Programmation fonctionnelle, Langage fonctionnel

Pascal Fares © Creative common

Un programme écrit en style fonctionnel se caractérise essentiellement par une chose :
l'absence d'effets de bord.

Un paradigme fonctionnel

Structurer le programme avec des fonctions et leur structure de contrôle principale : **la composition de fonction**

La fonction s'applique à un élément d'un ensemble (domaine) de départ et donne un unique résultat dans un autre ensemble (domaine) d'arriver.

f : D1 -> D2, g: D2 -> D3 alors g◦f D1 -> D3 tel que g◦f(x) = g(f(x))

Dans un langage de programmation fonctionnel les domaines (ensembles) sont des types

Association lexicale

Association lexicale: Quand on écrit une définition en utilisant des symbole et des opérateurs; on suppose que ces symboles ont une sémantique et on été défini auparavant

successeur: N -> N
x -> (x+1)

Par exemple, on utilise le symbole (le caractère) + ce symbole a une sémantique (que fait-il?) qui est une opération d'addition qui doit être déjà définie

Abstraction fonctionnelle

Un programme fonctionnel consiste en une expression E (représentant l'algorithme et les entrées). Cette expression E est sujette à des règles de réécriture : la réduction consiste en un remplacement d'une partie de programme fonctionnel par une autre partie de programme selon une règle de réécriture bien définie. Ce processus de réduction sera répété jusqu'à l'obtention d'une expression irréductible (aucune partie ne peut être réécrite). L'expression E* ainsi obtenue est appelée forme normale (fn) de E et constitue la sortie du programme.

Les fondement : Le lambda Calcul

Le λ-calcul a été inventé par le logicien américain Alonzo Church dans les années 1930

Fournir un fondement au mathématiques plus simple que la théorie des ensembles, et fondé sur la notion de fonction => résultat en calculabilité, étant équivalent aux machine de turing et donc aux algorithme impératif

Une théorie: Syntaxe, des règles d’inférences et une sémantique

la syntaxe : Les trois constructions principales du λ-calcul sont :

— **la variable** (il faut bien commencer quelque part) : nous les noterons x, y, z, etc.
— **l'application** : si u et v sont deux programmes, on peut considérer u comme une fonction et v comme un argument possible, et former l'application uv. Ceci correspond à la notation mathématique usuelle u(v) tout en nous économisant quelques parenthèses. On remarquera qu'il n'y a pas de contraintes de typage ici, et qu'il est donc tout à fait légal de former des auto-applications xx, par exemple.
— **l'abstraction** : si u est un programme dépendant (ou non) de la variable x, alors on peut former un nouveau programme λx: u, qui représente la fonction qui à x associe u. Par exemple, λx: x + 1 est intuitivement la fonction qui à x associe x + 1 — sauf que + et 1 ne font pas partie du vocabulaire décrit ci-dessus, faudra donc les définir avec du λ-calcul. Un point sur lequel nous reviendrons.

Calculs et réduction : Les règles essentiel du λ-calcul

α-renommage: (α) λx: u λy·(u[x ↦ y]) , *[x ↦ y] exprime remplacer x par y*

ceci exprime par exemple que λx: x + 1 et λy: y + 1 représente la même expression

β-réduction: (β) (λx: u) v -> u[x ↦ v] qui exprime que si on applique la fonction qui à x associe u à l'argument v, on obtient la même chose que si on calcule directement u avec x remplacé par v. Par exemple, (λx: x + 1)4 -> 4 + 1.

Python possède une définition lambda

lambda v : expression

exemple:

lambda x: x+1

pour appliquer la β-réduction

(lambda x: x+1)(2) -> python répondra 3

on peut nommer les lambda , tout simplement inc = lambda x: x+1

Dans lambda, pas d'état : pas d'affection, pas d'instructions tout est expression

Les lambda en python sont en fait des fonctions anonyme

inc = lambda x: x+1

est équivalent à

def inc(x): return x+1

```
inc = lambda x: x+1
applique10 = lambda f: f(10)

applique10(inc)
# >>> 11
```

La programmation fonctionnelle : quelques caractéristiques

1. Les fonctions sont de première classe (objets). Autrement dit, tout ce que vous pouvez faire avec les «données» peut être fait avec les fonctions elles-mêmes (telles que le passage d'une fonction à une autre fonction).
2. **La récursivité est utilisée comme structure de contrôle principale.** Dans certaines langues, aucune autre construction de «boucle» n'existe.
3. L'accent est mis sur le traitement des listes (par exemple, c'est la source du nom Lisp). Les listes sont souvent utilisées avec la récursion sur les sous-listes en remplacement des boucles.
4. Les langages fonctionnels «purs» évitent les effets de bords. Cela exclut le modèle presque omniprésent dans les langages impératifs d'affectation d'une première, puis d'une autre valeur à la même variable pour suivre l'état du programme.

La programmation fonctionnelle : quelques caractéristiques (suite)

5. La programmation fonctionnelle décourage ou interdit carrément les déclarations, et fonctionne à la place avec l'évaluation des expressions (fonctions plus arguments). Dans le cas pur, un programme est expression.
6. La programmation fonctionnelle se préoccupe de ce qui doit être calculé plutôt que de la manière dont il doit être calculé.
7. Une grande partie de la programmation fonctionnelle utilise des fonctions «d'ordre supérieur» (en d'autres termes, des fonctions qui agissent sur des fonctions qui fonctionnent sur des fonctions).

Python comme langage fonctionnel

La programmation fonctionnelle étant un style de programmation, elle pourrait être mise en œuvre avec tout langage de programmation.

Par contre pour être suffisamment puissante il est préférable de l'utiliser avec un langage

- permettant l'ordre supérieur
- et le polymorphisme. ou mieux le typage dynamique ou encore le duck typing

Python permet les 2, il possède aussi les fonctions anonyme lambda

Ordre supérieur et typage dynamique : Fonctionnel en Python

Python permet les 2, Les fonctions sont des entités comme toutes les autres, et le duck typing permet la souplesse au niveaux des “Abstraction fonctionnelle”

quand on défini par exemple **`EV if Cond else EF`**, on à une expression qui modélise:

si Cond est évalué à vraie, alors le résultat est celle de l’expression EV sinon l'évaluation est celle de l’expression EF . Dans ce cas peu importe les types de EV et EF. Le langage que l’on utilise devrait le permettre.

Je vous laisse réfléchir à cette définition

```
LunOuLautre = lambda c : lambda ev : lambda ef : ev if c else ef
```

Récursivité, récursivité terminale, itération

Sur un exemple nous allons montrer l'équivalence entre une fonction purement récursive et une fonction itérative issue de sa version récursive.

- Une fonction récursive terminale: Aucun traitement n'est effectué à la remontée d'un appel récursif (sauf le retour d'une valeur).
- Une fonction récursive non terminale: Le résultat de l'appel récursif est utilisé pour réaliser un traitement (en plus du retour d'une valeur).
    - Une fonction récursive terminale est en théorie plus efficace (mais souvent moins facile à écrire) que son équivalent non terminale : il n'y a qu'une phase de descente et pas de phase de remontée.
    - En récursivité terminale, les appels récursifs n'ont pas besoin d'être empilés dans la pile d'exécution, car l'appel suivant remplace simplement l'appel précédent dans le contexte d'exécution.
- Il est possible de transformer, de façon simple, une fonction récursive terminale en une fonction itérative : c'est la dérécursivation.

https://docs.python.org/dev/howto/functional.html

Les “callable”s en Python

1. Fonctions régulières créées avec def et nommées au moment de la définition
2. Fonctions anonymes créées avec lambda
3. Instances de classes qui définissent une méthode __call () __
4. Fermetures (Closure) renvoyées par les usines (Factory) de fonctions
5. *Méthodes statiques d'instances, soit via @staticmethod decoratoror, via la classe __dict__ (nous y reviendrons au moment de la synthèse du cours)*

Fonction nommée (def) et Lambda

```
>>> def hello1(name):
.....     return name

>>> hello2 = lambda name: name
```

lambda le corp est une expression qui est le résultat de la fonction.

Fonctionnel pure pas de modification d'états… avec lambda pas d'affectation possible

```
>>> lambda x: a=x
  File "<stdin>", line 1
SyntaxError: can't assign to lambda
```

On peut toujour utiliser def en s'imposat de ne pas utiliser les affectations…

Fermeture (closure) et instance fonctionnelle

```
# A class that creates callable adder instances
class Adder(object):
    def __init__(self, n):
        self.n = n
    def __call__(self, m):
        return self.n + m
add5_i = Adder(5)        # "instance" or "imperative"
```

Construit quelque chose appelable qui ajoute 5 à un argument transmis. Cela semble assez simple et assez mathématique

```
def make_adder(n):
    def adder(m):
        return m + n
    return adder
add5_f = make_adder(5)   # "functional"
```

ici aussi on construit quelque chose appelable qui ajoute 5 à un argument transmis. ici par une fermeture (n dans adder(m))

Les méthodes de classes

Toutes les méthodes de classes sont callables.

Pour la plupart, cependant, appeler une méthode d'une instance va à l'encontre des styles de programmation fonctionnels.

Habituellement, nous utilisons des méthodes parce que nous voulons référencer des données mutables regroupées dans les attributs de l'instance, et donc chaque appel à une méthode peut produire un résultat différent qui varie indépendamment des arguments qui lui sont passés.

Les générateurs (pas vraiment fonctionnel, mais evaluation lazy)

Des fonctions spéciales en Python sont celles qui contiennent une déclaration `yield` (rendre et attendre),

qui la transforme en **générateur.**

Ce qui est renvoyé en appelant une telle fonction n'est pas une valeur régulière, mais plutôt un **itérateur** qui produit ***une séquence de valeurs lorsque vous appelez la fonction next ()*** ou boucler dessus.

```python
def get_premier():
    """lazy Crible d'Eratosthenes

       il s'agit de supprimer d'une table des entiers de 2 à N tous les multiples d'un entier.
       En supprimant tous les multiples, à la fin il ne restera que les entiers qui ne sont multiples
       d'aucun entier, et qui sont donc les nombres premiers.
       On commence par rayer les multiples de 2, puis à chaque fois on raye les multiples du plus petit entier restant.
    """
    candidate = 2
    trouve = []
    while True:
        if all(candidate % prime != 0 for prime in trouve):
            yield candidate
            trouve.append(candidate)
        candidate += 1
```

Evaluation “Lazy” : Évaluation paresseuse

Évaluer les expressions uniquement si besoin.

Une puissante fonctionnalité de Python est son protocole d'itérateur:

L’utilisation du protocole de l’itérateur - et des nombreuses bibliothèques intégrées ou standard de Python - produit à peu près le même effet qu’une structure de donnée “Lazy”

Le protocol itérateur

La façon la plus simple de créer un itérateur - c'est-à-dire une séquence paresseuse (générée à la demande) - en Python est de définir une fonction de générateur.

Ou, techniquement, le moyen le plus simple consiste à utiliser l'un des nombreux objets itérables déjà produits par des modules intégrés ou la bibliothèque standard plutôt que de programmer un objet personnalisé.

De nombreux objets ont une méthode nommée .__ iter __ (), qui retourne un itérateur lors de son appel, généralement via la fonction intégrée iter (), ou encore plus souvent simplement en faisant une boucle sur l'objet (par exemple, for e in iterable: ...).

Qu'est-ce qu'un itérateur

Objet renvoyé par un appel à `iter(quelque chose)`, qui lui-même a

- une méthode nommée `.__ iter __ ()` qui renvoie simplement l'objet lui-même,
- et une autre méthode nommée `.__ next __ ()qui renverra les élément à la demande`.

La raison pour laquelle l'itérable lui-même a toujours une méthode .__ iter __ () est de rendre iter () idempotent.

pour vérifier par exemple qu'on a bien un itérable

Possède les 2 méthodes __iter__ et __next__

```
Python 3.6.8 (default, Oct  7 2019, 12:59:55)
[GCC 8.3.0] on linux
Type "help", "copyright", "credits" or "license" for m
>>> l=[1,2,3,4]
>>> l
[1, 2, 3, 4]
>>> iter(l)
<list_iterator object at 0x7f71a2024048>
>>> '__iter__' in dir(l)
True
>>> '__next__' in dir(l)
False
>>> it=iter(l)
>>> '__iter__' in dir(it) and '__next__' in dir(it)
True
>>>
```

```python
class Fibonacci(Iterable):
    def __init__(self):
        self.a, self.b = 0, 1
        self.total = 0
    def __iter__(self):
        return self
    def __next__(self):
        self.a, self.b = self.b, self.a + self.b
        self.total += self.a
        return self.a
    def running_sum(self):
        return self.total

# >>> fib = Fibonacci()
# >>> fib.running_sum()
# 0
# >>> for _, i in zip(range(10), fib):
# ...     print(i, end=" ")
# ...
# 1 1 2 3 5 8 13 21 34 55
# >>> fib.running_sum()
# 143
# >>> next(fib)
# 89
```

Fonction d’ordre supérieur

une fonction d'ordre supérieur est simplement une fonction qui prend une ou plusieurs fonctions comme arguments et / ou produit une fonction en conséquence.

Quelques fonctions utiles d'ordre supérieur sont contenues dans le module functools, et quelques autres sont intégrées. Il est courant de penser à map (), filter () et functools.reduce () comme les blocs de construction les plus élémentaires des fonctions d'ordre supérieur, et la plupart des langages de programmation fonctionnels utilisent ces fonctions comme primitives (parfois sous d'autres noms).

DONNÉES Entrée-> `FILTER -> MAP -> REDUCE` -> RÉSULTAT

currying :

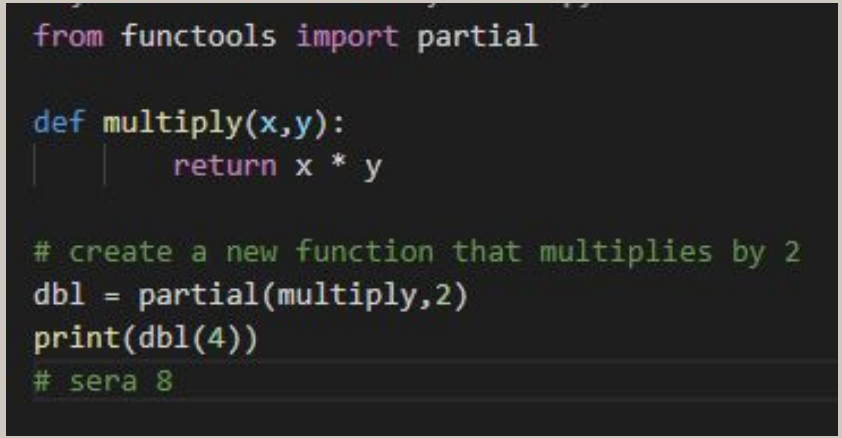
currying est functools.partial () en Python,

Une fonction qui prendra une autre fonction, avec zéro ou plusieurs arguments à pré-remplir, et retournera une fonction de moins d'arguments qui fonctionnera comme la fonction d'entrée le ferait lorsque ces arguments lui seraient transmis.

```python
from functools import partial

def multiply(x,y):
        return x * y

# create a new function that multiplies by 2
dbl = partial(multiply,2)
print(dbl(4))
# sera 8
```



Schémas de programmes

Schémas : Définitions récursives qui se ressemblent

La plupart des fonctions récursives

sur les listes ou les nombres que nous avons eu l'occasion de voir jusqu'à maintenant (par exemple les fonctions longueur, concat ou tri) et bien d'autres fonctions sur les listes (comme les fonctions somme et produit qui calculent respectivement la somme et le produit des éléments d'une liste de nombres)

ont des définitions récursives qui se ressemblent.

TOUJOURS LE MÊME SCHÉMA

```
def foncRec(l):
    if l:
        #Si la liste n'est pas vide
        return operation(tete(l),foncRec(reste(l)))
    else:
        #la séquence est vide
        return valeurInit
```

```
#l'exemple de la somme des élements d'une liste
def somme_l(l):
    if l:
        #ici l'opérateur est le +
        return l[0]+somme_l(l[1:])
    else:
        return 0
```

Ce shémas est un schémas de réduction

Voici un exemple d'implémentation de ce schémas

```
def reduire(op,vi,l):
    ''' op: operateur 2 parmètres élement et liste
      vi: valeur defaut si liste vide
      l: la sequence
    '''
    if l:
        print(f"reduire: tete={l[0]} reste={l[1:]}")
        return op(l[0], reduire(op, vi, l[1:]))
    else:
        return vi
```

reduce de Python liste -> scalaire

- Reduce en Python est différent de reduire définit précédemment

par exemple

- reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calcul ((((1+2)+3)+4)+5)
- essayer reduire(lambda x,y: 1+y,[1,2,3,100]) et reduce(lambda x,y: 1+y,[1,2,3,100])
    - reduce de Python traitement au niveaux des éléments reduire schémas de récursivité

Reduce de Python schémas fold_right, fold_left

Schémas fold_left et fold_right Dans le cas où l'opération binaire op est associative et a est un élément neutre pour cette opération, le résultat du schéma reduce peut s'exprimer comme suit en utilisant une notation mathématique sans parenthèses:

- e1 op e2 op ... op en op a
  - ou bien:
- a op e1 op e2 op ... op en où
- e1, e2,..., en sont les éléments de la liste
  - et l'on calcul op(op(op(a,e1),e2).....,en)
    - ou bien
  - op(e1,op(e2,.....,op(en,a))

Le shémas map

Applique une fonction à un argument à tous les éléments d’une liste. La fonction map de python est un générateur

exemple que donne ? : map(lambda x: x+1, range(10))

voir aussi : github, ShemasMap.py

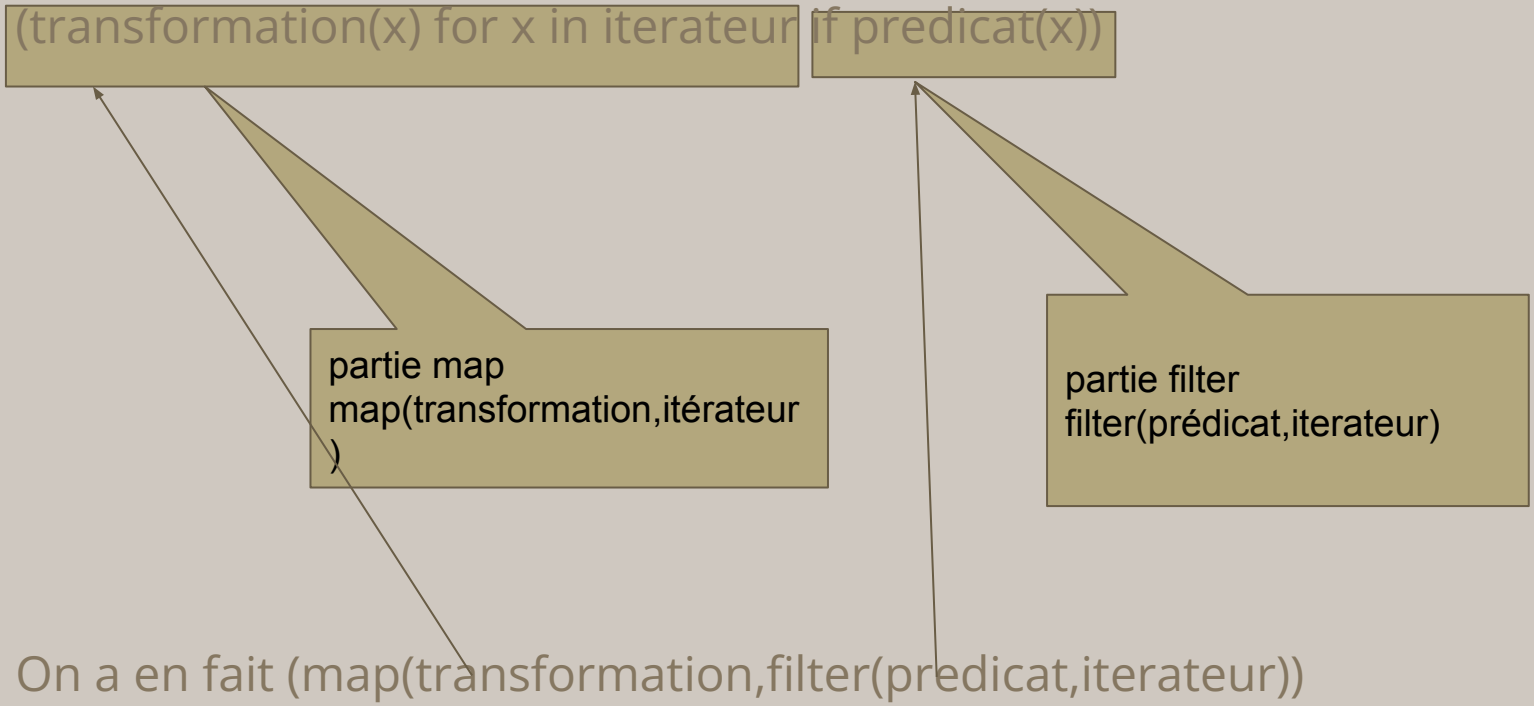
Le shémas filter

Filtre les éléments d’une liste renvoie une sous liste tel qu'il n’y ai que les éléments qui vérifient le prédicat (la condition)

exemple que donnerait? filter(lambda x: x>10,range(21))

voir aussi github ShemasFilter.py

Compréhension Vs Pure Fonctionnel



Exemple de fonction d’ordre supérieur

Voir l’exemple et discussion compose (dans le git)

Vers la programation réactive : Les traitements des flux

Filter -> map -> reduce

Source de donnée en entrée => Filter celle qui nous intéressent => modifié la donnée => donner une information en résultat

Question : Le flux, comment le récupérer?

Push/Pull : lire la donnée ou la recevoir par callback