Algorithme récursif

Introduction :

Dans ce cours nous nous apprêtons à étudier un concept important et parfois déroutant de la programmation informatique : celui de la récursivité, c'est-à-dire de la capacité d'une fonction ou d'un algorithme à s'appeler lui-même. Nous présenterons dans un premier temps ce principe et ses caractéristiques générales. Nous analyserons dans un deuxième temps les mécanismes qui permettent de le faire fonctionner ; et nous terminerons par une étude d'implémentations récursives.

1 Principe général

Commençons par un bref rappel sur les fonctions.

Rappel

Une fonction se définit de la manière suivante en Python :

```
def accueil():
    print('bonjour')
```

Elle s'appelle ensuite de la manière suivante :

```
accueil()
# affiche bonjour
```

On rappelle également que les fonctions peuvent comporter un ou plusieurs appels à d'autres fonctions. Notre fonction `accueil()` fait ainsi appel à la fonction `print()`, et pourrait de la même manière appeler toute autre fonction, qu'elle soit native ou définie par nous.

Ces rappels étant effectués, nous allons nous pencher sur le cas particulier d'une fonction qui, au lieu d'appeler une autre fonction, essaie de s'appeler elle-même.

a. Appels de fonction

Nous modifions notre fonction `accueil()` en ajoutant un appel à elle-même dans sa définition, après l'appel à `print()` :

```
def accueil():
    print('bonjour')
    accueil()
```

Observons maintenant ce qui se produit lors de l'appel de notre fonction ainsi modifiée.

Attention

> L'ordinateur va produire un grand nombre d'affichages avant d'afficher une erreur.

```
accueil()
```

affiche un très grand nombre de fois...

```
bonjour
bonjour
bonjour
bonjour
bonjour
bonjour
bonjour
bonjour
bonjour
bonjour
```

et finit par afficher le message suivant :

```
RecursionError: maximum recursion depth exceeded while calling a Python object
```

Le nombre exact de lignes d'affichage dépend de la configuration du langage Python utilisée, mais il est généralement de l'ordre de quelques milliers de lignes.

→ Si on fait abstraction de l'erreur qui a mis fin au programme, on observe que notre fonction a bien réussi à s'appeler elle-même, produisant un comportement assez analogue à celui d'une boucle infinie.

En réalité si le programme finit par s'arrêter en produisant une erreur, c'est parce qu'un mécanisme de protection a été intégré au langage pour limiter le nombre d'appels. Sans ce mécanisme que nous étudierons dans la deuxième partie du cours, la fonction s'appellerait elle-même **à l'infini jusqu'à saturation de la mémoire**.

Redéfinissons maintenant notre fonction avec la prise en compte d'un paramètre :

```
def accueil(prenom):
    print('bonjour', prenom)
    accueil(prenom)
```

Observons maintenant ce qui se produit lors de l'appel de notre fonction avec un argument.

```
accueil('Guido')
```

affiche un très grand nombre de fois :

```
bonjour Guido
bonjour Guido
bonjour Guido
bonjour Guido
bonjour Guido
bonjour Guido
bonjour Guido
```

et finit par afficher :

```
RecursionError: maximum recursion depth exceeded while calling a Python object
```

Là encore une erreur s'affiche après un grand nombre d'affichages, mais on constate que l'argument passé en paramètre a bien été pris en compte. Notre fonction est **récursive**.

b. Définition

Définition

> **Fonction récursive :**
>
> Une fonction récursive est une fonction qui s'appelle elle-même.

À retenir

> Autrement dit, une fonction récursive est une fonction qui s'**auto-référence**.

Notre fonction récursive actuelle n'est guère utile et finit par générer une erreur. Mais elle nous a permis d'illustrer qu'une fonction pouvait s'appeler elle-même. Ainsi que nous allons le découvrir dans la suite du cours, cette caractéristique peut s'avérer très intéressante.

Astuce

> On dit aussi que « pour bien comprendre la récursivité, il faut bien comprendre la récursivité », ce qui revient à donner une définition récursive de la récursivité.

Les références récursives de ce genre sont assez répandues en informatique.

→ Le sigle PHP est un sigle récursif, P.H.P. signifiant « PHP : Hypertext Preprocessor ».

Exemple

Certaines représentations graphiques illustrent le principe récursif, comme les images fractales ou le logo bien connu du fromage Vache qui rit, qui est mis en abyme dans les boucles d'oreille portées par la vache.

Intéressons-nous maintenant de plus près aux caractéristiques d'une fonction récursive utilisable sans erreur.

c. Caractéristiques d'une fonction récursive

Une fonction récursive se compose de deux parties :

- une **partie récursive** (également appelée cas récursif) ;
- une **partie terminale** (également appelée cas de base ou cas d'arrêt), destinée à stopper les appels récursifs quand une condition est atteinte.

La partie récursive de la fonction comporte un **appel récursif**. C'est la partie récursive qui permet à la fonction de s'auto-référencer.
La partie terminale d'une fonction récursive définit les conditions de terminaison de la fonction, de manière assez analogue aux conditions de sorties des boucles non bornées.

Notre fonction expérimentale ne comportait pas de partie terminale ou cas de base, ce qui la conduisait à s'appeler un nombre infini de fois. Elle ne comportait pas non plus de paramètres dans sa définition initiale, mais une fonction récursive peut tout à fait en posséder, comme n'importe quelle fonction, ainsi que nous l'avons constaté dans sa version modifiée. Dans une fonction récursive opérationnelle, la ou les valeurs passées en argument évoluent d'un appel à l'autre, jusqu'à finir par remplir la ou les conditions attendues par la partie terminale.

À retenir

> Une fonction récursive est obligatoirement composée d'une partie récursive et d'une partie terminale.
>
> - Si la partie terminale est manquante, la fonction s'appelle un nombre infini de fois.
> - Si la partie récursive est manquante, la fonction n'est pas récursive.

Définissons de manière récursive une fonction qui raccourcit une chaîne de caractères pour n'en conserver que la première lettre :

- La partie terminale, ou cas de base, vérifiera si la condition voulue est atteinte : une chaîne de caractères qui en comporte un seul.
- La partie récursive effectuera un appel à la fonction avec en paramètre une chaîne de caractères tronquée de son caractère final, et en retournera le résultat.

```
def raccourcit(chaine):
    if len(chaine) == 1:
        print(chaine)
    else:
        return raccourcit(chaine[:-1])

raccourcit('hello world')
# affiche h

raccourcit('z')
# affiche z
```

Nous obtenons bien le résultat attendu. Toutefois notre fonction est en l'état une procédure qui ne retourne rien : elle se borne à imprimer la chaîne finalement obtenue à l'issue des appels récursifs successifs.

Adaptons donc notre fonction pour qu'elle retourne les résultats obtenus :

```
def raccourcit(chaine):
    if len(chaine) == 1:
        return chaine
    else:
        return raccourcit(chaine[:-1])
```

```
print(raccourcit('hello world'))
# affiche h

print(raccourcit('z'))
# affiche z
```

À nouveau, notre fonction raccourcit bien les chaînes de caractères qu'on lui soumet, et retourne cette fois le caractère correspondant.

Toutefois nous n'avons pas prévu le cas d'une chaîne vide. Si nous appelons la fonction avec une chaîne vide avec `print(raccourcit(''))`, son exécution dure plusieurs secondes et finit par générer une erreur de type `RecursionError`. Analysons pourquoi.

Lors de l'appel initial, la longueur de la chaîne est évaluée. Comme elle ne vaut pas 1, nous ne sommes pas dans le cas terminal. La fonction passe alors dans la partie récursive, et effectue un appel sur la chaîne reçue en argument, tronquée de son dernier caractère.

→ Comme la troncature du dernier élément d'une chaîne vide est une chaîne vide, l'appel récursif n'évolue pas et s'effectue en permanence sur une chaîne vide.

Nous comprenons que, dans ces conditions, la fonction n'atteindra jamais la condition terminale où la longueur de chaîne est de longueur 1.
Une légère modification de la partie terminale de notre fonction permet de pallier ce problème : **au lieu d'une égalité stricte, nous retournons toute chaîne de longueur inférieure ou égale à 1 caractère**.

```
def raccourcit(chaine):
    if len(chaine) <= 1:
        return chaine
    else:
        return raccourcit(chaine[:-1])

print(raccourcit(''))
# n'affiche rien mais ne produit pas d'erreur
```

On vérifie que la fonction retourne bien un résultat et on évalue sa nature.

```
resultat = raccourci('')
print(resultat, len(resultat), type(resultat))
# affiche 0
```

Comme avec les conditions de sortie des boucles non bornées, une attention particulière doit être portée à la manière dont les conditions de terminaison sont évaluées.

À retenir

> L'appel effectué dans la partie récursive doit faire évoluer le paramètre de départ, sinon il n'y a pas de convergence vers la partie terminale.
> La partie terminale doit prendre en compte l'ensemble des cas de base possibles devant conduire à la terminaison de la fonction.

Maintenant que nous avons découvert les principes et caractéristiques générales de la récursivité des fonctions, intéressons-nous de plus près aux mécanismes qui en permettent l'usage.

2 | Fonctionnement récursif

Dans cette partie nous nous intéressons à la manière dont l'interpréteur Python gère la récursivité. Afin d'illustrer concrètement le fonctionnement, prenons appui sur le calcul de la factorielle d'un nombre.

Rappel

> En mathématique, la factorielle d'un nombre n, notée $n!$, est le produit de l'ensemble des entiers strictement positifs inférieurs ou égaux à n.
>
> → Ainsi factorielle 5 (notée $5!$) est égale à $5 \times 4 \times 3 \times 2 \times 1$.
>
> → $5! = 120$

Attention

> Notez la valeur particulière du cas $0! = 1$

a. Implémentation récursive

Considérons les caractéristiques mathématiques de $5!$:
$5! = 5 \times 4 \times 3 \times 2 \times 1$
$4 \times 3 \times 2 \times 1$ est égal à $4!$

Nous pouvons donc réécrire la première ligne ainsi :
$5! = 5 \times 4!$

$4!$ est égal à $4 \times 3 \times 2 \times 1$, que nous pouvons aussi écrire ainsi :
$4! = 4 \times 3!$

De la même manière :
$3! = 3 \times 2!$
$2! = 2 \times 1!$

Et par définition $1! = 1$

Nous disposons des éléments requis pour implémenter une version récursive du calcul de la factorielle d'un nombre :

- la partie terminale est $1! = 1$;
- la partie récursive s'applique ainsi à tout $n > 1$:
  $n! = n \times (n - 1)!$

→ L'implémentation en Python en est une transcription assez littérale.

```
def factorielle_recursive(nombre):
    if nombre == 1:
        return nombre
    else:
        return nombre * factorielle_recursive(nombre - 1)
```

Notre fonction produit bien le résultat attendu :

```
print(factorielle_recursive(5))
# affiche 120
```

Astuce

> L'instruction `return` termine l'exécution d'une fonction. Le recours à la clause `else` n'est donc pas nécessaire par rapport à notre instruction `if`.

On peut alléger notre code ainsi :

```
def factorielle_recursive(nombre):
    if nombre == 1:
        return nombre
    return nombre * factorielle_recursive(nombre - 1)
```

Notre fonction n'effectue aucune vérification sur le type ni sur la valeur entrée. On peut *a minima* modifier notre condition terminale :

```
if nombre == 1:
```

en :

```
if nombre <= 1:
```

Cela nous permet de calculer $0!$ (égal à 1), mais aussi d'empêcher des appels récursifs pour des nombres négatifs, dépourvus de sens mathématiques et qui conduiraient à une `RecursionError`. Idéalement on complèterait les vérifications avec la levée d'une exception pour toute saisie de nombre négatif (`ValueError`) ou pour tout type de valeur non entière (`TypeError`).

Même quand l'implémentation semble triviale, il est judicieux de bien réfléchir aux modalités quand plusieurs choix sont possibles.

b. Suivi des appels de fonction

Afin de pouvoir suivre le fonctionnement de notre programme récursif, nous insérons des affichages à différents endroits.

Astuce

> **Remarque :**

> Nous pourrions aussi utiliser un décorateur, une construction qui permet de modifier le comportement d'une fonction, mais leur emploi sort du cadre de ce cours.

Nous utilisons ici une variable globale qui nous servira à augmenter le décalage des messages affichés à chaque nouvel appel récursif, afin d'illustrer la cascade d'appels.

```
decalage = ' ' * 5
i = 0

def factorielle_recursive(nombre):
    global i
    print(decalage * i, 'réception d\'une demande de calcul de {}!'.format(nombre))
    if nombre <= 1:
        print(decalage * i, ' > cas terminal atteint (retourne 1) ')
        return nombre
    print(decalage * i, ' > cas récursif : appel de {}!, requis pour pouvoir retourner ensuite {}!\n'.format(nombre - 1, nombre))
    i += 1
    return nombre * factorielle_recursive(nombre - 1)

print(factorielle_recursive(5))
```

produit l'affichage suivant :

```
réception d'une demande de calcul de 5!
    > cas récursif : appel de 4!, requis pour pouvoir retourner ensuite 5!
        réception d'une demande de calcul de 4!
        > cas récursif : appel de 3!, requis pour pouvoir retourner ensuite 4!
            réception d'une demande de calcul de 3!
            > cas récursif : appel de 2!, requis pour pouvoir retourner ensuite 3!
                réception d'une demande de calcul de 2!
                > cas récursif : appel de 1!, requis pour pouvoir retourner ensuite 2!
                réception d'une demande de calcul de 1!
                > cas terminal atteint (retourne 1)

120
```

On observe des appels en cascade : chaque calcul de la factorielle d'un nombre donné reste suspendu en attente du résultat de celle de valeur immédiatement inférieure. L'interpréteur Python est donc contraint de garder la trace de chacun de ces appels en cascade.

→ Le mécanisme qui permet de suivre ces appels successifs et les retours attendus s'appelle une pile d'exécution.

c. Pile d'exécution

Définition

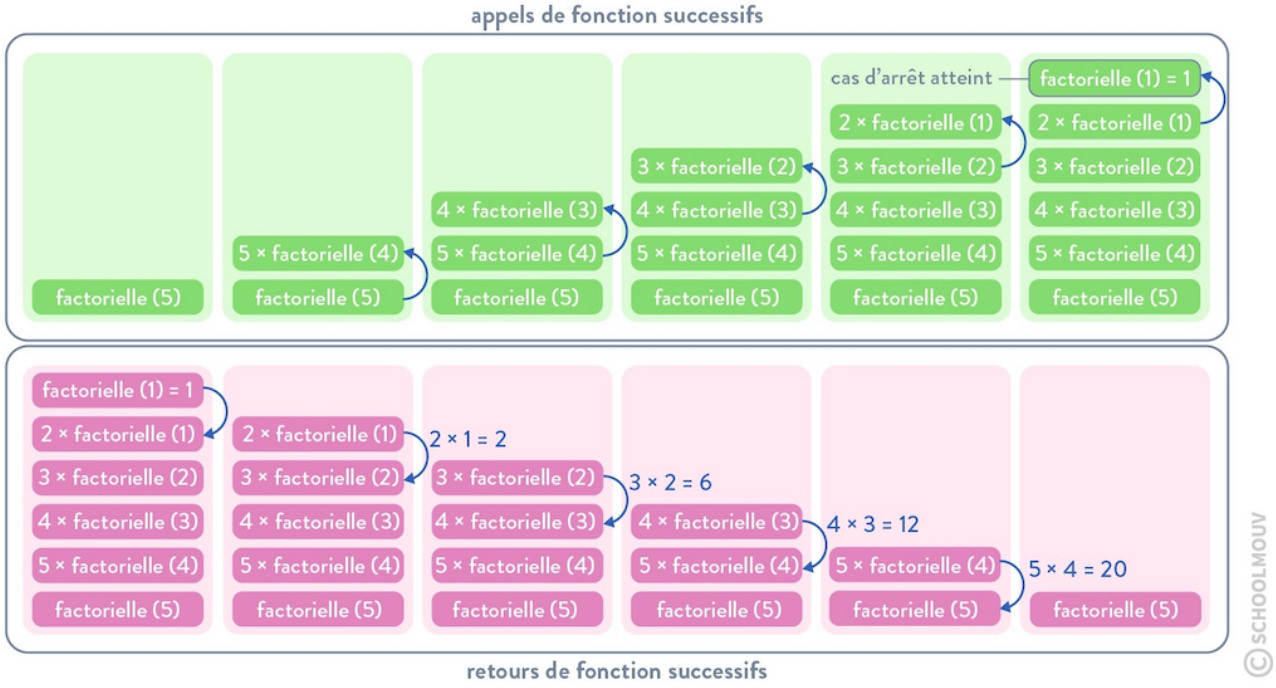
> **Pile d'exécution :**
>
> Une pile d'exécution est une structure de données chargée de garder la trace des appels de fonction successifs.

La **pile d'exécution** (ou *call stack* en anglais) est une pile type LIFO.

Définition

> **LIFO :**
>
> Le sigle LIFO signifie Last In, First Out. En français : dernier entré, premier sorti.
> Le dernier élément ajouté à la pile (ou « empilé ») sera donc le premier à en être sorti (ou « dépilé »).

La pile d'exécution **empile** les appels de fonction successifs correspondant aux cas récursifs, permettant à l'interpréteur Python de garder la trace des appels individuels pour pouvoir, le moment venu, gérer les retours correspondants. Une fois le cas terminal atteint, la pile est progressivement **dépilée** et chaque appel reçoit en retour le résultat qu'il attendait.



Ces appels étant potentiellement gourmands en mémoire, les langages qui permettent la récursivité de fonctions ou de programmes limitent le nombre d'appels récursifs.

Définition

> **Dépassement de pile :**
>
> Le dépassement ou débordement de pile est une erreur qui survient quand le nombre d'appels à stocker dans la pile dépasse la capacité de cette dernière.

→ En Python le débordement de pile génère l'erreur spécifique `RecursionError`.

L'expression anglaise équivalente est *stack overflow*. Cette erreur, redoutée des programmeur·se·s, est aussi le nom d'un célèbre site anglophone de questions/réponses ayant trait à la programmation informatique (http://www.stackoverflow.com/).

Maintenant que nous avons décrit les mécanismes permettant la récursivité, intéressons-nous aux implémentations récursives d'algorithmes.

3 Implémentations récursives

Dans cette partie nous nous intéressons aux implémentations récursives, dont nous étudierons différentes formes et cas d'usage. Nous les comparerons ensuite aux implémentations itératives.

a. Types d'implémentation récursive

Il existe différents types d'implémentations récursives. Les exemples que nous avons présentés jusqu'à présent étaient des cas de **récursivité simple**, également appelée **récursivité linéaire**. Mais la récursivité peut prendre d'autres formes.

Définition

> **Récursivité terminale :**
>
> Une fonction f récursive est dite « terminale » (*tail recursion* en anglais) si la valeur qu'elle retourne est directement la valeur obtenue par un appel récursif, sans qu'il n'y ait aucune opération sur cette valeur. Autrement dit, si la valeur retournée par f est de la forme `return f(...)`

Certains langages détectent et optimisent les implémentations de **récursivité terminale**.
En ce qui concerne plus particulièrement le langage Python, il n'a pas fait l'objet d'optimisations poussées de ses facultés récursives. L'auteur de Python, Guido van Rossum, ne souhaitait pas en étendre l'usage et s'en est expliqué à plusieurs reprises.

Définition

> **Récursivité mutuelle :**
>
> Une récursivité est dite « mutuelle » ou « croisée » lorsqu'elle est rendue possible par deux ou plusieurs fonctions effectuant des appels mutuels.

Nous créons deux fonctions qui retirent un caractère d'extrémité d'une chaîne : au début de la chaîne pour l'une, et à la fin de la chaîne pour l'autre. Le cas terminal est le caractère restant.

```
def grignote_debut(chaine):
    if len(chaine) < 2:
        return chaine
    else:
        return grignote_fin(chaine[1:])

def grignote_fin(chaine):
    if len(chaine) < 2:
        return chaine
    else:
        return grignote_debut(chaine[:-1])
```

Les deux fonctions sont **mutuellement récursives** : la récursivité est créée par les appels croisés d'une fonction à l'autre.

→ Il s'agit donc bien d'une **récursivité mutuelle** (ou croisée).

```
print(grignote_debut('récursivité'))
# affiche s

print(grignote_fin('récursivité'))
# affiche s
```

Le résultat est identique qu'on commence par le début ou la fin car la chaine est de longueur impaire. Ce n'est en revanche pas le cas avec une chaine de longueur paire.

```
print(grignote_debut('python'))
v# affiche h

print(grignote_fin('python'))
# affiche t
```

Définition

> **Récursivité multiple :**
>
> Une récursivité est dite « multiple" lorsqu'elle comporte plusieurs appels récursifs à eux-mêmes.

Le calcul d'un terme de la suite de Fibonacci est un exemple classique de **récursivité multiple**.
Les nombres de cette suite sont la somme des deux termes qui le précèdent. Sa définition mathématique par récurrence est $F(n) = F(n-1) + F(n-2)$, avec les deux premiers termes définis ainsi :

L'implémentation récursive de l'algorithme est assez immédiate :

```
def fibonacci(nombre):
    if nombre < 2:
        return nombre
    return fibonacci(nombre - 1) + fibonacci(nombre - 2)
```

Chaque exécution de la partie récursive entraîne deux appels récursifs, caractérisant une récursivité multiple. Le calcul récursif de la suite de Fibonacci est étudié de manière détaillée dans le cours sur la programmation dynamique.

b. Choix d'une implémentation récursive

L'implémentation récursive n'est pas une obligation mais un choix. En effet tout algorithme itératif peut être écrit en récursif et vice-versa.

Dans la partie précédente, nous avions créé une implémentation récursive du calcul de la factorielle d'un nombre.

```
def factorielle_recursive(nombre):
    if nombre <= 1:
        return nombre
    return nombre * factorielle_recursive(nombre - 1)
```

Nous pouvons implémenter le calcul de la factorielle par itération.

```
def factorielle_iterative(nombre):
    resultat = 1
    for i in range(1, nombre + 1):
        resultat *= i
    return resultat
```

Le choix entre une implémentation itérative ou récursive dépend de plusieurs facteurs :

- la facilité d'implémentation, plus ou moins évidente dans une forme ou une autre selon le cas ; parfois l'implémentation récursive est intuitive, parfois l'implémentation itérative l'est davantage ;
- les contraintes de mémoire et de performance ; la récursivité implique une consommation d'espace mémoire parfois importante pour le suivi des appels récursifs ;
- la nature des données sur lesquelles les traitements sont effectués ; par leur nature récursive, les systèmes de fichiers ou les conteneurs de données de type listes se prêtent particulièrement bien à des approches récursives.

 Exemple

> On souhaite créer une fonction capable d'aplatir des listes imbriquées entre elles, c'est-à-dire d'extraire les données atomiques contenues dans une liste pouvant être composée de listes imbriquées sur plusieurs niveaux, afin de les réunir dans une seule et même liste.
> Dans ce cas précis, l'implémentation récursive est assez intuitive :

```
def aplatissement(liste):
    resultat = []
    for element in liste:
        if isinstance(element, list):
            resultat += aplatissement(element)
        else:
            resultat += [element]
    return resultat

print(aplatissement([1, 2, [3, 4], [5, [6, 7, 8]], 9]))
# affiche [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

D'autres facteurs peuvent également entrer en ligne de compte :

- l'aisance du·de la développeur·se avec l'une ou l'autre approche ;

- les pratiques logicielles en vigueur dans une organisation ou un secteur d'activité.

À retenir

> La récursivité étant possible dans tous les langages de programmation généraliste et implémentée dans un certain nombre d'algorithmes, tout·e développeur·se doit connaître et maîtriser les bases de la récursivité.

Conclusion :

Nous avons présenté le concept de récursivité des fonctions, composées d'une partie récursive et d'une partie terminale. Nous avons ensuite détaillé la manière dont ces appels récursifs étaient accumulés dans une pile d'exécution jusqu'à l'obtention du résultat attendu. Enfin nous avons étudié différents types de récursivité et les facteurs pouvant influer sur le choix d'une implémentation itérative ou récursive.