TP Chapitre 4 : Récursivité

Objectifs du TP

Objectifs du TP. À l'issue de cette séance, tu dois être capable de :

— écrire une fonction récursive, c'est-à-dire une fonction qui s'appelle elle-même ;

— identifier le cas d'arrêt (ou cas de base) et l'appel récursif d'une fonction récursive ;

— analyser le fonctionnement d'un programme récursif : pile des appels, environnement propre à chaque appel, déroulé des retours ;

— vérifier que l'appel récursif modifie l'argument vers le cas d'arrêt, pour garantir que la fonction se termine ;

— comparer les versions itérative et récursive d'un même algorithme, et expliquer l'intérêt et les limites de la récursivité ;

— expliquer et éviter l'erreur `RecursionError` (limite de profondeur de la pile des appels).

Consignes générales. Tous les exercices se traitent en Python (console ou fichier). Enregistre régulièrement ton travail. Quand un programme ne fonctionne pas, lis le message d'erreur : il indique souvent la ligne et le type d'erreur. Attention à la `RecursionError` : elle signale une récursivité trop profonde, souvent à cause d'un cas d'arrêt manquant ou d'un appel récursif qui ne modifie pas l'argument. Pour chaque fonction récursive que tu écris, identifie systématiquement le cas d'arrêt et l'appel récursif. La partie B se fait *en autonomie* : tu écris des programmes entiers à partir d'une spécification, aucune ligne de code n'est fournie.

Partie A — Application guidée

Les exercices suivants se programment en Python. Ils sont classés du plus simple au plus difficile. Chaque exercice se termine par un *point de validation* (encadré vert) : le résultat exact que doit produire ton programme pour être juste. Complète les squelettes de code en respectant l'indentation : à chaque commentaire `# A COMPLETER`, écris le code Python qui réalise ce qui est décrit, à la place du commentaire.

Exercice A1 — La factorielle : première fonction récursive

Un gérant veut connaître le nombre de façons de ranger n articles sur une étagère : c'est la **factorielle** de n, notée $n!$, définie par $n! = n \times (n-1) \times \cdots \times 2 \times 1$. On peut aussi la voir de façon récursive : $n! = n \times (n-1)!$, avec $0! = 1$ et $1! = 1$. Par exemple, $5! = 5 \times 4! = 5 \times 4 \times 3! = \cdots = 120$.

Compléter la fonction `factorielle` : identifier le *cas d'arrêt* (le cas où la fonction répond directement, sans s'appeler elle-même) et l'*appel récursif* (l'appel de la fonction sur un paramètre modifié).

```
def factorielle(n):
    if n <= 1:
        return 1
    else:
        # A COMPLETER : renvoyer n * factorielle(n - 1)

print(factorielle(5))
```

Point de validation

Vérifie ton programme : `factorielle(0)` doit renvoyer 1, `factorielle(1)` doit renvoyer 1 et `factorielle(5)` doit renvoyer `120`.

Exercice A2 — Intérêts composés récursifs : le livret d'épargne

Léa place 200 € sur un livret d'épargne rémunéré à 3 % par an. Chaque année, les intérêts sont ajoutés au capital : le solde est multiplié par 1,03. On veut une fonction récursive `capital(annee)` qui renvoie le solde du livret après un nombre donné d'années : le solde après n années vaut le solde après $n-1$ années multiplié par 1,03, et le solde après 0 année vaut le dépôt initial.

Compléter la fonction : le *cas d'arrêt* correspond à l'année 0 (le dépôt initial, sans intérêts), l'*appel récursif* calcule le capital de l'année précédente puis applique les intérêts.

```
def capital(annee):
    if annee == 0:
        return 200
    else:
        # A COMPLETER : renvoyer capital(annee - 1) * 1.03

print(capital(5))
```

Point de validation

Vérifie ton programme : `capital(0)` doit afficher `200` ; `capital(5)` doit afficher environ `231.85` ; `capital(10)` environ `268.78` (un léger écart sur les décimales est normal, c'est la représentation des flottants).

Exercice A3 — Le total des abonnements : somme récursive d'une liste

Tu veux connaître le montant total versé pour un abonnement à un service en ligne : les paiements mensuels sont stockés dans une liste de nombres. On veut une fonction récursive `somme_paiements(liste)` qui additionne tous les montants de la liste. L'idée : la somme d'une liste vide vaut 0 (c'est le *cas d'arrêt*) ; sinon, la somme vaut le premier élément *plus* la somme du reste de la liste (*appel récursif*).

Compléter la fonction :

```
def somme_paiements(liste):
    if not liste:
        return 0
    else:
        # A COMPLETER : additionner liste[0] et la somme du reste liste[1:]

print(somme_paiements([10, 20, 30]))
```

Point de validation

Vérifie ton programme : `somme_paiements([])` doit renvoyer `0` ; `somme_paiements([10, 20, 30])` doit renvoyer `60` ; `somme_paiements([12.99, 12.99, 12.99])` doit renvoyer `38.97`.

Exercice A4 — Le décompte d'un abonnement : compte à rebours récursif

L'essai gratuit d'un service payant se termine dans n jours. On veut une fonction récursive `compte_a_rebours(n)` qui affiche le décompte des jours restants, puis le message `Abonnement termine !` quand il ne reste plus aucun jour. Le *cas d'arrêt* est $n = 0$: on affiche le message de fin sans appel récursif. Sinon, on affiche le nombre de jours restants puis on rappelle la fonction sur $n-1$.

Compléter la fonction :

```
def compte_a_rebours(n):
    if n == 0:
        print("Abonnement termine !")
```

```
    else:
        # A COMPLETER : afficher "Jours restants : n" puis appeler
        # compte_a_rebours(n - 1)

compte_a_rebours(3)
```

Point de validation

Vérifie ton programme : `compte_a_rebours(3)` doit afficher : `Jours restants : 3`, puis `Jours restants : 2`, puis `Jours restants : 1`, puis `Abonnement termine !`.

Partie B — En autonomie

Tu écris maintenant des programmes *entiers*, sans squelette. Pour chaque exercice : lis la spécification (entrées, traitement, sorties, contraintes), réponds d'abord aux questions de conception de l'encadré orange (sur papier ou à voix haute), puis écris ton programme dans un fichier et teste-le avec les valeurs de l'encadré vert. L'objectif est de concevoir toi-même la solution : ne demande pas de lignes de code toutes faites. Pour chaque fonction récursive, identifie le cas d'arrêt et l'appel récursif, et vérifie que l'appel récursif modifie l'argument vers le cas d'arrêt (sinon : `RecursionError`).

Exercice B1 — La suite de Fibonacci : versions récursive et itérative

La suite de Fibonacci est définie par $F(0) = 1$, $F(1) = 1$ et, pour tout entier positif n, $F(n) = F(n-1) + F(n-2)$. C'est un modèle classique de croissance (démographie, propagation d'une information, coût d'un algorithme).

Énoncé. Écrire deux fonctions qui renvoient $F(n)$:

— une fonction récursive `fibonacci_rec(n)` qui traduit directement la définition (deux appels récursifs) ;

— une fonction itérative `fibonacci_iter(n)` qui ne garde en mémoire que les deux derniers termes (une boucle `for` et une affectation simultanée `a, b = b, a + b`).

Puis, sur l'exemple $n = 6$, déterminer le nombre d'appels de la fonction déclenchés par la version récursive, en s'aidant de l'arbre des appels du cours.

Spécification.

— *Entrée* : un entier $n \geq 0$.

— *Traitement* : version récursive : cas d'arrêt $n \leq 1$ (renvoie 1), sinon $F(n-1) + F(n-2)$; version itérative : boucle `for` de n étapes en conservant les deux derniers termes.

— *Sortie* : la valeur $F(n)$, renvoyée par chacune des deux fonctions.

— *Contraintes* : les deux fonctions doivent renvoyer le même résultat pour toute valeur de n ; pas de liste pour mémoriser toute la suite.

Pour t'aider — questions de conception (sans donner le code)

— Pourquoi la version récursive est-elle si simple à écrire ? (elle traduit la définition presque mot pour mot)

— Combien d'appels déclenche `fibonacci_rec(4)` ? et `fibonacci_rec(6)` ? (aide-toi de l'arbre des appels du cours : `fibonacci(2)` y est calculé plusieurs fois)

— Quel est l'inconvénient du calcul récursif de Fibonacci ? (certains calculs sont répétés)

— Pour la version itérative : quelles variables faut-il conserver d'une étape à l'autre ? Pourquoi deux suffisent-elles ?

— Que renvoie `fibonacci_iter(0)` ? Vérifie que c'est bien 1.

Point de validation

Vérifie tes deux fonctions : `fibonacci_rec(6)` et `fibonacci_iter(6)` doivent toutes deux renvoyer `13` ; la version récursive déclenche **25 appels** de la fonction pour $n = 6$ (c'est le nombre d'appels indiqué dans le cours).

Exercice B2 — Répartition récursive de gains : quotient et reste par soustractions

Trois gagnants se partagent équitablement un gain de 2500 € : chacun reçoit la même part entière (le *quotient* de 2500 par 3), et il reste une somme non divisible (le *reste*). On rappelle la division euclidienne : $2500 = 3 \times 833 + 1$, donc chaque gagnant reçoit 833 € et il reste 1 €.

Énoncé. Écrire deux fonctions récursives qui calculent le quotient et le reste de la division entière de a par b ($a \geq 0$, $b > 0$) *sans utiliser* les opérateurs `//` ni `%`, uniquement par soustractions successives :

— `quotient_rec(a, b)` : tant qu'on peut soustraire b à a, on compte une soustraction de plus (le quotient est le nombre de soustractions) ; quand $a < b$, le quotient vaut 0 ;

— `reste_rec(a, b)` : quand $a < b$, le reste vaut a ; sinon, on soustrait b à a et on continue.

Spécification.

— *Entrées* : deux entiers $a \geq 0$ et $b > 0$.

— *Traitement* : récursivement, par soustractions successives de b à a, jusqu'à ce que $a < b$ (cas d'arrêt).

— *Sorties* : `quotient_rec` renvoie le nombre de soustractions (le quotient) ; `reste_rec` renvoie la valeur de a quand $a < b$ (le reste).

— *Contraintes* : interdiction d'utiliser `//` et `%` ; les deux fonctions sont récursives ; pour $a = 2500$ et $b = 3$, le nombre de soustractions (833) reste sous la limite de profondeur de récursivité, mais un très grand a peut la dépasser : c'est justement ce qu'on observe.

Pour t'aider — questions de conception (sans donner le code)

— Quel est le cas d'arrêt de `quotient_rec` ? Pourquoi la fonction doit-elle renvoyer 0 dans ce cas ?

— Pourquoi l'appel récursif soustrait-il b à a ? En quoi cela modifie-t-il l'argument vers le cas d'arrêt ?

— Quel est le lien entre le quotient et le nombre de soustractions effectuées ? (d'où vient le `1 +` de l'appel récursif ?)

— Comment adapter la fonction pour obtenir le reste ? Que doit renvoyer la fonction quand $a < b$?

— Que se passe-t-il avec $a = 3000$ et $b = 3$? (teste mentalement le nombre de soustractions, puis observe l'erreur Python — nomme-la)

Point de validation

Vérifie tes fonctions : `quotient_rec(10, 3)` doit renvoyer `3` et `reste_rec(10, 3)` doit renvoyer `1` ; `quotient_rec(2500, 3)` doit renvoyer `833` et `reste_rec(2500, 3)` doit renvoyer `1` (car $2500 = 3 \times 833 + 1$).

Exercice B3 — Remises successives : une remise après l'autre

Pendant les soldes, un magasin applique une remise de r % sur le prix affiché. On veut calculer le prix après n *remises successives* de r % (par exemple pendant une semaine de promos cumulées). Après une remise, le prix est multiplié par $1 - r/100$; après n remises, on applique cette multiplication n fois.

Énoncé. Écrire une fonction récursive `prix_remise(prix, remise, n)` qui renvoie le prix après n remises successives de `remise` % : cas d'arrêt $n = 0$ (le prix n'est pas modifié) ; sinon, on applique une remise au prix courant et on rappelle la fonction sur $n - 1$ (on peut soit décrémenter n, soit modifier le prix — choisis ta modélisation et explique-la).

Spécification.

— *Entrées* : un prix initial (flottant positif), un taux de remise `remise` (entier entre 0 et 100), un nombre de remises n (entier ≥ 0).

— *Traitement* : multiplier le prix par $1 -$ `remise`$/100$ à chaque remise ; récursion jusqu'à $n = 0$.

— *Sortie* : le prix final après les n remises (flottant).

— *Contraintes* : fonction récursive ; interdiction d'utiliser la puissance `**` (c'est justement ce que la récursion remplace ici) ; le prix ne doit jamais devenir négatif.

Pour t'aider — questions de conception (sans donner le code)

— Quel est le cas d'arrêt ? Que vaut le prix quand $n = 0$?

— Que devient le prix après une seule remise de `remise` % ? (écris la formule en français)

— Quel argument l'appel récursif doit-il modifier pour se rapprocher du cas d'arrêt : `n`, `prix`, ou les deux ? Justifie.

— Deux remises successives de 20 % équivalent-elles à une remise de 40 % ? (calcule $0{,}8 \times 0{,}8$ et compare à 0,6) — c'est un piège classique des soldes.

Point de validation

Vérifie ta fonction : `prix_remise(100, 20, 1)` doit renvoyer `80.0` ; `prix_remise(100, 20, 2)` doit renvoyer `64.0` ; `prix_remise(100, 20, 3)` doit renvoyer `51.2`.

Exercice B4 — Contrôle des transactions : une valeur est-elle dans la liste ?

Tu vérifies ta liste d'opérations bancaires : un remboursement de 42,50 € est-il bien arrivé sur ton compte ? Les opérations sont stockées dans une liste de montants. On veut une fonction récursive `est_dans_list(valeur, liste)` qui renvoie `True` si `valeur` est dans la liste, `False` sinon.

Énoncé. Écrire une fonction récursive `est_dans_list(valeur, liste)` : il y a *deux cas d'arrêt* — si la liste est vide, la valeur n'y est pas (renvoyer `False`) ; si le premier élément est égal à la valeur cherchée, elle y est (renvoyer `True`). Sinon, on cherche dans le reste de la liste (appel récursif sur `liste[1:]`).

Spécification.

— *Entrées* : une valeur (nombre) et une liste de nombres.

— *Traitement* : comparer le premier élément à la valeur ; si égal, renvoyer `True` ; sinon chercher dans le reste ; liste vide → `False`.

— *Sortie* : un booléen `True` ou `False`.

— *Contraintes* : fonction récursive ; pas de mot-clé `in` ; la liste n'est pas modifiée (on travaille sur des copies `liste[1:]`).

Pour t'aider — questions de conception (sans donner le code)

— Quels sont les deux cas d'arrêt ? Pourquoi la liste vide est-elle un cas d'arrêt ?

— Pourquoi tester le premier élément *avant* de faire l'appel récursif ? (que se passerait-il si on cherchait toujours dans le reste ?)

— Que se passe-t-il si la valeur n'est pas dans la liste ? (suis le déroulé jusqu'à la liste vide)

— L'opérateur `==` sur des flottants : avec `42.5`, `7.5`, `12.0`, aucune surprise ; pourquoi vaut-il mieux, en général, travailler en centimes (entiers) pour de l'argent ?

Point de validation

Vérifie ta fonction : `est_dans_list(42.5, [12.0, 7.5, 42.5, 3.0])` doit renvoyer `True` ; `est_dans_list(99.0, [12.0, 7.5, 42.5, 3.0])` doit renvoyer `False` ; `est_dans_list(1.0, [])` doit renvoyer `False`.

Partie C — Exercice de réflexion (à finir à la maison)

À finir à la maison

Mon épargne récursive : simulateur de capital avec objectif.

Contexte. Tu prépares un achat important (ordinateur portable, permis de conduire, voyage...). Tu ouvres un compte d'épargne avec un dépôt initial, tu verses chaque année une somme fixe (ton apport annuel), et le compte est rémunéré à un taux annuel : à la fin de chaque année, le capital est multiplié par $(1 + \text{taux})$, puis ton apport de l'année suivante s'y ajoute.

Objectif. Écrire un programme *récursif* qui simule l'évolution de ton épargne, année après année, et réponde à des questions comme : au bout de combien d'années mon objectif est-il atteint ? Combien ai-je versé au total ? Combien d'intérêts ai-je gagnés ? On pourra s'aider des exercices A2 (intérêts composés récursifs) et B2 (cas d'arrêt et garde-fou).

Questions à se poser avant de coder.

— Quelles sont les données d'entrée ? (dépôt initial, apport annuel, taux annuel, objectif)

— Comment écrire le solde après n années de façon récursive ? Quel est le cas d'arrêt ? Quel est l'appel récursif ? (revois l'exercice A2)

— Comment trouver l'année où l'objectif est atteint ? Quelle fonction récursive peut renvoyer cette année ? (le solde courant doit être transmis d'un appel à l'autre)

— Comment garantir que le programme se termine, même si l'objectif est irréaliste ? (garde-fou : nombre maximal d'années, par exemple 50 — sinon `RecursionError`)

— Comment vérifier que le programme est juste ? (penser à un cas simple : taux 0 % ; comparer avec la version itérative que tu sais déjà écrire)

Étapes suggérées.

1. Version 0, sans intérêts (taux 0 %) : après n années, le solde vaut dépôt $+n\times$ apport. Vérifier à la main (dépôt 1000, apport 120, 5 ans $\to$ 1600).
2. Version 1 : ajouter les intérêts annuels de 3 % — attention à l'ordre : intérêts puis apport, ou apport puis intérêts ? Choisis une convention et garde-la.
3. Version 2 : fonction récursive qui renvoie l'année où le solde atteint l'objectif, avec un garde-fou de 50 ans. Deux cas d'arrêt : objectif atteint (renvoyer l'année) ou garde-fou dépassé (renvoyer une valeur spéciale, par exemple -1).
4. Version 3 : afficher un bilan : année, capital, total versé, intérêts cumulés.

5. Bonus : tester le cas « doublement du capital » sans apport : avec 200 € à 3 % par an, le capital double-t-il bien en 24 ans (repère vu en Première) ? Que se passe-t-il si le taux passe à 4 % ? Compare ta version récursive avec une version itérative.

Contraintes. Données crédibles (taux entre 0 % et 5 %, apport entre 50 € et 300 € par an, objectif entre 500 € et 5000 €) ; garde-fou obligatoire ; programme testé avec taux 0 % (cas simple) et commenté ; les messages peuvent être écrits sans accents (ex. `Annee`, `interets`) si ton éditeur pose problème.

Fin du TP — la partie C est à finir à la maison.