

# Python : Quelques structures avancées

Matthieu Amiguet

2008 – 2009



---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

- 1 Exceptions
- 2 Surcharge d'opérateurs
- 3 Itérateurs
- 4 Générateurs
- 5 Gestionnaires de contexte

## Exceptions

3

```
try:
    x = 1/0
except:
    print "Undefined error!"
```

```
try:
    x = 1/0
except ZeroDivisionError:
    print "Division by zero!"
```

```
try:
    x = 1/0
except ZeroDivisionError, e:
    print e
```

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

## Exceptions – suite

4

```
try:
    x = raw_input()
    x = 1/0
except (ZeroDivisionError, IOError), e:
    print e
    raise # re-raise the exception
```

```
try:
    x = raw_input()
    x = 1/0
except ZeroDivisionError:
    print "Division by 0!"
except IOError:
    print "Input/Output problem!"
```

```

try:
    x = int(raw_input())
    y = 1/x
except ZeroDivisionError:
    pass # good idea?
except IOError:
    print "Input/Output problem!"
except: # catch all other exceptions
    print "Unknown problem!"
    raise
else:
    print "Everything went well"
finally:
    print "This is always executed"

```

---

---

---

---

---

---

---

---

---

---



---

---

---

---

---

---

---

---

---

---

- 1 Exceptions
- 2 Surcharge d'opérateurs
- 3 Itérateurs
- 4 Générateurs
- 5 Gestionnaires de contexte

La plupart des opérateurs peuvent être surchargés en (ré-)implémentant des méthodes dites "spécaiales" (dont le nom commence et finit par deux \_)

- $a+b \iff a.__add__(b)$  ou (à défaut)  $b.__radd__(a)$ 
  - idem avec `__sub__`, `__mul__`, `__mod__`,...
- $a==b \iff a.__eq__(b)$ 
  - idem avec `__lt__`, `__le__`,... (ou `__cmp__` à la place de tout ça...)
- $\text{print } a \iff \text{print str}(a) \iff \text{print } a.__str__()$ 
  - (cf. aussi `__repr__`)

---

---

---

---

---

---

---

---

---

---



---

---

---

---

---

---

---

---

---

---

- Émulation des listes/dictionnaires ...
  - $\text{len}(a) \iff a.__len__()$
  - $a[b] \iff a.__getitem__(b)$  ou  $a.__setitem__(b)$
- ... et des fonctions
  - $f(a) \iff f.__call__(a)$
  - pour une signature générique :
 

```
def __call__(a,b,*args,**kwargs):
```

    - `args` contient la *liste* des arguments positionnels supplémentaires
    - `kwargs` contient le *dictionnaire* des arguments nommés

---

---

---

---

---

---

---

---

---

---

```

class DynAttr():
    # only called when the attribute is not ←
    found
    def __getattr__(self, name):
        print "** Unknown attribute %s!" % name
        return None

    # always called!
    def __setattr__(self, name, value):
        print "Setting attribute", name
        # Avoid infinite recursion!
        self.__dict__[name] = value

```

---

---

---

---

---

---

---

---

---

---

```

>>> d=DynAttr()
>>> print d.a
** Unknown attribute a!
None
>>> d.a=2
Setting attribute a
>>> print d.a
2

```

- Peut être utilisé par exemple dans les ORM : nom de méthodes dynamiques

---

---

---

---

---

---

---

---

---

---

```

class C(object): #new-style class
    def getx(self): return self.__x
    def setx(self, value): self.__x = value
    def delx(self): del self.__x
    x = property(getx, setx, delx, "I'm the 'x' ←
    property.")

```

- Permet d'éviter les setters et getters
- ... quitte à les rajouter de manière transparente lors d'une refactorisation!

---

---

---

---

---

---

---

---

---

---

- 1 Exceptions
- 2 Surcharge d'opérateurs
- 3 **Itérateurs**
- 4 Générateurs
- 5 Gestionnaires de contexte

## Les boucles en python

13

- Les boucles `for` en python fonctionnent sur des listes...

```
for fruit in ['apple', 'orange', 'banana']:
    print fruit
```

- ... Mais aussi sur d'autres objets :

```
f = file('filename.txt')
for l in f: # pour chaque ligne du fichier
    print l
```

- On peut donc se demander
  - Comment ça marche
  - Comment implémenter ce comportement dans nos propres objets.

---

---

---

---

---

---

---

---

---

---

## Le protocole *itérable*

15

- Un objet sera *itérable* en python s'il possède une méthode `__iter__(self)` qui :
  - retourne un objet qui
    - implémente une méthode `next(self)`
    - Lève une exception `StopIteration` à la fin de l'itération
- NB : l'objet retourné peut être l'objet itérable lui-même (et c'est même souvent le cas !)
  - à condition qu'il implémente `next`, bien sûr.

---

---

---

---

---

---

---

---

---

---

## Derrière les boucles

14

- Que se passe-t-il lorsqu'on exécute le code suivant ?

```
for x in l:
    [do something...]
```

- Grosso modo, ceci :

```
iter = l.__iter__()
try:
    while 1:
        x = iter.next()
        [do something...]
except StopIteration: pass
```

---

---

---

---

---

---

---

---

---

---

## Exemple

16

```
class myrange:
    def __init__(self, stop):
        self.stop = stop
        self.current = -1
    def __iter__(self):
        return self
    def next(self):
        self.current += 1
        if self.current >= self.stop:
            raise StopIteration
        return self.current

for i in myrange(10):
    print i
```

- On peut rendre ses propres objets itérables
- La liste sur laquelle on itère n'a pas besoin d'être entièrement construite avant l'itération

```
from time import time

start = time()
for i in range(10000000):
    if i == 1: break

print time()-start
```

- Si maintenant on remplace la *liste* range par l'*itérateur* xrange, le résultat est nettement meilleur !
- On peut même itérer sur des listes infinies !

---

---

---

---

---

---

---

---

---

---



---

---

---

---

---

---

---

---

---

---

- 1 Exceptions
- 2 Surcharge d'opérateurs
- 3 Itérateurs
- 4 Générateurs
- 5 Gestionnaires de contexte

- Écrire des itérateurs selon la méthode décrite ci-dessus est souple et puissant, mais un peu fastidieux
- Les générateurs sont (à la base) une manière plus simple de générer des itérateurs...
- ... mais peuvent aussi trouver d'autres usages
- Un générateur ressemble à une fonction python
  - seule différence (syntaxique !) : utiliser yield plutôt que return

---

---

---

---

---

---

---

---

---

---



---

---

---

---

---

---

---

---

---

---

```
def myrange_gen(stop):
    i = 0
    while i < stop:
        yield i
        i+=1

for i in myrange_gen(10):
    print i

# Equivalent:
it = myrange_gen(10)
try:
    while True:
        print it.next()
except StopIteration:
    pass
```

## Qu'est-ce que ce `yield` ?

21

- Comme un `return`, il va quitter la fonction en retournant une valeur, MAIS...
- L'état d'exécution actuel va être sauvé et lors du prochain `next`, l'exécution reprendra *là où elle s'est arrêtée*
- On peut avoir un `return` dans un générateur, mais *sans argument*. Son effet sera de lever une exception `StopIteration`
- Comme `return`, `yield` peut
  - apparaître en plusieurs endroits du générateur
  - retourner plusieurs valeurs
- NB : l'état de la fonction est sauvegardé jusqu'à destruction de l'itérateur correspondant. Attention donc à l'utilisation mémoire, notamment avec la récursivité !

---

---

---

---

---

---

---

---

## Exemple

23

```
def echo():
    out = ""
    while True:
        out = (yield out).upper()

e = echo()
e.next()
while True:
    i = raw_input(">>> ")
    print e.send(i)
```

---

---

---

---

---

---

---

---

## Yield est une expression !

22

- Depuis python 2.5, `yield` est une expression
  - peut donc renvoyer une valeur...
  - oui mais... quelle valeur ?
- Les générateurs ont une méthode `send` qui agit comme `next`, mais en "injectant" une valeur dans l'expression du `yield`
  - `next()` devient alors synonyme de `send(None)`
- À noter que les générateurs ont aussi...
  - une méthode `throw`, pour lever une exception au niveau du `yield`
  - une méthode `close`, pour forcer l'arrêt du générateur

---

---

---

---

---

---

---

---

## Coroutines

24

- Avec ces fonctionnalités supplémentaires, les générateurs de python se rapprochent beaucoup des *co-routines*
- Les co-routines sont une *généralisation des sous-routines* qui acceptent plusieurs points d'entrée, de sortie et de suspension de l'exécution
- Peuvent être utiles pour
  - pseudo-parallélisme (non préemptif !)
  - patterns producteur-consommateur
  - "pipes"
  - ...

- 1 Exceptions
- 2 Surcharge d'opérateurs
- 3 Itérateurs
- 4 Générateurs
- 5 Gestionnaires de contexte

---

---

---

---

---

---

---

---

---

---



---

---

---

---

---

---

---

---

---

---

## try...finally

- Le try de python accepte une clause finally, qui sera exécutée dans tous les cas

```
f = file('test.txt')
try:
    for l in f:
        print l,
finally:
    f.close()
```

---

---

---

---

---

---

---

---

---

---

## With statement

- Ce motif étant courant lorsqu'on utilise un fichier, il existe un "raccourci" en utilisant les gestionnaires de contexte :

```
# The with statement will only be final in ↔
2.6:
from __future__ import with_statement

with file('test.txt') as f:
    for l in f:
        print l,
```

- ... et le fichier sera fermé dans tous les cas !

---

---

---

---

---

---

---

---

---

---

## Comment ça marche ?

```
with EXPR [as VAR]:
    [CODE...]
```

est transformé en (à peu près)...

```
VAR = EXPR
VAR.__enter__()
try:
    [CODE...]
finally:
    var.__exit__()
```

## Un gestionnaire de contexte possède... 29

- une méthode `__enter__(...)` qui sera exécutée à l'entrée du bloc `with`
  - Si `__enter__` renvoie une valeur, elle sera affectée à la variable après le `as`
- une méthode `__exit__(type, value, traceback)` qui sera exécutée à la sortie du bloc `with`
  - Si une exception est levée dans le bloc, elle sera passée dans les paramètres de `__exit__`. Sinon, les trois paramètres seront `None`
  - Si `__exit__` retourne `True`, l'exception sera supprimée. Sinon, elle sera à nouveau levée après l'exécution de `__exit__`

---

---

---

---

---

---

---

---

---

---

## Exemple 30

- On aimerait faire un gestionnaire de contexte pour gérer les transactions d'une base de données :

```
db_connexion = DatabaseConnection()
with db_connexion as cursor:
    cursor.execute('insert into ...')
    cursor.execute('delete from ...')
    [...]
```

- Si tout s'est bien passé, la fin du bloc doit exécuter un `commit`, et sinon un `rollback`

---

---

---

---

---

---

---

---

---

---

## Exemple – suite 31

```
class DatabaseConnection:
    [...]
    def __enter__(self):
        cursor = self.cursor()
        return cursor
    def __exit__(self type, value, tb):
        if tb is None:
            self.commit()
        else:
            self.rollback()
        # return False
```

---

---

---

---

---

---

---

---

---

---

## contextlib 32

- Le module `contextlib` permet de simplifier l'écriture de *context managers* en les écrivant comme une fonction...
- ... qui contient un `yield`
  - tout ce qui précède le `yield` est exécuté avant le bloc
  - tout ce qui suit est exécuté après

---

---

---

---

---

---

---

---

---

---

```

from contextlib import contextmanager

@contextmanager
def db_transaction (connection):
    cursor = connection.cursor()
    try:
        yield cursor
    except:
        connection.rollback()
        raise
    else:
        connection.commit()

db = DatabaseConnection
with db_transaction(db) as cursor:
    [...]

```

---

---

---

---

---

---

---

---

---

---

- Exceptions
  - <http://docs.python.org/tut/node10.html>
- Surcharge d'opérateurs
  - <http://docs.python.org/ref/specialnames.html>
  - <http://docs.python.org/ref/attribute-access.html>

---

---

---

---

---

---

---

---

---

---

- Les *context managers* permettent d'assurer simplement qu'un certain code sera toujours exécuté
- On peut obtenir un contrôle plus fin avec un `try...catch... else...finally`, mais dans beaucoup de cas le `with` peut faire l'affaire, avec une meilleure factorisation du code
- On peut aussi les utiliser pour des "wrappers" génériques
- Peut être utile pour
  - Les fichiers
  - Les bases de données
  - Les verrous (parallélisme...)
  - ...

---

---

---

---

---

---

---

---

---

---

- Itérateurs et générateurs
  - Tarek Ziadé, "Programmation Python", Eyrolles, 2006
  - <http://heather.cs.ucdavis.edu/~matloff/Python/PyIterGen.pdf>
  - <http://www.magellan.fpms.ac.be/python/quoi-de-neuf-dans-python-2.5/pep-342-nouvelles-fonctionnalites-des-generateurs/view>
- Gestionnaires de contexte
  - <http://docs.python.org/whatsnew/pep-343.html>
  - <http://effbot.org/zone/python-with-statement.htm>
  - <http://www.magellan.fpms.ac.be/python/quoi-de-neuf-dans-python-2.5/pep-343-1-instruction-with>

---

---

---

---

---

---

---

---

---

---