Le Langage Python

Yves Legrandgérard

ylg@pps.jussieu.fr

CNRS – Université Paris 7

Un bref historique

- Le langage Python a été conçu par *Guido Van Rossum* en 1990 au CWI a Amsterdam
- Connaît une large diffusion à partir de 1994
- La dernière version : 2.2.1 du 10/04/2002
- Disponible sur de nombreuses plateformes : UNIXs, DOS,

Quelques caractéristiques (1)

- Langage **interprété** (plus précisément, chaque programme Python est compilé en « *byte-code* », cette compilation étant effectuée lorsqu'un module est importé ou exécuté pour la 1ère fois et ceci de manière automatique)
- Langage **non typé**
- Langage de « haut niveau »
- Gestion automatisée de la mémoire

Quelques caractéristiques (2)

- Langage **orienté objet**
- Programmation **modulaire** : classes, modules, exceptions
- Intégration C et C++
- Chargement dynamique des modules C
- Rechargement dynamique des modules Python
- Comble le fossé entre les langages de script « traditionnels » et le langage C

Un premier exemple

```
# !/usr/bin/env python
import sys
file = open(sys.argv[1],
   'r')
while 1:
   line = file.readline()
   if not line:
      break
   if line[0] == '#':
      print line
```

L'exécution de la commande :

$ ex.py source.c > source.pp

extrait les lignes du fichier **source.c** qui commencent par le caractère # et les redirige vers le fichier **source.pp**.

Quelques éléments de syntaxe

```
# !/usr/bin/env python      ← Unix
# Ceci est un commentaire
x = 1 # Autre commentaire
y = 2; print z # Plusieurs
  instructions à la suite
if x < 1 and \ # Continuation de
  ligne
   y != 3:
    print 'Ok'
```

L'indentation (1)

```
# !/usr/bin/env python
import sys
file = open(sys.argv[1],
   'r')
while 1:
   line = file.readline()
   if not line:
      break
   if line[0] == '#':
      print line
```

- Il n'y a pas de début et de fin de bloc comme en C ou en Perl par exemple.
- C'est l'**indentation** qui permet à Python de repérer les débuts et fins de blocs.
- **Avantage** : cela impose un style de programmation permettant une (re)lecture plus aisée

L'indentation (2)

Langage C

```
if (cond1())
  if
    (cond2())
    {
     action();
     finish();
  }
else
    error();
```

Python

```
if cond1():
  if
    cond2():
     action()
     finish()
else:
    error()
```

Résout de manière élégante le problème du « *else pendant* », toujours source d'erreurs difficiles à détecter !

Comment exécuter Python

Script

```
# !/usr/bin/env
  python
# Le source hw.py
print 'Hello World!'
```

```
$ hw.py
Hello World!
```

Mode interactif

```
$ python
>>> print 'Hello
  World!'
Hello World!
>>>
```

Les instructions (1)

affectation

```
$ python
>>> x = 2
>>> y = 'chaine'
>>> z = 2.0e6
>>> print x, y, z
2 chaine 2000000.0
>>>
```

```
>>> a = b = c = 1
>>> print a, b, c
1 1 1
>>> x = 2
>>> x += 1 # *, ^ , |,...
>>> print x
3
>>> x, y = 1, 2 # tuple
```

Les instructions (2)
(structures de contrôle)

if, elif, else

```
# !/usr/bin/env
   python
if x = = 1:
  print 'valeur de x:
   1'
elif x = = 2:
  print 'valeur de x:
   2'
else:
```

- Comme en C, ≠ **0** et **0** sont respectivement **vrai** et **faux**
- Ainsi, une boucle

```
# !/usr/bin/env
   python
while 1:
   print 'Hello'
```

Les instructions (3)
(structures de contrôle)

not, and, or

```
# !/usr/bin/env python
if not x == 5 and (x < 10 or x >= 20):
    print x
if 2 <= y < 10: # 2 <= y and y < 10
    print y
if x == y == z == 0: # x == 0 and y ==
  0 and z == 0
    print 'erreur...'
```

Les instructions (4)
(structures de contrôle)

while

```
# !/usr/bin/env python
i = 0
while 1:
    if i == 10:
        break
    else:
        print i
        i = i + 1
```

for

```
$ python
>>> for i in range(2):
...     print i
0
1
>>> for i in
  range(1000):
...     pass # ne fait
  rien
>>>
```

Les nombres

Constantes	Opérations
1024, 3.14, 0177 0xFFFE, 0xabcd, 1000L 10e12, 6.62e-34	x + y, x – y x * y, x / y, x % y -x, + x abs(x), int(x), long(x) float(x), divmod(x), pow(x, y) x | y, x ^ y, x & y, ~ x x << n, x >> n

Les chaînes de caractères (1)

```
# !/usr/bin/env python
x, y  = 2.1, 'On obtient:'
print 'x = %.2f ' % x
z = '%s %.2f ' % (y, x)
print z + '\nFin du calcul'
```

```
>>> str.py
x = 2.10
On obtient: 2.10
Fin du calcul
```

- Les chaînes sont des listes particulières
- Les opérateurs **+**, *****, **[]**,... sont ainsi applicables aux chaînes
- Chaînes formatées (format presque identique à celui de *printf*

Les chaînes de caractères (2)

```
>>> x = "C'est un titre"
>>> y = '-' * (len(x) + 4)
>>> print '%s\n| %s |\n%s' % (y, x, y)
------------------
| C'est un titre |
------------------
>>> c = """  # chaîne de caractères sur
  plusieurs lignes
Un commentaire sur plusieurs
        lignes   avec des blancs      et des
  tabulations
"""
```

Les structures de données (1)

Listes

```
$ python
>>> L = ['item', 1, 5.0]
>>> L[2]
5.0
>>> 'item' in L
1
>>> for i in L:
...     print i,
item 1 5.0
```

```
>>> for c in 'Bonjour':
...     print c,
B o n j o u r # chaîne = liste
>>> L[0:2]
['item', 1]
>>> L[1:]
[1, 5.0]
>>> L[-1:]
[5.0]
```

Les structures de données (2)

Listes (suite)

```
>>> L = ['item', 1, 5.0]
>>> len(L)
3
>>> L[1:len(L)]
[1, 5.0]
>>> L[5]
IndexError:
        list index out of range
```

```
>>> L[-10:10]
['item', 1, 5.0] # pas d'erreur
>>> S = [3, 4]
>>> print L + S
['item', 1, 5.0, 3, 4]
>>> L.append(2)
['item', 1, 5.0, 2]
>>> del L[1]
>>> print L
['item', 5.0, 2]
```

Les structures de données (3)

Listes (suite et fin)

```
>>> L = [3]
>>> print L * 4
[3, 3, 3, 3]
>>> print L.count(3)
4
>>> L = [5, 3, 7, 0]
>>> L.sort()
>>> print L
[0, 3, 5, 7]
```

```
>>> L = [[1, 2], [3, 4, 5]]
>>> for i in L:
...     for j in i:
...         print j,
1 2 3 4 5
```

Encore d'autres méthodes applicables aux listes : *reverse()*, *index()*, *insert()*, *remove()*,...

Les structures de données (4)

Tuples

```
>>> T = (0, 2, 'item')
>>> for i in T:
...     print i,
0 2 'item'
>>> T[1] = 1
TypeError: objet doesn't support item assignment
>>> T = (3,) # tuple à 1 élt,
# attention (3) est l'entier 3 !
```

- Analogue aux listes à la différence (importante !) près que les **éléments** d'un tuple **ne sont pas modifiables**
- Les tuples fonctionnent donc comme les listes

```
# 1, 2, 3 est un tuple

x, y, z = 1, 2, 3
```

Les structures de données (5)

Dictionnaires

```
>>> D = {'Nom':
    'Dupont',
...          3:
    'premier'}
>>> print D[1],
    D['Nom']
premier Dupont
>>> for k in D:
...     print D[k],
premier Dupont
>>> D = {}  #
```

- Les dictionnaires sont des tableaux associatifs : {**clé**: *valeur*,...}
- Une clé peut être (presque) n'importe quel objet Python et pas nécessairement une chaîne de caractères

Les structures de données (6)

Dictionnaires (suite)

```
>>> D = {'Nom':
  'Dupont',
...          3:
  'premier'}
>>> print D.keys()
['Nom', 3]
>>> print D.values()
['Dupont', 'premier']
>>> print D.items()
[('Nom', 'Dupont'),
```

```
>>> if D.has_key(3):
...     print D[3]
premier
>>> D = {'Agent':
...        {'Nom':
  'Dupont',
...         'Prénom':
  'Jean',
...          'Age': 31
...        }
...      }
```

La gestion des erreurs (1)

```
# !/usr/bin/env python
t = [0, 'str', 5]
try:
    t[3] = 1
except IndexError:
    print 'indice
```

```
>>> ex.py
indice incorrect
```

```
# !/usr/bin/env python
t = [0, 'str', 5]
try:
    t[3] = 1
except IndexError,
    msg:
```

```
>>> ex.py
list assignment index
    out of
range
```

La gestion des erreurs (2)

```
# !/usr/bin/env python
t = [0, 'str', 5]
try:
    print a
    t[3] = 1
except IndexError,
   msg:
    print msg
except NameError,
   msg:
    print msg
except:
    print 'autre erreur'
else:
    print "pas d'erreur"
```

```
>>> ex.py
name 'a' is not defined
>>>
```

La gestion des erreurs (3)

```
# !/usr/bin/env python
FatalError = 'Erreur
   Fatale'
try:
   Code Python...
except (IndexError,
        NameError),
   msg:
   pass # ignore
   l'erreur
except:
   raise FatalError, \
        'autre erreur'
```

Si le *Code Python* du bloc **try** génère une exception autre que **IndexError** ou **NameError**, on aura :

```
>>> ex.py
Erreur Fatale: autre
   erreur
>>>
```

La gestion des erreurs (4)

```
# !/usr/bin/env python

fd = open('fichier', 'r')
try:
    parse(fd)
finally:
    fd.close()
```

- Le bloc **finally** est exécuté après l'exécution du bloc **try** qu'il y ait eu ou non erreur
- **finally** propage au niveau supérieur les erreurs éventuelles
- **finally** est incompatible avec **except** ou **else** dans le même bloc **try**

La gestion des erreurs (5)
(quelques exceptions prédéfinies)

- AttributeError
- EOFError
- IOError
- ImportError
- IndexError
- KeyError
- KeyBoardInterrupt
- MemoryError
- NameError
- OverflowError
- RuntimeError
- SyntaxError
- SystemError
- SystemExit
- TypeError
- ValueError
- ZeroDivisionError

Les fonctions (1)

```
>>> def echo(arg):
...     print arg
>>> echo(2.1)
2.1
>>> echo([1, 'item', -3.5])
[1, 'item', -3.5]
>>> echo({})
{}
>>>
```

Le nom d'une fonction est une variable, on peut en

```
>>> echo(echo)
<function echo at 0x009A37B8>
>>> x = echo
>>> x('item')
item
>>>
```

Les fonctions (2)

(l'instruction *return*, récursivité)

```
# !/usr/bin/env python
def exp(x, n):
    r = 1
    for i in range(n):
        r *= x
    return r
print exp(2, 32)
```

```
>>> exp.py
4294967296
```

- La valeur de retour d'une fonction peut être un objet Python quelconque
- **return** est optionnel
- Récursivité :

```
# !/usr/bin/env
python
def fact(n):
    if n == 0:
        return 1
    return n * fact(n -
1)
```

Les fonctions (3)
(l'instruction *global*)

```
# !/usr/bin/env python
MAX = 2
def gf():
    MAX = 4
gf()
print MAX
```

```
>>> gf.py
2
>>>
```

```
# !/usr/bin/env python
MAX = 2
def gf():
    global MAX
    MAX = 4
gf()
print MAX
```

```
>>> gf.py
4
>>>
```

Les fonctions (4)
(fonctions locales)

```
# !/usr/bin/env python
def t_sup(t):
    def sup(a, b):
        if a < b:
            return b
        return a
    r = t[0]
    for i in t[:-1]:
        r = sup(i, r)
    return r
```

```
print t_sup((3, 20, -1,
```

```
>>> lf.py
20
>>>
```

- A la différence du C, possibilité de définir des fonctions locales
- La fonction **sup** n'est pas connue au niveau global

Les fonctions (5)
(arguments facultatifs, nombre d'arguments variable)

```
>>> def af(a1, a2=2,
    a3=3):
...     print a2, a3
>>> af(1)
2 3
>>> af(1, 20, 30)
20 30
>>> af(1, 20)
20 3
>>>
```

```
>>> def av(a, *args):
...     for arg in args:
...         print arg,
>>> av(1)
>>> av(1, 'str', 2.0, 5)
str 2.0 5
>>>
```

Les fonctions (6)
(arguments facultatifs, nombre d'arguments variable)

```
>>> def afv(a1, a2=2,
    *args):
...     print a2,
...     for arg in args:
...         print arg,


>>> afv(1)
2
>>> afv(1, 20)
20
```

```
>>> afv(1, 20, 'str',
    -5.1)
20 str -5.1
>>>
```

- On peut mixer les deux formes (arguments facultatifs & optionnels)
- Les arguments facultatifs doivent être déclarés en premier

Les fonctions (7)

(arguments facultatifs, nombre d'arguments variable)

```
# !/usr/bin/env python
def ad(a, **dict): # dict est un
  dictionnaire
   print dict.items()
```

```
>>> dict.py
[('bg', 'vert'), ('gamma',
  0.98999999999999999),
 ('fg', 'noir')]
>>>
```

Les fonctions (8)
(arguments facultatifs, nombre d'arguments variable)

```
# !/usr/bin/env python
def afvd(a1, a2=3, *args, **dict):
    print a2,
    for arg in args:
        print arg,
    print dict.items()
afvd(1, 20, -5.1, 10, bg='vert', fg='noir')
```

Les arguments optionnels de type dictionnaire doivent être déclarés en dernier

Les fonctions (9)
(arguments facultatifs, nombre d'arguments variable)

```
>>> def fonct(x, y, z):
...     print x, y, z
>>> fonct(1, z=3,
    y=2)
1 2 3
>>> fonct(1, y=2, 3)
SyntaxError: non-
    keyword
arg after keyword arg
>>>
```

```
>>> fonct(1, y=2,
    x=1)
TypeError: fonct() got
multiple values for
    keyword
argument 'x'
>>>
```

Les fonctions (10)
(les fonctions anonymes : *lambda*)

```
f = lambda x, y: x * y
```

```
def f(x, y):
    return x * y
```

- Un exemple utile :

```
>>> def f(x, y):
...     Code Python
>>> f_x = lambda x:
    f(x, y= 0)
```

- L'exemple précédent peut aussi s'écrire :

```
>>> def f(x, y):
...     Code Python
>>> f_x = lambda x,
    y= 0: \
                f(x, y)
```

- Les fonctions anonymes s'utilisent également en conjonction avec **map**, **reduce** et **filter**

Les fonctions (11)
(programmation fonctionnelle : *map*)

```
# !/usr/bin/env python
def f(x):
    return x * x
s = (2, 3, 4, 5)
print map(f, s)
print map(None, s) #
```

```
>>> map.py
[4, 9, 16, 25]
[2, 3, 4, 5]
```

- Python offre les outils que l'on trouve dans tout langage fonctionnel tel **Scheme**
- Outre les fonctions ano- nymes, on dispose des fonctions **map**, **apply reduce** et **filter**

Les fonctions (12)
(programmation fonctionnelle : *map*)

```
# !/usr/bin/env python
def f(x, y):
    return x * y
s = (2, 3, 4, 5)
print map(None, s, map(lambda x: f(x, y= x), s))
print map(f, s, map(lambda x: f(x, y= x), s))
```

```
>>> map.py
[(2, 4), (3, 9), (4, 16), (5, 25)] # liste des
    arguments passés à f
[8, 27, 64, 125]
```

Les fonctions (13)

(programmation fonctionnelle: *apply*)

```
# !/usr/bin/env python
def echo(arg):
    print arg

def length(arg):
    len = 0
    for c in arg:
        len += 1
    print len
```

```
todo = ((echo, (3,)),
        (length,
   ('chaine',)))
for (f, a) in todo:
    apply(f, a) # f(a)
```

```
>>> ap.py
3
6
>>>
```

Les fonctions (14)
(programmation fonctionnelle : *reduce*)

```
# !/usr/bin/env python
def f(x, y): # fonction nécessairement à 2
   arguments
   return x * y
def fact(n): # une autre manière de coder n!
   return reduce(f, range(1, n + 1), 1) # valeur
   initiale nécess
```

```
>>> reduce.py
[1, 1, 2, 6, 24, 120]
```

Les fonctions (15)
(programmation fonctionnelle : *filter*)

```
# !/usr/bin/env python
MASK = 1 << 3
def f(x):
    return x & MASK
s = range(10)
print filter(None, s),
```

```
>>> filter.py
[1, 2, 3, 4, 5, 6, 7, 8, 9]
    [8, 9]
```

- ***filter****(fonct, seq)* construit une suite constituée des éléments de la suite *seq* pour lesquels la fonction *fonct* retourne *vrai* (≠ 0)
- Si *fonct* vaut *None*, seuls les éléments non nuls de la suite *seq* sont retournés

Les fonctions (16)
(les listes en « compréhension »)

- Les listes en compréhension fournissent un moyen plus explicite de construire des listes qu'avec *map*, *reduce*, *filter* et/ou *lambda*
- Une liste en compréhension est constituée d'une expression suivie d'une instruction *for*, elle-même éventuellement suivie d'une ou plusieurs instructions *for* ou *if*

Les fonctions (17)
(les listes en « compréhension »)

```
l1 = range(1, 6)
l2 = [x * x for x in l1]
l3 = [x * x for x in l1 if x != 1 and x != 5]
print l2, l3
l4 = [[x, x * x] for x in l3[:2]]
l5 = ['x=%d y=%d' % (x, y) for x in l1[:2] for y in
   l2[:2]]
```

[1, 4, 9, 16, 25] [4, 9, 16]

[[4, 16], [9, 81]] ['x=1 y=1', 'x=1 y=4', 'x=2 y=1', 'x=2 y=4']

Les modules (1)

- Les modules sont implémentés soit en Python (ce sont alors de simples fichiers texte contenant des instructions Python) soit en C
- Le chargement d'un module est indépendant de son implémentation
- Les modules disponibles dans la version 2.2.1 (il y en a des dizaines !)

Les modules (2)

(*import*)

```
# !/usr/bin/env python
# module essai.py
def prn():
    print 'essai.py'
```

```
>>> import essai
>>> essai.prn()
essai.py
>>>
```

La directive *import essai.py* :

2. recherche le fichier *essai.py* (à l'aide de la variable *sys.path*)
3. exécute le code
4. charge tous les objets définis dans un nouvel espace de noms (*essai*)
5. charge l'objet module essai dans l'espace de noms courant (*__main__*)

Les modules (3)

(*import* et *from*)

```
# !/usr/bin/env python
# module essai.py
x = 3
def prn():
    print 'essai.py'
```

```
>>> from essai import
   prn
>>> prn()
essai.py
>>>
```

- Comme précédemment pour les 3 premières étapes mais remplace l'étape 4 par une copie de la variable (ici *prn*)
- Pour charger

```
>>> from essai import
```

les objets d'un

```
>>> from essai import
*
```

Les modules (4)
(__main__)

```
# module essai.py
def prn():
    print 'essai.py'
prn()
```

- La fonction *prn* sera appelée dès que le module *essai.py* est importé pour la première fois
- Ce comportement n'est pas toujours souhaitable

```
# module essai.py
def prn():
    print 'essai.py'
if __name__ ==
    '__main__':
    prn()
```

- *prn* ne sera appelée que si

```
$ python essai.py
```

ou aussi (sous Unix)

```
$ essai.py
```

Les modules (5)

(*reload*)

```
# module essai.py
def prn():
    print 'essai.py'
```

```
>>> import essai
>>> essai.prn()
essai.py
```

On modifie maintenant le module essai.py...

```
# module essai.py
def prn():
    print 'Le module
```

```
>>> essai.prn()
essai.py
>>> reload(essai)
<module 'essai' from
    'essai.py'>
>>> essai.prn()
Le module essai.py
```

Les modules (6)

(*dir*, *__dict__*)

```
def prn(): # module
   essai.py
```

```
>>> print dir(essai)
['__builtins__',
   '__name__',
 '__file__', 'prn', '__doc__']
>>>  print essai.__file__
/users/ylg/essai.py
>>> print essai.__doc__
None
```

```
def prn(): # module
   essai.py
    print 'essai.py'
__doc__ = """\
```

```
>>> print
   essai.__doc__
```

essai.__doc__

↕

essai.__dict__['__doc__']

Les classes (1)
(généralités)

```
class Simple: # classe
    def show(self, arg): # méthode
        print arg
i = Simple() # i instance de la
                # classe Simple
```

```
>>> cl.py
essai
```

```
                # méthode
    show
```

- Les classes, comme les fonctions, sont définies au sein d'un module
- Outil de base pour la programmation orientée objet en Python
- Héritage et héritage multiple
- Analogue aux classes de C++, mais avec des

Les classes (2)
(généralités)

```
# !/usr/bin/env python
class Simple:
    def show(self, arg):
        print arg
x = Simple() # x instance
y = Simple() # y instance
x.label = 'x' # nouvel attribut
y.label = 'y' # nouvel attribut
```

- Les instances *x* et *y* ont un nouvel attribut *label* qui n'est pas un attribut de la classe *Simple*, chacune ayant sa propre copie locale
- On ne procède pas habituellement de cette manière : problème de cohérence

Les classes (3)
(généralités)

```
# !/usr/bin/env python
class Simple:
    def show(self, arg):
        print arg
    def set_label(self, label):
        self.label = label
    def get_label(self):

    self.show(self.label)
i = Simple()
i.set_label('x') # label est crée
```

- Ici l'attribut *label* est créé au sein de la classe *Simple*
- Quand une instance d'une classe appelle une métho-de, l'instance elle-même est passée (**de manière implicite**) comme premier argument de la méthode
- Cet argument a pour nom *self* par

Les classes (4)
(l'héritage)

```
class Simple:
    def show(self, arg):
        print arg
class
    SimpleFille(Simple):
    def set_label(self,
    label):
        self.label = label
    def get_label(self):

    self.show(self.label)
i = SimpleFille()
i.show('essai')
```

- La classe *SimpleFille* hérite de toutes les mé-thodes et attributs de la classe *Simple* sauf si elle les redéfinit
- Toutes les méthodes et attributs sont **virtuels** au sens de C++. Leurs réfé-rences ne sont examinées qu'à l'exécution (*run time*)

Les classes (5)
(l'héritage)

```
class Simple:
    def show(self, arg):
        print 'Simple', arg
class
    SimpleFille(Simple):
    def set_label(self,
    label):
        self.label = label
    def get_label(self):

    self.show(self.label)
    def show(self, arg):
        print
```

```
i = Simple()
i.show('i')
j= SimpleFille()
j.show('j')
```

```
>>> cl.py
Simple i
SimpleFille j
>>>
```

Les classes (6)
(l'héritage multiple)

```
class C1:
    def m(self):
        print 'C1.m'
class C2:
    def m(self):
        print 'C2.m'
class A(C1, C2):
    pass
class B(C2, C1):
    pass
```

```
a, b = A(), B()
a.m() # il y a conflit de
    noms,
```

```
>>> cl.py
C1.m
C2.m
```

Pour résoudre le conflit :

```
class B(C2, C1):
    m = C1.m
```

Les classes (7)
(constructeur : *__init__*)

```
class Packet:
    def __init__(self,
    data):
        self.data = data
class
    PacketWithHdr(Packe
    t):
    def __init__(self, hdr,
    data):
        self.hdr = hdr
        Packet.__init__(self,
    data)
pkt = Packet('data')
```

- Lorsqu'une instance est créée, le constructeur *__init__* est automatiquement appelé
- Un constructeur est une méthode comme une autre, en particulier du point de vue de l'héritage
- Contrairement à C++, Python n'invoque qu'**un seul** constructeur →

Les classes (8)
(destructeur : *__del__*)

```
class Classe:
    def __del__ (self):
        print self,
    'deleted'
x = Classe()
```

```
>>> cl.py
<__main__.Classe
    instance
 at 0x009E3168>
    deleted
>>>
```

- Lorsqu'une instance est détruite (i.e quand la dernière référence est supprimée), le destructeur *__del__* est automatique-ment appelé
- Comme pour les construc- teurs, **un seul** destructeur est invoqué lors de la suppression d'une instance dans le cas d'une classe fille

Les classes (9)
(surcharge d'opérateur/méthode)

```
class Packet:
    def __init__(self, data):
        self.data = data
    def __add__(self, data):
        ret = Packet(self.data)
        ret.data += data
        return ret
    def __repr__(self):
```

```
pkt = Packet('data1')
pkt += '-data2'
print pkt
```

```
>>> cl.py
Data: data1-data2
>>>
```

- Surcharge des méthodes *__add__* (+) et *__repr__* (*print*)

Les classes (10)
(surcharge d'opérateur/méthode)

```
PacketIPError = 'PacketIPError'
class PacketIP:
    def __init__(self, data):
        self.__dict__['type'], self.data = 'IPv4',
    data
    def __setattr__(self, nom, val):
        if nom == 'type': # analogue de private: en
    C++
            raise PacketIPError, 'attribut type non
    modifiable'
        self.__dict__[nom] = val # on n'écrit pas
    self.nom = val
```

Les classes (11)
(les méthodes prédéfinies)

Tous les types

__init__(self, args...)	constructeur
__del__(self)	destructeur
__repr__(self)	`self`, print self, repr(self)
__str__(self)	str(self), print self
__cmp__(self, other)	self > x, self == x, ...
__rcmp__(self, other)	x <= self, x != self, ...
__hash__(self)	dictionnary[self], hash(self]
__call__(self, *args)	self(args...)

Les classes (12)
(les méthodes prédéfinies)

Tous les types (suite)

__getattr__(self, nom)	self.nom
__setattr__(self, nom, val)	self.nom = val
	del self.nom

Collections

__len__(self)	len(self)
__getitem__(self, key)	self[key], x in self, for x in self
__setitem__(self, key, val)	self[key] = val

Les classes (13)
(les méthodes prédéfinies)

Collections (suite)	
__getslice__(self, l, h)	→ self[l:h]
__setslice__(self, l, h, seq)	→ self[l:h] = seq → del self[l:h]

Nombres		
__add__	__mul__	__mod__
__radd__	__rmul__	__rmod__
__sub__	__div__	__divmod__
__rsub__	__rdiv__	__pow__

Les classes (14)
(les méthodes prédéfinies)

Nombres (suite)		
__lshift__	__or__	__int__
__rlshift__	__ror__	__long__
__rshift__	__neg__	__float__
__rrshift__	__pos__	__oct__
__and__	__abs__	__hex__
__rand__	__invert__	
__xor__	__nonzero__	
__rxor__	__coerce__	

Les classes (15)
(les attributs prédéfinis)

```
class A:
    def pr(arg):
        print arg
class B(A): pass
print A.__dict__,
```

```
print B.__bases__
a, b = A(), B()
print a.__class__,
print isinstance(a, A),
print issubclass(B, A)
```

```
{'pr': <function pr at 0x009214F8>,
'__module__': '__main__', '__doc__': None} A

(<class __main__.A at 0x01041698>,)

__main__.A 1 1
```

Les fichiers (1)

```
# !/usr/bin/env python
fd = open('essai.py',
   'r')
print fd.read(),
fd.seek(0, 0)
```

```
>>> file.py
# !/usr/bin/env python
fd = open('essai.py',
   'r')
print fd.read()
```

```
fd.seek(0, 0)
print fd.read(3)
# !/
```

- Les autres méthodes :

readline()	*close()*
readlines()	*tell()*
write(string)	*isatty()*
writelines(list)	*flush()*

Les fichiers (2)

- On peut également accéder aux fichiers via le module *os*
- Dans ce cas, on dispose des appels système Unix standard : *read*, *write*, *lseek*, *dup*,... avec presque la même syntaxe et les mêmes arguments (la plupart sont également

```
# !/usr/bin/env python
import os
fd = os.open('essai', os.O_RDWR | os.O_TRUNC, 0644)
os.write(fd, 'essai')
os.close(fd)
```

La documentation en ligne (1)

```
>>> import __builtin__
>>> print
   dir(__builtin__)
['ArithmeticError', ...
 'zip']
>>> print chr.__doc__
chr(i) -> character

Return a string of one
   character with
   ordinal i;
   0 <= i < 256.
```

```
# !/usr/bin/env python
def f(c):
    """
f(x) -> retourne str(x)\
"""
    return str(x)
print f.__doc__
```

```
>>> doc.py
f(x) -> retourne str(x)
```

La documentation en ligne (2)

```
"""
Un petit module\
"""
class A:
    """
La classe A est vide\
"""
    pass
print A.__doc__,
print __doc__
```

```
>>> doc.py
La classe A est vide
Un petit module
```

- La documentation en ligne, très utile, est facile à mettre en œuvre
- A utiliser sans modération !
- Prise en compte par ***idle***

Étendre Python en C (1)

- Étendre Python en C est parfois nécessaire si l'on veut créer de nouveaux types de données ou accéder à des appels systèmes non déjà disponibles (il y en a de moins en moins !), etc…
- Si l'on sait programmer en C, cela ne présente pas de difficultés particulières
- L'API Python fournit tout ce dont on a besoin
- Le seul point un peu délicat est la gestion des références des objets

Étendre Python en C (2)

(un exemple : minimod.c)

```
# include <Python.h> // API Python
static PyObject *
MiniMod_hello(PyObject *self, PyObject *args)
{
    if (!PyArg_ParseTuple(args, "")) // pas
   d'argument
      return NULL; // exception gérée par
   l'interpréteur
    fprintf(stdout, "Hello World!\n");
    Py_INCREF(Py_None); // on crée une nouvelle
   référence
    return Py_None;          // de l'objet Py_None
   (None)
```

Étendre Python en C (3)

(un exemple : minimod.c)

```
static struct PyMethodDef MiniMod_methods[]
    = {
    {"hello", MiniMod_hello, METH_VARARGS},
    {NULL, NULL}
}; // déclaration des méthodes du module
    minimod
static char MiniMod_doc[] = "Un (tout) petit
    module";
void initminimod(void) // initialisation du module
    minimod
{
    Py_InitModule3("minimod",
    MiniMod_methods,
```

Étendre Python en C (4)
(un exemple : minimod.c)

```
$ gcc –shared –I<python_inc_dir> minimod.c –o
  minimod.so
$ python
>>> import minimod
>>> minimod.hello()
Hello World!
>>> print minimod.__doc__
Un (tout) petit module
```

- <python_inc_dir> est le catalogue où l'on trouve le fichier *Python.h* (par ex., */usr/local/include/python2.2*)

Conclusion
(qui n'engage que l'auteur)

- Vous pouvez oublier tous les autres langages de scripts (et même le C dans beaucoup de cas) et n'utiliser que Python !
- À mon sens, bien qu'ayant une approche très différente (langage *typé*), peut-être un seul « concurrent » intéressant : *ocaml* (Objective CAML). Mais *ocaml* est beaucoup moins diffusé