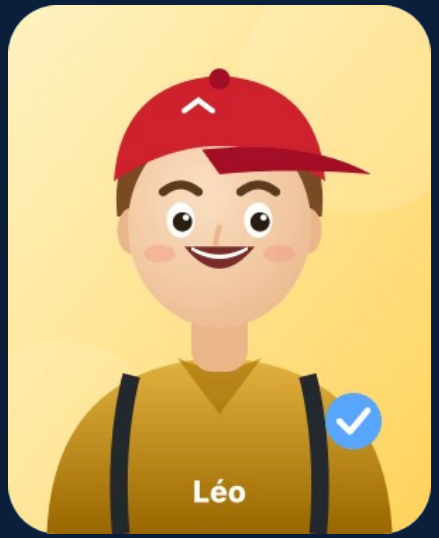
HEF-130 · MISSION 04 / 10

Mémoire et allocations

La mémoire est limitée, et la créer, la copier et la libérer coûte du temps. Savoir où elle part, c'est éviter pannes et lenteurs.



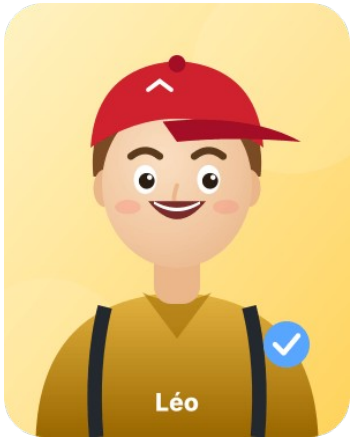


Badge : Économe de mémoire

100 min · avancé

L'histoire

Le traitement nocturne des certificats de HashBox lit un fichier d'inscriptions de 2 Go. Il fonctionne pendant des mois, puis un soir il est tué par le système : « out of memory ». Personne n'a changé le code. Le fichier a simplement doublé.



« Le code chargeait tout en mémoire, dit Sam. Tant que ça tenait, ça marchait. Il existe des façons de traiter des données plus grosses que la mémoire : flux, générateurs, structures compactes. Et d'autres fois, la mémoire n'est pas trop petite, elle fuit : un cache qui ne se vide jamais. »

Tu apprends à mesurer la mémoire avec `tracemalloc`, à comparer des structures de données, à utiliser des flux plutôt que des listes complètes, à repérer des fuites, et à borner un cache.

Ta mission

À la fin, je serai capable de…

1. mesurer la mémoire d'un programme avec `tracemalloc` et `sys.getsizeof`
2. comparer le coût mémoire de structures (liste, tuple, dict, classe, `__slots__`, `array`)
3. traiter de gros volumes avec des générateurs et des flux
4. détecter une fuite mémoire en comparant des instantanés
5. borner un cache (LRU) pour qu'il ne croisse pas sans limite

Défi d'échauffement

Défi d'échauffement

Tu déménages. Tu peux tout mettre dans un seul gros camion, ou faire des allers-retours en voiture avec quelques cartons. Quand le camion est trop petit, tu es bloqué ; avec des allers-retours, tu es lent mais tu peux déménager n'importe quelle quantité. Fais le parallèle avec lire un fichier de 2 Go en entier ou ligne par ligne.

Le concept

Où part la mémoire ?

Chaque objet Python a un **en-tête** (compteur de références, type) en plus de ses données. Quelques ordres de grandeur (CPython 64 bits) :

Objet	Taille approximative
`int` (petit)	28 octets
`str` d'un caractère	~50 octets
`list` vide	56 octets (+ 8 par élément)
`tuple` de 3 éléments	64 octets
`dict` vide	64 octets, ~230 avec quelques éléments
Instance de classe simple	~48 octets + son dictionnaire d'attributs (~100+ octets)

Le concept

Où part la mémoire ? (suite)

Une liste d'un million d'entiers occupe plus de **36 Mo** (8 par pointeur + 28 par entier), alors qu'un tableau compact de 4 octets par entier en prendrait **4 Mo**.

Mesurer

- `sys.getsizeof(objet)` : taille **de l'objet seul** (pas de ce qu'il contient).
- `tracemalloc` : trace chaque allocation Python ; on prend un **instantané** (`take_snapshot`), on lit le **courant** et le **pic** (`get_traced_memory`), on compare deux instantanés pour voir ce qui a augmenté.
- `memory_profiler`, `resource.getrusage` (pic de mémoire du processus).

Structures plus compactes

Besoin	Option	Gain
Beaucoup de nombres	`array.array('i')`, NumPy	×5 à ×10

Le concept

Structures plus compactes (suite)

Besoin	Option	Gain
Beaucoup d'objets identiques	classe avec `__slots__` (pas de dictionnaire par instance)	~×2 à ×3
Enregistrements immuables	`tuple`, `NamedTuple`	plus petits que `dict`
Chaînes répétées	**internement** (`sys.intern`), ou stocker un indice dans une table	selon répétition
Booléens en masse	`bytearray`, champs de bits	×8 ou plus
Texte volumineux	`bytes`, `memoryview` (sans copie)	évite les copies

Flux et générateurs

Le concept

Flux et générateurs (suite)

Charger tout en mémoire : `lignes = fichier.readlines()` → O(taille). Traiter ligne par ligne : `for ligne in fichier` → **O(1) en mémoire**. Un **générateur** produit les éléments un par un ; il remplace la liste complète quand on ne parcourt qu'une fois.

```
total = sum(int(l) for l in fichier)        # mémoire constante
total = sum([int(l) for l in fichier])      # construit toute la liste d'abord
```

Copies et vues

Découper (`tranche = donnees[a:b]`) **copie**. Pour de grosses données binaires, `memoryview` donne une **vue** sans copie. Les copies inutiles coûtent à la fois de la mémoire et du temps.

Ramasse-miettes et fuites

Le concept

Ramasse-miettes et fuites (suite)

CPython libère un objet quand son **compteur de références** tombe à zéro ; un ramasse-miettes détecte aussi les **cycles**. Une « fuite » en Python est presque toujours un objet **encore référencé** sans qu'on s'en serve :

- cache ou dictionnaire global qui ne se vide jamais ;
- liste de journaux/historique qui grandit ;
- écouteurs enregistrés jamais retirés ;
- références dans les exceptions et les fermetures.

Diagnostic : comparer deux instantanés `tracemalloc` pris à des moments différents ; les lignes qui grossissent sont les suspectes.

Le concept

Borner les structures

Toute structure qui croît avec le temps doit avoir une **limite explicite** : taille maximale, durée de vie (TTL), politique d'éviction (**LRU** : on retire l'élément le moins récemment utilisé). `functools.lru_cache(maxsize=...)` le fait pour une fonction.

Mémoire et performance

Moins de mémoire utilisée = **meilleure utilisation des caches du processeur** = souvent plus rapide. Mais attention aux compromis : un index en mémoire accélère les recherches mais consomme de la RAM.

> 🧠 ***Astuce*** *: demande-toi toujours « combien de données au pire ? ». Si la réponse est « ça dépend de l'entrée », fais un flux plutôt qu'une liste.*

Exemple pas à pas

Problème : mesurer la mémoire de plusieurs structures avec `tracemalloc`, traiter 1 million de lignes en flux, détecter une fuite par instantanés, et la corriger avec un cache LRU borné.

```
import array, collections, sys, tracemalloc

def pic_memoire(fonction):
    tracemalloc.start()
    try:
        resultat = fonction()
        courant, pic = tracemalloc.get_traced_memory()
    finally:
        tracemalloc.stop()
    return pic, resultat

# ---------------- 1. liste contre générateur ----------------
N = 500000
def avec_liste(): return sum([i * 2 for i in range(N)])
def avec_generateur(): return sum(i * 2 for i in range(N))
```

Exemple pas à pas (suite)

```
pic_l, r1 = pic_memoire(avec_liste)
pic_g, r2 = pic_memoire(avec_generateur)
print(f"liste : pic {pic_l / 1e6:6.2f} Mo | générateur : pic {pic_g / 1e6:6.3f} Mo
    (rapport x{pic_l / pic_g:.0f})")
assert r1 == r2 and pic_l > 100 * pic_g

# ---------------- 2. coût d'un million d'entiers ----------------
def liste_ints(): return list(range(1_000_000))
def tableau_ints(): return array.array("i", range(1_000_000))
pic_li, _ = pic_memoire(liste_ints)
pic_ta, _ = pic_memoire(tableau_ints)
print(f"liste d'entiers : {pic_li / 1e6:5.1f} Mo | array('i') : {pic_ta / 1e6:5.1f
   } Mo")
assert pic_li > 5 * pic_ta

# ---------------- 3. enregistrements : dict, classe, __slots__, tuple -----------
   -----
class Insc:
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    def __init__(self, membre, cours, montant): self.membre, self.cours, self.mont
   ant = membre, cours, montant

class InscSlots:
    __slots__ = ("membre", "cours", "montant")
    def __init__(self, membre, cours, montant): self.membre, self.cours, self.mont
   ant = membre, cours, montant

def construire(fabrique, n=100000):
    return lambda: [fabrique(i % 800, i % 90, i % 200) for i in range(n)]

mesures = {
    "dict":      pic_memoire(construire(lambda a, b, c: {"membre": a, "cours": b,
   "montant": c}))[0],
    "classe":    pic_memoire(construire(Insc))[0],
    "__slots__": pic_memoire(construire(InscSlots))[0],
    "tuple":     pic_memoire(construire(lambda a, b, c: (a, b, c)))[0],
}
for nom, pic in sorted(mesures.items(), key=lambda kv: kv[1]):
```

Exemple pas à pas

Exemple pas à pas (suite)

```
    print(f"  {nom:10s} {pic / 100000:6.0f} octets par enregistrement")
assert max(mesures["tuple"], mesures["__slots__"]) < mesures["classe"] < mesures["
   dict"]

# ---------------- 4. détecter une fuite par comparaison d'instantanés -----------
   -----
HISTORIQUE = []
def traiter_requete(i):
    HISTORIQUE.append({"requete": i, "details": f"{i}-" + "x" * 200})        # 🐛
   conservé pour toujours
    return i

tracemalloc.start()
for i in range(1000): traiter_requete(i)
avant = tracemalloc.take_snapshot()
for i in range(1000, 6000): traiter_requete(i)
apres = tracemalloc.take_snapshot()
croissance = apres.compare_to(avant, "lineno")
suspecte = croissance[0]
```

Exemple pas à pas (suite)

```
print(f"fuite suspectée : {suspecte.traceback[0].filename.split('/')[-1]} ligne {s
   uspecte.traceback[0].lineno}, +{suspecte.size_diff / 1e6:.2f} Mo")
tracemalloc.stop()
assert suspecte.size_diff > 1_000_000 and len(HISTORIQUE) == 6000

# ---------------- 5. la correction : un cache borné (LRU) ----------------
class LRU:
    def __init__(self, taille_max):
        self.taille_max, self.donnees = taille_max, collections.OrderedDict()
    def lire(self, cle):
        if cle in self.donnees:
            self.donnees.move_to_end(cle); return self.donnees[cle]
        return None
    def ecrire(self, cle, valeur):
        self.donnees[cle] = valeur; self.donnees.move_to_end(cle)
        if len(self.donnees) > self.taille_max:
            self.donnees.popitem(last=False)                    # on retire le
   moins récemment utilisé
```

Exemple pas à pas (suite)

```
cache = LRU(500)
def traiter_requete_v2(i):
    cache.ecrire(i, {"requete": i, "details": f"{i}-" + "x" * 200}); return i

tailles = []
for etape in range(5):
    for i in range(etape * 2000, (etape + 1) * 2000): traiter_requete_v2(i)
    tailles.append(len(cache.donnees))
print("taille du cache après chaque lot de 2000 requêtes :", tailles)
assert tailles == [500] * 5

# ---------------- 6. flux : la mémoire reste constante quelle que soit la taille
   du fichier ----------------
def lignes(n):
    for i in range(n): yield f"{i};{i % 97};{(i * 13) % 1000}"

def total_en_flux(n):
    return sum(int(l.split(";")[2]) for l in lignes(n))
```

Exemple pas à pas (suite)

```
p_petit, t1 = pic_memoire(lambda: total_en_flux(10_000))
p_grand, t2 = pic_memoire(lambda: total_en_flux(1_000_000))
print(f"flux : 10 000 lignes → pic {p_petit / 1e3:.1f} Ko | 1 000 000 lignes → pic
    {p_grand / 1e3:.1f} Ko")
assert p_grand < 5 * p_petit + 100_000                                  # 100 fois p
   lus de données, mémoire quasi constante
```

Lecture :

1. Le **générateur** consomme une fraction infime de la mémoire de la liste (le rapport est de plusieurs centaines).
2. Un `array('i')` stocke un million d'entiers en ~4 Mo contre ~36 Mo pour la liste (huit fois plus).
3. Pour 100 000 enregistrements : `__slots__` et `tuple` (≈ 90 octets chacun) < classe ordinaire (≈ 125) < `dict` (≈ 215), soit un facteur proche de 2,5 entre les deux extrêmes.

Exemple pas à pas (suite)

4. La comparaison d'instantanés **désigne la ligne qui fuit** (ici `HISTORIQUE.append`). La correction par un cache **LRU borné** garde la taille stable.
5. Le **flux** traite 100 fois plus de données avec une mémoire quasi inchangée.

À toi, avec aide

Estime et vérifie la mémoire pour stocker **2 millions de relevés de température** (un réel par relevé) :

1. dans une liste de `float` ;
2. dans un `array('d')` ;
3. dans un `bytearray` de relevés convertis en entiers 16 bits (précision 0,01 °C) via `struct` ou `array('h')`.

Donne la mémoire théorique de chaque option, vérifie-la avec `tracemalloc`, et dis laquelle choisir si la précision de 0,01 °C suffit.

À toi, avec aide · réponse

Indices progressifs

Indice 1

Un `float` Python occupe 24 octets + 8 octets de pointeur dans la liste.

Indice 2

`array('d')` : 8 octets par valeur ; `array('h')` : 2 octets.

Indice 3

Mesure le pic avec la fonction `pic_memoire`.

À toi, avec aide · réponse

Réponse

```
import array, tracemalloc, random
rng = random.Random(1)
N = 2_000_000

def pic(f):
    tracemalloc.start(); r = f(); courant, p = tracemalloc.get_traced_memory(); tr
   acemalloc.stop(); return courant, r

valeurs = [rng.uniform(-20, 40) for _ in range(N)]
m_liste, _ = pic(lambda: list(valeurs))                    # copie des pointeurs s
   eulement (les float existent déjà)
m_double, _ = pic(lambda: array.array("d", valeurs))
m_court, _ = pic(lambda: array.array("h", (round(v * 100) for v in valeurs)))
liste_totale = 8 * N + 24 * N                              # pointeurs + objets fl
   oat
print(f"liste de float : ~{liste_totale / 1e6:.0f} Mo (théorique) | array('d') : {
   m_double / 1e6:.1f} Mo | array('h') : {m_court / 1e6:.1f} Mo")
assert abs(m_double - 8 * N) < 0.01 * 8 * N and abs(m_court - 2 * N) < 0.10 * 2 *
   N
```

À toi, avec aide · réponse

Réponse (suite)

Liste de `float` : ≈ **64 Mo** (8 + 24 octets par valeur). `array('d')` : **16 Mo**. `array('h')` en centièmes de degré : **4 Mo**. Si 0,01 °C suffit, l'option 3 divise la mémoire par 16, et améliore aussi l'efficacité du cache processeur. Le prix : une conversion à l'entrée et à la sortie, et une plage limitée (entiers 16 bits : ±327,67 °C, largement suffisante).

À toi, en solo

Traiter un gros fichier sans le charger. Écris `analyser(chemin)` pour un fichier CSV de plusieurs millions de lignes `membre;cours;montant` qui calcule, en une seule passe et en **mémoire bornée** :

1. le nombre de lignes et le montant total ;
2. les 5 membres au montant total le plus élevé (en ne gardant en mémoire qu'une structure bornée : un tas de taille 5 *et* un dictionnaire par membre — explique pourquoi, avec 100 000 membres, cela reste raisonnable, et ce que tu ferais s'il y en avait 100 millions) ;
3. le nombre de membres distincts avec un ensemble **approximatif** économe (filtre de Bloom ou compteur HyperLogLog simplifié) et son erreur ;
4. teste sur un fichier généré (500 000 lignes) ; mesure le pic de mémoire avec `tracemalloc` et vérifie qu'il reste très inférieur à celui d'une version qui charge tout.

À toi, en solo · réponse

Pistes de correction

```
import heapq, math, os, random, tempfile, tracemalloc, hashlib
from collections import defaultdict

def generer(chemin, n=500000, graine=1):
    rng = random.Random(graine)
    with open(chemin, "w") as f:
        for _ in range(n):
            f.write(f"m{int(rng.paretovariate(1.2)) % 20000};HEF-{rng.randrange(90
   ):03d};{rng.randrange(1, 200)}\n")

class Compteur_distincts:
    # estimation de cardinalité de type « probabilistic counting » (registres de r
   ang maximal, HyperLogLog simplifié)
    def __init__(self, p=10):
        self.p, self.m = p, 1 << p
        self.registres = [0] * self.m
    def ajouter(self, x):
        h = int(hashlib.blake2b(x.encode(), digest_size=8).hexdigest(), 16)
        i, reste = h >> (64 - self.p), h & ((1 << (64 - self.p)) - 1)
```

Pistes de correction (suite)

```
        rang = (64 - self.p) - reste.bit_length() + 1
        self.registres[i] = max(self.registres[i], rang)
    def estimation(self):
        alpha = 0.7213 / (1 + 1.079 / self.m)
        e = alpha * self.m * self.m / sum(2.0 ** -r for r in self.registres)
        zeros = self.registres.count(0)
        return self.m * math.log(self.m / zeros) if e <= 2.5 * self.m and zeros el
   se e

def analyser(chemin):
    lignes, total, par_membre, distincts = 0, 0, defaultdict(int), Compteur_distin
   cts()
    with open(chemin) as f:
        for ligne in f:
            membre, _, montant = ligne.rstrip("\n").split(";")
            lignes += 1; total += int(montant); par_membre[membre] += int(montant)
   ; distincts.ajouter(membre)
    top5 = heapq.nlargest(5, par_membre.items(), key=lambda kv: kv[1])
    return lignes, total, top5, distincts.estimation(), len(par_membre)
```

À toi, en solo · réponse

Pistes de correction (suite)

```
def analyser_tout_charger(chemin):
    contenu = [l.rstrip("\n").split(";") for l in open(chemin)]
    return len(contenu), sum(int(c[2]) for c in contenu)

with tempfile.TemporaryDirectory() as d:
    chemin = os.path.join(d, "inscriptions.csv"); generer(chemin)
    tracemalloc.start(); r = analyser(chemin); _, pic_flux = tracemalloc.get_trace
   d_memory(); tracemalloc.stop()
    tracemalloc.start(); analyser_tout_charger(chemin); _, pic_tout = tracemalloc.
   get_traced_memory(); tracemalloc.stop()
lignes, total, top5, estime, exact = r
print(f"{lignes} lignes, total {total}, top 5 {[m for m, _ in top5]}")
print(f"membres distincts : exact {exact}, estimé {estime:.0f} (erreur {abs(estime
    - exact) / exact:.1%})")
print(f"pic mémoire : flux {pic_flux / 1e6:.1f} Mo contre {pic_tout / 1e6:.1f} Mo
   en chargeant tout")
assert lignes == 500000 and pic_flux < pic_tout / 10 and abs(estime - exact) / exa
   ct < 0.10
```

Pistes de correction (suite)

Le flux ne conserve qu'une **ligne** à la fois plus le dictionnaire des totaux par membre (100 000 membres : quelques Mo). Avec **100 millions** de membres, le dictionnaire exact ne tiendrait plus : on passerait à un **algorithme approximatif** pour les plus gros (*Count-Min Sketch* + tas) ou à un traitement **distribué/par partitions** (hachage du membre pour répartir). L'estimateur de cardinalité fournit une valeur à quelques pourcents près avec une **mémoire fixe de 1024 registres**.

Pièges classiques

Piège	Pourquoi c'est un problème	Comment l'éviter
Charger tout un fichier en mémoire	Ça tient aujourd'hui, pas demain	Flux ligne par ligne, générateurs
Caches et historiques non bornés	Fuite mémoire lente puis arrêt brutal	Taille max, TTL, éviction LRU
Copier des données sans nécessité	Mémoire et temps gaspillés	Vues (`memoryview`), tranches avec prudence
Ne pas mesurer	On optimise la mauvaise structure	`tracemalloc`, instantanés comparés

Mini-boss

Mini-boss

Budget mémoire du traitement des certificats. Un traitement lit un fichier de 5 millions d'inscriptions (génère-le à échelle réduite, 500 000 lignes) et doit produire des statistiques par cours, par mois et par membre, avec un budget de **50 Mo** de pic.

1. écris la version naïve (tout charger) et mesure son pic ;
2. identifie les trois plus gros consommateurs avec des instantanés `tracemalloc` ;
3. réécris en flux avec des structures compactes (`__slots__` ou tuples, `array`, internement des chaînes répétées) ;
4. borne tout cache et expose la taille de chaque structure ;
5. prouve l'équivalence des résultats avec la version naïve ;
6. tableau : version, pic mémoire, temps, facteur de réduction ;
7. extrapole : à 5 millions de lignes, la version finale respecte-t-elle le budget ? Justifie.

Mini-boss · réponse

Pistes de correction

Un bon travail : mesure du pic (naïf), comparaison de snapshots désignant les listes complètes, les dictionnaires par enregistrement, les chaînes dupliquées; version finale avec lecture en flux, `sys.intern` pour les codes de cours, `array` pour les montants, cache LRU; résultats identiques; tableau des facteurs; extrapolation par régression linéaire entre deux tailles (la mémoire de la version finale doit rester presque constante sauf pour le dictionnaire des membres, dont on calcule la taille en fonction du nombre de membres distincts).

Quiz éclair

Quiz éclair

1. Pour traiter un fichier plus gros que la mémoire, on utilise :

A. un flux ou un générateur, ligne par ligne
B. une liste de toutes les lignes
C. un second écran

2. Une fuite mémoire en Python correspond le plus souvent à :

A. un manque de RAM sur la machine
B. des objets encore référencés (cache, historique) qu'on n'utilise plus
C. une erreur de syntaxe

3. `__slots__` permet de :

A. accélérer le réseau

Quiz éclair

Quiz éclair (suite)

B. chiffrer les attributs
C. réduire la mémoire des instances en évitant un dictionnaire par objet

4. 🔁 *Révision mission 03* : Dans un profil, `tottime` mesure :

A. le temps passé dans la fonction elle-même, sans ses sous-appels
B. le temps total du programme
C. le nombre d'appels

5. 🔁 *Révision mission 03* : La première étape d'une optimisation est de :

A. réécrire le code en C
B. profiler pour identifier le point chaud
C. ajouter du cache partout

Quiz éclair · réponse

Réponses

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

Je réfléchis

1. Quelle structure de ton code grossit avec le temps sans jamais être vidée ?
2. Si tes données doublaient demain, qu'est-ce qui casserait en premier côté mémoire ?

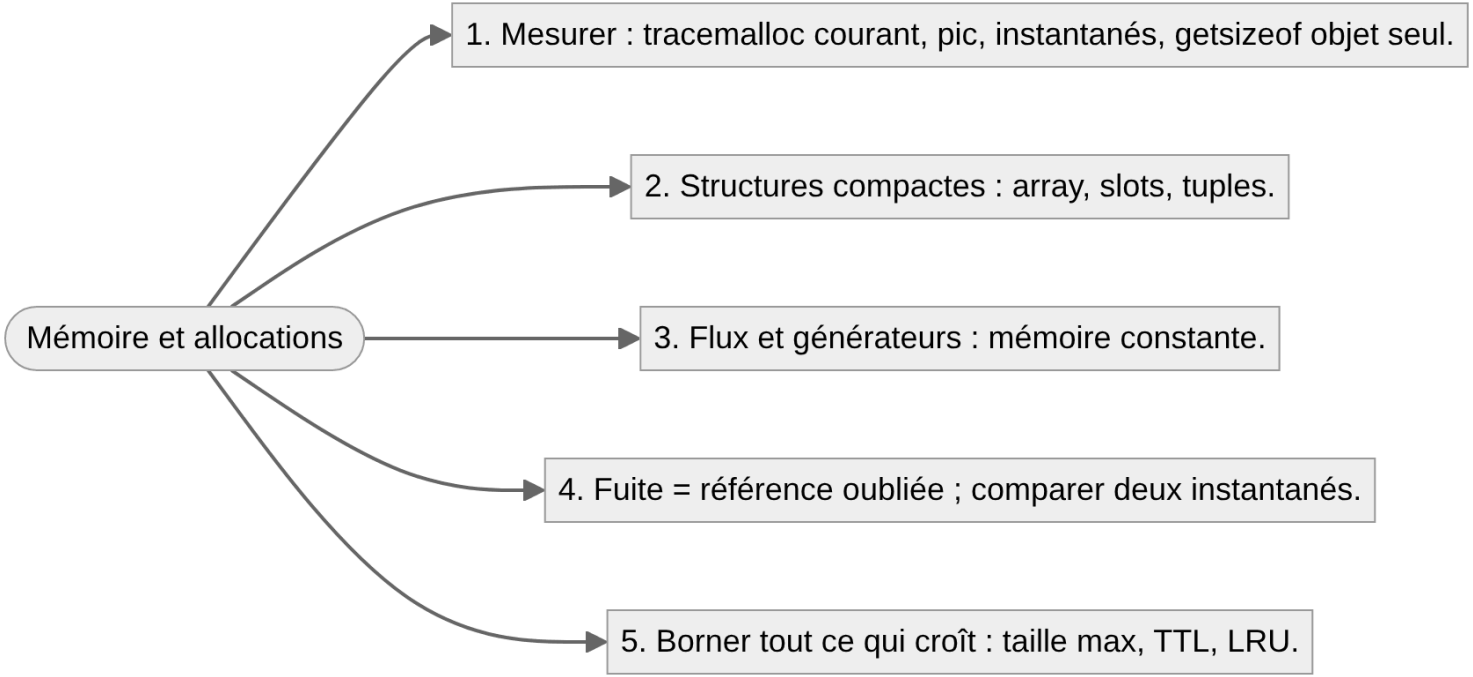
À retenir

À retenir

1. Mesurer : `tracemalloc` (courant, pic, instantanés), `getsizeof` (objet seul).
2. Structures compactes : `array`, `__slots__`, tuples.
3. Flux et générateurs : mémoire constante.
4. Fuite = référence oubliée ; comparer deux instantanés.
5. Borner tout ce qui croît : taille max, TTL, LRU.

À retenir

Le schéma de la mission



MISSION ACCOMPLIE

+100 XP

Badge débloqué : Économe de mémoire

Prochaine mission : I/O et stockage

