COURS

Python — Programmation orientée objet et récursivité

1re

Programme de première générale, enseignement de spécialité — BO spécial n°1 du 22 janvier 2019

> Affectez une liste à une nouvelle variable, modifiez la seconde, et regardez la première : elle a changé aussi. Ce n'est pas un piège du langage, c'est ce que fait une affectation — elle donne un second nom au même objet. Beaucoup de bugs incompréhensibles tiennent à cette seule phrase.

1 Variables, noms et objets

RÈGLE

Affecter une variable ne **copie pas** l'objet : cela lui donne un **second nom**, et modifier l'un modifie ce que voit l'autre. Pour les objets **immuables** — entiers, chaînes, tuples — la question ne se pose pas, puisqu'on ne peut pas les modifier en place. Pour les objets **modifiables** — listes, dictionnaires — elle se pose à chaque affectation, à chaque passage en argument de fonction et à chaque valeur par défaut.

$$b = anecréepasunnouvelobjet : aetbnommentlemême$$

LE PIÈGE QUI COÛTE LE PLUS DE POINTS

Croire qu'affecter une liste à une nouvelle variable en crée une copie indépendante. — Les deux noms désignent le **même objet en mémoire** : toute modification faite par l'un est visible par l'autre. L'effet se propage aux **arguments de fonction** — une fonction qui modifie une liste reçue modifie celle de l'appelant — et il est d'autant plus déroutant qu'il ne se manifeste pas avec des entiers ou des chaînes, sur lesquels les débutants testent d'abord. Pour obtenir une vraie copie, il faut la demander explicitement.

Le réflexe : Objet **modifiable** : se demander si l'on veut un **alias** ou une **copie**.

2 La programmation orientée objet

DÉFINITION

Une **classe** est un **modèle** : elle décrit les **attributs** — les données — et les **méthodes** — les opérations — que partageront ses objets. Une **instance** est un objet construit à partir de ce modèle. Le constructeur initialise les attributs propres à chaque instance : deux objets d'une même **classe** partagent leurs méthodes et possèdent chacun **ses** valeurs d'attributs.

DÉFINITION

L'**encapsulation** consiste à regrouper dans un même objet les données et les opérations qui les manipulent, et à **n'exposer** qu'une interface d'usage. Son intérêt n'est pas la dissimulation : c'est de garantir que l'objet reste dans un état **cohérent**. Si toute modification passe par une méthode, cette méthode peut vérifier ce qu'elle reçoit ; si l'attribut est modifié directement de l'extérieur, aucune vérification n'est possible.

DÉFINITION

L'**héritage** permet à une classe de **réutiliser** les attributs et méthodes d'une autre, en les **spécialisant**. La classe dérivée peut ajouter des membres et **redéfinir** une méthode existante. La relation à vérifier est « **est un** » : un carré est un rectangle, donc l'héritage se discute. En revanche une voiture n'est pas un moteur — elle en **possède** un, et c'est une composition, non un **héritage**.

3 La récursivité

RÈGLE

Une fonction **récursive** s'appelle elle-même. Elle exige **deux** éléments, et l'oubli du premier est l'erreur la plus fréquente : un **cas de base**, qui renvoie une valeur sans appel récursif, et un **appel récursif** qui se rapproche **strictement** de ce cas de base. Sans cette progression, les appels s'empilent jusqu'à saturer la **pile d'appels**.

$$casdebase + progressionverscecas = terminaison$$

EXEMPLE

Écrire la factorielle de façon récursive et vérifier sa terminaison.

1. **Cas de base** : la **factorielle** de 0 vaut 1, sans aucun appel récursif.
2. **Appel récursif** : la factorielle de n vaut n multiplié par la factorielle de n − 1.
3. La progression est stricte — n diminue de 1 à chaque appel — donc le cas de base est atteint. Pour n = 4 :
$$4 \times 3 \times 2 \times 1 = 24$$

factorielle(4) = 24

Chaque appel non terminé occupe une place dans la **pile d'appels**, ce qui limite la profondeur possible. Une **récursivité** naïve sur la suite de Fibonacci recalcule en outre les mêmes valeurs un nombre considérable de fois : elle est correcte et **inutilisable** au-delà de quelques dizaines de termes. Correction et efficacité sont deux propriétés distinctes.

4 Exceptions et modules

PROPRIÉTÉ

Une **exception** signale une situation qu'une fonction ne peut pas traiter elle-même : fichier absent, division par zéro, conversion impossible. Le bloc *try/except* permet de la **rattraper** et de décider quoi faire. Deux règles d'usage : rattraper le type d'**exception précis** que l'on sait traiter, et ne jamais rattraper toutes les exceptions pour n'en rien faire — un programme qui masque ses erreurs échoue silencieusement, ce qui est pire qu'un arrêt net.

Un **module** est un fichier de code réutilisable qu'on importe. La bibliothèque standard en fournit pour les mathématiques, l'accès au système de fichiers, la lecture de fichiers tabulaires, les dates. Importer un **module** existant plutôt que réécrire une fonction n'est pas de la paresse : le code importé est testé par bien plus d'utilisateurs que le vôtre ne le sera.

ÉCRIRE UNE FONCTION RÉCURSIVE QUI SE TERMINE

1. Écrire d'abord le **cas de base** : quelle est la plus petite entrée, et que renvoie-t-on alors ?
2. Formuler le cas général **en fonction d'un cas plus petit**, sans chercher à dérouler les appels.
3. Vérifier que chaque appel **se rapproche strictement** du cas de base.
$$4 \times 3 \times 2 \times 1 = 24$$
n décroît de 1 à chaque appel et atteint 0.
4. Tester sur la plus petite entrée, puis sur la suivante : les deux premiers cas valident la structure.
5. Vérifier la **profondeur** atteinte, et le nombre d'appels répétés avant de conclure à l'efficacité.

À VÉRIFIER

- Ai-je distingué un **alias** d'une **copie** pour mes objets modifiables ?
- Ma fonction modifie-t-elle une liste reçue en argument ? Est-ce voulu ?
- Ma **classe** distingue-t-elle **attributs** et **méthodes** ?
- Mon **héritage** passe-t-il le test « **est un** » ?
- Ma fonction **récursive** a-t-elle un **cas de base** atteint à coup sûr ?
- Mes **exception**s rattrapées sont-elles **précises** et effectivement traitées ?

À RETENIR

- Une affectation donne un **second nom**, elle ne **copie pas**.
- Objets **immuables** : entiers, chaînes, tuples. **Modifiables** : listes, dictionnaires.
- Une fonction qui modifie une liste reçue modifie celle de l'**appelant**.
- **Classe** : le modèle. **Instance** : l'objet construit. **Attributs** et **méthodes**.
- L'**encapsulation** garantit un état **cohérent**, elle ne dissimule pas.
- **Héritage** : vérifier la relation « **est un** » — sinon c'est une composition.
- **Récursivité** : un **cas de base** et une **progression stricte** vers lui.
- Sans cas de base, la **pile d'appels** sature.
- Une récursivité peut être **correcte** et **inutilisable** : Fibonacci naïf.
- Rattraper une **exception précise** et la traiter ; jamais masquer en silence.
- Importer un **module** éprouvé pour ce qui est difficile à écrire correctement.