?

NSI terminale - JB Duthoit

"Pour bien comprendre la récursivité, vous devez d'abord bien comprendre ce qu'est la récursivité." "To understand recursion, one must first understand recursion." Stephen Hawking

Une poupée russe, c'est une poupée avec une poupée russe à l'intérieur !



2.1.1 Introduction

Analyse d'un programme

Analyser et tester ce programme

```
def f():
    for i in range(1,6):
        print(f"{i} affiché par la fonction f")

def g():
    for i in range(1,6):
        print(f"{i} affiché par la fonction g")
        if i == 3:
            f()
g()
```

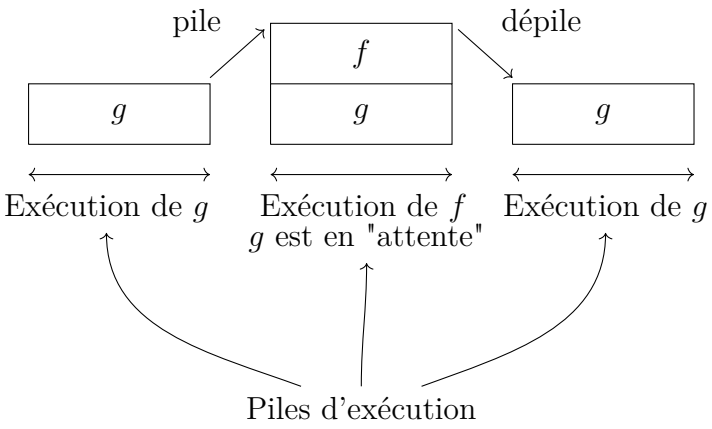
Dans l'exemple ci-dessus, nous avons une fonction `g` qui appelle une autre fonction `f`. La principale chose à retenir de cet exemple est que l'exécution de `g` est interrompue pendant l'exécution de `f`.
Une fois l'exécution de `f` terminée, l'exécution de `g` reprendra là où elle avait été interrompue.

☛ Analyse ce programme dans Python Tutor.

Les piles d'exécution

Pour gérer ces fonctions qui appellent d'autres fonctions, le système utilise une **"pile d'exécution"**.
Une pile d'exécution permet d'enregistrer des informations sur les fonctions en cours d'exécution dans un programme.

On parle de **pile**, car les exécutions successives "s'empilent" les unes sur les autres. Si nous nous intéressons à la pile d'exécution du programme étudié ci-dessus, nous obtenons le schéma suivant :

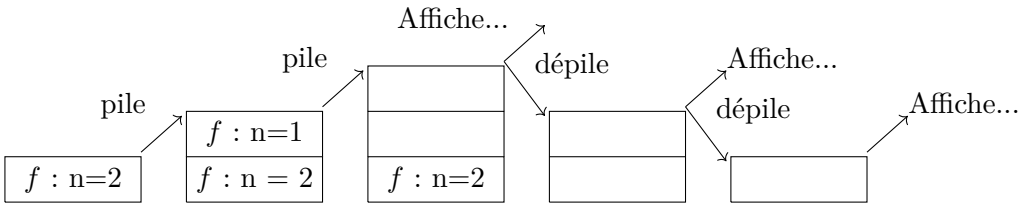


Lorsque l'on fait fonctionner un programme récursif, il utilise forcément des piles d'exécution.

2.1.2 Un premier exemple

Analyser, essayer de prévoir , puis tester ce programme :
Créer les différentes piles en complétant le schéma suivant :

```
def f(n):
    if n>0:
        f(n-1)
    print(n)
```



☛ Étudier ce programme dans Python Tutor.

Remarque

Il ne faut jamais oublier qu'à chaque nouvel appel de la fonction f le paramètre n est différent. La variable n est différente pour chaque fonction. D'une façon générale, il faut retenir que les variables définies dans une fonction ne sont utilisables que dans cette fonction.

2.1.3 Définition

La récursivité est une démarche qui fait référence à l'objet même de la démarche à un moment du processus.

En d'autres termes, c'est une démarche dont la description mène à la répétition d'une même règle.

Il existe de nombreux programmes ou algorithmes récursifs, avec notamment :

- tri fusion
- l'algorithme de remplissage d'une zone délimitée (algorithme de diffusion)

Et aussi des structures récursives :

- liste
- Arbres ...

Remarque

La structure récursive des listes
De manière formelle, les listes sont des structures de données dynamiques et récursives. Elles se définissent ainsi :
Une liste d'éléments de type T est :

- soit la liste vide
- soit un couple (t,R) où t est de type T et R est une liste d'éléments de type T

Dans le cas d'une liste non vide (t,R) :

- t est le premier élément de la liste
- R est la liste des éléments qui suivent t

On parle pour de telles structures de listes chaînées, qui se distingue donc des listes par tableaux.

Les listes chainées sont par exemple beaucoup plus efficaces que les listes par tableaux lorsqu'il s'agit de supprimer un élément de la liste, ou d'en insérer un.

Définition

Un algorithme de résolution d'un problème P sur une donnée a est dit **récursif** si parmi les opérations utilisées pour le résoudre, on trouve la résolution du même problème P sur une donnée b.

2.1.4 Un exemple de situation qui pourrait être résolue par récursivité

On pourrait utiliser la récursivité pour calculer la dérivée d'une fonction par exemple :

```
Entrée : f (une fonction dérivable)
Sortie : f ' (la fonction dérivée)
derivee(f) =
     si f est une fonction élémentaire de base
```

```
      renvoyer sa dérivée
sinon si f == u+v
      renvoyer derivee(u) + derivee(v)
sinon si f == u * v
      renvoyer derivee(u)*v + u*derivee(v)
sinon si ....
```

Exercice 2.1

CONSTRUCTION D'UN PROGRAMME RÉCURSIF QUI DONNE LA FACTORIELLE D'UN ENTIER

La factorielle est une fonction mathématique qui prend comme paramètre un entier n et qui renvoie le produit des nombres entiers strictement positifs inférieurs ou égaux à n.

Ainsi, la factorielle de 3 est : $3 \times 2 \times 1 = 6$; la factorielle de 4 est $4 \times 3 \times 2 \times 1 = 24$; la factorielle de 5 est $5 \times 4 \times 3 \times 2 \times 1 = 120$.

On note la factorielle de n par $n!$, on a donc $5! = 120$, $3! = 6$...

Ainsi $n!$ est défini par :

- $0! = 1$ (par définition)
- Pour tout entier $n > 0$, $n! = n \times (n \text{˘} 1)!$

1. On note `fact(n)` la fonction récursive python qui permet de renvoyer la factorielle de `n`. Compléter :
   a) `fact(0)=.....`
   b) `fact(n)=.............`
2. Représenter l'exécution de cette fonction à l'aide de piles.
3. Écrire cette fonction python, et testez-la.

2.1.5 Les éléments pour un "bon" programme récursif

Une première erreur classique !

Analyser puis tester le code suivant :

```
def f():
    print ("Bonjour !")
    f()
f()
```

Dans le cas où une fonction s'appelle elle-même, on retrouve le même système de pile d'exécution.
Dans l'exemple traité ci-dessus, les appels s'enchainent sans rien pour mettre un terme à cet enchainement, la taille de la pile d'exécution augmente sans cesse, nous n'avons pas de "dépilement" juste des "empilements".
Le système interrompt donc le programme en générant une erreur quand la pile d'exécution dépasse une certaine taille.

☛ Quand on écrit une fonction récursive, il est donc nécessaire de bien penser à mettre en place une structure qui à un moment ou à un autre **mettra fin** à ces appels récursifs.

⚠ **Insistons :** chaque appel possède son propre environnement, donc ses propres variables.

⚠ En Python, la taille de la pile des appels récursifs est limitée. Si la récursivité est trop profonde et que l'on atteint cette limite, on déclenche une RecursionError.
La valeur de cette pile peut être obtenue par :

```
>>> import sys
>>> sys.getrecursionlimit()
```

Si on veut augmenter cette profondeur on peut utiliser la fonction sys.setrecursionlimit (cependant il faut rester raisonnable) :

```
>>> sys.setrecursionlimit(1500)
```

pour mettre 1500 au lieu d 1000 par exemple !

Une autre erreur classique...

```
def fact(n):
    if n <= 1:
        return 1
    else:
        n * fact(n-1)
```

Analyser ce programme et trouver son erreur !

Éléments caractéristiques d'un bon programme récursif

- Il faut au moins une situation qui ne consiste pas en un appel récursif

  ```
  if n <= 1:
      return 1
  ```

  Cette situation est appelée situation de terminaison ou situation d'arrêt.

- Chaque appel récursif doit se faire avec des données qui permettent de se rapprocher d'une situation de terminaison

  ```
  return n * fact(n-1)
  ```

  Il faut s'assurer que la situation de terminaison est atteinte après un nombre **fini** d'appels récursifs.

  ☞ La preuve de terminaison d'un algorithme récursif se fait en identifiant la construction d'une suite strictement décroissante d'entiers positifs ou nuls.

⚠ Ces éléments sont nécessaires mais malheureusement pas suffisants pour réussir un programme récursif !

2.1.6 Les différents types de récursivité

Récursivité terminale

Un algorithme récursif simple est terminal lorsque l'appel récursif est le dernier calcul effectué pour obtenir le résultat. Il n'y a pas de "calcul en attente".
L'avantage est qu'il n'y a rien à mémoriser dans la pile.

```
def est_present(c,s):
    if s == '':
        return False
    elif c == s[0]:
        return True
    else:
        return est_present(c,s[1:])
```

Récursivité multiple

Les expressions qui définissent une fonction peuvent aussi dépendre de plusieurs appels à la fonction en cours de définition.
Par exemple, la fonction fibonacci(n), qui doit son nom au mathématicien Leonardo Fibonacci, est définie récursivement, pour tout entier naturel n, de la manière suivante :

$$fibonacci(n) = \begin{cases} 0 \quad si \quad n = 0 \\ 1 \quad si \quad n = 1 \\ fibonacci(n-1) + fibonacci(n-2) \quad si \quad n \geq 2 \end{cases}$$

Récursivité croisée

Voici deux fonctions nommées pair et impair déterminant la parité ou l'imparité d'un entier.

```
def pair(n):
  if n == 0:
    return True
  else:
    return impair(n-1)

def impair(n):
  if n  == 0:
    return False
  else:
    return pair(n-1)
```

2.1.7 Conclusion

La récursivité offre donc au programmeur un autre moyen, souvent élégant et concis, de résoudre des problèmes.

Par exemple :

- dans la programmation des jeux solitaires du type Sudoku, labyrinthes...

- dans la programmation des jeux à deux joueurs du type Échecs, Dames, Othello...
- et dans bien d'autres domaines encore