Récursivité

1 - Considérations générales

Lorsqu'on désire faire effectuer un travail à une fonction `f()` et qu'elle dispose de 3 fonctions `a() b() c()` pour l'aider, il existe deux manières d'envisager les choses.

1.1 Stratégie itérative

`f()` fait tout le travail et lance elle-même les appels à `a() b() c()`, utilisent leurs réponses pour construire sa propre réponse.

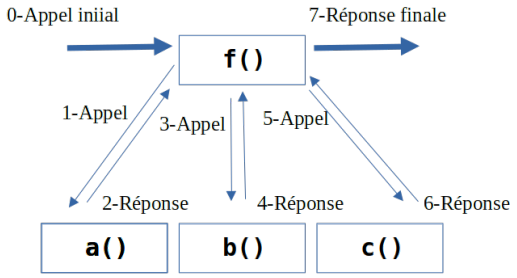

Figure 35.1 – Itératif

1.2 Stratégie récursive

`f()` lance un appel à `a()` qui va lancer un appel à `b()` qui va lancer un appel à `c()`. `c()` répond à `b()` qui peut alors construire sa propre réponse et répondre à `a()`. `a()` peut alors construire sa réponse et répondre à `f()`. `f()` pourra à son tour utiliser la réponse de `a()` pour construire sa propre réponse finale.

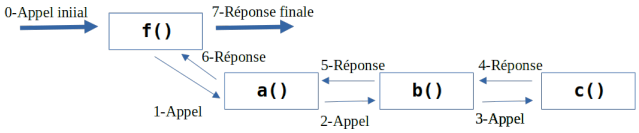

Figure 35.2 – Récursif

2 - Fonction récursive

2.1 Définition

Une **fonction récursive** est une fonction qui peut **lancer un appel à une autre instance d'elle même**.

Elle doit posséder au moins deux modes de calcul :

- Un **cas récursif** où on a besoin de lancer à autre appel à la fonction (en modifiant éventuellement les paramètres d'appel)
- Un **cas de base** non récursif qui permet d'obtenir une possibilité d'arrêt. On obtient ce cas lorsque la condition d'arrêt est validée. Il faut être vigilant sur cette condition d'arrêt : elle doit pouvoir être atteinte, sinon on obtient une boucle infinie d'appels récursifs.

2.2 Rappel : return None

Une fonction Python qui ne rencontre par de **`return`** avant de se terminer renvoie en réalité la valeur **`None`** comme si vous aviez placé **`return None`**.

2.3 Exemple sans return

```
1  def p(x :int) -> None :
2      """Affiche un truc"""
3      print(f"Appel de p({x})")
4      if x == 0 :
5          pass
6      else :
7          p(x-1)
8
9  p(5)
```

Déroulé :

L1 (définition de p())

L9 (appel à p(5)) - L1(x=5) - L3 - L4 - L6 - ...

L7 (appel à p(4)) - L1(x=4) - L3 - L4 - L6 - ...

L7 (appel à p(3)) - L1(x=3) - L3 - L4 - L6 - ...

L7 (appel à p(2)) - L1(x=2) - L3 - L4 - L6 - ...

L7 (appel à p(1)) - L1(x=1) - L3 - L4 - L6 - ...

L7 (appel à p(0)) - L1(x=0) - L3 - L4 - L5 - L7 (retour de None de p(0) à p(1))

L7 (retour None de p(1) à p(2))

L7 (retour None de p(2) à p(3))

L7 (retour None de p(3) à p(4))

L7 (retour None de p(4) à p(2))

L9 (retour None de p(5))

Pas de L10 : fin du programme.

2.4 Exemple avec return

On veut calculer ici 1+2+3+4... +n.

Pour rappel :

$$\sum_{x=0}^{n} x = \frac{n(n+1)}{2}$$

```python
def f(n :"int POSITIF NON NUL") -> int :
    """Somme des n premiers entiers"""

    print(f"Appel de f({n})")
    if n == 0 :
        return 0
    else :
        return n + f(n-1)

f(5)
```

Déroulé :

L1 (définition de f())

L10 (appel à p(5)) - L1(n=5) - L3 - L4 - L6 - ...

L7 (appel à p(4)) - L1(n=4) - L3 - L4 - L6 - ...

L7 (appel à p(3)) - L1(n=3) - L3 - L4 - L6 - ...

L7 (appel à p(2)) - L1(n=2) - L3 - L4 - L6 - ...

L7 (appel à p(1)) - L1(n=1) - L3 - L4 - L6 - ...

L7 (appel à p(0)) - L1(n=0) - L3 - L4 - L5 (retour de 0 de f(0) à f(1))

L7 (fin du calcul 1+0=1 et retour 1 de p(1) à p(2))

L7 (fin du calcul 2+1=3 et retour 3 de p(2) à p(3))

L7 (fin du calcul 3+3=6 et retour 6 de p(3) à p(4))

L7 (fin du calcul 4+6=10 et retour 10 de p(4) à p(5))

L7 (fin du calcul 5+10=15 et retour 15 de p(5) à la ligne 9

L9 (réception du 15) et Fin.

Pour information, la **version itérative équivalente** est :

```python
def f(n :"int POSITIF NON NUL") -> int :
    """Somme des n premiers entiers"""
    somme = 0
    for x in range(n+1) :
        somme = somme + x
    return somme

f(5)
```

3 - Empilement et dépilement

3.1 Empilement

Chaque appel de fonction est stocké dans la **pile d'exécution** (**call stack**). On place notamment dans cette zone mémoire les arguments reçus et la zone mémoire où fournir la réponse.

L'**empilement** consiste à placer en mémoire les différents appels jusqu'à atteindre le cas de base qui va stopper les appels récursifs et provoquer un premier retour de réponse.

3.2 Dépilement

Le **dépilement** est le phénomène inverse : les fonctions obtiennent une réponse de la fonction à laquelle elles ont fait appel et peuvent ainsi fournir elles-même une réponse à l'objet qui les a appeler.

3.3 Exemple avec 1+2+3+...n

```
Appel à f(5) va renvoyer 5 + f(4)                Début de l'empilement
    Appel à f(4) va renvoyer 4 + f(3)
        Appel à f(3) va renvoyer 3 + f(2)
            Appel à f(2) va renvoyer 2 + f(1)
                Appel à f(1) va renvoyer 1 + f(0)
                    Appel à f(0) renvoie 0       Cas de base
                Appel à f(1) renvoie 1 + 0 = 1
            Appel à f(2) renvoie 2 + 1 = 3
        Appel à f(3) renvoie 3 + 3 = 6
    Appel à f(4) renvoie 4 + 6 = 10
Appel à f(5) renvoie 5 + 10 = 15                 Fin du dépilement
```