Programmation fonctionnelle : activités de découverte

Trois styles, une pensée différente

Remarque 1 (Avant de commencer). Ces activités ont pour but de vous faire *découvrir* les idées-clés de la programmation fonctionnelle avant l'étude du cours. Un même problème peut s'écrire de façons très différentes en Python : avec des variables modifiables et des boucles (style *impératif*), ou au contraire en composant des fonctions, sans modifier quoi que ce soit (style *fonctionnel*). Nous allons comparer ces deux approches sur des problèmes simples, pour mettre en évidence ce qui change vraiment : la manière de *penser* le calcul.

① Un même problème, trois styles

On dispose d'une liste d'entiers représentant des notes sur 20 : `notes = [8, 15, 12, 7, 18, 9, 14]`. On cherche à en calculer la somme.

Exercice 1 *Somme en style impératif* **(2 points)** On écrit la fonction `somme_imp` qui renvoie la somme des éléments d'une liste en utilisant une boucle et un accumulateur :

```
def somme_imp(liste):
    total = 0
    for x in liste:
        total = total + x
    return total
```

1. Recopier et compléter le tableau suivant qui donne l'état de la variable `total` *après chaque itération* de la boucle lors de l'appel `somme_imp([8, 15, 12, 7, 18, 9, 14])`.

étape	1	2	3	4	5	6	7
x	8	15	12	7	18	9	14
`total` **après**							

2. Combien de fois la variable `total` est-elle *modifiée* au cours du calcul ? Que contient-elle à la fin ?

Exercice 2 *Somme en style fonctionnel récursif* **(3 points)** On veut désormais écrire la même fonction, mais *sans aucune boucle et sans aucune variable modifiable*. Une seule écriture reste possible : la récursivité.

On remarque que la somme d'une liste se décompose en deux cas :

— **cas de base** : la somme d'une liste vide vaut 0 ;

— **cas général** : la somme d'une liste non vide est son premier élément, plus la somme du reste de la liste.

En Python, le premier élément de la liste `L` se note `L[0]` et le reste se note `L[1:]`.

1. Compléter la fonction suivante, en remplaçant chaque groupe de pointillés par l'expression correcte :

```
def somme_rec(liste):
    if liste == []:
```

```
        return ........
    else:
        return ........ + somme_rec(........)
```

2. Détailler sur le papier la cascade d'appels déclenchée par `somme_rec([8, 15, 12])`. Combien d'appels récursifs sont effectués pour une liste de longueur n ?
3. La fonction `somme_rec` utilise-t-elle une *variable modifiable* ? Une *boucle* ? Justifier.

Exercice 3 *La même chose* **(1 point)** Python dispose nativement d'une fonction `sum` qui renvoie la somme des éléments d'une liste. En une ligne, on peut donc écrire :

```
somme = sum(notes)
```

1. Comparer cette écriture aux deux versions précédentes. Laquelle est la plus courte ? La plus lisible ?
2. La fonction `sum` est-elle une *instruction* ou une *expression* ? En quoi cela ressemble-t-il à l'écriture mathématique $\sum_{i=0}^{n-1} \texttt{notes}[i]$?

Remarque 2 (Premier bilan). En style **impératif**, on décrit le calcul comme une succession d'*instructions* qui modifient l'état de variables. En style **fonctionnel**, on ne modifie rien : on compose des fonctions et on écrit des *expressions* qui « sont » la valeur cherchée. La boucle disparaît au profit de la récursivité ou d'une fonction d'ordre supérieur.

② De `def` à `lambda`

En style fonctionnel, une fonction est une *expression* comme une autre. On peut la définir par le mot-clé `def`, comme d'habitude, mais aussi de façon plus compacte grâce à l'expression conditionnelle `a if cond else b` et à la notation `lambda`.

Exercice 4 *Trois écritures équivalentes* **(3 points)** Voici trois écritures Python de la fonction « valeur absolue ».

(A) Avec `if/else` classique.

```
def val_abs(x):
    if x >= 0:
        return x
    else:
        return -x
```

(B) Avec expression conditionnelle.

```
def val_abs(x):
    return x if x >= 0 else -x
```

(C) Avec `lambda`.

```
val_abs = lambda x: x if x >= 0 else -x
```

1. Vérifier, sur les entiers -5, 0 et 3, que les trois versions renvoient bien la même valeur.

2. On remarque que la ligne `return x if x >= 0 else -x` n'est pas une instruction `if/else` classique mais une *expression conditionnelle* : elle produit directement une valeur (soit x, soit $-x$) sans « entrer » ni dans un bloc `if` ni dans un bloc `else`. À quelle construction mathématique cette expression ressemble-t-elle ?
3. Réécrire les trois fonctions suivantes dans les deux autres écritures :
   a) `est_pair(n)` qui vaut `True` si n est pair, `False` sinon ;
   b) `maximum(a, b)` qui renvoie le plus grand des deux nombres a et b ;
   c) `prix_avec_remise(p)` qui vaut p si $p < 50$, et $0{,}9 \times p$ sinon.

Exercice 5 *Pourquoi `lambda` ?* **(2 points)** Voici deux appels à la fonction `sorted` de Python, qui trie une liste. Le paramètre `key` (nommé) attend une *fonction* qui calcule la « clé » utilisée pour la comparaison.

```
animaux = ['veau', 'vache', 'cochon', 'anaconda', 'chat', 'chien']
print(sorted(animaux, key=len))
print(sorted(animaux, key=lambda mot: mot[-1]))
```

1. Dans le premier appel, quelle fonction passe-t-on en paramètre ? Selon quel critère les animaux sont-ils alors triés ?
2. Dans le second appel, le paramètre `key` reçoit une fonction définie « sur place » avec la notation `lambda`. Selon quel critère les animaux sont-ils maintenant triés ?
3. Quel est l'intérêt d'utiliser une `lambda` plutôt que de définir la fonction avec `def` avant l'appel ?

③ Transformer, filtrer : `map`, `filter` et les compréhensions

On reprend la liste des notes : `notes = [8, 15, 12, 7, 18, 9, 14]`. On va appliquer deux opérations très courantes :

— **transformer** chaque note (par exemple, la doubler) ;

— **sélectionner** certaines notes (par exemple, celles supérieures ou égales à 10).

Chaque opération peut s'écrire de trois façons : avec une boucle, avec une fonction d'ordre supérieur (`map` ou `filter`), ou avec une compréhension de liste.

Exercice 6 *Doubler toutes les notes* **(3 points)**

1. Recopier et compléter chacune des trois écritures suivantes pour la liste `notes = [8, 15, 12, 7, 18, 9, 14]`.

   (A) Boucle `for` + `append`.

```
doubles = []
for n in notes:
    doubles.append(........)
```

   (B) Compréhension de liste.

```
doubles = [........ for n in notes]
```

(C) `map` + `lambda`.

```
doubles = list(map(lambda n: ........, notes))
```

2. Donner la valeur de `doubles` (la même pour les trois versions).
3. Quelle est la différence, du point de vue du style de programmation, entre la version (A) d'une part, et les versions (B) et (C) d'autre part ?

Exercice 7 *Sélectionner les notes supérieures ou égales à* 10 **(3 points)**

1. Compléter les trois écritures suivantes pour obtenir, dans chaque cas, la liste des notes supérieures ou égales à 10.

(A) Boucle `for` + `append`.

```
bonnes = []
for n in notes:
    if ........:
        bonnes.append(n)
```

(B) Compréhension de liste avec `if`.

```
bonnes = [n for n in notes if ........]
```

(C) `filter` + `lambda`.

```
bonnes = list(filter(lambda n: ........, notes))
```

2. Donner la valeur de `bonnes`.
3. Quelle différence y a-t-il entre ce que fait `map` et ce que fait `filter` ?

④ Composer : la moyenne des bonnes notes en une expression

On cherche maintenant à calculer *la moyenne des notes supérieures ou égales à* 10. Il s'agit donc d'enchaîner deux étapes : sélectionner les bonnes notes, puis en calculer la moyenne.

Exercice 8 *Chaîner deux opérations* **(3 points)**

1. Écrire une fonction `moyenne_bonnes_imp(notes)` dans un style *impératif* qui utilise une boucle, un compteur et un accumulateur.
2. Écrire une fonction `moyenne_bonnes_fonc(notes)` dans un style *fonctionnel* qui enchaîne `filter` pour sélectionner, puis `sum` et `len` pour obtenir la moyenne. On écrira la fonction en une seule ligne de `return`.
3. Écrire une troisième version `moyenne_bonnes_comp(notes)` qui utilise une compréhension de liste avec condition à la place du `filter`.
4. Vérifier que les trois fonctions renvoient la même valeur sur `notes = [8, 15, 12, 7, 18, 9, 14]`.

Exercice 9 *Effets de bord et fonctions pures* **(2 points)** Une fonction est dite **pure** si elle renvoie toujours la même valeur pour les mêmes arguments et si elle ne modifie rien en dehors d'elle-même (pas d'affichage, pas de modification de variable globale, pas de modification des arguments reçus).

Considérons les deux fonctions suivantes, qui ajoutent un élément à une liste :

Version (A).

```
def ajouter_A(x, L):
    L.append(x)
    return L
```

Version (B).

```
def ajouter_B(x, L):
    return L + [x]
```

1. Dans une console Python, on exécute le scénario suivant :

   ```
   ma_liste = [1, 2, 3]
   print(ajouter_A(4, ma_liste))
   print(ma_liste)
   ```

   Que s'affiche-t-il ? Même question en remplaçant `ajouter_A` par `ajouter_B`.

2. Laquelle des deux fonctions est une fonction pure ? Justifier.

3. En style fonctionnel, on préfère toujours créer de nouvelles structures plutôt que de modifier les existantes. Pourquoi cette règle rend-elle les programmes plus *prévisibles* ?

5 Pour aller plus loin

Exercice 10 *Combiner `map` et `filter`* **(2 points)** On souhaite obtenir la liste des notes *supérieures ou égales à* 10, *ramenées sur* 10 (en les divisant par 2).

1. Écrire cette opération en chaînant `filter` et `map`, à partir de `notes = [8, 15, 12, 7, 18, 9, 14]`. *Indication : on applique `filter` d'abord, puis `map` sur le résultat.*

2. Écrire la même opération avec une compréhension de liste à condition.

3. Vérifier que les deux versions donnent le même résultat.

Exercice 11 *La fonction `reduce`* **(3 points)** La fonction `reduce` du module `functools` généralise l'accumulation : elle applique une fonction binaire « deux à deux » pour réduire toute une liste à une seule valeur.

```
from functools import reduce
total = reduce(lambda a, b: a + b, [3, 5, 2, 4], 0)
```

Le calcul s'effectue en chaîne : 0 + 3 = 3, puis 3 + 5 = 8, puis 8 + 2 = 10, puis 10 + 4 = 14. Le troisième argument 0 est la valeur initiale de l'accumulateur.

1. Réécrire la fonction `somme(notes)` en une seule ligne avec `reduce`.

2. Définir, à l'aide de `reduce`, une fonction `produit(liste)` qui renvoie le produit des éléments de `liste`. Quelle valeur initiale faut-il choisir ?

3. Définir, à l'aide de `reduce`, une fonction `maximum(liste)` qui renvoie le plus grand élément d'une liste non vide. *Indication : la fonction native `max(a, b)` convient comme fonction binaire.*

Remarque 3 (Une fonction qui renvoie une fonction). En style fonctionnel, une fonction est un objet de première classe : on peut la passer en paramètre, mais aussi la *renvoyer* comme résultat d'une autre fonction. Voici une fonction `multiplicateur` qui prend un entier k et renvoie une nouvelle fonction multipliant son argument par k :

```
def multiplicateur(k):
    return lambda x: k * x

double = multiplicateur(2)
triple = multiplicateur(3)
print(double(5))   # 10
print(triple(5))   # 15
```

La fonction renvoyée « se souvient » de la valeur de k utilisée pour la fabriquer, même après que `multiplicateur` a terminé son exécution. Ce mécanisme s'appelle une *fermeture* (en anglais : *closure*). Il est très utilisé en programmation fonctionnelle pour créer dynamiquement des fonctions paramétrées.

Remarque 4 (Bilan des activités). Nous avons comparé, sur une même liste de notes, trois manières d'écrire un calcul :

— en style **impératif**, avec des boucles, des variables modifiables et des instructions successives ;
— en style **fonctionnel** explicite, avec la récursivité et les fonctions d'ordre supérieur `map`, `filter` ;
— en style **pythonique**, avec les compréhensions de liste qui combinent les deux approches.

L'essentiel du paradigme fonctionnel tient en trois idées :

1. une fonction est une *expression* comme une autre (notations `lambda`, expressions conditionnelles) ;
2. on ne modifie pas l'état : on construit de nouvelles valeurs (fonctions *pures*) ;
3. la composition de fonctions remplace la séquence d'instructions.

Le notebook Capytale associé vous permettra d'exécuter et de comparer toutes ces versions en Python.