Programmation fonctionnelle

Chapitre 24

NSI2

21 février 2022

Tout comme la programmation récursive ou orientée objet, la *programmation fonctionnelle* est un des multiples paradigmes de programmation.

Tout comme la programmation récursive ou orientée objet, la *programmation fonctionnelle* est un des multiples paradigmes de programmation.

Python, langage multi paradigme, autorise la programmation fonctionnelle, seule ou même combinée aux autres méthodes de programmation.

Tout comme la programmation récursive ou orientée objet, la *programmation fonctionnelle* est un des multiples paradigmes de programmation.

Python, langage multi paradigme, autorise la programmation fonctionnelle, seule ou même combinée aux autres méthodes de programmation.

« Une fonction est une donnée comme les autres. »

Fonction comme valeur passée à une autre fonction

Un concept déjà rencontré

Nous avons déjà utilisé la fonction `sorted` de PYTHON pour trier des listes suivant différents critères.

Un concept déjà rencontré

Nous avons déjà utilisé la fonction `sorted` de PYTHON pour trier des listes suivant différents critères.
Par exemple pour trier une liste de paires d' `int` dans l'ordre croissant suivant leur deuxième élément, on écrit ceci

Un concept déjà rencontré

Nous avons déjà utilisé la fonction `sorted` de PYTHON pour trier des listes suivant différents critères.
Par exemple pour trier une liste de paires d' `int` dans l'ordre croissant suivant leur deuxième élément, on écrit ceci

```
def f(t: tuple) -> int:
    return t[1]


l1 = [(1, 2), (2, 0), (10, 1)]
l2 = sorted(l1, key=f)
print(l2)
# result : [1, 3, 7, 13, 21, 31]
```

Analyse

On a bel et bien passé la fonction `f` *comme paramètre* à la fonction `sorted` .

Analyse

On a bel et bien passé la fonction `f` *comme paramètre* à la fonction `sorted` .

Ainsi `f` est considérée comme une variable. D'ailleurs elle a un type :

Analyse

On a bel et bien passé la fonction `f` *comme paramètre* à la fonction `sorted` .
Ainsi `f` est considérée comme une variable. D'ailleurs elle a un type :

```
>>> type(f)

<class 'function'>
```

Un autre exemple

La programmation fonctionnelle nous autorise à créer et utiliser des fonctions prenant d'autres fonctions comme paramètres :

Un autre exemple

La programmation fonctionnelle nous autorise à créer et utiliser des fonctions prenant d'autres fonctions comme paramètres :

```
def make_table(f, a: int, b: int) -> list:
    return [f(x) for x in range(a, b + 1)]


def g(x: float) -> float:
    return x ** 2 + x + 1


print(make_table(g, 0, 5))

# result : [1, 3, 7, 13, 21, 31]
```

Analyse

La fonction `make_table`

Analyse

La fonction `make_table`

- prend en paramètre une fonction `f` et deux `int` `a` et `b`;

Analyse

La fonction `make_table`

- prend en paramètre une fonction `f` et deux `int` `a` et `b`;
- renvoie une liste qui est le tableau de valeurs de `f` pour les valeurs entières entre `a` et `b` compris.

Aller plus loin

On peut imaginer des exemples plus utiles.

Aller plus loin

On peut imaginer des exemples plus utiles.

Imaginons qu'on dispose d'une liste d'éléments x_i et d'une opération *associative* sur ces éléments notée $\star$.

Aller plus loin

On peut imaginer des exemples plus utiles.

Imaginons qu'on dispose d'une liste d'éléments x_i et d'une opération *associative* sur ces éléments notée $\star$.

On voudrait calculer $x_1 \star x_2 \ldots \star x_n$.

Aller plus loin

On peut imaginer des exemples plus utiles.

Imaginons qu'on dispose d'une liste d'éléments x_i et d'une opération *associative* sur ces éléments notée $\star$.

On voudrait calculer $x_1 \star x_2 \ldots \star x_n$.

La fonction `compute` du programme suivant permet de le faire :

Programme

```
def compute(operator, l: list) -> float:
    result = l[0]
    for i in range(1, len(l)):
        result = operator(result, l[i])
    return result


def product(x: float, y: float) -> float:
    return x * y


print(compute(product, [1, 2, 3, 4, 5]))
# result : 120
```

les fonctions lambda

Idée

Il est assez fastidieux d'avoir à définir des fonctions à l'aide de `def`, lorsqu'elle sont juste destinées à être passées en paramètre.

Idée

Il est assez fastidieux d'avoir à définir des fonctions à l'aide de `def`, lorsqu'elle sont juste destinées à être passées en paramètre.

Dans ce cas une *fonction lambda* est plus simple à utiliser.

Idée

Il est assez fastidieux d'avoir à définir des fonctions à l'aide de `def`, lorsqu'elle sont juste destinées à être passées en paramètre.

Dans ce cas une *fonction lambda* est plus simple à utiliser.

Reprenons l'exemple de `sorted` :

Idée

Il est assez fastidieux d'avoir à définir des fonctions à l'aide de `def`, lorsqu'elle sont juste destinées à être passées en paramètre.

Dans ce cas une *fonction lambda* est plus simple à utiliser.

Reprenons l'exemple de `sorted` :

```
>>> f = lambda t: t[1]
>>> print(sorted([(1, 2), (3, 0), (5, 1)], key=f))

[(3, 0), (5, 1), (1, 2)]
```

Mieux

Et on peut même faire plus court :

```
>>> print(sorted([(1, 2), (3, 0), (5, 1)], key=lambda t: t[1]))

[(3, 0), (5, 1), (1, 2)]
```

lambda fonction

Une lambda fonction (fonction anonyme) est une fonction définie à la volée et soumise à la syntaxe suivante :

lambda fonction

Une lambda fonction (fonction anonyme) est une fonction définie à la volée et soumise à la syntaxe suivante :

```
lambda var_1,...,var_n : <expression utilisant ces variables>
```

lambda fonction

Une lambda fonction (fonction anonyme) est une fonction définie à la volée et soumise à la syntaxe suivante :

```
lambda var_1,...,var_n : <expression utilisant ces variables>
```

Il n'y a pas de nom de fonction, pas de **`return`** et l'expression doit s'écrire comme une seule ligne.

Exemples

- `lambda x, y : x *  y` pour le produit de 2 nombres ;

Exemples

- `lambda x, y : x * y` pour le produit de 2 nombres ;
- pour faire la somme des 10 premiers éléments d'une liste :

```
lambda s : sum(s[i] for i in range(len(s)) if i < 10)
```

Fonction comme valeur renvoyée par une fonction

Exemple

Avec une fonction lambda, on peut rapidement écrire une fonction `affine_function` qui

Exemple

Avec une fonction lambda, on peut rapidement écrire une fonction `affine_function` qui

- en entrée prend deux `float` `m` et `p`;

Exemple

Avec une fonction lambda, on peut rapidement écrire une fonction `affine_function` qui

- en entrée prend deux `float` `m` et `p`;
- en sortie renvoie la fonction qui à tout `float` `x` renvoie le `float` `m*x+p`;

Exemple

Avec une fonction lambda, on peut rapidement écrire une fonction `affine_function` qui

- en entrée prend deux `float` `m` et `p`;
- en sortie renvoie la fonction qui à tout `float` `x` renvoie le `float` `m*x+p`;

```
def affine_function(m: float, p: float):
    return lambda x: m * x + p


f = affine_function(2, 3)
print(f(1))  # result : 2*1+3 = 5
```

Sans lambda fonction

Rien ne nous oblige à utiliser une lambda fonction, mais c'est plus long sans :

Sans lambda fonction

Rien ne nous oblige à utiliser une lambda fonction, mais c'est plus long sans :

```
def affine_function(m: float, p: float):
    def f(x: float) -> float:
        return m * x + p

    return f
```

Fonction renvoyée comme valeur de retour

Rien n'interdit de créer une fonction prenant une fonction en paramètre et renvoyant une fonction en sortie.

Fonction renvoyée comme valeur de retour

Rien n'interdit de créer une fonction prenant une fonction en paramètre et renvoyant une fonction en sortie.

```
def double_function(f):
    return lambda x: 2 * f(x)


def g(x: float) -> float:
    return x * x


df = double_function(g)
print(df(10)) # result 200
```

Fonction renvoyée comme valeur de retour

Rien n'interdit de créer une fonction prenant une fonction en paramètre et renvoyant une fonction en sortie.

```
def double_function(f):
    return lambda x: 2 * f(x)


def g(x: float) -> float:
    return x * x


df = double_function(g)
print(df(10)) # result 200
```

On peut même fabriquer des exemples vraiment intéressants.

Propriété de maths (rappel ?)

Soit f une fonction définie sur un intervalle I non réduit à un point et x un réel appartenant à cet intervalle.

Propriété de maths (rappel ?)

Soit f une fonction définie sur un intervalle I non réduit à un point et x un réel appartenant à cet intervalle.

On dit que f est *dérivable en* x si et seulement si la quantité

Propriété de maths (rappel ?)

Soit f une fonction définie sur un intervalle I non réduit à un point et x un réel appartenant à cet intervalle.

On dit que f est *dérivable en* x si et seulement si la quantité

$$\frac{f(x+h)-f(x)}{h}$$

Propriété de maths (rappel ?)

Soit f une fonction définie sur un intervalle I non réduit à un point et x un réel appartenant à cet intervalle.

On dit que f est *dérivable en* x si et seulement si la quantité

$$\frac{f(x+h)-f(x)}{h}$$

tend vers une valeur finie quand $x+h \in I$, $h \to 0$, $h \neq 0$.

Propriété de maths (rappel ?)

Soit f une fonction définie sur un intervalle I non réduit à un point et x un réel appartenant à cet intervalle.

On dit que f est *dérivable en* x si et seulement si la quantité

$$\frac{f(x+h)-f(x)}{h}$$

tend vers une valeur finie quand $x+h \in I$, $h \to 0$, $h \neq 0$.
Lorsque c'est le cas on note $f'(x)$ ce nombre et on l'appelle *nombre dérivé* de f en x.

Propriété de maths (rappel ?)

Soit f une fonction définie sur un intervalle I non réduit à un point et x un réel appartenant à cet intervalle.

On dit que f est *dérivable en* x si et seulement si la quantité

$$\frac{f(x+h)-f(x)}{h}$$

tend vers une valeur finie quand $x+h \in I$, $h \to 0$, $h \neq 0$.
Lorsque c'est le cas on note $f'(x)$ ce nombre et on l'appelle *nombre dérivé* de f en x.

Si f est dérivable en tout réel $x \in I$ alors on peut définir la *fonction* dérivée de f.

Avec Python

Plutôt qu'un passage à la limite, effectuons le calcul du taux d'accroissement

$$\frac{f(x+h)-f(x)}{h}$$

pour une valeur de h très petite, cela nous donnera dans beaucoup de cas une bonne approximation de $f'(x)$ et conduit à ceci :

Dérivation « numérique »

```
def differentiate(f):
    h = 1e-7
    return lambda x: (f(x + h) - f(x)) / h


def g(x: float) -> float:
    return x ** 2 + x * 4 + 2


g_prime = differentiate(g)

print(g_prime(2))

# g'(x) = 2x + 4 so g'(2) = 8
# numerical result given : 8.00000009348878
```

La philosophie de la programmation fonctionnelle

Origines de la programmation fonctionnelle

Le plus ancien langage basé sur la programmation fonctionnelle est Lisp, inventé en 1958 par John McCarthy, dont on a déjà parlé précédemment.

Origines de la programmation fonctionnelle

Le plus ancien langage basé sur la programmation fonctionnelle est Lisp, inventé en 1958 par John McCarthy, dont on a déjà parlé précédemment.

Cependant ce paradigme de programmation découle directement des travaux d'Alonzo Church, mathématicien américain qui inventa le système du *lambda-calcul* dans les années 30.

Origines de la programmation fonctionnelle

Le plus ancien langage basé sur la programmation fonctionnelle est Lisp, inventé en 1958 par John McCarthy, dont on a déjà parlé précédemment.

Cependant ce paradigme de programmation découle directement des travaux d'Alonzo Church, mathématicien américain qui inventa le système du *lambda-calcul* dans les années 30.

Idée

L'idée de la programmation fonctionnelle est de recourir le plus possible aux fonctions et de faire en sorte que celle-ci ne produisent pas d'*effets de bord* (traduction un peu trop littérale du terme Anglais *side effect* qui signifie en réalité *effet secondaire*).

Idée

L'idée de la programmation fonctionnelle est de recourir le plus possible aux fonctions et de faire en sorte que celle-ci ne produisent pas d'*effets de bord* (traduction un peu trop littérale du terme Anglais *side effect* qui signifie en réalité *effet secondaire*).

L'idée est qu'une fonction évaluée avec un jeu de paramètres doit *toujours donner le même résultat*.

Effet de bord ?

Voici un exemple de fonction induisant un effet de bord.

```
L = [1, 2]


def f(x: int) -> int:
    L[0] += 1
    return x + L[0]


print(f(10))  # result : 12
print(f(10))  # result : 13
```

Quelle horreur !

Il n'est pas acceptable que l'évaluation de `f(10)` donne deux valeurs différentes !

Quelle horreur !

Il n'est pas acceptable que l'évaluation de `f(10)` donne deux valeurs différentes !

Ainsi la programmation fonctionnelle interdit ce genre de fonctions.

Quelle horreur !

Il n'est pas acceptable que l'évaluation de `f(10)` donne deux valeurs différentes !

Ainsi la programmation fonctionnelle interdit ce genre de fonctions.

L'effet de bord est dû au fait que le type `list` est *mutable*, c'est-à-dire qu'on peut changer les éléments d'une liste après sa création.

Un problème ? Une solution...

Pour contourner le problème, la programmation fonctionnelle interdit les types mutables.

Un problème ? Une solution...

Pour contourner le problème, la programmation fonctionnelle interdit les types mutables.

Dans un langage PYTHON *purement* fonctionnel, il n'y aurait pas de type `list`, simplement le type `tuple` (qui est non-mutable).

… Avec des conséquences notables !

Suivant ce paradigme, l'exécution d'un programme est l'évaluation d'une ou plusieurs expressions, qui sont souvent des applications de fonctions à des valeurs passées en paramètre, tout en retenant bien que les fonctions elles-mêmes sont des valeurs.

… Avec des conséquences notables !

Suivant ce paradigme, l'exécution d'un programme est l'évaluation d'une ou plusieurs expressions, qui sont souvent des applications de fonctions à des valeurs passées en paramètre, tout en retenant bien que les fonctions elles-mêmes sont des valeurs.

La programmation fonctionnelle dite *pure* bannit également l'idée de réaffectation : les variables créés et initialisées sont également non-mutables. C'est un peu comme si « les variables étaient des constantes ».

… Avec des conséquences notables !

Suivant ce paradigme, l'exécution d'un programme est l'évaluation d'une ou plusieurs expressions, qui sont souvent des applications de fonctions à des valeurs passées en paramètre, tout en retenant bien que les fonctions elles-mêmes sont des valeurs.

La programmation fonctionnelle dite *pure* bannit également l'idée de réaffectation : les variables créés et initialisées sont également non-mutables. C'est un peu comme si « les variables étaient des constantes ».

Hors de question d'écrire `i = i + 1` ou bien même `for i in range(10)`. C'en est fini des boucles !

La catastrophe ?

Dès lors on peut se demander s'il est vraiment possible de programmer suivant ce paradigme :

La catastrophe ?

Dès lors on peut se demander s'il est vraiment possible de programmer suivant ce paradigme :

- pas d'effets de bord ;

La catastrophe ?

Dès lors on peut se demander s'il est vraiment possible de programmer suivant ce paradigme :

- pas d'effets de bord ;
- des variables non-mutables ;

La catastrophe ?

Dès lors on peut se demander s'il est vraiment possible de programmer suivant ce paradigme :

- pas d'effets de bord ;
- des variables non-mutables ;
- pas de boucles.

La catastrophe ?

Dès lors on peut se demander s'il est vraiment possible de programmer suivant ce paradigme :

- pas d'effets de bord ;
- des variables non-mutables ;
- pas de boucles.

La réponse est oui, et on a démontré que l'on peut programmer autant de choses qu'en suivant les autres paradigmes que sont la POO ou la programmation impérative. Voici un exemple d'algorithme programmé en impératif puis en fonctionnel

Version impérative

```
def somme(lst):
    s = 0
    for e in lst:
    s = s + e
    return s

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

Version fonctionnelle

```
def somme(lst):
    return 0 if not lst else lst[0] + somme(lst[1:])

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

Analyse

On constate que le deuxième programme ne modifie aucune variable et ne contient pas de boucle, celle-ci a été remplacée par de la récursivité, c'est souvent le cas en programmation fonctionnelle.

Avantages de la programmation fonctionnelle

- les programmes sont plus concis ;

Avantages de la programmation fonctionnelle

- les programmes sont plus concis ;
- on peut utiliser des *fonctions d'ordre supérieur* c'est à dire des fonctions qui prennent d'autres fonctions en paramètre et/ou renvoient des fonctions en valeur de sortie ;

Avantages de la programmation fonctionnelle

- les programmes sont plus concis ;
- on peut utiliser des *fonctions d'ordre supérieur* c'est à dire des fonctions qui prennent d'autres fonctions en paramètre et/ou renvoient des fonctions en valeur de sortie ;
- le fait qu'il n'y ait pas d'effets de bords rend les programmes

Avantages de la programmation fonctionnelle

- les programmes sont plus concis ;
- on peut utiliser des *fonctions d'ordre supérieur* c'est à dire des fonctions qui prennent d'autres fonctions en paramètre et/ou renvoient des fonctions en valeur de sortie ;
- le fait qu'il n'y ait pas d'effets de bords rend les programmes
  - plus sûrs ;

Avantages de la programmation fonctionnelle

- les programmes sont plus concis ;
- on peut utiliser des *fonctions d'ordre supérieur* c'est à dire des fonctions qui prennent d'autres fonctions en paramètre et/ou renvoient des fonctions en valeur de sortie ;
- le fait qu'il n'y ait pas d'effets de bords rend les programmes
  - plus sûrs ;
  - plus simple à débugger ;

Avantages de la programmation fonctionnelle

- les programmes sont plus concis ;
- on peut utiliser des *fonctions d'ordre supérieur* c'est à dire des fonctions qui prennent d'autres fonctions en paramètre et/ou renvoient des fonctions en valeur de sortie ;
- le fait qu'il n'y ait pas d'effets de bords rend les programmes
  - plus sûrs ;
  - plus simple à débugger ;
  - plus simples à prouver.

Avantages de la programmation fonctionnelle

- les programmes sont plus concis ;
- on peut utiliser des *fonctions d'ordre supérieur* c'est à dire des fonctions qui prennent d'autres fonctions en paramètre et/ou renvoient des fonctions en valeur de sortie ;
- le fait qu'il n'y ait pas d'effets de bords rend les programmes
  - plus sûrs ;
  - plus simple à débugger ;
  - plus simples à prouver.