

Fonction récursive

1 | Rappel : définir et appeler une fonction

La **définition** d'une fonction, introduite par `def`, indique les opérations à effectuer et nomme ses paramètres. Elle ne calcule encore aucun résultat. Un **appel** exécute ces opérations avec les valeurs données aux paramètres.

```
def double(x):
    return 2 * x
```

Une fois la fonction définie, elle peut être utilisée. Voici plusieurs appels de cette même fonction :

- `double(3)` renvoie 6 : le paramètre `x` reçoit la valeur 3 ;
- `double(5)` renvoie 10 : le paramètre `x` reçoit la valeur 5 ;
- ou avec une variable :

```
a = 4
print(double(a))  # renvoie puis écrit dans la console 8
```

2 | Les fonctions récursives

Certaines définitions ramènent un problème au même problème, mais à des rangs inférieurs. Par exemple, la somme des entiers de 0 à n est définie par

$$S(n) = \sum_{k=0}^{n} k$$

Elle vérifie la relation

$$S(0) = 0 \qquad \text{et} \qquad S(n) = n + S(n-1) \quad \text{pour } n \geq 1 \qquad \text{car} \qquad \sum_{k=0}^{n} k = n + \sum_{k=0}^{n-1} k$$

Définition

Fonction récursive

Une fonction **récursive** est une fonction qui **s'appelle elle-même**.
Sa définition doit comporter

- au moins un **cas de base**, dont le résultat est connu sans nouvel appel ;
- un **cas récursif**, qui ramène le calcul à une entrée plus petite.

Exemple

```
def somme(n):              # precondition : n est un entier >= 0
    if n == 0:             # cas de base
        return 0
    else :
        return n + somme(n - 1)  # cas recursif
```

L'instruction `return` met fin à l'appel courant. Le `else` après ce premier `return` est donc facultatif.

2.1 Implémentation

La **conception** et la définition d'une **fonction récursive** sont à rapprocher de la **récurrence en mathématiques** :

- La fonction récursive f doit prendre en charge un paramètre d'appel n, **décrémenté à chaque appel de la fonction**.
- La fonction doit prévoir un **cas de base** (condition d'arrêt ou cas simple), $f(0)$ par exemple, pour lequel la fonction renvoie un résultat sans nécessiter un appel récursif.
- La fonction doit prévoir un **appel récursif** (conduisant à la terminaison), $f(n-1)$ par exemple.

On peut donc définir la **structure générale** suivante (les valeurs à renvoyer sont à adapter au problème) :

```
def fct(n):
    if n == 0:              # cas de base
        return valeur
    else:
        return fct(n - 1)   # appel recursif
```

2.2 Suivre les appels et les retours

Reprenons l'exemple de la somme :

```
def somme(n):             # precondition : n est un entier >= 0
    if n == 0:           # cas de base
        return 0
    else :
        return n + somme(n - 1)  # cas recursif
```

- Pour calculer $S(3)$, on calcule d'abord $S(2)$, qui nécessite $S(1)$, et ainsi de suite jusqu'à $S(0)$.
- **Chaque appel dispose de ses propres variables locales** : le `n` de `somme(3)` vaut toujours 3 pendant que `somme(2)` s'exécute avec son propre `n` égal à 2.
- L'appel `somme(3)` doit attendre la valeur de `somme(2)` pour calculer $3+S(2)$.
- Les appels en attente sont conservés dans la **pile d'appels**.

Descente : chaque colonne ajoute un appel en attente ; le cas de base est atteint à la dernière colonne.

1^{er} appel ($n=3$)	2^{e} appel ($n=2$)	3^{e} appel ($n=1$)	4^{e} appel ($n=0$)
$S(3)=?$	$S(3)=?$	$S(3)=?$	$S(3)=?$
	$S(2)=?$	$S(2)=?$	$S(2)=?$
		$S(1)=?$	$S(1)=?$
			$S(0)=0$

Lorsque $S(0)$ est connu, les appels en attente sont résolus dans l'ordre inverse : c'est la **remontée** :

1^{er} appel ($n=3$)	2^{e} appel ($n=2$)	3^{e} appel ($n=1$)	4^{e} appel ($n=0$)
$S(3)=$	$S(3)=?$	$S(3)=?$	$S(3)=?$
	$S(2)=$	$S(2)=?$	$S(2)=?$
		$S(1)=$	$S(1)=?$
			$S(0)=0$

3 | Approche itérative ou récursive ?

Considérons la suite définie par

$$\begin{cases} u_0 = 1, \\ u_{n+1} = \sqrt{1 + u_n}, \quad n \in \mathbb{N}. \end{cases}$$

Les deux fonctions suivantes calculent le même terme.

Version itérative

```python
from math import sqrt

def terme_iteratif(n):
    u = 1
    for _ in range(n):
        u = sqrt(1 + u)
    return u
```

Dans la version itérative, u vaut d'abord $u_0 = 1$, puis chaque tour de boucle calcule le terme suivant. C'est-à-dire qu'on calcule les termiens les uns après les autres en **partant du premier**.

Version récursive

```python
from math import sqrt

def terme_recursif(n):
    if n == 0:
        return 1
    precedent = terme_recursif(n - 1)
    return sqrt(1 + precedent)
```

Dans la version récursive, on calcule le n-ième terme par rapport au terme précédent. Autrement dit, on **commence par le le n-ième terme** et on remonte au premier, qui est le seul connu.

Question 1 : Compléter le tableau de suivi de `terme_recursif(2)` : indiquer l'appel effectué par chaque fonction, puis la valeur qu'elle renvoie.

Appel en cours	Appel suivant (descente)	Remontée
`terme_recursif(2)`		$u_2 =$
`terme_recursif(1)`		$u_1 =$
`terme_recursif(0)`	Aucun	$u_0 =$

3.1 Avantages et inconvénients

3.1.1 Avantage

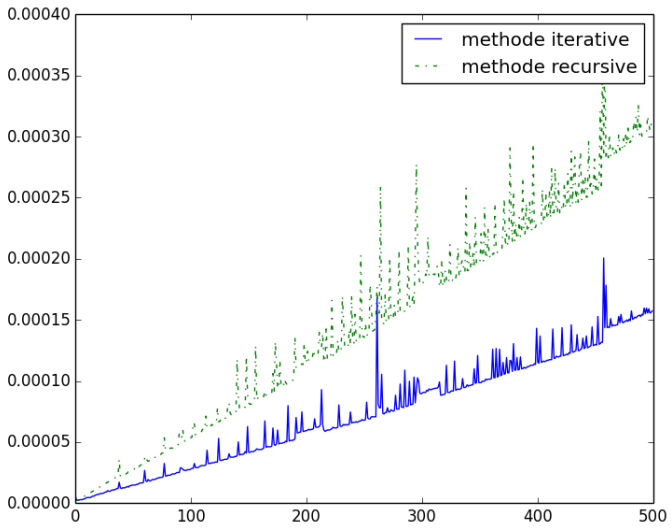
Lorsque le choix se présente, **une fonction récursive n'est pas forcément de meilleure efficacité qu'une fonction itérative bien écrite**. Le choix de définir une fonction sous sa forme récursive tient souvent de la **simplicité à traduire directement le problème mathématique** étudié sous cette forme et de sa **lisibilité**.

3.1.2 Inconvénients

Complexité temporelle – temps de calcul

Pour la suite u_n ci-dessus, les deux versions effectuent n calculs de racine : leur complexité temporelle est donc $O(n)$. **La version récursive ajoute** cependant la **gestion** de $n+1$ **appels de fonction**. Sur les mesures ci-dessous, réalisées pour cet exemple, elle prend davantage de temps que la version itérative.

Les durées exactes et les irrégularités du tracé dépendent bien sûr de l'ordinateur et des conditions de mesure mais la logique persiste : **les appels nécessitent des manipulations de mémoire** qui coûte forcément du temps.



Comparaison expérimentale des temps de calcul des deux versions.

Complexité spatiale - mémoire utilisée
La version itérative ne conserve qu'un nombre fixe de variables : elle utilise une mémoire supplémentaire en $O(1)$. La version récursive conserve les appels en attente dans une **pile d'appels** ; pour calculer u_n, sa profondeur est proportionnelle à n, d'où une mémoire supplémentaire en $O(n)$.
Si cette pile devient trop grande, Python interrompt le calcul avec une **erreur de profondeur de récursion**. Autrement dit, il y a des sécurités pour éviter de saturer l'espace mémoire.

4 Terminaison et correction

4.1 Rappels

- **Variant de boucle :** quantité entière, positive ou nulle tant que la boucle peut recommencer, et qui décroît strictement à chaque itération. Il sert à prouver que la boucle **termine** : une suite d'entiers naturels strictement décroissante ne peut pas être infinie.
- **Invariant de boucle :** propriété des variables vraie avant la première itération et qui reste vraie après chaque itération. Combinée à la condition de sortie, elle sert à démontrer que le résultat calculé est **correct**.

Pour une fonction récursive, on recherche de même une quantité qui décroît à chaque appel et une propriété que les valeurs renvoyées doivent vérifier.

> **Remarque :** Il arrive très souvent avec les fonctions récursives qu'on puisse prouver la terminaison et la correction avec la même récurrence.

4.2 Terminaison

La preuve se mène comme celle d'une boucle `while` et utilise souvent comme **variant de boucle le paramètre d'appel qui est décrémenté à chaque appel récursif** de la fonction.

Exemple

Pour la fonction somme arithmétique : n est un variant de boucle, comme entier positif dont les valeurs successives $n-1$ prises à chaque appel forment une suite de valeurs strictement décroissantes.

4.3 Correction

La preuve se mène **souvent par récurrence** sur un variant bien choisi :

- **Initialisation :** On montre que le résultat est correct pour le (ou les) cas de base.
- **Transmission :** On admet que si le résultat renvoyé est correct pour k (ou $k-1$) alors il l'est encore pour $k+1$ (ou k).
- **Sortie :** Elle est généralement immédiate si le variant a bien été choisi car on connaît alors l'expression de $f(n)$.

Exemple

Pour la fonction somme arithmétique : on choisit comme propriété $P(n)$: pour tout $n \in \mathbb{N}$, `somme(n)` renvoie $\sum_{k=0}^{n} k$.

a) **Initialisation :** Pour $n = 0$, la fonction exécute `return 0` donc la propriété est vérifiée.

b) **Transmission :** On admet que $P(n)$ est vraie et que `somme(n)` renvoie $\sum_{k=0}^{n} k$. Donc, pour $n+1$, la fonction renvoie $(n+1) + \sum_{k=0}^{n} k = \sum_{k=0}^{n+1} k$.

c) **Sortie :** Lorsque la fonction termine, on a `somme(n)` $= \sum_{k=0}^{n} k$.

Remarques :

- Si l'appel récursif fait intervenir $f(n-1)$, une récurrence simple suffit généralement.
- Si l'appel récursif fait intervenir $f(n-1)$ et $f(n-2)$, une récurrence double est nécessaire généralement.
- Si l'appel récursif fait intervenir $f(k)$ avec $k < n$, k non déterminé, alors une récurrence forte est généralement nécessaire.

5 Exercices d'application

Pour construire une fonction récursive, préciser sa **précondition**, son **cas de base** et son **appel vers un cas plus petit**. Vérifier que cet appel atteint le cas de base et que le résultat est correct.

5.1 Factorielle

Question 2 : Écrire une fonction récursive `factorielle(n)` qui renvoie $n!$ pour tout entier $n \geq 0$. Préciser le cas de base.

Question 3 : Montrer la correction totale de cette fonction, c'est-à-dire qu'elle termine et qu'elle est correcte.

Question 4 : Compléter le tableau d'appels pour `factorielle(3)`.

Appel en cours	Appel suivant	Valeur renvoyée
`factorielle(3)`		
`factorielle(2)`		
`factorielle(1)`		
`factorielle(0)`	Aucun	

5.2 Puissance

Question 5 : Écrire une fonction récursive `puissance(x, n)` qui renvoie x^n pour $n \in \mathbb{N}$. Combien de multiplications effectue-t-elle ?

Question 6 : Montrer la correction totale de `puissance(x, n)` pour $n \in \mathbb{N}$.

5.3 Somme d'une liste

Question 7 : Écrire une fonction récursive `somme_prefixe(L, n)` qui renvoie la somme des n premiers éléments d'une liste numérique `L`, avec $0 \leq n \leq$ `len(L)`.

Question 8 : Pour `L = [2, 4, 1]`, compléter le tableau des appels de `somme_prefixe(L, 3)` et des valeurs renvoyées lors de la remontée.

Valeur de `n`	Appel suivant	Valeur renvoyée
3		
2		
1		
0	Aucun	

5.4 Plusieurs appels récursifs

La suite de Fibonacci est définie par $F_0 = 0$, $F_1 = 1$ et $F_n = F_{n-1} + F_{n-2}$ pour $n \geq 2$.

Question 9 : Écrire une fonction récursive `fibonacci(n)` qui renvoie F_n pour $n \in \mathbb{N}$. Identifier les cas de base et le cas récursif dans le code.

Question 10 : Déterminer combien de fois `fibonacci(2)` est appelé pendant le calcul de `fibonacci(4)`. Pour répondre à cette question, réaliser un tableau des appels.