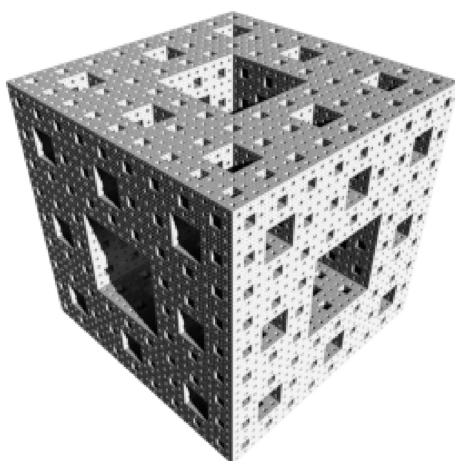
Chapitre 1

Récursivité



Le mot **récursivité** vous fait certainement penser au mot **récurrence** que vous avez déjà sûrement rencontré en Mathématiques : les termes sont proches et la différence entre les deux minimes.
Tout comme on peut définir une suite par récurrence, on définit une **fonction récursive** de la manière suivante :

> **Définition.** Une fonction est dite **récursive** si elle s'appelle elle-même.
> On parle alors d'**appel récursif** de la fonction.

Dans ce cours, nous allons découvrir comment programmer des fonctions récursives et dans quels cas les utiliser.

1.1 Fonctions récursives

Considérons la suite mathématique suivante :

$$\begin{cases} u_0 & = & 3 \\ u_{n+1} & = & 2u_n - 2 \end{cases}$$

Pour calculer le terme u_{10}, il est nécessaire de connaître u_9 (puisque $u_{10} = 2u_9 + 2$), mais pour connaître u_9 est il nécessaire de connaître u_8 et ainsi de suite... Classiquement, on se sert d'une **boucle itérative** pour calculer la valeur de u_{10} :

```
u = 3

for i in range(10):
    u = 2*u - 2

print(u)
```

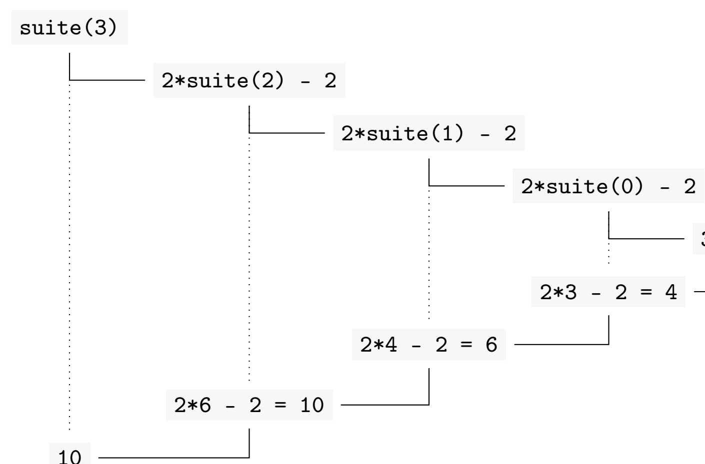
On peut même définir une fonction `suite(n)` qui retourne la valeur de u_n pour tout entier n :

```
def suite(n):
    u = 3

    for i in range(n):
        u = 2*u - 2

    return u
```

Cette fonction peut également se définir de manière récursive :

```
def suite(n):
    if n == 0:
        return 3
    else:
        return 2*suite(n-1) - 2
```

Avec la récursivité, la boucle a disparu ! La suite est codée de façon « plus naturelle ».
Schématiquement, voici ce que donne le calcul de `suite(3)` :



La fonction `suite` est ainsi appelée 4 fois :

- Calcul de `suite(3)` : nécessite le calcul de `suite(2)`
- Calcul de `suite(2)` : nécessite le calcul de `suite(1)`
- Calcul de `suite(1)` : nécessite le calcul de `suite(0)`
- Calcul de `suite(0)` : le résultat est 3

Le cas $n = 0$ est appelé **cas de base** : c'est un cas essentiel, sans lequel les appels récursifs à la fonction `suite` se feraient à l'infini. Toute fonction récursive inclut un cas de base et se code de la manière suivante :

```
def fonction_recursive():
    if cas de base:
        instruction de base
    else:
        appel récursif
```

Une fois le cas de base atteint, on « remonte » dans les calculs :

- Calcul de `suite(1)` : $2 \times 3 - 2 = 4$
- Calcul de `suite(2)` : $2 \times 4 - 2 = 6$
- Calcul de `suite(3)` : $2 \times 6 - 2 = 10$

Autre exemple « classique » : le calcul de la **factorielle** d'un nombre. La factorielle d'un entier naturel n est définie comme étant le produit des entiers de 1 à n, et on la note $n!$:

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

En remarquant que $n! = n \times (n-1)!$ et $1! = 1$, on peut alors donner une fonction récursive calculant la factorielle de n :

```python
# Version itérative

def factorielle(n):
    f = 1

    for i in range(2,n+1):
        f = f * i

    return f
```

```python
# Version récursive

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

Question 01 Quel est le cas de base dans la version récursive de `factorielle` ?

Question 02 Par convention, on définit $0! = 1$. Corriger la fonction `factorielle` pour prendre en compte ce cas particulier.

Question 03 La version récursive de `factorielle` peut-elle générer une boucle infinie ? Si oui, donner une valeur de `n` correspondante. Comment peut-on empêcher cela ?

Si l'on tente de calculer 2000! avec notre fonction récursive, le programme affiche une erreur :

```
RecursionError: maximum recursion depth exceeded in comparison
```

Cette erreur provient du fait que Python limite le nombre d'appels récursifs. Pour connaître cette limite, on peut lui demander explicitement avec la fonction `getrecursionlimit()` du module `sys` :

```python
import sys
print(sys.getrecursionlimit())
```

```
1000
```

Heureusement, il est possible de modifier cette valeur, avec la fonction `setrecursionlimit()` et calculer 2000! :

```python
import sys
sys.setrecursionlimit(10000)

print(factorielle(2000))
```

```
331627...00000
```

Question 04 On considère la fonction récursive suivante, où `a` et `b` désignent des entiers naturels **non nuls** :

```python
def f(a,b):
    if b == 1:
        return a
    else:
        return a + f(a, b-1)
```

1. Quel résultat retourne `f(9,7)` ? On effectuera le calcul **à la main**.
2. Quel est le rôle de la fonction `f` ?
3. Quel est le cas de base dans cette fonction ? Est-il toujours atteint ?

1.2 Itératif VS Récursif

Alors, faut-il préférer la récursivité à l'itératif (boucle `for`) ? « La question est vite répondue »

Un programme récursif est souvent **plus court** et **plus naturel** à coder qu'un programme itératif (bon point pour les programmeurs qui sont fainénats) mais il faut faire **attention aux boucles infinies** qui pourraient apparaître.

L'inconvénient majeur des programmes récursifs réside dans leur excessive **lenteur**, comparés à leurs analogues itératifs. Considérons le programme suivant, qui calcule la factorielle de `N` avec les deux versions de la fonction `factorielle` vues précédemment, et affiche le temps d'éxécution de chacune :

```
import time, sys

def factorielle(n):
    f = 1
    for i in range(2,n+1):
        f = f * i
    return f

def factorielle_recur(n):
    if n == 1:
        return 1
    else:
        return n * factorielle_recur(n-1)

N = 1000
sys.setrecursionlimit(2*N) # On se laisse juste assez d'appels récursifs

T = time.time()
factorielle(N)
print("Itératif : {}".format(time.time() - T))

T = time.time()
factorielle_recur(N)
print("Récursif : {}".format(time.time() - T))
```

Pour `N = 1000` , les temps d'éxécution (en secondes) sont très courts mais l'itératif est plus rapide :

```
Itératif : 0.0005857944488525391
Récursif : 0.001522064208984375
```

Pour `N = 20000` , l'écart reste faible :

```
Itératif : 0.17080307006835938
Récursif : 0.20796823501586914
```

Pour de grandes valeurs de `N` , une erreur survient :

```
Itératif : 1.2242422103881836
Erreur de segmentation (core dumped)
```

Cette erreur est causée par un trop grand nombre d'appels récursifs. La fonction `setrecursionlimit` ne peut rien y faire!

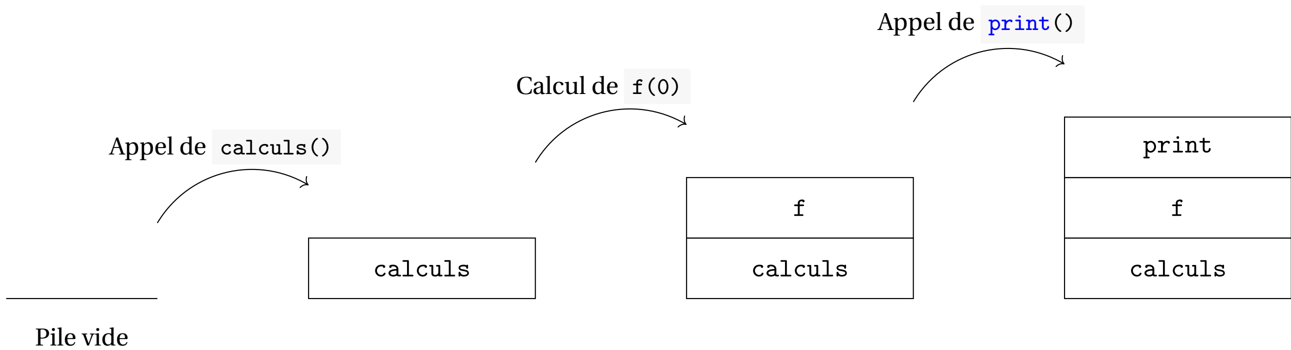
Pile d'éxécution

Lorsqu'une fonction est appelée, Python va garder une trace de l'endroit où chaque fonction active doit retourner à la fin de son exécution (les fonctions actives sont celles qui ont été appelées, mais n'ont pas encore terminé leur exécution).
Ces informations sont enregistrées dans la **pile d'éxécution** (ou *call stack*).

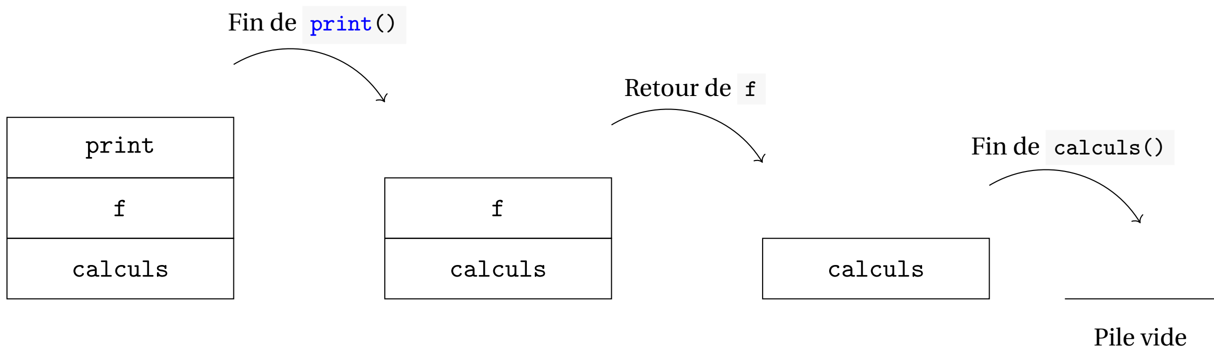
Sans entrer dans les détails, prenons un programme simple pour découvrir son fonctionnement :

```
def f(x):
    return 1/x

def calculs():
    print(f(0))

calculs()
```

Dans ce programme, l'ordre d'appel de chacune des fonctions est le suivant : `calculs`, `f` et enfin `print`.
La pile d'éxécution se remplit alors de la manière suivante :



Lorsque chaque fonction termine son éxécution, elle est **dépilée** :



Remarquons qu'ici, le calcul de `f(0)` retourne une erreur, car on divise par 0. On peut alors « voir » la pile d'éxécution : Python nous dresse le chemin suivi depuis le début du programme jusqu'à l'erreur, ligne par ligne :

```
Traceback (most recent call last):
  File "recursivite.py", line 7, in <module>
    calculs()
  File "recursivite.py", line 5, in calculs
    print(f(0))
  File "recursivite.py", line 2, in f
    return 1/x
ZeroDivisionError: division by zero
```

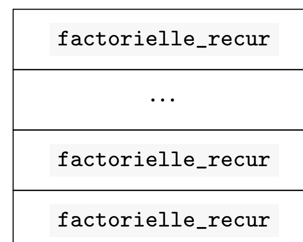
La fonction `print` n'apparaît pas dans cette liste, car elle n'a pas eu le temps d'être appelée : le calcul de `f(0)` a provoqué une erreur avant l'appel de `print`.

Revenons à notre fonction `factorielle_recur`. Que se passe-t-il lorsqu'on demande à Python de calculer la valeur de `factorielle_recur(20000)` ?

- On appelle la fonction `factorielle_recur` : elle est empilée dans la pile d'éxécution.
- Le calcul fait intervenir `factorielle_recur(19999)` : on empile une seconde fois la fonction `factorielle_recur`
- Le calcul fait intervenir `factorielle_recur(19998)` : nouvel empilement...

La taille de la pile d'éxécution augmente jusqu'à dépasser une valeur critique : c'est ce qui cause l'*erreur de segmentation*.

On a un **débordement de la pile d'éxécution** : notre programme Python essaie d'accéder à des adresses mémoires qu'il ne connaît pas, et le programme s'interrompt.



On saisit alors le problème : ces empilements prennent un peu de temps mais surtout beaucoup de place en mémoire !

En itératif, ce problème n'existe pas : la pile d'éxécution n'est pas modifiée, et Python gère très bien les très grands nombres.

Il est possible de modifier la taille maximale de la pile, mais le programme risque d'occuper de plus en plus de place en mémoire... Ce n'est pas une bonne solution, mais la voilà, par curiosité :

```
import resource
resource.setrlimit(resource.RLIMIT_STACK, [0x10000000, resource.RLIM_INFINITY])
```

Un mauvais exemple : Fibonacci

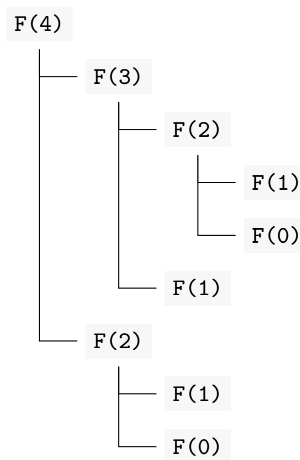
Vous connaissez sûrement la *suite de Fibonacci*, c'est la suite (F_n) dont chaque terme s'obtient en ajoutant les deux termes précédents, en partant de 1 et 1. On la définit naturellement par récurrence :

$$\begin{cases} F_0 = F_1 = 1 \\ F_{n+2} = F_{n+1} + F_n \end{cases}$$

En Python, on code facilement cette suite en récursif :

```
def F(n):
    if n == 0 or n == 1:
        return 1
    else:
        return F(n-1) + F(n-2)
```

Combien de fois la fonction `F` est-elle appelée lors du calcul de `F(4)` ?



La réponse n'est ni 4, ni 5... mais plutôt 9 fois !

Question 05 Combien de fois la fonction `F` est-elle appelée lors du calcul de `F(5)` ? de `F(6)` ? de `F(10)` ?

C'est exemple est clairement un mauvais exemple à programmer en récursif : le nombre d'appels à la fonction explose littéralement, et le temps d'éxécution s'envole.

Question 06 Coder la suite de Fibonacci en itératif, et comparer les temps d'éxécution de chacune des deux versions.

En résumé

Que retenir de la récursivité ? Qu'il faut faire attention au nombre d'appels récursifs ! Tant que ce dernier reste contenu, il y a peu d'incidence sur le déroulement du programme, et ce dernier gagne souvent en clarté. Mais lorsque le nombre d'appels est grand, on privilégie une programmation itérative...
Lorsque nous manipulerons des structures de données hiérarchiques, nous découvrirons les bienfaits de la récursivité... D'ici là, patience !

1.3 Exercices

Exercice 01 On considère la fonction récursive suivante :

```python
def f(a):
    if a == 1:
        return 1
    else:
        if a % 2 == 0:
            return f(a // 2)
        else:
            return f(3 * a + 1)
```

1. Calculer (à la main) $f(3)$ et $f(40)$.
2. Est-on certain que la fonction précédente se termine ?
3. Écrire un programme Python permettant de calculer les valeurs de $f(a)$ pour a allant de 1 à 10 000.

   Vérifier que ce programme se termine.

☞ *En septembre 2020, personne n'a réussi à trouver une valeur de a pour laquelle $f(a)$ génère une boucle infinie... Les mathématiciens pensent qu'une telle valeur n'existe pas : c'est la **conjecture de Syracuse**.*

Exercice 02 Écrire une fonction récursive `puissance(x,y)` qui retourne la valeur de x^y (où x et y sont deux nombres entiers naturels), sans utiliser l'opération `**`.

Exercice 03 Écrire une fonction récursive `somme(n)` qui retourne la somme des n premiers entiers.

```
>>> somme(3) # 1 + 2 + 3
6
>>> somme(10) # 1 + 2 + ... + 10
55
```

☞ *Sauriez-vous coder cette fonction en une ligne, en utilisant vos connaissances en Mathématiques ?*

Exercice 04 Écrire une fonction récursive `somme(n, k)` qui retourne la somme des puissances k-ièmes n premiers entiers.

```
>>> somme(3, 1) # 1 + 2 + 3
6
>>> somme(3, 2) # 1^2 + 2^2 + 3^2
14
```

Exercice 05 Écrire une fonction récursive `somme(L)` qui retourne la somme des éléments d'une liste `L`, sans utiliser la fonction `sum`.

Exercice 06 Pour savoir si un entier est divisible par 3, on dispose du critère de divisibilité bien connu :

« Un nombre est divisible par 3 si la somme de ses chiffres est encore divisible par 3 »

Écrire une fonction récursive `divisiblePar3(n)` qui retourne `True` si `n` est divisible par 3, et `False` sinon.