Récursivité

Cours

Introduction

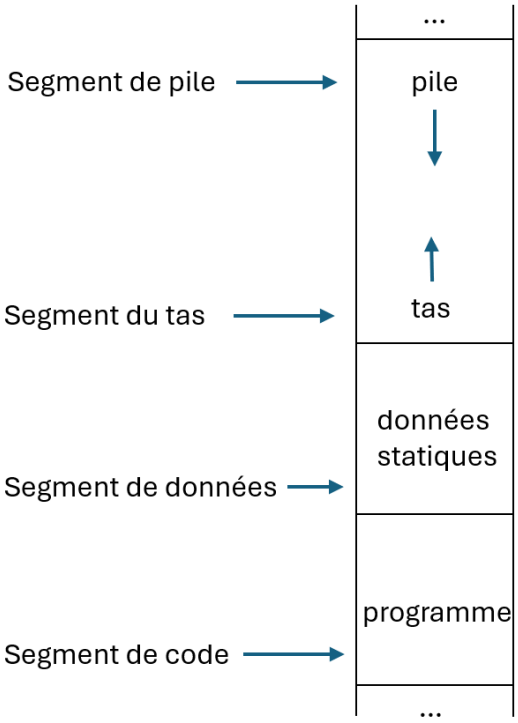
Organisation de la mémoire

La mémoire d'un ordinateur contient à la fois le programme à exécuter et les données que ces programmes doivent manipuler.

La mémoire d'un ordinateur peut être vue comme un tableau de cases mémoires élémentaires, appelées mot mémoire.

Chaque case possède une adresse unique à laquelle on se réfère pour accéder à son contenu (en écriture ou en lecture).

Le tableau ci-dessous décris l'organisation de l'espace mémoire d'un programme actif, plus généralement appelé *processus*, c'est-à-dire un programme en cours d'exécution.



Cet espace et découpé en quatre parties, on dit aussi segment de mémoire :

- **Le segment de code** :
  - Contient les instructions du programme ;

- **Le segment de données** :
  - Contient les données dont l'adresse (en mémoire) et la valeur sont connues au moment de l'initialisation - on parle de données statiques ;

- **Le segment de pile** :
  - Contient l'espace mémoire alloué dynamiquement par un programme ;
  - Utilisée au moment de l'appel de fonctions d'un programme ;
  - La gestion en pile libère la mémoire au moment où la fonction se termine ;

- **Le segment du tas** :
  - Zone mémoire qui contient toutes les données allouées dynamiquement ;
  - Durée de vie non liée à l'exécution des fonctions ;
  - Dont le type impose qu'elle soit allouée dans cette zone mémoire - exemple : taille peut évoluer (tableaux Python).

Exemple sur la gestion du segment de pile

Considérons les deux fonctions suivantes en Python et observons la gestion de la pile décrite lors de l'appel f(1, 5).

```
def g(x, y) :
    return 100 + y

def f(x, y) :
    z= x+y
    u = g(x-1, y*2)
    return z + u
```

La pile (A) représente l'environnement lors de l'appel à f(1, 5).

Les premières cases mémoires contiennent :

- Une sauvegarde des registres du microprocesseur (reg),
- Un espace pour la valeur de retour de la fonction (ret),
- Les valeurs des paramètres
- Un espace pour la variable locale (ne contient aucune valeur au départ).

La pile (B) représente l'environnement lors de l'appel g(x – 1 , y * 2).

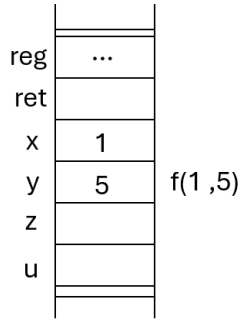
Pour réaliser cet appel :

- Un nouvel environnement est alloué sur la pile ;
- Avec une sauvegarde des registres et les valeurs des arguments x (0) et y (10) de g ;
- Se termine en renvoyant la valeur 100 + y qui est stockée dans son espace de retour (ret).

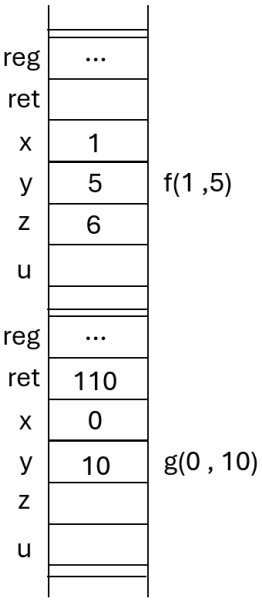
La pile (C) montre enfin l'environnement de f(1, 5) après le retour de g0, 10).

- L'espace alloué pour l'appel à g a été supprimé de la pile ;
- La valeur de retour (110) a été récupérée et stockée dans la case mémoire de la variable locale u et que la somme z + u (116) est stockée dans l'espace de retour.

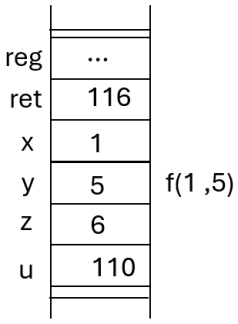
(A)

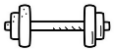


(B)



(C)



 **Activité :** Exercice 1

Le principe

On aborde dans ce chapitre la programmation à l'aide de fonctions récursives.

Une fonction récursive est une fonction qui s'appelle elle-même.

Il s'agit à la fois :

- D'un style de programmation
- D'une technique pour résoudre certains problèmes qu'il n'est parfois pas facile de traiter en programmant uniquement avec des boucles.

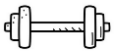
La somme des n premiers entiers

Pour définir la somme des n premiers entiers, on a l'habitude d'écrire la formule :

$$0+1+2+3+4+...+n$$

Une solution :

- Utiliser une **boucle for** pour parcourir tous les entiers i entre 0 et n.

 **Activité :** Coder la fonction somme(n) permettant de calculer la somme des n premiers entiers, à l'aide d'une boucle for.

```
def somme(n) :
```

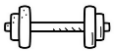
➔ On peut remarquer que ce code Python n'est pas directement lié à la formule de départ.

Il existe une autre manière d'aborder ce problème :

➔ Définir la fonction récursive somme(n) qui, pour tout entier naturel n, donne la somme des n premiers entiers de la manière suivante :

si n=0 : somme (n) = 0

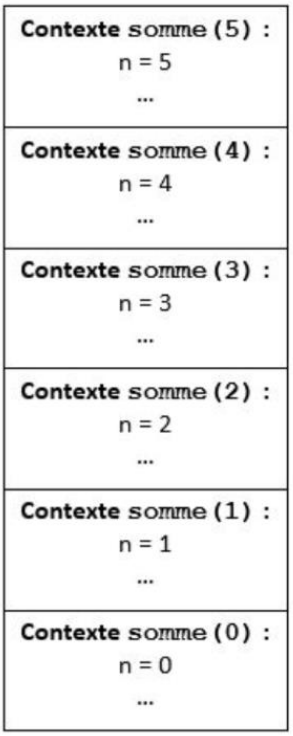
si n>0 : somme (n) = n+somme(n-1)

 **Activité :**

1. Écrire la fonction récursive somme(n) qui calcule la somme des n premiers entiers.
2. Dessiner l'arbre d'appels de cette fonction pour l'appel somme(5).

```
def somme(n) :
```

L'organisation de la mémoire lors de l'appel à somme(5) peut être représentée par la pile ci-dessous :



- ⇨ Un nouvel environnement d'exécution va être alloué dans la pile pour chacun de ces appels.
- ⇨ Après le dernier appel, la pile contient les contextes d'exécution pour les six appels à la fonction somme.
- ⇨ Lors de l'exécution d'une fonction récursive, chaque appel récursif conduit à un empilement du contexte dans la pile d'exécution.
- ⇨ Lorsque la condition d'arrêt de la récursivité se produit : les différents contextes sont progressivement dépilés pour poursuivre l'exécution de la fonction.

Attention : Puisque les appels récursifs sont effectués sur une zone de mémoire plutôt limitée : la pile d'exécution ne doit pas dépasser une certaine limite. Par défaut, elle est de 1000 dans l'implémentation courante de Python.
Il est cependant possible de modifier le plafond du nombre d'appels :

```
import sys
sys.setrecursionlimit(3500)

def somme(n) :



somme(3000)
```

Les principes de la programmation récursive

La programmation récursive est une approche où une fonction s'appelle elle-même pour résoudre un problème.

Elle est particulièrement utile pour résoudre des problèmes qui peuvent être décomposés en sous-problèmes similaires.

Voici les grands principes de la programmation récursive :

1. Cas de Base (Condition d'arrêt)

- **Définition :** Le cas de base est la condition qui permet de mettre fin à la récursion. C'est une situation où le problème est suffisamment simple pour être résolu directement, sans appel récursif supplémentaire.

2. Appel Récursif

- **Définition :** Si le cas de base n'est pas satisfait, la fonction s'appelle elle-même avec un sous-ensemble du problème initial. Cela crée une série d'appels imbriqués jusqu'à ce que le cas de base soit atteint.

3. Décomposition du Problème

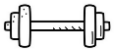
- **Définition :** Un problème complexe est décomposé en sous-problèmes plus simples, souvent similaires au problème initial. Cette répétition de la décomposition est essentielle pour la récursion.

4. Convergence

- **Définition :** Pour qu'une fonction récursive fonctionne correctement, chaque appel récursif doit se rapprocher du cas de base. Cela garantit que la récursion finira par s'arrêter.

5. Empilement des Appels (Stack)

- **Définition :** Chaque appel récursif est empilé dans une pile d'appels, qui est gérée automatiquement par le système d'exécution du programme. Lorsque le cas de base est atteint, les appels sont "désempilés" et les résultats sont combinés pour produire la solution finale.

 **Activité :** Exercices 2 et 3.

Exemple : Le calcul de la factorielle d'un nombre

Le calcul de la factorielle d'un nombre est une opération mathématique qui consiste à multiplier ce nombre par tous les entiers positifs inférieurs ou égaux à lui. La factorielle d'un nombre entier positif n est notée n!

Définition Formelle

- **Pour un entier positif n :**

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$$

- **Cas particulier :**

$$0! = 1$$

Par convention, la factorielle de 0 est définie comme étant égale à 1.

Exemple de Calculs

- $5!$:

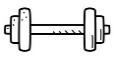
$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

- $3!$:

$$3! = 3 \times 2 \times 1 = 6$$

- $0!$:

$$0! = 1$$

 **Activité :** Déterminer la fonction permettant de calculer la factorielle d'un nombre n de manière récursive.

```
def factorielle(n):
```

La récursivité terminale

La récursivité terminale (ou **récursion terminale**) est un type particulier de récursion où l'appel récursif est la dernière opération effectuée par la fonction. Cela permet à certains compilateurs ou interpréteurs d'optimiser la récursion pour éviter des débordements de pile, en réutilisant la même mémoire pour chaque appel récursif.

Exemple de Récursivité Terminale : Calcul de la Factorielle

Un exemple classique de récursivité terminale est le calcul de la factorielle. Comparons une version standard et une version terminale de cette fonction.

```
def factorielle(n):
```

Ici, la multiplication n* doit encore être effectuée après le retour de l'appel récursif, donc ce n'est pas de la récursivité terminale.

Version Récursive Terminale (Optimisée)

Pour que la fonction soit récursive terminale, on peut introduire un accumulateur qui garde en mémoire le produit partiel des calculs.

```
def factorielle_terminale(n, accumulateur=1):
    if n == 0:
        return accumulateur  # Cas de base
    else:
        return factorielle_terminale(n - 1, n * accumulateur)  # Appel récursif terminal
```

- ⇨ **Cas de base :** Si n=0, on retourne l'accumulateur, qui contient le résultat final.
- ⇨ **Appel récursif terminal :** La fonction s'appelle avec n−1 et l'accumulateur mis à jour, ce qui fait en sorte que l'appel récursif est la dernière opération.

 **Activité :** Exercices 4, 5, 6 , 7.

D'autres formes de récursivité

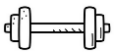
Formulations récursives

Prenons comme exemple l'opération de puissance n-ième d'un nombre x, c'est à dire la multiplication répétée n fois de x avec lui-même, que l'on écrit habituellement de la manière suivante, avec par convention, que la puissance de x pour n = 0 vaut 1 :

$x^n = \underbrace{x * .... * x}_{\text{n fois}}$

Pour écrire une version récursive de x^n , on va définir une fonction puissance(x, n) en commençant par chercher les cas de base à cette opération. Ici, le cas de base évident est celui pour n=0. Pour définir la valeur de puissance(x,n) pour un entier n strictement positif, on suppose que l'on connaît le résultat de X à la puissance n – 1, c'est à dire la valeur de puissance(x, n-1). On obtient la définition suivante :

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

 **Activité :** Déterminer la fonction puissance(x, n) pour un entier n strictement positif de manière récursive.

```
def puissance(x, n):
```

Définitions récursives plus riches

Cas de base multiples

La définition de la fonction puissance(x,n) n'est pas unique. On peut par exemple identifier deux cas de base « faciles », celui pour n = 0 mais également celui pour n = 1, avec puissance(x, 1) = x. Ce deuxième cas de base a l'avantage d'éviter de faire la multiplication (inutile) x * 1 de la définition précédente. Ainsi on obtient la définition suivante avec deux cas de base :

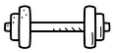
$$\text{puissance}(x,n) = \begin{cases} 1 & \text{si } n = 0, \\ x & \text{si } n = 1, \\ x * \text{puissance}(x, n-1) & \text{si } n > 1. \end{cases}$$

On pourrait continuer à ajouter des cas de base pour n = 2, n= 3, mais cela n'apportera rien à la définition et cela ne réduirait pas le nombre de multiplications à effectuer.

Double récursion

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 aux mathématiciens Leonardo Fibonacci, elle définit récursivement, pour tout entier naturel n, de la manière suivante :

$$\text{fibonacci}(n) = \begin{cases} 0 & \text{si } n = 0, \\ 1 & \text{si } n = 1, \\ \text{fibonacci}(n-2) + \text{fibonacci}(n-1) & \text{si } n > 1. \end{cases}$$

 **Activité :** Ecrire le détail des calculs et des appels récursifs pour les fonctions fibonacci suivantes :

fibonacci(0) =

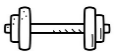
fibonacci(1) =

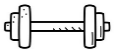
fibonacci(2) =

fibonacci(3) =

fibonacci(4) =

fibonacci(5) =

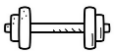
 **Activité :** Dessiner l'arbre d'appel fibonacci(5)

 **Activité :** Déterminer la fonction fibonacci(x) de manière récursive.

Récursion imbriquée

Les occurrences de la fonction en cours de définition peuvent également être imbriquées. Par exemple, la fonction $f_{91}(n)$ est définie avec deux occurrences imbriquées, de la manière suivante :

$$f_{91}(n) = \begin{cases} n-10 & \text{si } n > 100, \\ f_{91}(f_{91}(n+11)) & \text{si } n <= 100. \end{cases}$$

 **Activité :** Déterminer la fonction f_{91}de manière récursive.

```
def f91(n):



# Exemple de test
for i in range(90, 110):
   print(f"f91({i}) = {f91(i)}")
```

Récursion mutuelle.

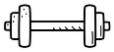
Il est également possible, et parfois nécessaire, de définir plusieurs fonctions récursives en même temps, quand ces fonctions font référence les unes aux autres. On parle alors de définitions récursives mutuelles.

Par exemple, les fonctions F(n) et M(n) ci-dessous, inventées par Douglas Hofstadter sont définies par récursion mutuelle de la manière suivante :

Ces fonctions sont définies en tandem et sont basées sur une dépendance mutuelle :

$$F(n) = \begin{cases} 0 & \text{si } n = 0, \\ n - M(F(n-1)) & \text{si } n > 0. \end{cases}$$

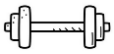
$$M(n) = \begin{cases} 1 & \text{si } n = 0, \\ n - F(M(n-1)) & \text{si } n > 0. \end{cases}$$

 **Activité :** Déterminer les fonctions hofstadter_female(n) et hofstadter_male(n) de manière récursive.

```
def hofstadter_female(n):




def hofstadter_male(n):
```

 **Activité :** Exercices 8,9,10,11 et 12.

Limites de la programmation récursive

1. Consommation de mémoire (pile d'exécution)
Chaque appel récursif crée une nouvelle entrée dans la pile d'exécution. Cela peut entraîner un dépassement de la pile (stack overflow) si la profondeur de la récursion est trop importante.

2. Risque de boucles infinies
Si la condition de sortie (ou base) n'est pas correctement définie ou atteinte, la fonction récursive peut se retrouver dans une boucle infinie, entraînant une consommation excessive de mémoire.

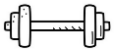
3. Complexité en temps
Certaines fonctions récursives peuvent recalculer plusieurs fois les mêmes résultats, ce qui peut causer une augmentation substantielle du temps d'exécution. Par exemple, la récursion naïve pour le calcul des nombres de Fibonacci : des sous-problèmes sont recalculés plusieurs fois.

4. Difficulté de compréhension
Pour certaines personnes, les programmes récursifs peuvent être plus difficiles à comprendre et à déboguer.

5. Performances (surcharge des appels)
Les appels de fonctions récursives peuvent être moins efficaces que les boucles itératives, car chaque appel de fonction ajoute une surcharge liée à la gestion de la pile d'exécution (sauvegarde du contexte, allocation de la pile, etc.).

6. Optimisation limitée
Sans optimisation de récursion terminale, la gestion de la pile peut être coûteuse.

En conclusion, bien que la récursion puisse apporter des solutions élégantes et concises pour certains problèmes, elle doit être utilisée avec précaution en raison de ces inconvénients, notamment en ce qui concerne la gestion de la mémoire et la performance.

 **Préparer une fiche de synthèse de la leçon ainsi qu'une carte mentale.**