Python – Cours 2

Jean-Yves Thibon

Université Paris-Est Marne-la-Vallée

ESIPE/IMAC 3

Au menu :

- Les exceptions
- Commentaires et docstrings
- L'itération en python
- Fichiers et répertoires
- Expressions régulières
- Syntaxe objet par l'exemple

Exceptions I

Mots clés : `try - except - raise -finally`

```
def f():
    try:
        return 1/0
    except:
        return 2+[2]
    finally:
        return 42

>>> f()
42
```

Exceptions II

Plus sophistiqué :

```
try:
    buffer += socket.read(4096)
    # ...
except OSError,e:
    if e.errno!=errno.EAGAIN: raise
except DiskFullException:
    print "Le disque est plein."
except:
    save_the_work()
    do_something_with(sys.exc_info)
```

Exceptions III

Différents niveaux de contrôle :

```
try:
    raise EnvironmentError(666,
                           'External program crashed',
                           'hello.o')
except EnvironmentError, e:
    print e
    print e.args
    print e.errno, e.strerror, e.filename


>>>
[Errno 666] External program crashed: 'hello.o'
(666, 'External program crashed') i
666 External program crashed hello.o
>>>
```

Exceptions IV

La clause `finally` permet de terminer proprement (en refermant fichiers, sockets, etc. par exemple).

```
try:
    du_code()
    raise Exception("bonjour")
finally:
    # sera exécuté dans tous les cas
    # (et avant les éventuels gestionnaires d'exception)
    save_work_in_progress()
    print "au revoir"
```

Commentaires et docstrings

```
# avec des dièses
"ou bien des chaînes littérales"

""" ou encore, sur plusieurs lignes,
avec des triple-quotes """

def racine_carree(x):
"""Cette fonction calcule la racine carrée
           du nombre fourni comme paramètre."""
   return x**0.5

help(racine_carree)
help.__doc__
```

L'itération en Python I

Syntaxe : `for i in` < *iterable* > : < *faireqqchose* >

Les listes, tuples, chaînes et dictionnaires sont itérables.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> [x*x for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [x*x for x in range(10) if x%2]
[1, 9, 25, 49, 81]
>>> map(lambda x:x*x, range(10))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> filter(lambda x:x%2,[1, 4, 2, 3, 7, 8, 13, 42])
[1, 3, 7, 13]
>>> h={"un":1, "deux":2, "trois":3}
>>> for key in h: print key,
un trois deux
```

L'itération en Python II

Les objets itérables possèdent une méthode `__iter__()` qui renvoie un *itérateur*. Un itérateur possède une méthode `next()` qui renvoie l'élément suivant.

Les *générateurs* permettent de créer des itérateurs sur des objets qui n'ont pas besoin d'être contruits à l'avance. La syntaxe est identique à celle des fonctions, avec `yield` au lieu de `return`.

```
def fib():
  a, b = 0, 1
    while 1:
      yield b; a, b = b, a+b

>>> F=fib()
>>> [F.next() for i in range(10)]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
>>> [F.next() for i in range(10)]
[89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765]
```

L'itération en Python III

```
>>> for (i, f) in zip(range(10), fib()):
              print "F[%d]=%d" % (i, f)
...
F[0]=1
F[1]=1
F[2]=2
F[3]=3
F[4]=5
F[5]=8
F[6]=13
F[7]=21
F[8]=34
F[9]=55
>>> zip("abracadabra", range(6))
[('a', 0), ('b', 1), ('r', 2), ('a', 3), ('c', 4), ('a', 5)]
```

La bibliothèque (modules)

À explorer : (`http://doc.python.org/lib/`)

```
sys : sys.argv, sys.stdin... principalement
os : fonctions "bas niveau" (appels système)
re : expressions régulières
math : flottants, complexes
pickle, cPickle : (dé)sérialisation
shelve : dictionnaires persistants
urllib, urllib2 : http
HTMLParser : comme son nom l'indique ...
xmlrpclib : "Remote Procedure Call" (web services)
cgi : Common Gateway Interface
sqlite3 : Bases de données SQLite
socket : réseau bas niveau
```

Il y a de nombreux paquetages

`http ://pypi.python.org/pypi` (environ 25000) pour à peu près n'importe quoi ...

Manipulation de fichiers

```
>>> f=open(filename,mode='r') # comme fopen(3)
>>> f.read() # lit tout le fichier
>>> f.read(1024) # lit 1024 octets
>>> f.readline() # lit une ligne (avec le \n)
>>> f.write('toto')
>>> f.close()
>>> import sys
# On a sys.stdin, sys.stdout, sys.stderr
>>> for truc in sys.stdin: # itère sur les lignes
>>> sys.stdout = open('/tmp/log.txt','a')
```

Manipulation de répertoires

```
>>> import os
>>> print os.listdir(os.environ['HOME'])
>>> for name in os.listdir(os.environ['HOME']):
...     path=os.path.join(os.environ['HOME'],name)
...     print "%s: %d octets"%(name,os.stat(path).st_size)
>>> for dirpath, dirnames, filenames in os.walk('/'):
...     for dirname in dirnames:
...         print os.path.join(dirpath,dirname)
...     for filename in filenames:
...         print os.path.join(dirpath,filename)
>>> os.open('/tmp/toto',os.O_RDWR|os.O_CREAT,0644)
3
>>> os.write(3, 'coin\n')
```

Noter en particulier `os.walk` : parcours récursif d'un répertoire.

`os.path` est un module distinct.

Expressions régulières

Module `re`

`re.match`(regex, chaîne) et `re.search`(regex, chaîne) retournent (en cas de match) un objet de type `Match`.

`re.match` ne cherche qu'au début de la chaîne.

`re.search` cherche à n'importe quelle position.

`re.findall` retourne une liste de toutes les occurences trouvées.

Recherches multiples : utiliser `re.compile`. Pratique : les raw strings

```
>>> "abc\n\1def"
'abc\n\x01def'
>>> r"abc\n\1def"
'abc\\n\\1def'
>>> re.search(r't(.)t\1', 'le tutu')
<_sre.SRE_Match object at 0xb7de5de0>
>>> _.groups()
('u',)
```

Un test commode I

Les expressions régulières sont puissantes, mais d'un maniement délicat. Le petit test suivant permet de visualiser leur effet.

```
import re
def re_show(pat, s):
  print re.compile(pat, re.M).sub("{\g<0>}",
                                  s.rstrip()),'\n'

s="le cheval de mon cousin ne mange du foin que le dimanche"
p = r'(\b\w*in\b)'
>>> re_show(p,s)
le cheval de mon {cousin} ne mange du {foin} que le dimanche
>>> t = """ceci \\ est un \\backslash
mais cela \\n n'est pas un \n retour chariot"""

>>> t
"ceci \\ est un \\backslash\nmais cela \\n n'est pas un \n
                                                retour chariot"
```

Un test commode II

```
>>> print t
ceci \ est un \backslash
mais cela \n n'est pas un
 retour chariot

>>> re_show('\n|\\\\','t)
ceci {\} est un {\}backslash{
}mais cela {\}n n'est pas un {
} retour chariot
>>> re_show(r'\n|\\',t) # autre version
```

Métacaractères

Ce sont les suivants

```
. ^ $ * + ? { } [ ] \ | ( ) \A \Z \b \B
```

"`[`" et "`]`" spécifient une classe de caractères : `[abcd]` ou `[a-d]`. Les métacaractères ne sont pas actifs dans une classe : `[a-c(?$]` contient les caractères `a,b,c,(,?,$`. Un `^` au début définit le complémentaire : les caractères non alphanumériques sont `[^a-zA-Z0-9]`. Le `\` est le caractère d'échappement.

```
>>> re_show(r'[\n\\]',t)
ceci {\} est un {\}backslash{
}mais cela {\}n n'est pas un {
} retour chariot
```

Classes prédéfinies

```
\d  equivalent à [0-9].
\D  equivalent à [^0-9].
\s  equivalent à [ \t\n\r\f\v].
\S  equivalent à [^ \t\n\r\f\v].
\w  equivalent à [a-zA-Z0-9_].
\W  equivalent à [^a-zA-Z0-9_].
```

Ces séquences peuvent être incluses dans une classe. Par exemple, `[\s\\]` contient tous les blancs et le backslash

```
>>> re_show(r'[\s\\]',t)
ceci{ }{\}{ }est{ }un{ }{\}backslash{
}mais{ }cela{ }{\}n{ }n'est{ }pas{ }un{ }{
}{ }retour{ }chariot
```

Métacaractères (suite) I

Le métacaractère "." matche tout sauf le retour chariot.
Il existe un mode `re.DOTALL` où il matche tout caractère.
Le " ?" matche 0 ou 1 fois.
L'étoile < *regexp* >`*` signifie 0 ou plusieurs fois < *regexp* >.
Le plus < *regexp* >`+` signifie 1 ou plusieurs fois < *regexp* >.
Les accolades < *regexp* >`{m,n}` signifient au moins *m* fois et au plus *n* fois (< *regexp* >`{n}` pour exactement *n* fois).

```
>>> re_show(r'o*', 'boogie-woogie')
{}b{oo}g{}i{}e{}-{}w{oo}g{}i{}e{}
>>> re_show(r'o+', 'boogie-woogie')
b{oo}gie-w{oo}gie
>>> re_show(r'o{2,4}',
            'oh boogie-woogie wooof woooof wooooof')
oh b{oo}gie-w{oo}gie w{ooo}f w{oooo}f w{oooo}of
```

Métacaractères (suite) II

La barre verticale $< R1 > | < R2 >$ matche $< R1 >$ ou $< R2 >$. On peut mettre une expression entre parenthèses pour lui appliquer un opérateur comme $*$ ou $+$

```
>>> s = 'baaababaaababbaababbbabaabaabbbaaaababa'
>>> re_show(r'(a|ba)+',s)
{baaababaaaba}b{baaba}bb{babaabaa}bb{baaababa}
```

Les parenthèses servent aussi à indiquer des *groupes* (voir plus loin). `^` et `$` marquent respectivement le début et la fin d'une ligne.

```
>>> re_show('^m|sh$|^\s',t)
ceci \ est un \backsla{sh}
{m}ais cela \n n'est pas un
{ }retour chariot
```

Le type `RegexObject` I

Les expressions régulières, jusqu'ici données sous forme de chaînes, peuvent être *compilées*. Le résultat est une instance de la classe `RegexObject`.

Le module `re` exporte des fonctions ayant les mêmes noms que les méthodes des Regex Objects :

```
>>> s='baaababaaababbaababbbabaabaabbbaaaababa'
>>> r = re.compile('((a|ba)+)')
>>> r.findall(s)
[('baaababaaaba', 'ba'), ('baaba', 'ba'),
       ('babaabaa', 'a'), ('baaaababa', 'ba')]
```

Les méthodes `r.match` (matche au début de la chaîne) et `r.search` (matche n'importe où dans la chaîne) retournent un `MatchObject` ou `None`.

Le type `RegexObject` II

La méthode `r.findall` trouve toutes les occurences (non recouvrantes) de l'expression et retourne une liste. `r.finditer` retourne un itérateur.
`r.split` casse la chaîne selon les occurences de `r`.
`r.sub` et `r.subn` remplacent les occurences de `r` par une chaîne fixée, ou leur appliquent une fonction.

```
>>> re.split(r'\s|\\',t)
['ceci', '', '', 'est', 'un', '', 'backslash', 'mais',
   'cela', '', 'n', "n'est", 'pas', 'un', '', '',
                                    'retour', 'chariot']
>>> re.sub(r'\s+|\\', ' ',t)
"ceci   est un  backslash mais cela  n n'est pas
                                     un retour chariot"
```

Le type `MatchObject` I

Les méthodes les plus importantes d'un `MatchObject m` sont :

- `m.group()` ou `m.group(0)` retourne la chaîne matchée
- `m.start()` retourne la position de départ du match
- `m.end()` retourne la position de la fin du match
- `m.span()` retourne un couple `(start, end)`
- `m.groups()` : une expression peut comporter des *groupes*, délimités par des parenthèses ...

```
>>> s='baaababaaababbaababbbabaabaabbbaaaababa'
>>> p = re.compile('((a|ba)+)')
>>> m = p.match(s)
>>> m.group()
'baaababaaaba'
>>> m.span()
(0, 12)
>>> p.sub(lambda x:'.'*len(x.group(0)),s)
'............b.....bb........bb.........'

>>> p = re.compile('(a(b)c)d(e)')
```

Le type `MatchObject` II

```
>>> m = p.match('abcde')
>>> m.groups()
('abc', 'b', 'e')
>>> m.group(0)
'abcde'
>>> m.group(1)
'abc'
>>> m.group(2)
'b'
>>> m.group(3)
'e'
```

Le type `MatchObject` III

Les groupes peuvent être nommés en utilisant la syntaxe `(?P<name>...)`

```
s="le cheval de mon cousin ne mange du foin que le dimanche"
p = re.compile(r'(\b\w*in\b).*(\bd\w*e\b)')
m = p.search(s)
>>> m.groups()
('cousin', 'dimanche')
p = re.compile(r'(?P<qui>\b\w*in\b).*(?P<quand>\bd\w*e\b)')
m = p.search(s)
>>> m.group('quand')
'dimanche'
```

Application d'une fonction aux groupes I

Syntaxe : `r.sub(f, s, count=0)`
où `f` accepte comme argument un MatchObject et retourne la chaîne à utiliser en remplacement du match.

```
import re
s='''<html>
      <body>
      <H1>Mon beau cours de python</H1>
      <H2>Blabla</H2> et ri et
ra  patati et patata
</body>
</html>'''

def capitalize(m):
    return ' '.join([w.capitalize()
                     for w in m.group(1).split()])
p=re.compile(r'<h1>(.*?)</h1>', re.I|re.M)

print re.findall(p,s)
print p.sub(capitalize,s)
```

Application d'une fonction aux groupes II

```
>>>
['Mon beau cours de python']
<html>
      <body>
      Mon Beau Cours De Python
      <H2>Blabla</H2> et ri et
ra  patati et patata
</body>
</html>
>>>
```

Glouton, ou pas ?

Les opérateurs non-gloutons `*?`, `+?`, `??`, or `{m,n}?`, matchent aussi peu de texte que possible

```
>>> s = '<html><head><title>Title</title>'
>>> print re.match('<.*>', s).group()
<html><head><title>Title</title>
>>> print re.match('<.*?>', s).group()
<html>
```

Options de compilation

- `I` ou `IGNORECASE` : insensible à la casse
- `L` ou `LOCALE` : rend `\w, \W, \b, \B` dépendants de ce que la `locale` courante considère comme alphanumérique
- `M` ou `MULTILINE` : `^, $` matchent le début et la fin de chaque ligne, en plus du début et de la fin de la chaîne
- `S` ou `DOTALL` : le point . matche un retour chariot
- `U` ou `UNICODE` : rend `\w, \W, \b, \B` dépendants de ce qu'Unicode considère comme alphanumérique
- `X` ou `VERBOSE` : les espaces blancs sont ignorés et `#` est interprété comme le début d'un commentaire

Mode verbeux

Les options de compilation sont aussi utilisables avec les chaînes (syntaxe `(?iLmsux)`). En mode verbeux, les espaces sont ignorés et `#` signale un commentaire :

```
pat_url = re.compile(  r'''
                 (?x)( # verbose identify URLs within text
    (http|ftp|gopher) # make sure we find a resource type
                   :// # needs to be followed by ://
       (\w+[:.]?){2,} # at least two domain groups: (gouv.)(fr)
                  (/?| # just the domain name (maybe no /)
              [^ \n\r"]+ # or stuff then space, newline, tab, quote
                  [\w/]) # resource name ends in alphanumeric or /
    (?=[\s\.,>)'"\]]) # assert: followed by white or clause ending
                    ) # end of match group'''')
```

Pickle : mise en bocal

Pour sauvegarder des données au format Python

```
>>> import pickle
>>> f=open('/tmp/toto','w') # sauvegarde
>>> pickle.dump(47,f)
>>> pickle.dump(sys.argv,f)
>>> pickle.dump({'canard':'coin',
                 'chat':'miaou'},f)
>>> f=open('/tmp/toto')     # rechargement
>>> try:
...     while 1: print pickle.load(f)
... except EOFError: pass
```

On ne peut pas tout sauver (sockets, modules...).

Shelve I

Table de hachage stockée sur disque
Restrictions : les mêmes que pour pickle
De plus, les clés doivent être des chaînes

```
>>> import shelve
>>> d = shelve.open('titi')
>>> d['ga'] = (1,2)
>>> d['bu']= 'abracadabra'
>>> d['zo'] = 3.1415926535
>>> d['meu'] = {'a':'b'}
>>> d.close()
```

```
>>> d = shelve.open('titi')
>>> d.keys()
['ga', 'bu', 'zo', 'meu']
>>> d['bu']=range(3)
```

Shelve II

```
>>> d['bu'].append(8)
>>> d['bu']
[0, 1, 2]
```

Un piège : `writeback=False` par défaut.

Etude de cas : métadonnées mp3 I

Cet exemple illustre l'utilisation des classes pour structurer un programme analysant un fichier binaire.

Il est tiré du livre de Mark Pilgrim "Dive into Python" :

Construire une classe pour stocker les métadonnées d'un fichier mp3 (ancienne norme ID3v1.0, la nouvelle est accessible à http ://www.id3.org/id3v2-00).

Ici, le problème est trivial (la nouvelle norme serait plus complexe). L'intérêt de l'exemple est de fournir un modèle, et surtout d'illustrer quelques pythonismes.

Il s'agit simplement de lire les 128 derniers octets du fichier, et de les séparer en champs contenant les informations suivantes :

Etude de cas : métadonnées mp3 II

Field	Length	Offsets
Tag	3	0-2
Songname	30	3-32
Artist	30	33-62
Album	30	63-92
Year	4	93-96
Comment	30	97-126
Genre	1	127

La signification du tag *genre* est donnée dans le document cité.

Etude de cas : métadonnées mp3 III

Par exemple :

```
[jyt@kastelberg AUDIO]$ hexdump -C 18\ -\ L\'alcool.mp3 |tail
001dd560  00 00 00 54 41 47 4c 27  61 6c 63 6f 6f 6c 00 00  |...TAGL'alcool..|
001dd570  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
001dd580  00 00 00 00 4c 65 73 20  34 20 42 61 72 62 75 73  |....Les 4 Barbus|
001dd590  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
001dd5a0  00 00 4c 61 20 50 69 6e  63 65 20 c0 20 4c 69 6e  |..La Pince À Lin|
001dd5b0  67 65 20 31 00 00 00 00  00 00 00 00 00 00 00 00  |ge 1............|
001dd5c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
001dd5e0  00 12 ff                                          |..ÿ|
001dd5e3
```

Etude de cas : métadonnées mp3 IV

Le script devra avoir pour effet d'afficher les métadonnées de tous les fichiers mp3 du répertoire passé en argument (par défaut, le répertoire courant)

```
[jyt@kastelberg AUDIO]$ python fileinfo.py
album=La Pince À Linge 1
comment=
name=./18 - L'alcool.mp3
title=L'alcool
artist=Les 4 Barbus
year=
genre=255
```

... et aussi être utilisable comme module.

Etude de cas : métadonnées mp3 V

Analyse du problème

- La classe `MP3FileInfo` devra dériver du type `dict`, et avoir une clef pour chaque TAG
- C'est même plus clair d'avoir un type intermédiaire `FileInfo` qui se contente d'affecter un attribut `name` à l'appel de `__init__`.
- `MP3FileInfo` devra surcharger la méthode spéciale `__setitem__` des dictionnaires
- Elle pourra le faire au moyen d'une méthode privée `__parse` qui se chargera d'analyser le fichier
- Pour la lisibilité, on définira un dictionnaire `tagDataMap` ayant pour clefs les tags et pour valeurs des triplets `(debut,fin, filtre)`.

Etude de cas : métadonnées mp3 VI

- Le filtre dit quoi faire de la chaine récupérée entre début et fin
- il y a beaucoup d'octets nuls '00'. On appliquera une fonction `stripnulls`,
- sauf pour le tag 'genre' qui doit renvoyer un entier entre 0 et 255 (`ord`).

Etude de cas : métadonnées mp3 VII

A ce stade, on peut donc coder ceci :

```
class FileInfo(dict):
    "store file metadata"
    def __init__(self, filename=None):
        self["name"] = filename

def stripnulls(data):
    "strip whitespace and nulls"
    return data.replace("\00", "").strip()

tagDataMap = {"title"   : (  3,  33, stripnulls),
              "artist"  : ( 33,  63, stripnulls),
              "album"   : ( 63,  93, stripnulls),
              "year"    : ( 93,  97, stripnulls),
              "comment" : ( 97, 126, stripnulls),
              "genre"   : (127, 128, ord)}
```

Etude de cas : métadonnées mp3 VIII

Le dictionnaire tagDataMap sera inclus dans le code de `MP3FileInfo`. Toute la logique est dans la méthode (privée) `__parse`

```
class MP3FileInfo(FileInfo):
    tagDataMap = ...
            ....

    def __parse(self, filename):
            ....
# analyse si item=fichier et sinon affecte normalement
    def __setitem__(self, key, item):
        if key == "name" and item: self.__parse(item)
        FileInfo.__setitem__(self, key, item)
```

Etude de cas : métadonnées mp3 IX

Finalement, le code de `__parse` :

```
def __parse(self, filename):
    "parse ID3v1.0 tags from MP3 file"
    self.clear()   # Les dictionnaires ont "clear"
    try:
        fsock = open(filename, "rb", 0) # b pour windows
        try:
            fsock.seek(-128, 2)  # recule de 128, 2=fin
            tagdata = fsock.read(128) # lit les 128 derniers
        finally: fsock.close()
        if tagdata[:3] == "TAG":
            for tag, (start, end, parseFunc) in \
                                    self.tagDataMap.items():
                self[tag] = parseFunc(tagdata[start:end])
    except IOError: pass
```

Etude de cas : métadonnées mp3 X

Pour utiliser cette classe et d'autres nommées `XYZFileInfo` (exercice : essayer de décrypter ce code, puis voir l'explication dans le livre)

```
def listDirectory(directory, fileExtList):
    "get list of file info objects for files of particul
    fileList = [os.path.normcase(f)
                for f in os.listdir(directory)]
    fileList = [os.path.join(directory, f)
                for f in fileList
                if os.path.splitext(f)[1] in fileExtList
    def getFileInfoClass(filename,
                         module \
                          =sys.modules[FileInfo.__module
        "get file info class from filename extension"
        subclass = "%sFileInfo" % os.path.splitext(
                                    filename)[1].upper
        return hasattr(module, subclass) \
               and getattr(module, subclass) \
               or FileInfo
    return [getFileInfoClass(f)(f) for f in fileList]
```

Etude de cas : métadonnées mp3 XI

Finalement, pour utiliser le programme comme un script :

```
if __name__ == "__main__":
    for info in listDirectory(".", [".mp3"]): (1)
        print "\n".join(["%s=%s" % (k, v)
                         for k, v in info.items()])
        print
```

Le code complet, commenté en détail, se trouve au chapitre 5 de Dive into Python.

Commentaires : retour sur les classes I

Mécanisme de définition des classes simple, analogue à celui des fonctions. Une classe peut hériter d'une ou plusieurs autres, définies par l'utilisateur, ou prédéfinies ("built-in")
La méthode spéciale `__init__` est appelée immédiatement après la création de l'instance (joue le rôle d'un constructeur pour les classes "old style" en python 2). Elle peut prendre un nombre arbitraire d'arguments.
Les méthodes spéciales ont des noms qui commencent et se terminent par deux soulignés. Elles permettent la surcharge des opérateurs.
Le premier argument de toute méthode d'une classe est toujours une référence à l'instance courante de la classe (conventionnellement appelée `self`).

Commentaires : retour sur les classes II

Pour un dictionnaire, `d.__setitem__(k,v)` équivaut à `d[k]=v`. Avec la surcharge :

```
>>> import fileinfo
>>> mp3file = fileinfo.MP3FileInfo()
>>> mp3file
{'name': None}
>>> mp3file["name"]="18 - L'alcool.mp3"
>>> mp3file
{'album': 'La Pince \xc0 Linge 1', 'comment': '',
'name': "18 - L'alcool.mp3", 'title': "L'alcool",
'artist': 'Les 4 Barbus', 'year': '', 'genre': 255}
```

Commentaires : retour sur les classes III

Accès aux attributs :

```
>>> fileinfo.MP3FileInfo.tagDataMap
{'title': (3, 33, <function stripnulls at 0260C8D4>),
'genre': (127, 128, <built-in function ord>),
'artist': (33, 63, <function stripnulls at 0260C8D4>),
'year': (93, 97, <function stripnulls at 0260C8D4>),
'comment': (97, 126, <function stripnulls at 0260C8D4>),
'album': (63, 93, <function stripnulls at 0260C8D4>)}
>>> m = fileinfo.MP3FileInfo()
>>> m.tagDataMap
{'title': (3, 33, <function stripnulls at 0260C8D4>),
'genre': (127, 128, <built-in function ord>),
'artist': (33, 63, <function stripnulls at 0260C8D4>),
'year': (93, 97, <function stripnulls at 0260C8D4>),
'comment': (97, 126, <function stripnulls at 0260C8D4>),
'album': (63, 93, <function stripnulls at 0260C8D4>)}
```

Commentaires : retour sur les classes IV

`MP3FileInfo` est la classe elle-même. `tagDataMap` est un attribut de la classe. Il est accessible à toutes ses instances.

```
>>> class counter:
...     count = 0
...     def __init__(self):
...         self.__class__.count += 1
...
>>> counter
<class __main__.counter at 010EAECC>
>>> counter.count
0
>>> c = counter()
>>> c.count
1
>>> counter.count
1
>>> d = counter()
>>> d.count
2
>>> c.count
```

Commentaires : retour sur les classes V

```
2
>>> counter.count
2
```

`count` est un attribut de la classe `counter`.
`__class__` est un attribut prédéfini de toute instance d'une classe. C'est une référence à la classe dont `self` est une instance.
`count` est accessible par référence directe à la classe, avant même la création d'une instance. Chaque instanciation incrémente `count`, ce qui affecte la classe elle-même.
Sans l'attribut `__class__`, on aurait

```
class compteur:
    compte = 0
    def __init__(self):
        self.compte +=1

>>> a=compteur()
>>> a.compte
1
>>> b=compteur()
```

Commentaires : retour sur les classes VI

```
>>> b.compte
1
>>> compteur.compte
0
>>>
```

Commentaires : retour sur les classes VII

Fonctions privées (inacessibles en dehors de leur module)
Méthodes privées (inacessibles en dehors de leur classe)
Attributs privés (inacessibles en dehors de leur classe)
Leurs noms commencent (mais ne finissent pas !) par deux soulignés : `__parse`.

```
>>> m = fileinfo.MP3FileInfo()
>>> m.__parse("18 - L'alcool.mp3")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'MP3FileInfo' object has no attribute '__parse
>>> m._MP3FileInfo__parse("18 - L'alcool.mp3")
>>> # On peut, mais il ne faut pas !
```