MINISTÈRE
DE L'ÉDUCATION
NATIONALE,
DE LA JEUNESSE
ET DES SPORTS

Liberté
Égalité
Fraternité

VOIE GÉNÉRALE

2DE | 1RE | TLE

Numérique et sciences informatiques

ENSEIGNEMENT
SPÉCIALITÉ

LE PARADIGME FONCTIONNEL

SOMMAIRE

Introduction

Le paradigme fonctionnel est un paradigme de programmation qui reprend les principes du lambda-calcul introduit par Church dans les années 1930.

L'idée fondamentale du lambda-calcul est de considérer que les fonctions sont des données comme les autres. Ainsi, elles peuvent être par exemple passées en paramètre à d'autres fonctions.

D'autres principes découlent également de la thèse de Church :

- les fonctions sont des fonctions au sens mathématique du terme : elles se contentent de renvoyer une valeur en fonction de leurs arguments ;
- il n'y a pas de notion « d'état », ni à l'extérieur des fonctions, ni dans les fonctions. Un programme n'est donc qu'une composition de fonctions.

Le paradigme fonctionnel a d'abord été implanté au sein de langages dédiés, plus ou moins « purement fonctionnel ». Parmi les langages dits fonctionnels, on peut citer :

- LISP (List Processing) : 1958 ;
- SML (Standard Meta Language) : 1983 ;
- CAML (Categorical Abstract Machine Language) : 1987, puis son extension objet OCAML ;
- Haskell : 1990 ;
- Clojure : 2007.

Mais certains aspects du paradigme fonctionnel ont fini par être intégrés dans des langages impératifs, car ils présentent certains avantages :

- fonctions pures ;
- fonctions d'ordre supérieur ;
- lambda-expressions ;
- évaluation paresseuse.

La mise en œuvre de ces principes dans les langages fonctionnels est présentée en annexe A., et un survol de ce que cela peut donner en Haskell est donné dans l'annexe B.. Le reste du document décrit quelques principes de base permettant de manipuler ce paradigme en Python. L'annexe C. montre comment il est possible d'écrire du « fonctionnel objet » en Python.

Mise en œuvre du paradigme fonctionnel en Python

Fonctions pures

Une **fonction pure** est une fonction qui ne modifie rien ; elle ne fait que renvoyer des valeurs en fonction de ses paramètres.

Les modifications qu'une fonction peut effectuer sur l'état du système sont appelées **effets de bord**. Un affichage à l'écran est un exemple d'effet de bord.

En Python, rien n'impose d'implanter des fonctions pures. Notamment, étant donné la façon dont les arguments sont passés à une fonction en Python (utilisation d'une copie de la référence initiale), rien n'interdit qu'une fonction modifie l'objet référencé par l'un de ses paramètres.

Voici un tel exemple :

```
def retirer_dernier(liste) :
      liste.pop()
```

On utilise cette fonction ainsi :

```
l = [1, 2, 3]
retirer_dernier(l)
```

L'inconvénient de ce type de fonction est qu'elles complexifient énormément la compréhension du code. Une fonction pure doit renvoyer la valeur calculée sans modifier ses paramètres. Ainsi, on peut réécrire le traitement précédent de la façon suivante :

```
def retirer_dernier_pure(liste) :
      retour = liste[:]
      retour.pop()
      return retour
```

Cette fonction s'utilise ainsi :

```
l1 = [1, 2, 3]
l2 = retirer_dernier_pure(l1)
```

Dans ce dernier cas, le fait que l'appel à *retirer_dernier_pure* ne modifie par *l1* est bien plus intuitif.

L'exemple suivant est encore plus parlant. On définit la fonction suivante :

```
def somme(liste) :
    s = 0
    while len(liste) > 0 :
        dernier_element = liste.pop()
        s += dernier_element
    return s
```

Cette fonction calcule bien la somme des éléments de la liste passée en paramètres :

```
l1 = [1, 2, 3]
s = somme(l1)
print(s)
```

```
6
```

Ce qui est beaucoup moins clair, en revanche, c'est le résultat de l'exécution suivante :

```
l1 = [1, 2, 3]
s1 = somme(l1)
print(s1)
s2 = somme(l1)
print(s2)
```

```
6
0
```

En effet, la fonction `somme` modifie la liste passée en paramètre. Pour faciliter l'écriture de fonction pures, on peut utiliser des objets non mutables. En Python, les objets non mutables sont des objets qu'on ne peut modifier.

Exemples :

```
>>> a = 1
>>> id(a)
94014566454016
>>> a += 1
>>> id(a)
94014566454048
>>> c = «toto»
>>> id(c)
139886778417072
>>> c += «titi»
>>> id(c)
139886778429488
```

Comme on peut le voir, après « modification » de la valeur de la variable, celle-ci référence un objet différent. En effet, en Python, les entiers et les chaînes de caractères sont des objets non mutables.

A contrario, certaines données en Python sont mutables :

```
>>> b = [1]
>>> id(b)
139886801700256
>>> b += [2]
>>> id(b)
139886801700256
>>> print(b)
[1, 2]
```

Sur ce dernier exemple, c'est bien la liste de départ qui a été modifiée.

Pour faciliter l'écriture de fonctions pures en Python, on peut :

- utiliser au maximum des données non mutables ;
- copier systématiquement au début des fonctions les paramètres référençant des données mutables et utiliser ces copies dans la fonction.

La fonction *retirer_dernier_pure* ci-dessus est bien une fonction pure. Cependant, si on veut suivre complètement le paradigme fonctionnel, il ne faut pas utiliser une séquence d'instruction mais uniquement de la composition de fonction. On peut alors réécrire cette fonction ainsi :

```
def retirer_dernier_fonctionnelle(liste) :
       return liste[:-1]
```

S'interdire les séquences d'instruction n'est pas forcément à rechercher en Python, mais essayer de n'écrire que des fonctions pures permet de limiter les risques de bugs et facilite la relecture des programmes (voir la ressource sur la gestion des bugs). Il s'agit donc d'un style de programmation à privilégier.

Fonctions d'ordre supérieur

Une **fonction d'ordre supérieur** est une fonction qui, parmi ses paramètres, prend une fonction.

Python fournit, en mode basique ou par l'intermédiaire de bibliothèques standards, plusieurs fonctions d'ordre supérieur. Certaines de ces fonctions sont très utiles pour appliquer des traitements aux listes. Voici quelques-unes de ces fonctions avec des exemples d'utilisation.

Fonction `map`

La fonction `map` est une fonction qui permet d'appliquer un *traitement* à tous les éléments d'une liste[1]. Cette fonction ne modifie pas la liste de départ : elle renvoie un objet (itérable) encapsulant le résultat (le résultat n'est pas construit à l'appel) ; les valeurs sont calculées lorsqu'elles sont requises ; c'est une mise en œuvre du principe d'*évaluation paresseuse*. Le traitement est bien sûr spécifié via une fonction.

Exemple :

```
def carre(n) :
    return n * n
l1 = [1, 2, 3, 4]
res = map(carre, l1)
print(type(res))
```

Produit l'affichage suivant :

```
<class 'map'>
```

1. En fait, d'un *iterable*.

Par la suite, les instructions suivantes :

```
l2 = list(res)
print(l2)
```

Donnent maintenant l'affichage :

```
[1, 4, 9, 16]
```

Comme on peut le voir, c'est la **fonction** (c'est-à-dire son nom) qui est passée en paramètre, pas son appel (il n'y a pas de parenthèses après le nom de la fonction).

Fonction `filter`

La fonction `filter` est un autre exemple de fonction d'ordre supérieur s'appliquant à des listes. Elle prend en premier paramètre une fonction à valeur booléenne appelée *filtre*, et une liste[2] en deuxième paramètre. En résultat, elle renvoie un *iterable* ne contenant que les valeurs de la liste pour lesquels le filtre renvoie la valeur `True`.

Exemple :

```
def est_pair(n) :
    return n % 2 == 0
l1 = [1,3, 4, 6, 7, 9, 10]
res = filter(est_pair, l1)
l2 = list(res)
```

On obtient l'affichage suivant :

```
[4, 6, 10]
```

Fonction `reduce`

La fonction `reduce` du module `functools` est une fonction d'ordre supérieur qui permet d'effectuer un *pliage* (ou *réduction*) d'une liste[3]. Un pliage consiste à itérer un calcul sur les éléments d'une liste. On peut résumer ainsi l'algorithme de la fonction `reduce`[4] :

```
reduce(fonction, liste, init) :
       accumulateur = init
    for element in liste :
        accumulateur = fonction(accumulateur, element)
    return accumulateur
```

2. En fait, n'importe quel *iterable*.
3. En fait, n'importe quel *iterable*.
4. C'est une version légèrement simplifiée.

Ainsi, pour calculer la somme ou le produit des éléments d'une liste, on peut utiliser `reduce` ainsi :

```
from functools import reduce

def addition(x, y) :
    return x + y
def multiplication(x, y) :
    return x * y
l1 = [1, 2, 3, 4]
somme = reduce(addition, l1, 0)
produit = reduce(produit, l1, 1)
```

Il est alors possible d'expliquer pas à pas ce qu'il se passe pour le calcul de la somme par exemple (*fonction* représente donc l'addition) :

Instruction	Init	Element	Accumulateur
Entrée dans reduce	0		
accumulateur = init	0		0
for element in liste	0	1	0
accumulateur = `fonction`(accumulateur, element)	0	1	1
for element in liste	0	2	1
accumulateur = `fonction`(accumulateur, element)	0	2	3
for element in liste	0	3	3
accumulateur = `fonction`(accumulateur, element)	0	3	6
for element in liste	0	4	6
accumulateur = `fonction`(accumulateur, element)	0	4	10

Intérêt des fonctions d'ordre supérieur

Les fonctions d'ordre supérieur permettent de définir des comportements génériques paramétrés par un comportement défini à la volée. La fonction `sorted` est un exemple de fonction d'ordre supérieur fonctionnant ainsi : le paramètre nommé `key` permet de spécifier le critère de comparaison pour paramétrer la fonction de tri. Dans le cas des fonctions sur les listes présentées ci-dessus, cela permet d'écrire des traitements sur les listes sans écrire de boucle, ce qui simplifie la structure du code et réduit les risques d'erreur.

Les fonctions d'ordre supérieur ne sont pas explicitement au programme de NSI. On évite donc d'évaluer les élèves sur le sujet. Pour autant, elles permettent d'éclairer l'intérêt du paradigme fonctionnel, et une présentation de celles-ci semble pertinente. On peut se limiter à `map` et `filter`, qui sont plus abordables que `reduce`.

Lambda-expressions (hors programme)

Dans les exemples de la partie précédente sur les fonctions d'ordre supérieur, il fallait à chaque fois commencer par définir la fonction qu'on souhaitait utiliser, ce qui rend l'écriture un peu lourde. Dans le paradigme fonctionnel, il existe un opérateur pour définir des fonctions que l'on peut stocker dans des variables ou passer en paramètre : il s'agit de l'opérateur `lambda`.

La syntaxe générale est la suivante :

```
lambda args:expr
```

où `args` est une liste d'arguments et `expr` la valeur de la fonction selon ses arguments.

Exemples :

```
(lambda x : x + 1)(5)
```

```
6
```

```
(lambda x,y : x + y)(4, 5)
```

```
9
```

On peut bien sûr stocker ces fonctions dans des variables :

```
successeur = lambda x : x + 1
successeur(4)
```

```
5
```

Il est également possible de passer ces fonctions en paramètre aux fonctions d'ordre supérieur. Ainsi, les exemples de la partie 2.2 peuvent être réécrits ainsi :

- carrés d'une liste :

```
map(lambda x : x*x, l1)
```

- nombres pairs d'une liste :

```
filter(lambda x : x % 2 == 0, l1)
```

- somme des éléments d'une liste :

```
reduce(lambda x,y : x + y, l1, 0)
```

- produit des éléments d'une liste :

```
reduce(lambda x,y : x + y, l1, 1)
```

Évaluation paresseuse (hors programme)

Introduction

Dans le paradigme fonctionnel, l'évaluation paresseuse est souvent utilisée. Ce mécanisme consiste à ne calculer une valeur que lorsque celle-ci est nécessaire. On retrouve cette mise en œuvre dans le fonctionnement des fonctions `map` et `filter` de Python par exemple, et c'est ce qui justifie que ces fonctions ne renvoient pas une liste mais un objet itérable. Ainsi, c'est lors de l'appel à *next* que la valeur suivante est calculée.

On peut mettre en évidence l'évaluation paresseuse en ajoutant un affichage dans la fonction `carre` :

```
def carre_affichage(n) :
    carre = n * n
    print(n, «au carré vaut»,carre)
    return carre
l1 = [1, 2, 3, 4]
res = map(carre_affichage, l1)
print(«On vient d'appliquer map»)
for val in res :
    print(«élément», val)
```

L'affichage produit est le suivant :

```
On vient d'appliquer map
1 au carré vaut 1
élément 1
2 au carré vaut 4
élément 4
3 au carré vaut 9
élément 9
4 au carré vaut 16
élément 16
```

Comme on peut le voir, l'affichage n'est produit que lorsque les éléments résultants sont requis par l'appel à la fonction `next` générée par le `for`, ce qui signifie bien que l'exécution de la fonction `carre_affichage` n'a lieu qu'à ce moment.

Intérêt de l'évaluation paresseuse

On se propose de définir la classe `limite` suivante, qui permet, à partir d'un itérable `source`, de renvoyer un autre itérable fait des `longueur` premiers éléments de `source` :

```
class limite :
    def __init__(self, source, longueur) :
        self.longueur = longueur
        self.elements_vus = 0
        self.iterateur = iter(source)

    def __iter__(self) :
        return self

    def __next__(self) :
        if self.elements_vus == self.longueur :
            raise StopIteration
        self.elements_vus += 1
        return next(self.iterateur)
```

On effectue alors la suite d'instructions suivante :

```
la = [i for i in range(100)]
lb = map(carre_affichage, la)
lc = limite(lb, 4)
l = list(lc)
```

On constate, via l'affichage suivant, que le calcul des carrés, bien que déclaré avant la limitation du nombre d'éléments, n'est finalement effectué que pour les 4 éléments retenus au final :

```
0 au carré vaut 0
1 au carré vaut 1
2 au carré vaut 4
3 au carré vaut 9
```

En guise de conclusion…

Outre la lisibilité, l'intérêt des fonctions pures est indéniable sur la correction du code produit, mais il s'agit d'un principe qui n'est pas propre à la programmation fonctionnelle.

Par ailleurs, il est préférable d'avoir des fonctions qui renvoient des chaînes de caractères à afficher plutôt que des fonctions faisant un affichage direct car cela augmente grandement les possibilités de réutilisation (stocker les affichages dans un fichier si nécessaire par exemple).

On note par ailleurs que, s'il existe des langages fonctionnels orientés objet, paradigme fonctionnel et paradigme objets sont relativement antagonistes en Python, puisque la mise en œuvre du paradigme objet en Python impose de modifier `self`. On peut cependant réussir à conjuguer les 2 paradigmes (voir l'annexe C.).

Comme nous avons pu le voir en 2.2.4, pour du traitement de flux de données, le paradigme fonctionnel est particulièrement adapté : la spécification du traitement devient très déclarative, ce qui la rend plus lisible, moins susceptible de présenter des erreurs, et potentiellement plus efficace grâce à l'évaluation paresseuse. De façon plus générale, quand on écrit un programme qui permet à l'utilisateur de définir des éléments de traitement à la volée, le paradigme fonctionnel est plutôt le bienvenu. À contrario, si le développement envisagé demande de manipuler de nombreux types de données fortement structurés, le paradigme objet peut apparaître comme plus adapté. Enfin, pour des programmes très simples (scripts par exemple), on peut se contenter de l'utilisation du paradigme impératif non objet.

Exemples théoriques sur le paradigme fonctionnel

Un premier exemple

Dans les langages dits « fonctionnels », on n'utilise en général pas les parenthèses pour mentionner les paramètres des fonctions. On peut définir une fonction ainsi :

```
f x = x + 1
```

En mathématique, on note cela ainsi :

$f : x \mapsto x + 1$

et l'application de la fonction est réalisée ainsi :

```
f 4
```

Lorsqu'on définit une fonction, les paramètres de la fonction (x dans l'exemple précédent) sont considérés comme des *variables liées* dans le corps fonction. Si des variables sont mentionnées dans la fonction sans apparaître dans les paramètres, on parle de *variable libre*.

Composition de fonctions

Dans le paradigme fonctionnel, il n'y a pas de séquentialité ; on ne fait que composer des fonctions. Ainsi pour augmenter une valeur de 1, puis multiplier la valeur obtenue par 2, on procède ainsi :

```
f x = x + 1
g x = 2 * x
g (f 4)
```

Dans l'exemple ci-dessus, les parenthèses sont utilisées pour indiquer la priorité : on applique d'abord la fonction f à 4 avant de passer le résultat en argument de g.

Typage

Les langages fonctionnels sont en général des langages fortement typés, avec un système d'*inférence de type* inhérent à l'interpréteur. Cela signifie que l'interpréteur est capable de déduire le type d'une fonction que l'utilisateur définit.

Ainsi, dans l'exemple précédent, si les opérateurs « + » et « * » sont définis comme des opérateurs sur les entiers, les types de f et g sont automatiquement déduits par l'interpréteur ainsi :

type de `f` : `int → int`

type de `g` : `int → int`

On peut remarquer l'utilisation du symbole « → » pour séparer le type du paramètre du type du résultat.

Les fonctions sont des données

Considérons l'exemple suivant :

```
f x = x + 1
```

Ici, on définit la fonction *f*. Cela signifie que, maintenant, la variable *f* désigne la fonction qui à x associe x + 1. Ainsi, comme toute variable en programmation fonctionnelle, *f* a un type (ici, `int → int`).

En mathématiques, on note cela ainsi :

```
f : N → N
    x ↦ x + 1
```

Forme curryfiée

On définit maintenant la fonction g suivante :

```
h x y = x + y
```

On peut appliquer *h* à 2 arguments ainsi :

```
h 2 3
```

Mais quel est le type de la fonction *h* ? Dans les langages fonctionnels, une telle définition est en général interprétée ainsi : *h* est la fonction qui à *x* associe la fonction qui à *y* associe *x + y*. Il faut donc interpréter cette définition ainsi :

```
h : N  → (N →  N)
    x ↦ (y ↦ x+y)
```

Ainsi, *h* est une fonction à 1 argument qui renvoie une fonction. En effet, dans le paradigme fonctionnel, une fonction est une donnée et peut, à ce titre, être renvoyée par une autre fonction comme toute autre donnée.

On appelle cette représentation d'une fonction classiquement vue comme une fonction à 2 arguments la forme *curryfiée*.

Application partielle

La fonction *h* précédente peut donc être vue comme une fonction à 1 argument. On peut donc définir la variable *h2* suivante :

```
h2 = h 2
```

h2 contient ainsi le résultat de l'application de *h* à la valeur 2. Ainsi *h2* est la fonction qui à y associe 2 + y. Mathématiquement :

```
h2 : N → N
     y ↦ 2 + y
```

Fonction en paramètre

Tout comme une fonction peut renvoyer une fonction, une fonction peut aussi prendre une fonction en paramètre. Considérons la fonction g suivante :

```
g f x = 2 * (f x)
```

La fonction g est une fonction qui prend une fonction en paramètre (f) et qui renvoie une fonction qui applique f à son paramètre, puis renvoie le double du résultat obtenu. En effet, g peut être traduite mathématiquement ainsi :

```
g: (T → N) → (T → N)
     f     ↦ (x ↦ 2 * f(x))
```

Dans cette définition, *T* représente un type quelconque. C'est ce qu'on peut appeler une *variable de type*.

On peut ainsi écrire une fonction permettant de faire de la composition de fonctions :

```
comp f g x = f (g x)
```

La définition mathématique de *comp* est alors :

```
comp : (B → C) → (( A → B) → (A → C))
          f     ↦(    g      ↦ f(g(x)))
```

Là encore, *A*, *B* et *C* sont des variables de type.

L'opérateur → est en général considéré comme associatif à droite. On peut donc également noter le type de `comp` ainsi :

```
comp : (B → C) → (A → B) → A → C)
```

Lambda expressions

Jusqu'à présent, pour passer une fonction en paramètre, il fallait que la fonction ait déjà été définie et stockée dans une variable. Cependant, dans les langages fonctionnels, cela n'est pas nécessaire, on peut définir des fonctions anonymes, en utilisant le caractère grec lambda : λ, d'où le terme lambda-expression. La syntaxe est alors la suivante : λ *x.(expr)*. On peut par exemple utiliser la fonction `comp` définie précédemment ainsi :

```
comp (λ x.(2*x)) (λ x.(x+1)) 4
```

L'évaluation de l'expression précédente consiste à appliquer (λ x.(x+1)) à 4 (ce qui donne 5), puis (λ x.(2*x)) au résultat, ce qui vaut 10.

Fonctions d'ordre supérieur

On appelle fonction d'ordre supérieur une fonction qui prend en paramètre une autre fonction. C'est donc le cas par exemple de la fonction `comp` présentée ci-dessus.

Les langages fonctionnels utilisent beaucoup la structure de liste (on note *[A]* une liste d'éléments de type *A*). Des fonctions d'ordre supérieur très pratiques sur les listes sont ainsi présentes dans la plupart de ces langages :

```
map : [A] → (A → B) → [B]
```

La fonction *map* est une fonction qui prend en paramètre une liste *l* d'éléments de type *A* et une fonction *f* de *A* vers *B* et qui renvoie une liste d'éléments de type *B* obtenus en appliquant *f* à chacun des éléments de *l*.

Exemples :

```
map [1,2,3,4,5] (λx.(x%2 == 0))
=> [false, true, false, true, false]
```

```
map [1,2,3,4,5] (λx.(x * x))
=> [1, 4, 9, 16, 25]
```

```
filter : [A] → (A → Boolean) → [A]
```

La fonction *filter* prend une liste *l* d'éléments et une fonction *f* à valeur booléenne, et renvoie la liste des éléments de *l* qui vérifient *f*.

Exemple :

```
map [1,2,3,4,5] (λx.(x%2 == 0))
=> [2,4]
```

```
reduce : [A] → (A → B → B) → B → B
```

La fonction reduce (pliage) permet de calculer une valeur à partir des éléments d'une liste en itérant un calcul sur les éléments de la liste en stockant les résultats intermédiaires dans un accumulateur.

Exemple : calcul du produit des éléments d'une liste :

```
reduce [1, 2, 3, 4, 5] (λx.(λy.(x*y))) 1
=> 120
```

Évaluation paresseuse

Une caractéristique fondamentale des langages fonctionnels est la mise en œuvre de l'évaluation paresseuse : ce mécanisme permet de ne calculer une valeur que lorsqu'elle est vraiment utile. Ainsi, on peut tout à fait définir une liste infinie… tant qu'on n'a pas besoin de toutes ses valeurs.

Illustration avec Haskell

Le langage Haskell est un langage fonctionnel moderne, efficace et relativement agréable à utiliser. De plus, un interpréteur en ligne existe à l'adresse suivante : https://repl.it/languages/haskell.

Sur la page qui s'affiche, dans la partie de droite, taper « `ghci` » pour lancer l'interpréteur Haskell.

On peut alors définir la fonction successeur ainsi :

```
succ x = x + 1
```

Le type de `succ` peut être obtenu ainsi :

```
: t succ
```

```
succ :: Num a => a -> a
```

Cela signifie que `succ` est une fonction qui prend un paramètre d'un type numérique quelconque et renvoie une valeur du même type.

On retrouve en Haskell la fonction `map` :

```
: t map
```

```
map :: (a -> b) -> [a] -> [b]
```

```
doubler x = 2 * x
map doubler [1, 2, 3]
```

```
[2, 4, 6]
```

Et bien sûr, on retrouve également en Haskell les lambda-expressions :

```
map (\x -> x * x) [1, 2, 3]
```

```
[1, 4, 9]
```

Pour calculer la somme des éléments d'une liste, on peut utiliser soit la récursivité :

```
somme liste = if (length liste == 0) then 0 else head liste + somme (tail
liste)
: t somme
```

```
Num p => [p] -> p
```

```
somme [1, 2, 3]
```

```
6
```

Soit une fonction de réduction :

```
foldl (+) 0 [1,2,3]
```

```
6
```

Ici, foldl est une fonction d'ordre supérieur qui prend en premier argument la fonction de réduction (ici, on utilise l'opérateur + en le précisant entre parenthèses), en deuxième argument la valeur avec laquelle initialiser l'accumulateur, et en troisième argument la liste à réduire.

Paradigme objet fonctionnel

Il est possible d'écrire des classes qui ne modifient pas les instances après leur initialisation. Pour ce faire, on utilise une terminologie différente pour les accesseurs en écriture, qui doivent alors renvoyer une « copie modifiée » de l'objet de départ, comme dans la classe `Personne` ci-dessous :

```
class Personne :
    def __init__(self, nom, prenom) :
        self.nom = nom
        self.prenom = prenom

    def avec_nom(self, nom) :
        retour = Personne(nom, self.prenom)
        return retour

    def avec_prenom(self, prenom) :
        retour = Personne(self.nom, prenom)
        return retour

    def __str__(self) :
        return self.prenom + « « + self.nom
```

Que l'on va utiliser ainsi :

```
p = Personne(«Durand», «Jacques»)
p2 = p.avec_nom(«Martin»)
print(p)
```

```
Jacques Durand
```

```
print(p2)
```

```
Jacques Martin
```

On peut également mettre cela en œuvre sur les opérateurs, comme dans la classe `Heure` ci-dessous :

```
class Heure :
    def __init__(self, heures, minutes) :
        self.heures = heures
        self.minutes = minutes

    def avec_heures(self, heures) :
        retour = Heure(heures, self.minutes)
        return retour

    def avec_minutes(self, minutes) :
        retour = Heure(self.heures, minutes)
        return retour

    def __str__(self) :
        return «{:02d}:{:02d}».format(self.heures, self.minutes)

    def __add__(self, minutes) :
        nouvelles_minutes = self.minutes + minutes
        if nouvelles_minutes >= 60 :
            delta_heure = nouvelles_minutes // 60
            nouvelles_minutes = nouvelles_minutes % 60
            nouvelles_heures = (self.heures + delta_heure) % 24
        else :
            nouvelles_heures = self.heures
        return Heure(nouvelles_heures, nouvelles_minutes)
```

On peut alors utiliser cette classe ainsi :

```
heure1 = Heure(22,40)
heure2 = heure1
heure1 = Heure(22,40)
heure2 = heure1
print (id(heure1))
```

```
139829759335952
```

```
print (id(heure2))
```

```
139829759335952
```

```
heure1 += 130
print (id(heure1))
```

```
139829759335760
```

```
print (heure1)
```

```
00:50
```

```
print (heure2)
```

```
22:40
```

Comme on peut le voir, en ajoutant 130 minutes à `heure1`, on ne modifie pas `heure1`, mais on crée une nouvelle instance de la classe `Heure`.