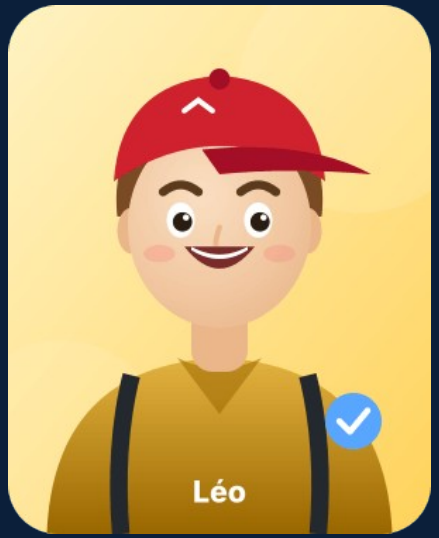
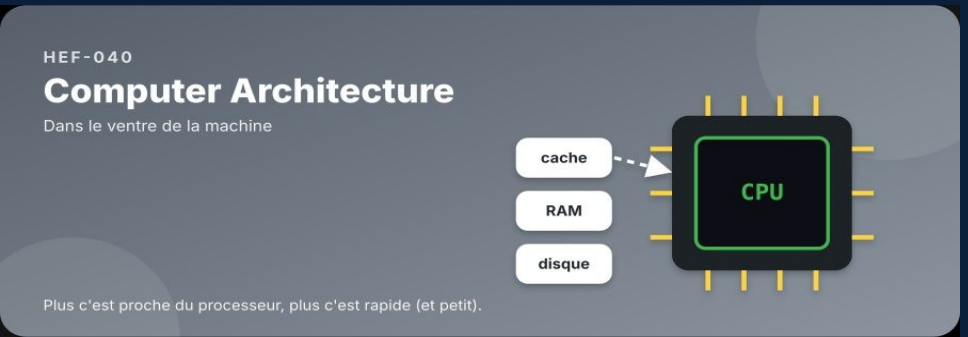
HEF-040 · MISSION 08 / 10

Un programme en cours d'exécution

Entre ton fichier de code et les instructions du CPU, il se passe beaucoup de choses. Regardons sous le capot.





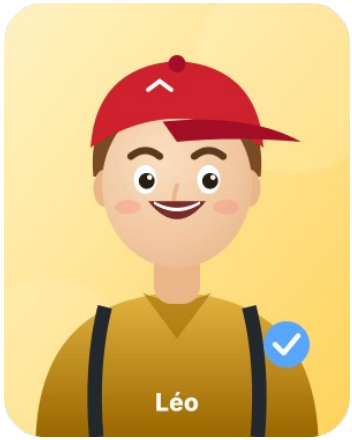
Badge : Anatomiste de processus

75 min · débutant

L'histoire

Kodi lance un script Python qui plante avec `RecursionError`. Un autre programme, écrit en C par un collègue, plante avec `segmentation fault`. Même idée (trop d'appels), deux erreurs différentes.

> *« Pour comprendre, il faut voir un programme comme la machine le voit, dit Sam : du code, des données, une pile, un tas. »*



Tu apprends comment un programme est organisé en mémoire pendant qu'il tourne.

Ta mission

À la fin, je serai capable de…

1. décrire le chemin du code source à l'exécution (compilation, interprétation, chargement)
2. décrire la disposition mémoire : code, données, tas, pile
3. expliquer ce qu'est un cadre de pile (stack frame)
4. expliquer le dépassement de pile et la récursion
5. lire le bytecode Python avec le module `dis`

Défi d'échauffement

Défi d'échauffement

Tu lis une recette à voix haute à quelqu'un qui cuisine. La recette dit : « Pour la sauce, voir la recette de la page 12 ». Il faut interrompre la recette en cours, aller faire la sauce, puis **revenir exactement où tu en étais**. Comment retenir où tu en étais, si la sauce elle-même renvoie à une autre recette ? Que se passe-t-il si les renvois s'enchaînent sans fin ?

Le concept

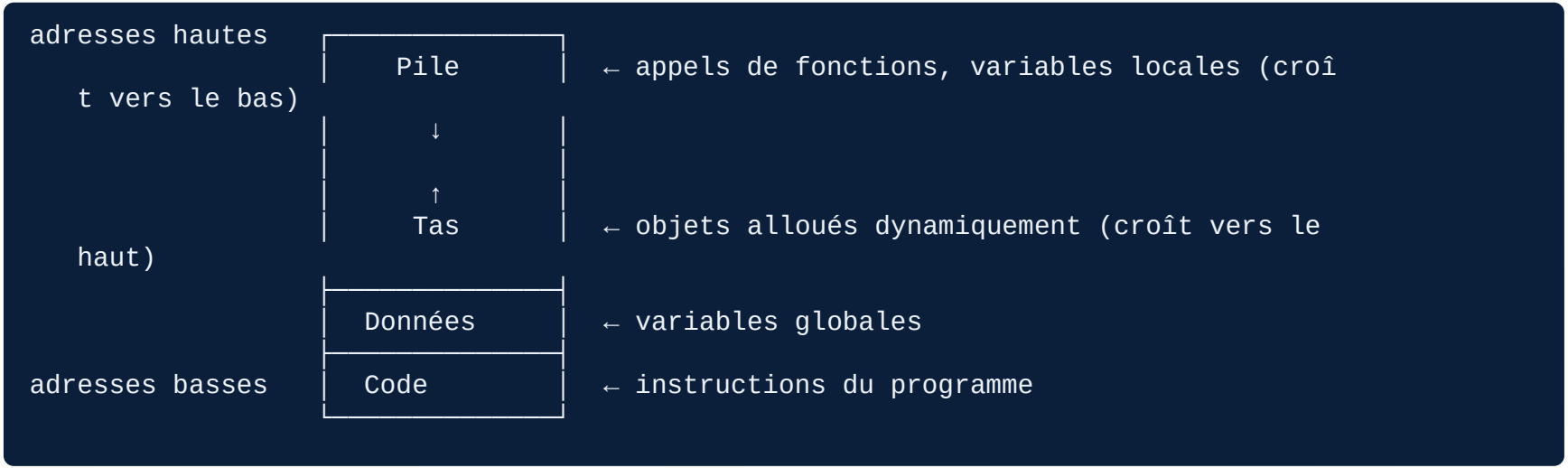
Du code à l'exécution

Étape	Rôle
Écriture	code source (texte)
Compilation	traduction du code source en langage machine (C, Rust, Go) ou en bytecode (Java, Python)
Édition de liens	assemblage des morceaux et des bibliothèques
Chargement	le système d'exploitation copie le programme en mémoire et lance le **processus**
Exécution	le CPU exécute (ou une machine virtuelle interprète)

Compilé : traduit à l'avance en code machine (rapide). **Interprété** : lu et exécuté par un autre programme (flexible, plus lent). Python compile vers du **bytecode** exécuté par une machine virtuelle.

Le concept

La mémoire d'un processus



Le concept

La pile d'appels

Chaque appel de fonction crée un **cadre de pile** (*stack frame*) qui contient :

- les **paramètres** ;
- les **variables locales** ;
- l'**adresse de retour** (où reprendre après la fonction).

Quand la fonction se termine, son cadre est **dépilé** et l'exécution reprend à l'adresse de retour.

```
def c():
    return 1

def b():
    return c() + 1
```

Le concept

La pile d'appels (suite)

```
def a():
    return b() + 1

print(a())    # 3 : pile pendant l'appel de c : a, b, c
```

Dépassement de pile

La pile a une taille limitée. Une récursion trop profonde la remplit :

- en C : `segmentation fault` (accès hors de la zone autorisée) ;
- en Python : `RecursionError` (une limite logicielle protège la machine).

Le concept

Dépassement de pile (suite)

```
import sys
print(sys.getrecursionlimit())     # 1000 par défaut en général
```

La pile et le tas

	Pile	Tas
Contenu	variables locales, appels	objets de durée variable
Gestion	automatique (entrée/sortie de fonction)	allocation à la demande
Vitesse	très rapide	plus lente
Risque	dépassement	fuites, fragmentation

En Python, presque tout est un **objet sur le tas** ; les variables sont des **références**.

Le concept

Voir le bytecode

```
import dis

def somme(a, b):
    return a + b

dis.dis(somme)
```

Le résultat liste des instructions de la machine virtuelle (`LOAD_FAST`, `BINARY_OP` ou `BINARY_ADD`, `RETURN_VALUE`…), qui ressemblent à notre mini-processeur.

🧠 ***Astuce*** *: quand ta récursion est « trop profonde », pense à la pile : chaque appel y laisse un cadre jusqu'à sa fin.*

Exemple pas à pas

Exemple pas à pas

Problème : voir la pile d'appels grandir, puis provoquer un dépassement contrôlé.

```
import sys

profondeur_max = 0

def descendre(n):
    global profondeur_max
    profondeur_max = max(profondeur_max, n)
    return descendre(n + 1)

try:
    descendre(0)
except RecursionError:
    print("dépassement de pile à une profondeur de", profondeur_max)

print("limite configurée :", sys.getrecursionlimit())
```

Exemple pas à pas (suite)

Lecture : la profondeur atteinte est proche de la limite (un peu en dessous, car le programme principal utilise déjà quelques cadres). Chaque appel consomme de la pile ; la limite protège la machine.
Version itérative (sans pile d'appels) :

```
def compter_iteratif(n):
    total = 0
    for i in range(n + 1):
        total += i
    return total

def compter_recursif(n):
    return 0 if n == 0 else n + compter_recursif(n - 1)

print(compter_iteratif(10000))      # 50005000
try:
    print(compter_recursif(10000))
except RecursionError:
```

Exemple pas à pas (suite)

```
    print("la version récursive dépasse la pile")
```

Leçon : une boucle utilise un cadre de pile constant ; une récursion en utilise un par niveau.

À toi, avec aide

Examine le bytecode de cette fonction avec `dis` :

```
def carre_plus_un(x):
    return x * x + 1
```

1. Combien d'instructions de calcul vois-tu ?
2. Retrouve la correspondance avec notre mini-processeur (LOAD, MUL, ADD…).
3. Que contient une pile de machine virtuelle à chaque instant ?

À toi, avec aide · réponse

Indices progressifs

Indice 1

Importe `dis` et appelle `dis.dis(carre_plus_un)`.

Indice 2

Cherche des instructions contenant `LOAD`, `MULTIPLY` ou `BINARY_OP`, `ADD`, `RETURN`.

Indice 3

La machine virtuelle Python est une machine à pile : les opérandes sont empilés puis consommés.

À toi, avec aide · réponse

Réponse

```
import dis

def carre_plus_un(x):
    return x * x + 1

dis.dis(carre_plus_un)
```

On observe en gros : charger `x` (deux fois), multiplier, charger la constante 1, additionner, retourner. Deux opérations de calcul (la multiplication et l'addition). La pile de la machine virtuelle contient les opérandes en attente : après `LOAD x, LOAD x` elle contient deux valeurs ; `MULTIPLY` les remplace par leur produit ; etc. Le nom exact des instructions dépend de la version de Python.

À toi, en solo

Simule la pile d'appels. Écris un petit interpréteur qui exécute `factorielle(4)` en gérant lui-même une pile de cadres (liste de dictionnaires `{"fonction": …, "n": …, "retour": …}`).
À chaque empilement/dépilement, affiche l'état de la pile. Vérifie que la profondeur maximale est n + 1 et que le résultat est 24.

À toi, en solo · réponse

Pistes de correction

```
def factorielle_avec_pile(n):
    pile = []
    max_profondeur = 0
    # descente : on empile un cadre par valeur
    for k in range(n, -1, -1):
        pile.append({"fonction": "fact", "n": k})
        max_profondeur = max(max_profondeur, len(pile))
        print("empile  ", [c["n"] for c in pile])
    # remontée : on dépile et on combine
    resultat = 1                    # fact(0) = 1
    pile.pop()
    print("dépile  ", [c["n"] for c in pile], "→ fact(0) = 1")
    while pile:
        cadre = pile.pop()
        resultat = cadre["n"] * resultat
        print("dépile  ", [c["n"] for c in pile], "→ fact(%d) = %d" % (cadre["n"],
    resultat))
    return resultat, max_profondeur
```

Pistes de correction (suite)

```
r, p = factorielle_avec_pile(4)
assert r == 24 and p == 5
print(r, p)
```

Profondeur maximale n + 1 = 5 : exactement ce que fait le vrai processus avec sa pile d'appels.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Récursion sans limite	Dépassement de pile	Prévoir un cas de base et une profondeur raisonnable, ou utiliser une boucle
Confondre pile et tas	Mauvais modèle mental de la mémoire	Pile = appels et locales ; tas = objets
Croire qu'un langage est « soit compilé soit interprété »	La réalité est un spectre (bytecode, JIT)	Voir la chaîne complète
Oublier que les variables Python sont des références	Modifications inattendues d'objets partagés	Distinguer objet et nom

Mini-boss

Mini-boss

Autopsie d'un processus Python.

1. Écris un programme qui crée une liste de 1 million d'entiers, une fonction récursive de profondeur 500, et un dictionnaire de 100 000 entrées.
2. Mesure avec `tracemalloc` la mémoire utilisée sur le **tas** par chaque structure.
3. Utilise `sys.getsizeof` et `id` pour montrer que deux variables peuvent désigner le **même objet**.
4. Utilise `dis` pour comparer le bytecode d'une boucle `for` et d'une compréhension de liste.
5. Rédige un schéma de la mémoire au moment de l'appel le plus profond (pile et tas).

Mini-boss · réponse

Pistes de correction

```
import sys, tracemalloc, dis

def mesurer(f):
    tracemalloc.start()
    objet = f()
    courant, pic = tracemalloc.get_traced_memory()
    tracemalloc.stop()
    return objet, courant // 1024

liste, ko1 = mesurer(lambda: list(range(1_000_000)))
dico, ko2 = mesurer(lambda: {i: i for i in range(100_000)})
print("liste :", ko1, "Ko ; dictionnaire :", ko2, "Ko")

a = [1, 2, 3]
b = a
print(a is b, id(a) == id(b))          # même objet
c = list(a)
print(a is c)                          # copie : autre objet
```

Mini-boss · réponse

Pistes de correction (suite)

```
def boucle(n):
    r = []
    for i in range(n):
        r.append(i * 2)
    return r

def comprehension(n):
    return [i * 2 for i in range(n)]

dis.dis(boucle)
dis.dis(comprehension)

def profond(n):
    return 0 if n == 0 else 1 + profond(n - 1)
print(profond(500))
```

Le schéma attendu montre la pile avec ~500 cadres de `profond`, et le tas contenant la liste, le dictionnaire et les entiers référencés.

Quiz éclair

Quiz éclair

1. Un cadre de pile contient notamment :

A. le code source
B. les fichiers ouverts
C. les variables locales et l'adresse de retour

2. Un `RecursionError` en Python signale :

A. une pile d'appels trop profonde
B. un fichier manquant
C. une erreur de syntaxe

3. En mémoire, les objets de taille variable créés pendant l'exécution se trouvent surtout :

A. dans les registres

Quiz éclair (suite)

B. sur le tas
C. dans le code

4. 🔁 *Révision mission 07* : Une interruption est :

A. une panne
B. un type de mémoire
C. un signal d'un périphérique vers le CPU

5. 🔁 *Révision mission 06* : Un tampon (buffer) sert à :

A. regrouper des données avant de les envoyer
B. effacer des fichiers
C. chiffrer les données

Quiz éclair · réponse

Réponses

1. **C**.
2. **A**.
3. **B**.
4. **C**.
5. **A**.

Je réfléchis

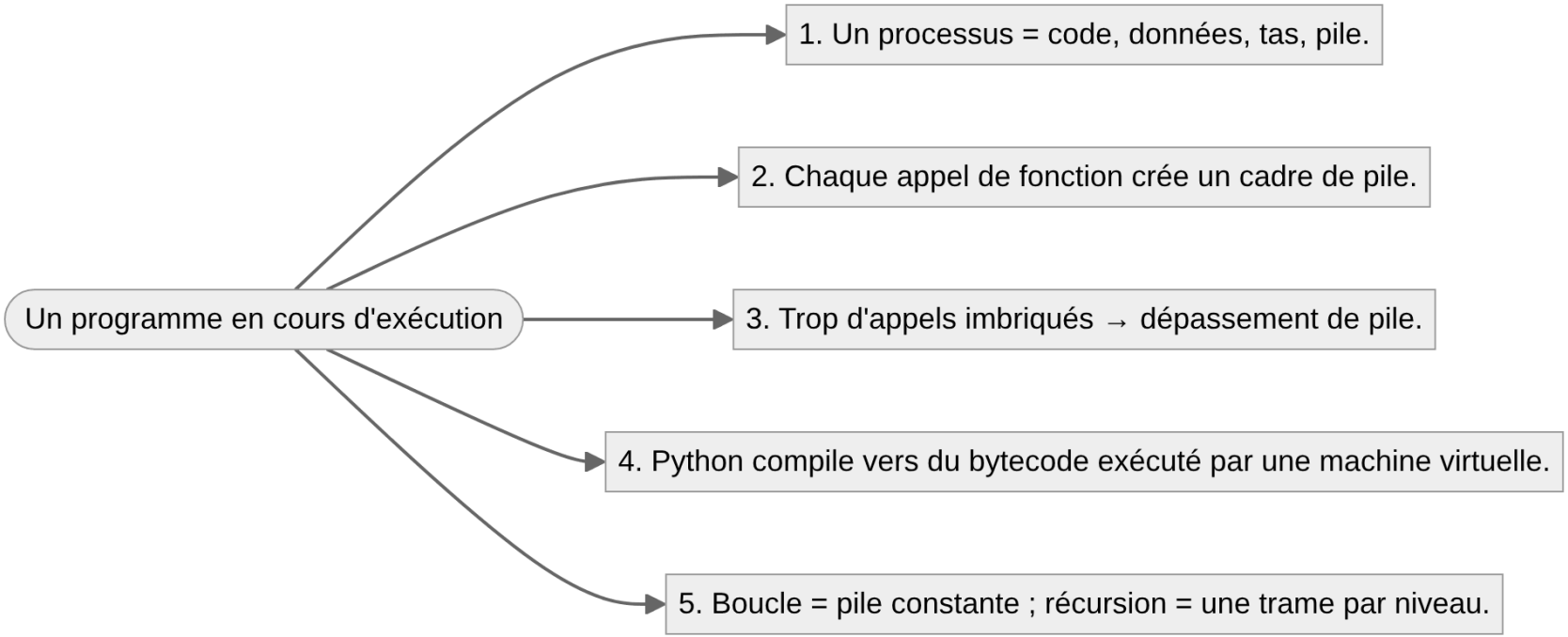
1. Quelle situation de ta vie ressemble à une « pile d'appels » (tâches interrompues par d'autres tâches) ?
2. Comment le fait de savoir ce qu'est la pile t'aide-t-il à lire un message d'erreur ?

À retenir

1. Un processus = code, données, tas, pile.
2. Chaque appel de fonction crée un cadre de pile.
3. Trop d'appels imbriqués → dépassement de pile.
4. Python compile vers du bytecode exécuté par une machine virtuelle.
5. Boucle = pile constante ; récursion = une trame par niveau.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Anatomiste de processus

Prochaine mission : Performance et limites physiques

