Python – Cours 5

Jean-Yves Thibon

Université Paris-Est Marne-la-Vallée

ESIPE/IMAC3

- Décorateurs
- Méthodes d'instance, de classe, statiques
- Itération itérateurs et itertools
- Autres optimisations
- Extensions C/C++

Décorateurs I

Un décorateur sert à envelopper un fonction dans une autre :

```
@dec
def f(arg1, arg2):
    pass

# est équivalent à

def f(arg1, arg2):
    pass
f = dec(f)
```

où `dec` prend une fonction comme argument et retourne une fonction.

Décorateurs II

Un petit test pour comprendre ce qui se passe :

```
def trace(f):
    def traced(*args, **kwargs):
        print '>>'
        f(*args, **kwargs)
        print '<<'
    return traced

@trace
def f1(truc):
    print 'truc:', truc

@trace
def f2(x, y):
    print 'x:', x, 'y:', y
    f1((x, y))
```

Décorateurs III

Résultat :

```
>>> f1(3)
>>
truc: 3
<<
>>> f2('toto',666)
>>
x: toto y: 666
>>
truc: ('toto', 666)
<<
<<
>>>
```

Décorateurs IV

Exemple : mise en cache des valeurs d'une fonction récursive (*memoization*)

```
def memoize(f):
    cache = {}
    def memoized(*args):
        try:
            return cache[args]
        except KeyError:
            result = cache[args] = f(*args)
            return result
    return memoized
```

Décorateurs V

Avec la suite de Fibonacci

```
@memoize
def fib(n):
    if n<2: return n
    else: return fib(n-1)+fib(n-2)

>>> fib(200)
280571172992510140037611932413038677189525L
```

Décorateurs VI

Mesure du temps d'éxécution d'une fonction :

```
import time

def timeit(f):
    def timed(*args, **kw):
        ts = time.time()
        result = f(*args, **kw)
        te = time.time()
        print '%r (%r, %r) %2.2f sec' % \
              (f.__name__, args, kw, te-ts)
        return result
    return timed
```

Décorateurs VII

Résultat :

```
@timeit
def g(t):
    print 'début'
    time.sleep(t)
    print 'fin'

>>> g(3.1415926535)
début
fin
'g' ((3.1415926535,), {}) 3.17 sec
>>>
```

Décorateurs VIII

```
class C(object):
    @timeit
    def __init__(self):
        time.sleep(2.718281828)
        print 'Fini !'

>>> c=C()
Fini !
'__init__' ((<__main__.C object
       at 0x7f84c9ca8c90>,), {}) 2.73 sec
```

Décorateurs IX

Vérifier le type d'un argument :

```
def require_int(f):
    def wrapper (arg):
        assert isinstance(arg, int)
        return f(arg)
    return wrapper

@require_int
def h(n):
    print  n, " est un entier."
```

Décorateurs X

```
>>> h(5)
5  est un entier.
>>> h('toto')

Traceback (most recent call last):
  File "<pyshell#67>", line 1, in <module>
    h('toto')
  File "/home/jyt/dec.py", line 67, in wrapper
    assert isinstance(arg, int)
AssertionError
```

Décorateurs XI

Les décorateurs peuvent être empilés :

```
@timeit
@require_int
def rien(n):
    time.sleep(n)
    print 'Fini'

>>> rien(3)
Fini
'wrapper' ((3,), {}) 3.02 sec
>>> rien(3.1)
Traceback (most recent call last):
  File "<pyshell#69>", line 1, in <module>
    rien(3.1)
  File "/home/jyt/dec.py", line 44, in timed
    result = f(*args, **kw)
  File "/home/jyt/dec.py", line 67, in wrapper
    assert isinstance(arg, int)
AssertionError
```

Décorateurs avec arguments I

```
@dec(argA, argB)
def f(arg1, arg2):
    pass

# est équivalent à

def f(arg1, arg2):
    pass
f = dec(argA, argB)(f)
```

C'est donc équivalent à créer une fonction composée `f = dec(argA, argB)(f)` Autrement dit, `dec(argA, argB)` doit être un décorateur.

Décorateurs avec arguments II

Ajouter un attribut à une fonction :

```
def add_attr(val):
    def decorated(f):
        f.attribute = val
        return f
    return decorated

@add_attr('Nouvel attribut')
def f():
    pass

>>> f()
>>> print f.attribute
Nouvel attribut
```

Décorateurs avec arguments III

Tester le type de la valeur retournée par une fonction :

```
def return_bool(bool_value):
    def wrapper(func):
        def wrapped(*args):
            result = func(*args)
            if result != bool_value:
                raise TypeError
            return result
        return wrapped
    return wrapper

@return_bool(True)
def always_true():
    return True

@return_bool(False)
def always_false():
    return True
```

Décorateurs avec arguments IV

```
>>> always_true()
True
>>> always_false()

Traceback (most recent call last):
  File "<pyshell#72>", line 1, in <module>
    always_false()
  File "/home/jyt/dec2.py", line 48, in wrapped
    raise TypeError
TypeError
>>>
```

Décorateurs définis par des classes I

La seule contrainte sur l'objet retourné par un décorateur est qu'il se comporte comme une fonction (*duck typing*), autement dit qu'il soit *callable*.
C'est le cas de toute classe possédant la méthode spéciale `__call__`.

```
class MyDecorator(object):
    def __init__(self, f):
    # faire quelquechose avec f ...
   def __call__(*args):
    # faire autre chose avec args
```

Décorateurs définis par des classes II

```
class Memoized(object):
    def __init__(self, f):
        self.f = f
        self.cache = {}
    def __call__(self, *args):
        if args in self.cache:
            return self.cache[args]
        else:
            value = self.f(*args)
            self.cache[args] = value
            return value
    def __repr__(self):
        '''Return the function's docstring.'''
        return self.f.__doc__
```

Décorateurs définis par des classes III

Permettre à une fonction de compter combien de fois elle a été appélée :

```
class countcalls(object):
    def __init__(self, func):
        self.__func = func
        self.__numcalls = 0
    def __call__(self, *args, **kwargs):
        self.__numcalls += 1
        return self.__func(*args, **kwargs)
    def count(self):
        return self.__numcalls
```

Décorateurs définis par des classes IV

```
@countcalls
def p(): print '*',

@countcalls
def q(): print '+'

for i in range(5):
    p()
for i in range(4):
    p();q()
print
* * * * * * + * + * + * +
>>> p.count()
9
>>> q.count()
4
```

Variables de classes, méthodes de classes et méthodes statiques I

On a déjà vu la différence entre les variables des classes et celles des instances :

```
class A(object):
    c = 0
    def __init__(self):
        self.c +=1

class B(object):
    c = 0
    def __init__(self):
        B.c +=1
```

Variables de classes, méthodes de classes et méthodes statiques II

```
>>> a=A(); b=A()
>>> a.c, b.c, A.c
(1, 1, 0)
>>> x=B(); y=B()
>>> x.c, y.c, B.c
(2, 2, 2)
>>>
```

Variables de classes, méthodes de classes et méthodes statiques III

Les méthodes normales sont des méthodes d'instances. Leur premier argument doit être l'instance elle-même, conventionnellement appelée `self`.
Mais il existe aussi des méthodes de classes. On les définit comme les méthodes d'instance, leur premier argument est alors la classe elle-même, conventionnellement appelée `cls`, puis on les passe à la fonction `classmethod`

```
class ASimpleClass(object):
    description = 'a simple class'
    def show_class(cls, msg):
        print '%s: %s' % (cls.description , msg, )
        show_class = classmethod(show_class)
```

Variables de classes, méthodes de classes et méthodes statiques IV

C'est plus clair avec un décorateur. Python fournit

```
@classmethod

class ASimpleClass(object):
    description = 'a simple class'
    @classemethod
    def show_class(cls, msg):
        print '%s: %s' % (cls.description , msg, )
```

Variables de classes, méthodes de classes et méthodes statiques V

Par exemple, la classe `B` qui tient un compte de ses instances pourrait s'écrire

```
class B(object):
    c = 0
    def __init__(self):
        B.c += 1

    @classmethod
    def compte_instances(cls):
        print 'instances : %d' % (cls.c, )
```

Variables de classes, méthodes de classes et méthodes statiques VI

Une méthode statique ne prend ni une instance ni la classe comme premier paramètre. Elle se définit à l'aide de la fonction `staticmethod` ou du décorateur `@staticmethod`

```
class B(object):
    c = 0
    def __init__(self):
        B.c += 1

    @staticmethod
    def compte_instances():
        print 'instances : %d' % (B.c, )

>>> a=B(); b=B(); c=B()
>>> B.compte_instances()
instances : 3
```

Itération, itérateurs et `itertools` I

- Syntaxe de l'itération : `for x in <quelquechose>`
- `quelquechose` peut être une liste, un tuple, une chaîne, un dictionnaire, un fichier ouvert, un ensemble ...
- Ces objets itérables possèdent une méthode spéciale `__iter__`
- On l'appelle au moyen de la fonction `iter`
- Elle retourne un itérateur qui possède une méthode `next`

```
>>> ll = [1, 2, 3, 4, 5]
>>> it = ll.__iter__()
>>> it.next()
1
>>> it.next()
2
```

Itération, itérateurs et `itertools` II

Normalement, on écrit plûtôt

```
>>> ll = [1, 2, 3, 4, 5]
>>> it = iter(ll)
>>> it.next()
1
>>> it.next()
2
>>> for x in it: print x,

3 4 5
>>>
```

On remarque que l'itérateur se consume au fur et à mesure que `next` est appelée. Si on continue :

Itération, itérateurs et `itertools` III

```
>>> it.next()

Traceback (most recent call last):
  File "<pyshell#52>", line 1, in <module>
    it.next()
StopIteration
>>>
```

```
for x in obj:
   # faire quelque chose
```

est équivalent à

Itération, itérateurs et `itertools` IV

```
_iter = iter(obj)
while 1:
    try:
        x = _iter.next()
    except StopIteration:
        break
    # faire quelque chose
```

Itération, itérateurs et `itertools` V

On peut définir des classes qui supportent l'itération : il suffit d'implémenter les méthodes `__iter__` et `next` :

```
class countdown(object):

    def __init__(self,n):
        self.count = n

    def __iter__(self):
        return self

    def next(self):
        if self.count <= 0:
            raise StopIteration
        r = self.count
        self.count -= 1
        return r
```

Itération, itérateurs et `itertools` VI

```
>>> c=countdown(10)
>>> c.next()
10
>>> list(c)
[9, 8, 7, 6, 5, 4, 3, 2, 1]
>>>
```

Itération, itérateurs et `itertools` VII

- Les boucles longues sont peu efficaces et sont une des principales causes de lenteur en Python.
- Pour des boucles sur les entiers, on utilisera `xrange` (un générateur écrit directement en C) plutôt que `range`.
- Pour des itérations plus compliquées, on pourra utiliser le module `itertools`, qui propose des version optimisées d'opérations courantes, et de nombreuses fonctionnalités commodes.

Itération, itérateurs et `itertools` VIII

On peut chainer des itérateurs :

```
>>> from itertools import *
>>> it=chain(range(5),range(5,-1,-1))
>>> list(it)
[0, 1, 2, 3, 4, 5, 4, 3, 2, 1, 0]
>>>
```

ou les tricoter ... (`zip` retourne une liste)

```
>>> it=izip(range(5),range(5,-1,-1))
>>> list(it)
[(0, 5), (1, 4), (2, 3), (3, 2), (4, 1)]
>>>
```

Itération, itérateurs et `itertools` IX

`count(start,step=1)` engendre les entiers à partir de `start` avec les pas `step`. La fonction `islice(iterable,[start],stop,[step])` remplace `iterable[start :stop :step]` :

```
>>> it=islice(count(5),7,17,2)
>>> list(it)
```

Avec les mots binaires de 5 digits, on pourrait faire

```
>>> words = product(*['01' for i in range(5)])
>>> ll=islice(words,16,24)
>>> list(ll)
[('1', '0', '0', '0', '0'), ('1', '0', '0', '0', '1'),
('1', '0', '0', '1', '0'),
('1', '0', '0', '1', '1'), ('1', '0', '1', '0', '0'),
('1', '0', '1', '0', '1'),
('1', '0', '1', '1', '0'), ('1', '0', '1', '1', '1')]
>>>
```

Itération, itérateurs et `itertools` X

On remarquera au passage la fonction `product` qui renvoie le produit cartésien (les tuples) d'un nombre arbitraire d'itérables. On peut s'en servir pour construire les mots de longueur donnée sur un alphabet, comme dans ce craqueur de mots de passe basique :

```
from crypt import crypt

def words(alphabet,length):
    return product(*[alphabet for i in range(length)])

def crack(password, salt, alphabet,length):
    ww = words(alphabet,length)
    for w in ww:
        p = crypt(''.join(w),salt)
        if p == password:
            print 'Password found: ', ''.join(w)
            return ''.join(w)
```

Itération, itérateurs et `itertools` XI

Par exemple,

```
from string import lowercase
>>> pw = 'toto'
>>> slt ='XY'
>>> h = crypt(pw,slt)
>>> h
'XYwNfZo28h1a6'
>>> 'XYwNfZo28h1a6'
'XYwNfZo28h1a6'
>>> crack(h,'XY',lowercase,4)
Password found:  toto
'toto'
```

Ce n'est pas très efficace (!) mais c'est simple ...

Itération, itérateurs et `itertools` XII

La fonction `tee(iterateur,n=2)` retourne *n* copies identiques de l'itérateur

```
>>> it = islice(count(), 5)
>>> i1, i2, i3 = tee(it,3)
>>> [(i1.next(), i2.next(),i3.next()) for i in range(5)]
[(0, 0, 0), (1, 1, 1), (2, 2, 2), (3, 3, 3), (4, 4, 4)]
>>> list(it)       # entièrement consommé ...
[]
>>>
```

Itération, itérateurs et `itertools` XIII

La fonction `imap` fonctionne comme `map`, mais s'arrête lorsque l'un des itérateurs est entièrement consommé (au lieu d'insérer des `None`) :

```
>>> map(lambda x,y: x*y, range(4), range(4))
[0, 1, 4, 9]
>>> it=imap(lambda x,y: x*y, range(4), range(8))
>>> list(it)
[0, 1, 4, 9]
>>> map(lambda x,y: x*y, range(4), range(8))

Traceback (most recent call last):
  File "<pyshell#67>", line 1, in <module>
    map(lambda x,y: x*y, range(4), range(8))
  File "<pyshell#67>", line 1, in <lambda>
    map(lambda x,y: x*y, range(4), range(8))
TypeError: unsupported operand type(s) for *:
                    'NoneType' and 'int'
>>>
```

Itération, itérateurs et `itertools` XIV

La fonction `starmap` fonctionne comme `imap`, mais calcule `f(*i)`

```
>>> it = izip('abcd', xrange(1,5))
>>> ff = starmap(lambda x,y: x*y,it)
>>> list(ff)
['a', 'bb', 'ccc', 'dddd']
>>>
```

La fonction `cycle` répète indéfiniment un itérateur fini

```
>>> it = cycle('bla')
>>> [it.next() for i in range(12)]
['b', 'l', 'a', 'b', 'l', 'a', 'b', 'l', 'a', 'b', 'l',
>>>
```

Itération, itérateurs et `itertools` XV

et la fonction `repeat` fait ce qu'on imagine :

```
>>> it=repeat('bla',4)
>>> list(it)
['bla', 'bla', 'bla', 'bla']
>>>
```

On l'utilise en combinaison avec `imap` ou `izip`

```
>>> it = izip(xrange(5), repeat(2), 'abcd')
>>> list(it)
[(0, 2, 'a'), (1, 2, 'b'), (2, 2, 'c'), (3, 2, 'd')]
>>>
```

Itération, itérateurs et `itertools` XVI

Le filtrage s'effectue au moyen des fonctions `dropwhile, takewhile, ifilter, ifilterfalse`

```
>>> it=takewhile(lambda x:x*x<100, count())
>>> list(it)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> it=dropwhile(lambda x:x<0, range(-6,5))
>>> list(it)
[0, 1, 2, 3, 4]
>>> it=ifilter(lambda x:x%2, xrange(10))
>>> list(it)
[1, 3, 5, 7, 9]
>>> it=ifilterfalse(lambda x:x%2, xrange(10))
>>> list(it)
[0, 2, 4, 6, 8]
```

Itération, itérateurs et `itertools` XVII

Plus complexe : `groupby` construit un itérateur qui renvoie les clés et groupes consécutifs d'un itérable.La clé `key` est une fonction qui calcule une valeur sur chaque élément. Par défaut, c'est l'identité.
On pourra donc écrire

```
>>> ll = [1,2,2,2,1,1,4,4,5,5,2,2,1,1,3,3,1,1]
>>> it=groupby(ll)
>>> [k for k,g in it]
[1, 2, 1, 4, 5, 2, 1, 3, 1]
# le résultat complet serait
 >>> [(k, list(g))  for k,g in it]
[(1, [1]), (2, [2, 2, 2]), (1, [1, 1]), (4, [4, 4]), (5, [5, 5]),
(2, [2, 2]), (1, [1, 1]), (3, [3, 3]), (1, [1, 1])]

>>> ll = [('a',1), ('b',2), ('c',2), ('d',1), ('e',2), ('f',1)]
>>> it=groupby(ll, lambda x: x[1])
>>> [(k, list(g))  for k,g in it]
[(1, [('a', 1)]), (2, [('b', 2), ('c', 2)]), (1, [('d', 1)]),
      (2, [('e', 2)]), (1, [('f', 1)])]
>>>
```

Itération, itérateurs et `itertools` XVIII

Finalement, on dispose de quelques fonctions combinatoires basiques :

```
>>> it=combinations('abcd',2)
>>> list(it)
[('a', 'b'), ('a', 'c'), ('a', 'd'), ('b', 'c'),
('b', 'd'), ('c', 'd')]
>>> it=combinations_with_replacement('abcd',2)
>>> list(it)
[('a', 'a'), ('a', 'b'), ('a', 'c'), ('a', 'd'),
('b', 'b'), ('b', 'c'),
 ('b', 'd'), ('c', 'c'), ('c', 'd'), ('d', 'd')]
>>> it=permutations(range(1,4))
>>> list(it)
[(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2),
```

Autres optimisations I

- Le module `operator` propose des fonctions optimisées pour remplacer les opérateurs standards de Python (ex. `add(x,y)`.
- Le module `collections` fournit des structures de données hautes performances pour remplacer `dict`, `list`, `set`, `tuple`:`namedtuple()`, `deque`, `Counter`, `OrderedDict`, `defaultdict`.
- Le module `array` fournit des tableaux optimisés pour des types de données basiques (caractères, entiers, flottants ...)

Le module `ctypes` I

Il permet d'utiliser des bibliothèques partagées, avec des types de données compatibles au C.

```
>>> from ctypes import *
>>> cdll.LoadLibrary("libc.so.6")
<CDLL 'libc.so.6', handle 7f9d03130000 at 7f9d030bb710>
>>> printf=libc.printf
>>> printf("%s\n", "Hello world:")
13  # depuis idle, le résultat s'affiche sur le terminal
>>> print libc.time(None)
1381746728
```

Types

`c_int, c_char, c_bool ..., Structure, pointer ...,`

Le module `ctypes` II

Pour de longues itérations, on peut écrire en C la fonction critique et la compiler sous forme *dll/shared object*
Un petit test de performances :

```
/* rien.c
 compiler avec
   gcc -Wall -fPIC -c rien.c
   gcc -shared -Wl,-soname,librien.so.1
                    -o librien.so.1.0   *.o
*/

int rien(int n){
    int i=0;
    while (1==1) {
        i++;
        if(i>n){ return(i); }
    }
}
```

Le module `ctypes` III

```
from ctypes import *
cdll.LoadLibrary("./librien.so.1.0")
librien=CDLL("./librien.so.1.0")

def rien(n):
    i=0
    while 1==1:
        i+=1
        if i>n: return i

from time import time
print "Avec le C :"
a=time()
librien.rien(10000000)
print time()-a
print "En pur Python :"
a=time()
rien(10000000)
print time()-a
```

Le module `ctypes` IV

Le verdict est sans appel ...

```
Avec le C :
0.0196750164032
En pur Python
0.450246095657
# si on remplace while 1==1: par while 1:
# c'est un peu mieux :
En pur Python :
0.896075963974
```

Cython I

Pour étendre Python avec du code C ou C++, il vaut mieux utiliser Cython (`http ://cython.org`)

- Cython est un sur-langage de Python (basé sur Pyrex) avec les types de données du C
- Il possède un compilateur optimisé
- Presque tout code Python est aussi du code Cython valide
- En déclarant les types, on obtient du code très efficace

Cython II

Après avoir installé Cython, on peut reprendre l'exemple précédent. On crée un fichier `rien.pyx`

```
# rien.pyx
def rien(int n):
    cdef int i=0
    while 1==1:
        i+=1
        if i>n: return i
```

Le code est le même, à ceci près que les types (`int`) de `n` et `i` ont été déclarés.

Cython III

Pour compiler, il faut, dans le même répertoire, un fichier `setup.py` structuré ainsi

```
# setup.py
from distutils.core import setup
from Cython.Build import cythonize

setup( ext_modules = cythonize("rien.pyx") )
```

On compile avec la commande

```
python setup.py build_ext --inplace
```

On peut alors importer `rien` comme un module ordinaire

Cython IV

```
#testrien.py
from time import time
import rien

def pyrien(n):
    i=0
    while 1==1:
        i+=1
        if i>n: return i

print "Avec le C :"
a=time()
rien.rien(10000000)
print time()-a
print "En pur Python :"
a=time()
pyrien(10000000)
print time()-a
```

Cython V

Le résultat est plûtôt convaincant

```
>>>
>>>
Avec le C :
0.00649309158325
En pur Python :
0.505649805069
>>>
```

Notons au passage que si on n'avait pas déclaré les types, l'effet aurait été beaucoup moins bon

```
>>>
Avec le C :
0.225028038025
En pur Python :
0.465053796768
```