Python et la Programmation fonctionnelle

Matthieu Amiguet

2009 – 2010

haute école arc ingénierie
neuchâtel berne jura saint-imier le locle delémont

1 Programmation fonctionnelle ?
2 Les fonctions comme entité de premier ordre
3 Lambda expressions
4 Des boucles sans boucles
5 Décorateurs

Programmation fonctionnelle 3

- L'expression "programmation fonctionnelle" peut se comprendre de plusieurs façons
  - Interprétation la plus forte : programmation effectuée uniquement à l'aide de fonctions au sens mathématique du terme ("programmation fonctionnelle *pure*")
  - Interprétation la plus faible : programmation utilisant les fonctions (au sens informatique du terme) da manière prépondérante
  - Entre deux : utilisation des fonctions (informatiques) comme des *objets de premier ordre*

Programmation fonctionnelle "pure" 4

- Au sens le plus strict :
  - Pas d'effets de bord
  - Pas d'affectation de variables
  - Pas de boucles (utilisation systématique de la récursivité)
- Avantages
  - Parallélisation
  - Vérification formelle
- Python ne s'y prête pas bien. On utilise plutôt des langages spécialisés
  - LISP, Scheme (hybrides)
  - ML, Caml, . . . (hybrides)
  - Haskell (pur)

BY-NC-SA distribué sous licence creative common | détails sur www.matthieuamiguet.ch

Python et la programmation fonctionnelle 5

- Python n'est donc pas un langage fonctionnel au sens le plus strict du terme
- Par contre, il offre des outils traditionnels de la programmation fonctionnelle
  - Fonctions comme entités de premier ordre
  - `map`, `filter`, ...
  - Lambda fonctions
  - ...
- C'est ces aspects "orientés fonctionnel" que nous allons étudier

1 Programmation fonctionnelle ?
2 Les fonctions comme entité de premier ordre
3 Lambda expressions
4 Des boucles sans boucles
5 Décorateurs

Une fonction est un objet comme un autre ! 7

- En python, les fonctions sont des objets (comme d'ailleurs les modules, les classes, et... tout !)
- On peut donc
  - Les affecter
  - Les passer en paramètre
  - ...
- Au fait, on a vu qu'une fonction est simplement un objet qui implémente la méthode `__call__` !
  - ⇝ notion de `callable`

Callable 8

```
>>> callable(2)
False
>>> callable(callable)
True
>>> dir(callable)
['__call__', '__class__', '__cmp__',
'__delattr__', '__doc__', '__getattribute__',
'__hash__', '__init__', '__module__',
'__name__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__self__',
'__setattr__', '__str__']
>>>
```

BY-NC-SA distribué sous licence creative common | détails sur www.matthieuamiguet.ch

Exemple 1 9
Affectation

```
import sys

old_exit = sys.exit

def my_exit(i=0):
    print "sys.exit called with argument", i
    old_exit(i)

sys.exit = my_exit

while True:
    x = raw_input(">>> ")
    if x == "": sys.exit(0)
    print x
```


Exemple 2 10
Passage en paramètre

```
from math import cos

def apply(f,x):
    return f(x)

x = apply(cos, 0)
print x
```


Exemple 3 11
Passage en paramètre

```
list = [
    'Bonjour',
    'au revoir!',
    'Au revoir'
    ]

list.sort(key=str.lower)

print list
```


Exemple 4 12
Attributs

```
def test(x):
    """ Ceci est une fonction test """
    return test.param + x

test.param = 2

print test.__doc__
print test(3)
```

BY-NC-SA distribué sous licence creative common | détails sur www.matthieuamiguet.ch

Exemple 5
Fonction comme objet


```
class adder:
    def __init__(self,n):
        self.n = n

    def __call__(self,x):
        return self.n+x

plus3 = adder(3)

print plus3(2)
```

1 Programmation fonctionnelle ?
2 Les fonctions comme entité de premier ordre
3 Lambda expressions
4 Des boucles sans boucles
5 Décorateurs

Trier des chaînes selon leur 2e lettre ?


```
list = [
    'Bonjour',
    'Au revoir',
    'Hello',
    'Bye'
    ]

def second_pos(s):
    return s[1]

list.sort(key=second_pos)

print list
```


Une solution plus légère


```
list = [
    'Bonjour',
    'Au revoir',
    'Hello',
    'Bye'
    ]

list.sort(key=lambda s: s[1])

print list
```

BY-NC-SA distribué sous licence creative common | détails sur www.matthieuamiguet.ch

Lambda expressions 17

- L'expression `lambda args : expr(args)` est équivalente à

```
def anonymous(args):
    return expr(args)
```

... sauf qu'on a pas besoin de donner un nom à la fonction

  - La lambda-expression retourne simplement un pointeur sur l'objet fonction correspondant
  - C'est pourquoi on appelle parfois les lambda-expressions des *fonctions anonymes*

- En python, le corps d'une lambda-expression est *limité à une seule expression*

Exemple 18
Dictionnaires comme "switch"

```
ops = {
    'plus' : lambda x,y: x+y,
    'moins' : lambda x,y: x-y
}

op = raw_input('Quelle operation ? ')

print ops[op](2,3)
```

1 Programmation fonctionnelle ?
2 Les fonctions comme entité de premier ordre
3 Lambda expressions
4 Des boucles sans boucles
5 Décorateurs

Outils fonctionnels standard 20

- Dans les langages fonctionnels, les boucles sont facilement remplacées par des appels à des fonctions s'appliquant à des fonctions
- Trois grands classiques
  - `map(function, list)` applique successivement `function` aux éléments de la liste et retourne la liste des résultats
  - `filter(function, list)` retourne une liste des éléments de `list` pour lesquels `function` est vrai
  - `reduce(bin_func, list)` applique `bin_func` aux deux premiers éléments de la liste, puis au résultat et au 3e élément, puis... et retourne le résultat final.

BY-NC-SA distribué sous licence creative common | détails sur www.matthieuamiguet.ch

Exemples 21

```
l = [1,2,3,4]

print "list: ", l

print "squares:", map(lambda x: x**2, l)
print "odd numbers:", filter(lambda x: x%2, l)
print "sum:", reduce(lambda x,y: x+y, l)
```


List comprehensions 22

- Les fonctions ci-dessus sont des classiques de la programmation fonctionnelle
- Mais python, en s'inspirant de Haskell, offre une alternative combinant `map` et `filter` dans une nouvelle syntaxe : les *list comprehensions*
- Syntaxe python :
  `[expr(x) for x in list]`
  $\Leftrightarrow$ `map(expr,list)`
  `[expr(x) for x in list if test(x)]`
  $\Leftrightarrow$ `map(expr,filter(test,list))`

Exemple 23

```
l = [1,2,3,4]

print "list: ", l

print "squares:", [x**2 for x in l]
print "odd numbers:", [x for x in l if x%2==1]

l2 = [x**2 for x in l if x%2==1]
print "squares of odd numbers:", l2

# Equivalent
l2 = []
for x in l:
    if x%2 ==1:
        l2.append(x**2)
print "squares of odd numbers:", l2
```


Generator expressions 24

- Si, dans une *list comprehension*, on remplace les crochets par des parenthèses, on obtient un *générateur*
- Exemple : afficher toutes les lignes non vides d'un fichier texte en enlevant les espaces en début et en fin de ligne et ceci *sans charger tout le fichier en mémoire d'un coup* :

```
with  file('test.txt') as f:
    nbl = (l.strip() for l in f if l != "\n")
    for l in nbl:
        print l
```

BY-NC-SA distribué sous licence creative common | détails sur www.matthieuamiguet.ch

Avantages 25

- Les constructions que nous avons vu dans ce chapitre permettent de résumer une boucle de plusieurs lignes de code en une expression
  - Moins à taper
  - Moins de risque d'erreur
  - Plus lisible (?)
- De plus, une boucle python est implémentée... en python, alors que les constructions ci-dessus sont implémentées en C
  - Elles sont donc (sensées être) plus rapides ! (particulièrement les *list comprehensions*)

1 Programmation fonctionnelle ?
2 Les fonctions comme entité de premier ordre
3 Lambda expressions
4 Des boucles sans boucles
5 Décorateurs

Fonctions de fonction 27

- Puisque les fonctions sont des objets de premier ordre, elles peuvent
  - être passées en paramètre
  - être retournées par une fonction
- On peut donc faire des fonctions qui transforment des fonctions
  - Pendre une fonction en paramètre...
  - ... la modifier...
  - ... et la retourner !

Exemple 1 28

```
def deco(f):
    "useless transformation of a function"
    f.__doc__ += ", but decorated"
    return f


def f(x):
    "useless function"
    return x

f = deco(f)
```

BY-NC-SA distribué sous licence creative common | détails sur www.matthieuamiguet.ch

Exemple 1 – résultat 29

```
C :\> pydoc exemple1
[...]
FUNCTIONS
    deco(f)
        useless transformation of a function
    f(x)
        useless function, but decorated
```


Exemple 2 30

```
def wrapper(f):
    "slightly more useful function"
    def inner(x):
        print "(Entering %s)" % f.__name__,
        result = f(x)
        print "(Exiting %s with result %s)" % ( ↩
            f.__name__, result),
        return result
    return inner

def f(x):
    "useless function"
    return x
f = wrapper(f)

print "f(1) = ", f(1)
```


Exemple 2 – résultat 31

```
C :\> python example2.py
f(1) = (Entering f) (Exiting f with result 1)
1
```


Décorateurs 32

- En python, une fonction qui modifie une fonction est appelée un *décorateur*
- Pour éviter la syntaxe suivante, un peu lourde :

```
def f(x):
    [...]
f = decorator(f)
```

Python propose la syntaxe *équivalente* suivante

```
@decorator
def f(x):
    [...]
```

BY-NC-SA distribué sous licence creative common | détails sur www.matthieuamiguet.ch

Sucre syntaxique 33

- Les décorateurs ne sont que du “sucre syntaxique”
  - On peut s’en passer et utiliser la syntaxe équivalente
- Mais la syntaxe proposée
  - rend la décoration plus visible (surtout pour une longue fonction)
  - simplifie l’utilisation des décorateurs pré-existants (?)
    - cf. p. ex. `contextlib.contextmanager`, `classmethod`, `staticmethod`, ...

Plusieurs décorateurs... 34

- Les décorateurs peuvent être chaînés !

```
@dec1
@dec2
@dec3
def f(x):
    [...]

# Equivalent:

f = dec1(dec2(dec3(f)))
```


Décorateurs avec arguments 35

- Supposons que l’on veuille écrire quelque chose comme ça :

```
@accepts(int)
def succ(x):
    return x+1
```

  - On doit vérifier si le type de l’argument est un `int` et lever une exception sinon
- Alors la fonction `accepts` doit
  - Prendre un type en argument
  - Retourner un décorateur, c’est-à-dire une fonction qui
    - prend une fonction en argument et
    - retourne une fonction “décorée”

Exemple 36

```
def accepts(type):
    """Decorator verifying that the argument
        is of given type"""
    def deco(f):
        def inner(arg):
            if not isinstance(arg,type):
                raise TypeError
            return f(arg)
        return inner
    return deco
```

BY-NC-SA distribué sous licence creative common | détails sur www.matthieuamiguet.ch

Un peu de cosmétique... 37

Pour que l'application d'un décorateur soit transparente dans toutes les situations, on devrait compléter un peu le code :

```
def wrapper(f):
    def inner(x):
        print "(Entering %s)" % f.__name__,
        result = f(x)
        print "(Exiting %s with result %s)" %  (f. ↩
            __name__, result),
        return result
    inner.__doc__ = f.__doc__
    inner.__name__ = f.__name__
    inner.__dict__.update(f.__dict__)
    return inner
```


Pour aller plus loin 38

- Tarek Ziadé, "Programmation Python", Eyrolles, 2006
- `http://docs.python.org/tut/tut.html`
- `http://wiki.python.org/moin/PythonDecoratorLibrary`
- `decorator` module :
  - ⇝ module (tiers) proposant des décorateurs simplifiant l'écriture de décorateurs (!) `http://www.phyast.pitt.edu/~micheles/python/documentation.html`

BY-NC-SA distribué sous licence creative common | détails sur www.matthieuamiguet.ch