Modularité.	Utiliser des API (*Application Programming Interface*) ou des bibliothèques. Exploiter leur documentation. Créer des modules simples et les documenter.	
Récursivité.	Écrire un programme récursif. Analyser le fonctionnement d'un programme récursif.	Des exemples relevant de domaines variés sont à privilégier.

Python 35 - Récursivité avec Python

I – Principe de la récursivité

1.1 – Fonction récursive

Une fonction récursive est une fonction qui, pour répondre, peut éventuellement faire appel à une autre instance (version) d'elle-même.

Pour ne pas boucler à l'infini, elle doit posséder au moins deux modes de calcul :

- Un **cas récursif** où elle lance à autre appel à la fonction.
- Un **cas de base** non récursif qui permet de répondre. On obtient ce cas lorsque la **condition d'arrêt** est validée.

EXEMPLE Voir DOC 1.

```
1 def f(x):
2     '''Fonction récursive fournissant la somme des n premiers entiers'''
3     print(f"Appel de f({x})")
4     if x == 0:
5         return 0
6     else:
7         return x + f(x-1)
```

Si on lance l'appel à **f(5)** :

→ on active **f(5)**

→ le paramètre x référence 5 dans l'espace des noms de f(5)

→ elle affiche "**Appel de f(5)**"

→ elle répondra **5 + f(4)** → f(5) en attente de … f(4)

EXEMPLE Voir DOC 1.

```
1 def f(x):
2     '''Fonction récursive fournissant la somme des n premiers entiers'''
3     print(f"Appel de f({x})")
4     if x == 0:
5         return 0
6     else:
7         return x + f(x-1)
```

Si on lance l'appel à **f(5)** :

→ on active **f(5)** → f(5) en attente de … 5 + f(4)

→ on active **f(4)**

→ le paramètre x référence 4 dans l'EdN de f(4)

→ elle affiche "**Appel de f(4)**"

→ elle répondra 4 **+ f(3)** → f(4) en attente de … 4+f(3)

EXEMPLE Voir DOC 1.

```
1 def f(x):
2     '''Fonction récursive fournissant la somme des n premiers entiers'''
3     print(f"Appel de f({x})")
4     if x == 0:
5         return 0
6     else:
7         return x + f(x-1)
```

Si on lance l'appel à **f(5)** :

→ on active **f(5)** → f(5) en attente de … 5 + f(4)

→ on active **f(4)** → f(4) en attente de … 4 + f(3)

→ on active **f(3)**

→ le paramètre x référence 3 dans l'EdN de f(3)

→ elle affiche "**Appel de f(3)**"

→ elle répondra 3 **+ f(2)** → f(3) en attente de … 3+f(2)

EXEMPLE Voir DOC 1.

```
1 def f(x):
2     '''Fonction récursive fournissant la somme des n premiers entiers'''
3     print(f"Appel de f({x})")
4     if x == 0:
5         return 0
6     else:
7         return x + f(x-1)
```

Si on lance l'appel à **f(5)** :

→ on active **f(5)** → f(5) en attente de … 5 + f(4)

→ on active **f(4)** → f(4) en attente de … 4 + f(3)

→ on active **f(3)** → f(3) en attente de … 3 + f(2)

→ on active **f(2)**

→ le paramètre x référence 2 dans l'EdN de f(2)

→ elle affiche "**Appel de f(2)**"

→ elle répondra 2 **+ f(1)** → f(2) en attente de … 2+f(1)

EXEMPLE Voir DOC 1.

```
1 def f(x):
2     '''Fonction récursive fournissant la somme des n premiers entiers'''
3     print(f"Appel de f({x})")
4     if x == 0:
5         return 0
6     else:
7         return x + f(x-1)
```

Si on lance l'appel à **f(5)** :

→ on active **f(5)** → f(5) en attente de … 5 + f(4)

→ on active **f(4)** → f(4) en attente de … 4 + f(3)

→ on active **f(3)** → f(3) en attente de … 3 + f(2)

→ on active **f(2)** → f(2) en attente de … 2 + f(1)

→ on active **f(1)**

→ le paramètre x référence 1 dans l'EdN de f(1)

→ elle affiche "**Appel de f(1)**"

→ elle répondra 1 **+ f(0)** → f(1) en attente de … 1+f(0)

EXEMPLE Voir DOC 1.

```
1 def f(x):
2     '''Fonction récursive fournissant la somme des n premiers entiers'''
3     print(f"Appel de f({x})")
4     if x == 0:
5         return 0
6     else:
7         return x + f(x-1)
```

Si on lance l'appel à **f(5)** :

→ on active **f(5)** → f(5) en attente de … 5 + f(4)

→ on active **f(4)** → f(4) en attente de … 4 + f(3)

→ on active **f(3)** → f(3) en attente de … 3 + f(2)

→ on active **f(2)** → f(2) en attente de … 2 + f(1)

→ on active **f(1)** → f(1) en attente de … 1 + f(0)

→ on active **f(0)**

→ le paramètre x référence 1 dans l'EdN de f(0)

→ elle affiche "**Appel de f(0)**"

→ elle **répond 0** car il s'agit du cas de base

Si on lance l’appel à **f(5)** :

→ on active **f(5)**	→ `f(5) en attente de … 5 + f(4)`
→ on active **f(4)**	→ `f(4) en attente de … 4 + f(3)`
→ on active **f(3)**	→ `f(3) en attente de … 3 + f(2)`
→ on active **f(2)**	→ `f(2) en attente de … 2 + f(1)`
→ on active **f(1)**	→ `f(1) en attente de … 1 + f(0)`
→ on active f(0)	**→ f(0) répond 0.**

A partir de là, il suffit de remonter dans les appels.

Si on lance l'appel à **f(5)** :

→ on active **f(5)**	→ `f(5) en attente de … 5 + f(4)`
→ on active **f(4)**	→ `f(4) en attente de … 4 + f(3)`
→ on active **f(3)**	→ `f(3) en attente de … 3 + f(2)`
→ on active **f(2)**	→ `f(2) en attente de … 2 + f(1)`
→ on active **f(1)**	→ `f(1) en attente de … 1 + f(0)`
→ on active f(0)	**→ f(0) répond 0.**

A partir de là, il suffit de remonter dans les appels.

f(0) répond 0 à f(1)

Si on lance l'appel à **f(5)** :

→ on active **f(5)**	→ `f(5) en attente de … 5 + f(4)`
→ on active **f(4)**	→ `f(4) en attente de … 4 + f(3)`
→ on active **f(3)**	→ `f(3) en attente de … 3 + f(2)`
→ on active **f(2)**	→ `f(2) en attente de … 2 + f(1)`
→ on active **f(1)**	→ `f(1) en attente de … 1 + f(0)`
→ on active f(0)	**→ f(0) répond 0.**

A partir de là, il suffit de remonter dans les appels.

f(0) répond 0 à f(1)

f(1) évalue 1 + 0 et répond 1 à f(2)

Si on lance l'appel à **f(5)** :

→ on active **f(5)**	→ `f(5) en attente de … 5 + f(4)`
→ on active **f(4)**	→ `f(4) en attente de … 4 + f(3)`
→ on active **f(3)**	→ `f(3) en attente de … 3 + f(2)`
→ on active **f(2)**	→ `f(2) en attente de … 2 + f(1)`
→ on active **f(1)**	→ `f(1) en attente de … 1 + f(0)`
→ on active f(0)	**→ f(0) répond 0.**

A partir de là, il suffit de remonter dans les appels.

f(0) répond 0 à f(1)

f(1) évalue 1 + 0 et répond 1 à f(2)

f(2) évalue 2 + 1 et répond 3 à f(3)

Si on lance l'appel à **f(5)** :

→ on active **f(5)**	→ `f(5) en attente de … 5 + f(4)`
→ on active **f(4)**	→ `f(4) en attente de … 4 + f(3)`
→ on active **f(3)**	→ `f(3) en attente de … 3 + f(2)`
→ on active **f(2)**	→ `f(2) en attente de … 2 + f(1)`
→ on active **f(1)**	→ `f(1) en attente de … 1 + f(0)`
→ on active f(0)	**→ f(0) répond 0.**

A partir de là, il suffit de remonter dans les appels.

f(0) répond 0 à f(1)

f(1) évalue 1 + 0 et répond 1 à f(2)

f(2) évalue 2 + 1 et répond 3 à f(3)

f(3) évalue 3 + 3 et répond 6 à f(4)

Si on lance l'appel à **f(5)** :

→ on active **f(5)**	→ `f(5) en attente de … 5 + f(4)`
→ on active **f(4)**	→ `f(4) en attente de … 4 + f(3)`
→ on active **f(3)**	→ `f(3) en attente de … 3 + f(2)`
→ on active **f(2)**	→ `f(2) en attente de … 2 + f(1)`
→ on active **f(1)**	→ `f(1) en attente de … 1 + f(0)`
→ on active f(0)	**→ f(0) répond 0.**

A partir de là, il suffit de remonter dans les appels

f(0) répond 0 à f(1)

f(1) évalue 1 + 0 et répond 1 à f(2)

f(2) évalue 2 + 1 et répond 3 à f(3)

f(3) évalue 3 + 3 et répond 6 à f(4)

f(4) évalue 4 + 6 et répond 10 à f(5)

Si on lance l'appel à **f(5)** :

→ on active **f(5)**	→ `f(5) en attente de … 5 + f(4)`
→ on active **f(4)**	→ `f(4) en attente de … 4 + f(3)`
→ on active **f(3)**	→ `f(3) en attente de … 3 + f(2)`
→ on active **f(2)**	→ `f(2) en attente de … 2 + f(1)`
→ on active **f(1)**	→ `f(1) en attente de … 1 + f(0)`
→ on active f(0)	**→ f(0) répond 0.**

A partir de là, il suffit de remonter dans les appels

f(0) répond 0 à f(1)

f(1) évalue 1 + 0 et répond 1 à f(2)

f(2) évalue 2 + 1 et répond 3 à f(3)

f(3) évalue 3 + 3 et répond 6 à f(4)

f(4) évalue 4 + 6 et répond 10 à f(5)

f(5) évalue 5 + 10 et répond 15 à … l'endroit où on l'a appelée.

On notera deux points importants :

* **il faut s’assurer de pouvoir atteindre le cas de base** (sinon c’est une boucle infinie). On retrouve donc la notion de preuve de terminaison.

On notera deux points importants :

* **il faut s’assurer de pouvoir atteindre le cas de base** (sinon c’est une boucle infinie). On retrouve donc la notion de preuve de terminaison.

* tant qu’une fonction attend une réponse d’une autre fonction, elle est bloquée.

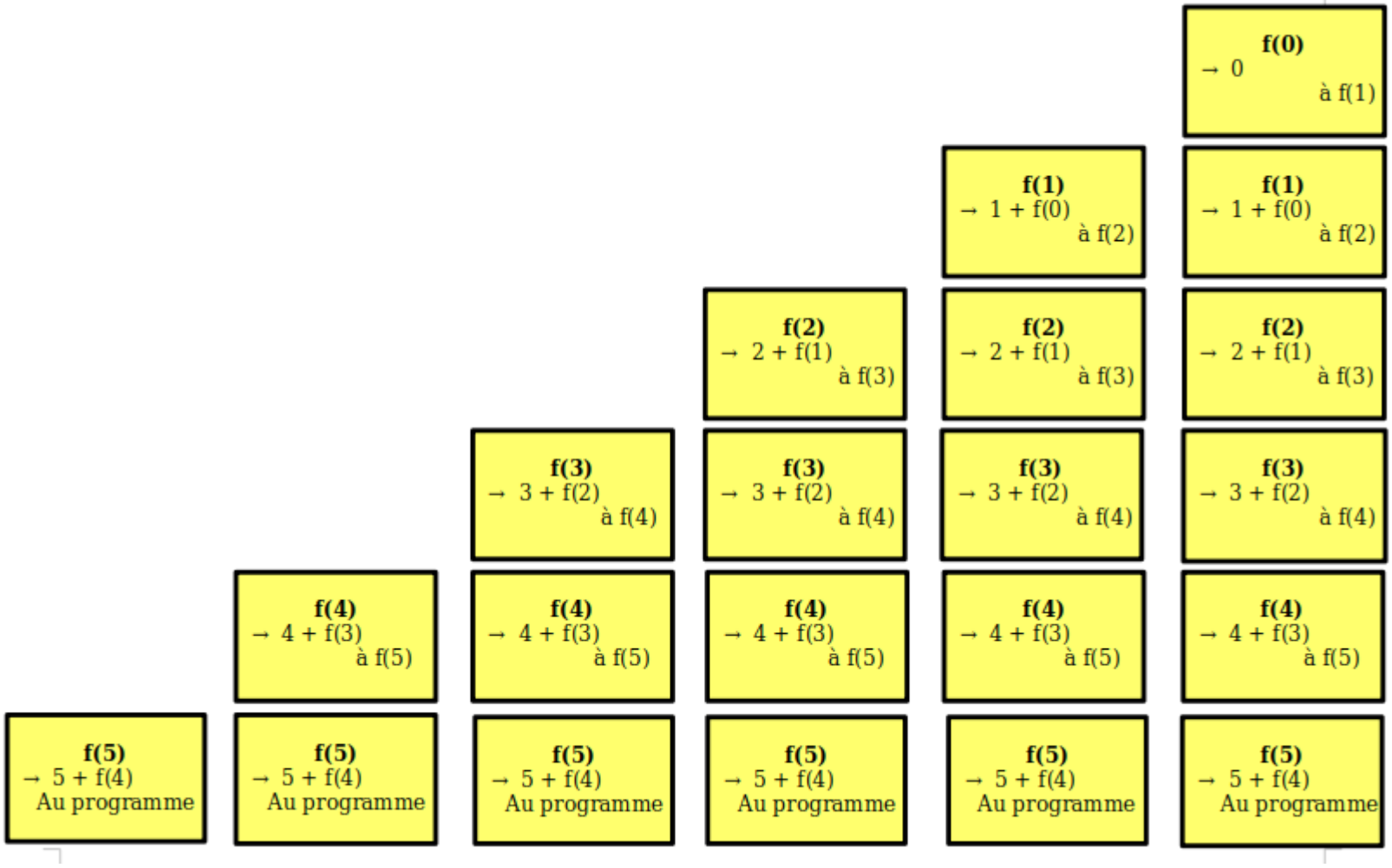
II – Empilement

La **pile d'exécution** est la structure de données dans laquelle on enregistre les informations sur les appels de fonctions dans un programme informatique. C'est grace à elle qu'on sait "qui appelle qui" et quelle réponse donner à qui.

Chaque appel de fonction génère son bloc.

Chaque bloc contient les informations :
* les **paramètres** reçus lors de l'appel
* l'**adresse de réponse** (le bloc de dessous normalement)
* les ***variables locales*** de cet appel

FIG 2 : Création progressive de la pile d’appel



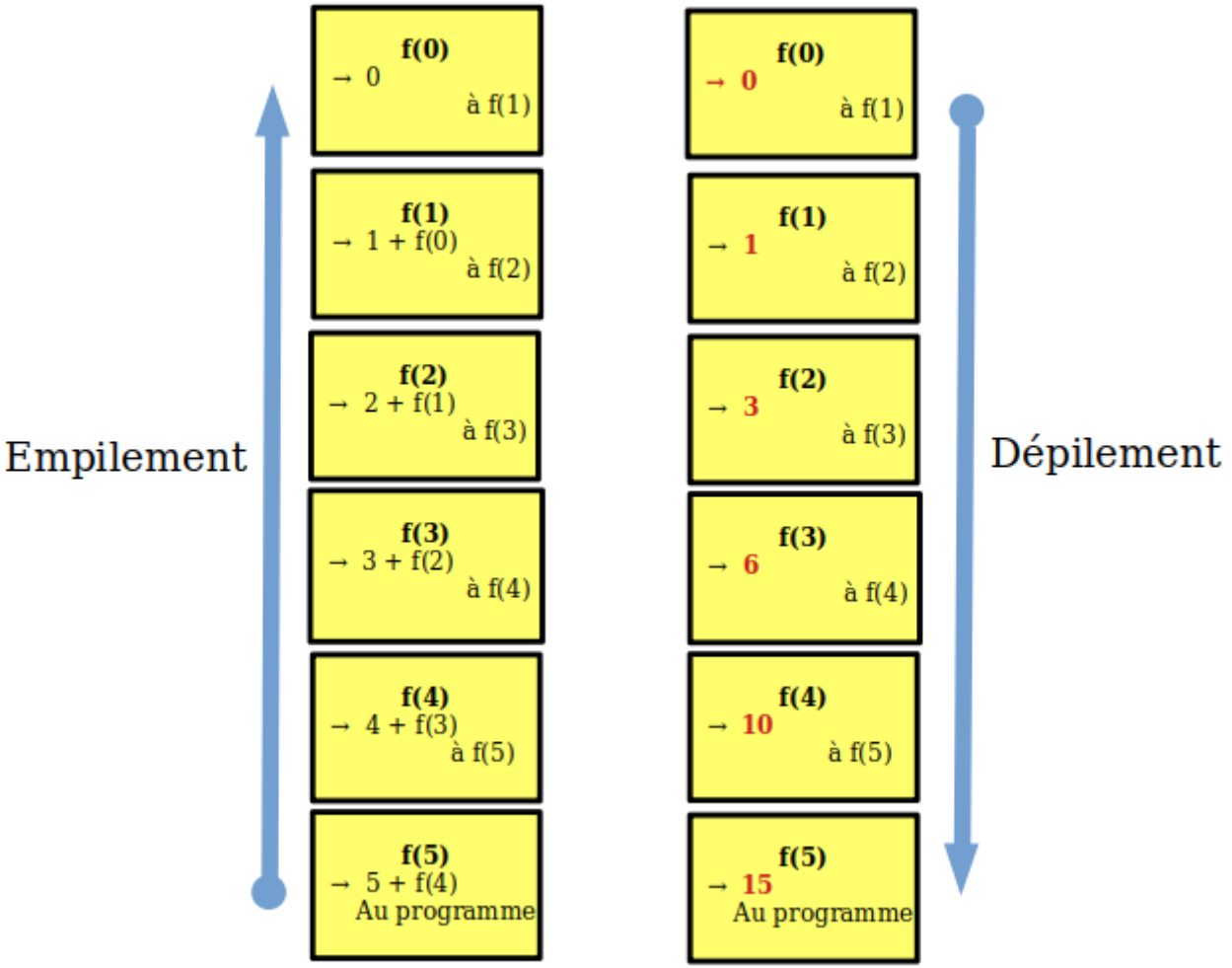
III – Dépilement

FIG 3 :

Lorsqu'on arrive enfin à un cas de base, le bloc du haut peut donner sa réponse au bloc en dessous et disparaît.

Il ne reste au final en mémoire que le bloc de f(5) qui pourra finir son exécution et disparaître après avoir répondu.

C'est pour cela que les espaces des noms disparaissent.



Compléments

Le problème de la récursivité est donc l'empilement des blocs puisqu'ils prennent tous de la place en mémoire.

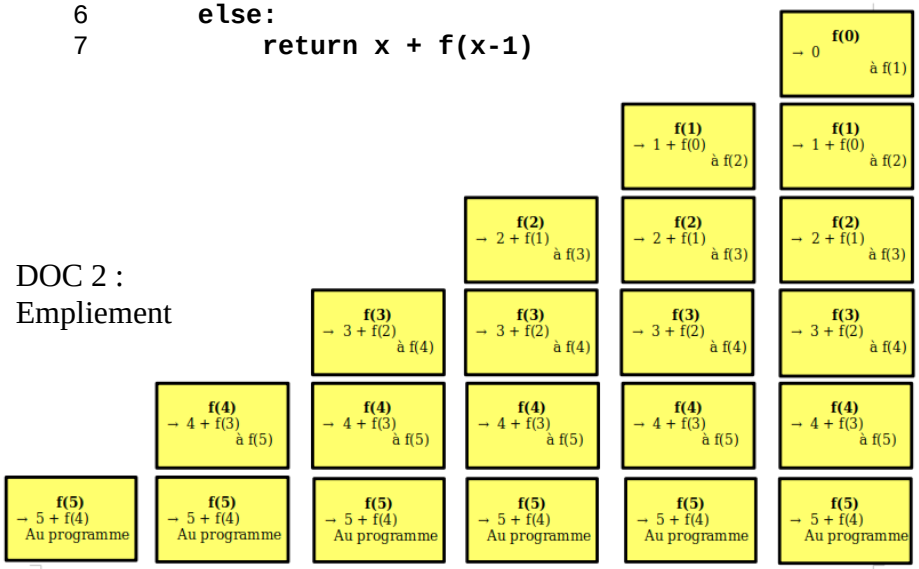
Python limite par défaut le nombre d'appels récursifs à 2000.

Vous verrez l'année prochaine à la notion de **récursivité terminale** : il s'agit des appels récursifs où on n'est pas obligé de garder en mémoire le bloc car une fois arrivé au return, il n'y a plus aucun calcul à faire. On peut donc juste garder en mémoire le fait de savoir à qui répondre.

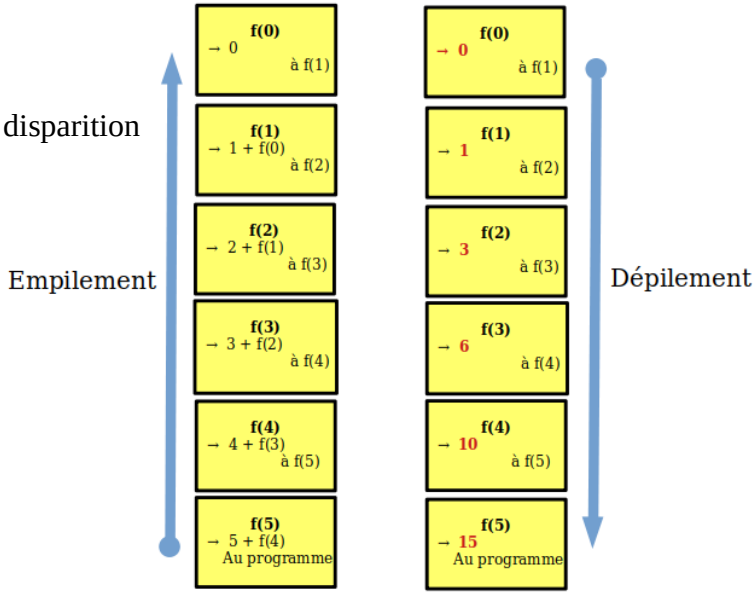
EXEMPLE (DOC 1)

```
1   def f(x):
2       '''Fonction récursive fournissant la somme des n premiers entiers'''
3       print(f"Appel de f({x})")
4       if x == 0:
5           return 0
6       else:
7           return x + f(x-1)
```

DOC 2 : Empliement



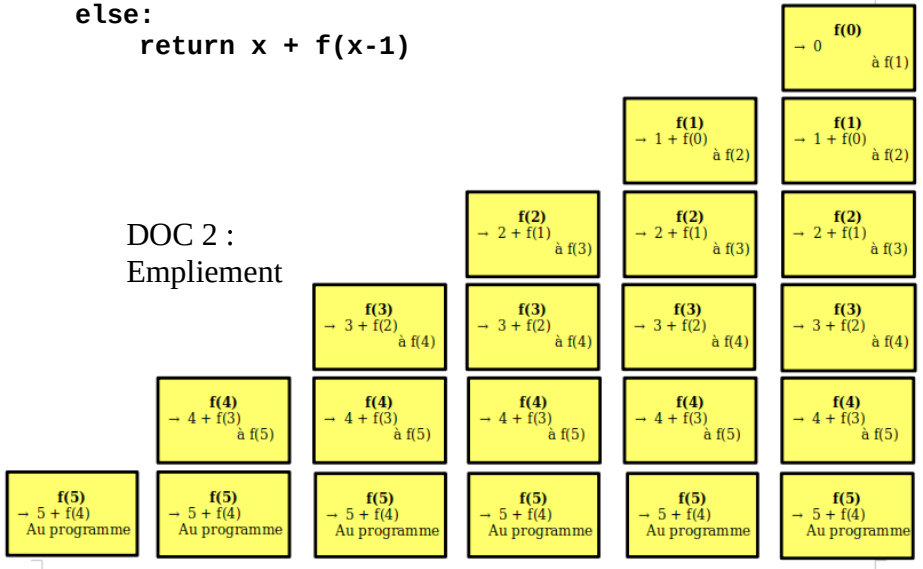
DOC 3 : Apparition et disparition des blocs



EXEMPLE (DOC 1)

```
1   def f(x):
2       '''Fonction récursive fournissant la somme des n premiers entiers'''
3       print(f"Appel de f({x})")
4       if x == 0:
5           return 0
6       else:
7           return x + f(x-1)
```

DOC 2 : Empliement



DOC 3 : Apparition et disparition des blocs