« Le Python, c'est bon ! »

Une présentation du langage

Decotigny (LLR, 15/06/2009)

Plan

- Python, c'est quoi, pour faire quoi, comment ?
- Quelques caractéristiques
- Types de base
- structures de contrôle
- Fonctions
- Programmation objet
- Modules, packages, ...

Python, c'est quoi ?

- C'est un ”récent” (1991 !) langage de script
  - Automatisation de tâches rébarbatives ”système”
  - Prototypage facile
  - Remplace perl, sed, grep, ... mais pas forcément (c)sh
- C'est un vrai langage de programmation
  - Permet d'écrire de vrais programmes complexes
    - Structurés en modules (eq. .o, #include, ...)
    - Programmation objet
- Ça peut utiliser de vraies bibliothèques
    - ROOT, calcul numérique, simulation, GUIs, etc.
    - Inversement : peut servir dans d'autres programmes (configuration, ...)

Python, c'est aussi...

- Un langage facile à apprendre
- Une doc bien fournie (en ligne, fonction `help()`, programme `pydoc`)
- Une communauté très active
  - forums + archives sur le web
  - Un environnement qui évolue (langage + interpréteur + types/modules standards). Actuellement : transition python 2.x (recommandé & présenté ici : 2.4 ou +, actuel : 2.6) → 3.x
- Un gourou (Guido van Rossum)
- Une philosophie (le Zen) : `import this`
- Des références aux Monty Python

Python, pour faire quoi ?

- Écrire des programmes d'analyse/remplacement de textes, outils ”serveur”/”ligne de commande”
  - Exemple : scripts de configuration, services web, ...
- Écrire des scripts d'analyse de données
  - scipy, numpy, pylab, ROOT, ...
- Écrire des interfaces graphiques
  - PyGTK, PyQt, pyGTK, wxPython, tix, ...
- Et tout celà :
  - Sans trop se soucier de l'OS (unix, windows, Mac)
  - Sans se soucier des erreurs bêtes (débordements de tableaux, fuites de mémoire, erreurs de pointeur, …)
    - Automatique : Conteneurs robustes, comptage de références, GC, ...

Python, ça s'utilise comment ?

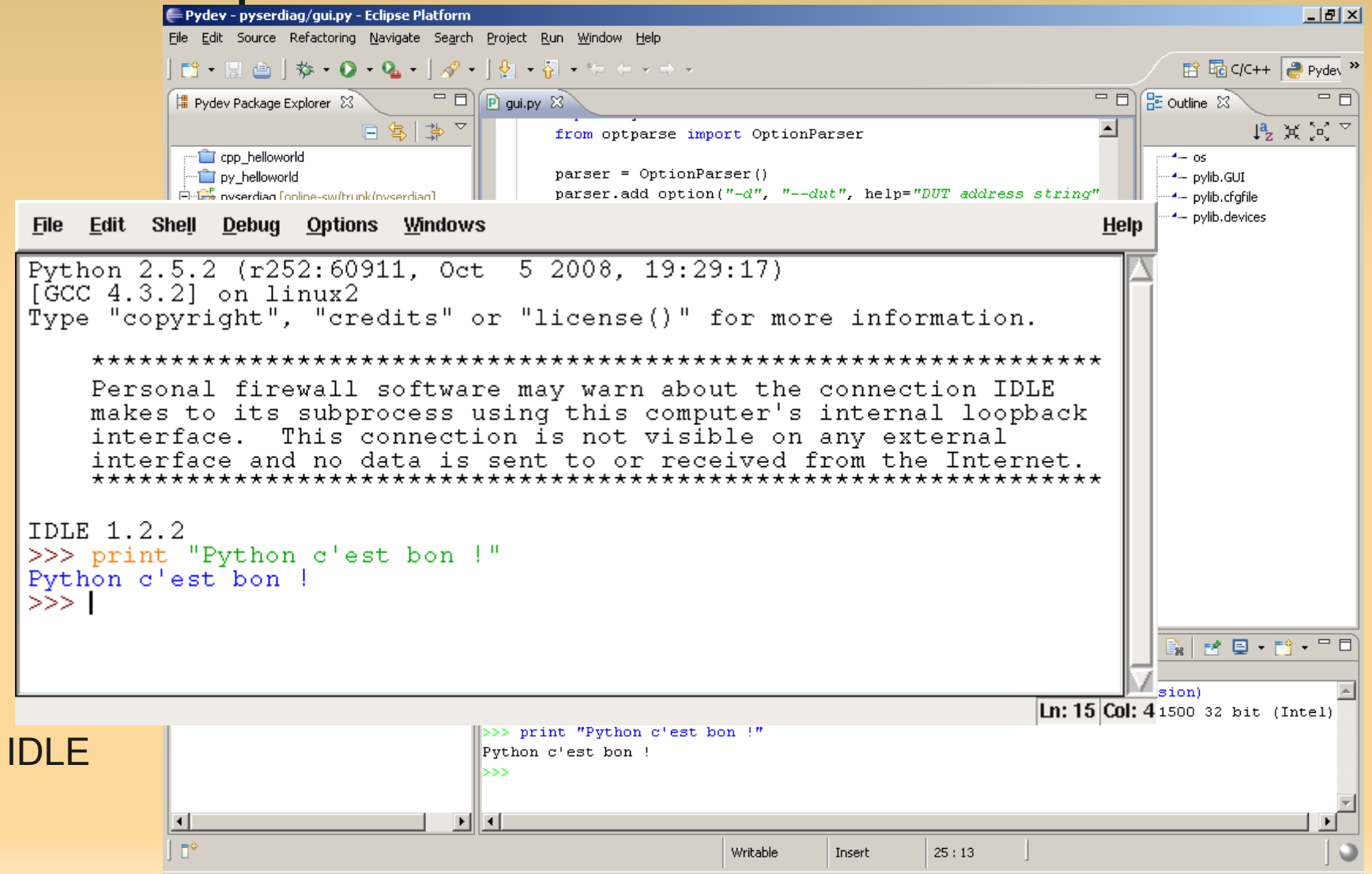
- En ligne de commande :

```
[pollindd] svn/pollin1/2009/intro_python >python                    1050
Python 2.5.2 (r252:60911, Oct  5 2008, 19:29:17)
[GCC 4.3.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> def factorielle(n):
...     if n == 1:
...         return 1
...     else:
...         return n*factorielle(n-1)
...
>>> factorielle(142)
269536413788816277658850750803729026709400851689139611079208959973446471
698867057212879731934194897340243610609741027716867307764822172854447798
758612548486829479004434022298980073807909182159855712819266715648000000
00000000000000000000000000L
>>> print "On appuie sur Ctrl-D pour quitter..."
On appuie sur Ctrl-D pour quitter...
>>>
[pollindd] svn/pollin1/2009/intro_python >                          1051
```

Mais sans les couleurs en réalité (FYI : j'utilise Pygments pour la coloration dans ces slides)

Python, ça s'utilise comment ?

- Avec un environnement de développement adapté :



IDLE

PyDev/eclipse

Python, ça s'utilise comment ?

- Avec vos éditeurs favoris...

```
def factorielle(n):
        if n == 1:
                return n
        else:
                return n*factorielle(n-1)
~
~
~
~
~
~
                                1,1           All
```

```
File  Edit  Options  Buffers  Tools  Python  Help
def factorielle(n):
        if n == 1:
                return n
        else:
                return n*factorielle(n-1)

print factorielle(142)

--:--  f.py          All (8,0)      (Python)--
Python 2.5.2 (r252:60911, Oct  5 2008, 19:2
9:17)
[GCC 4.3.2] on linux2
Type "help", "copyright", "credits" or "lic
ense" for more information.
>>> 269536413788816277658850750803729026709
400851689139611079208959973446471469886705
212879731934194897340243610609741027716867
077648221728544477989758612548486829479004
340222989800738079091821598557128192667156
800000000000000000000000000000000000
>>>
-u:**  *Python*      Bot (21,4)      (Inferior
```

Quelques caractéristiques (1/4)

- Un langage de programmation orienté objet
  - Syntaxe simple et ”intuitive”
    - Peu verbeux (pas assez ?) et clair à relire
    - Débarrasse des ”détails” techniques (malloc, free, les tableaux ont des bornes, ...)
      - Focalise sur l'algorithmique, limite les erreurs de programmation graves
  - Un langage implicitement typé
    - Implicitement = code peu déclaratif
    - Typé = tous les objets ont une structure connue à l'exécution
      - Supporte/encourage l'introspection (late binding)
    - Mais nécessite de la discipline pour une maintenance facile
      - Fonctions déclarées sans prototype (... ça change avec python 3)
  - Un langage orienté objet : encapsulation, héritage
  - Un langage riche: exceptions, générateurs, fonctionnel...

Quelques caractéristiques (2/4)

- Un interpréteur / machine virtuelle
  - Cycle de mise au point rapide (pas de compilation)
  - Mais *relativement* lent à exécuter
    - Et pourtant ! Python est très utilisé pour le calcul numérique (c'est que la partie python s'occupe surtout du séquencement, pas des calculs numériques grain fin)
  - Un interpréteur de référence : cPython
    - Fonctionne à l'identique sous windows, unix, macos, … (en principe)
    - D'autres environnements : jython, PyPy, Parrot, ...
  - Passage par un bytecode (facultatif) pour éviter un parsing systématique : fichiers .pyc et .pyo

Quelques caractéristiques (3/4)

- Un langage + librairies très bien documentés
  - http://python.org/doc/
  - La grosse partie de cette présentation : http://docs.python.org/tutorial/
- Un langage qui encourage la programmation modulaire (modules, packages, ...)
- La possibilité de ”binder” avec des bibliothèques
  - Permet à du python d'utiliser virtuellement toutes les bibliothèques existantes : pyROOT, PyQt, pyGTK, ...
  - Mais ”traduction” manuelle nécessaire, même si des outils facilitent la tâche (cAPI, swig, boost::python, sip, ctypes...)
  - Énoooooorme corpus déjà disponible (> java ???)

Quelques caractéristiques (4/4)

- Une bibliothèque de ”*modules*” standards
  - Très très fournie : http://docs.python.org/library/
- Plein de modules externes pour tous usages
  - Cheeseshop : http://pypi.python.org/
  - ASPN python cookbook : http://code.activestate.com/recipes/langs/python/

Utiliser l'interpréteur (1/5)

- En mode interactif :

```
>>> Bonjour Monde !
  File "<stdin>", line 1
    Bonjour Monde !
                  ^
SyntaxError: invalid syntax
>>> 'Bonjour Monde !'
'Bonjour Monde !'
>>> print 'Bonjour Monde !'
Bonjour Monde !
>>> 42
42
>>> print 42
42
>>> import math
>>> print math.sqrt(1764)
42.0
>>>
```

Utiliser l'interpréteur (2/5)

- En ligne de commande :

```
[pollindd] ~ >python -c 'Bonjour Monde !'                              1044
  File "<string>", line 1
    Bonjour Monde !
                 ^
SyntaxError: invalid syntax
zsh: exit 1     python -c 'Bonjour Monde !'
[pollindd] ~ >python -c 'print "Bonjour Monde !"'                      1045
Bonjour Monde !
[pollindd] ~ >python -c 42                                             1046
[pollindd] ~ >python -c 'print 42'                                     1047
42
[pollindd] ~ >python -c 'import math ; print math.sqrt(1764)'          1048
42.0
[pollindd] ~ >                                                         1049
```

Utiliser l'interpréteur (3/5)

- Le premier module... ou presque

```
[pollindd] ~ >cat > premier.py                                  1050

'Bonjour Monde !'

print 'Hello world !'

42

print 42


import math
print math.sqrt(1764)

Bonjour Monde !

[pollindd] ~ >python ./premier.py                               1051
  File "./premier.py", line 14
    Bonjour Monde !
                ^
SyntaxError: invalid syntax
zsh: exit 1     python ./premier.py
[pollindd] ~ >                                                  1052
```

Zut alors !

Utiliser l'interpréteur (4/5)

- Le premier module... qui marche !

```
[pollindd] ~ >cat > premier-ok.py                                    1052

'Bonjour Monde !'

print 'Hello world !'

42

print 42


import math
print math.sqrt(1764)

[pollindd] ~ >python ./premier-ok.py                                 1053
Hello world !
42
42.0
[pollindd] ~ >                                                       1054
```

Utiliser l'interpréteur (5/5)

- Un programme = module exécutable (*shebang*)

```
[pollindd] ~ >cat > py-shebang.py                                    1002
#! /usr/bin/env python

'Bonjour Monde !'

print 'Hello world !'

42

print 42

import math
print math.sqrt(1764)

[pollindd] ~ >chmod 755 py-shebang.py                                1003
[pollindd] ~ >./py-shebang.py                                        1004
Hello world !
42
42.0
[pollindd] ~ >                                                       1005
```

Les types de base : scalaires

- Utiliser python comme une calculatrice

```
>>> 40+2 # Ceci est un commentaire !
42
>>> (1+2+3+4) / 5
2
>>> 7/3 # Division entiere
2
>>> 7 % 3 #  Modulo
1
>>> 7./3 # Division flottante
2.3333333333333335
>>> var1 = 7 # var1 est un entier
>>> var2 = 3
>>> var1 / var2
2
>>> float(var1) / var2
2.3333333333333335
>>> 2**128 # Puissances
236118324143482260684 8L
>>> x = y = z = 0
>>> x
0
>>> y
0
>>> z
0
>>> y = 1
>>> x
0
>>> y
1
>>> z
0
>>>
```

```
>>> int(3.99)
3
>>> round(3.99)
4.0
>>> int(round(3.99))
4
>>> int(1e9)
1000000000
>>> x = int(1e9)
>>> x*x*x  # Pas de saturation en python
1000000000000000000000000000L
>>> int(1e27) # Mais ATTENTION aux arrondis !
1000000000000000013287555072L
>>>
```

commentaires
int/long, float, complex
assignements de variables
conversion, construction, arrondis
premiers attributs d'objets

Variables, fonctions, … : attention à la casse (majuscules/minuscules) !

```
>>> 1j * 1j
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> (3+1j)*3
(9+3j)
>>> a = _ # Dernier resultat (console seule)
>>> a.real # attribut '.real' de la variable
9.0
>>> a.imag
3.0
>>> abs(a) # Le module
9.486832980505138 1
>>>
```

Les types de base : scalaires

- Expressions booléennes :

```
>>> True
True
>>> False
False
>>> 1 == 1
True
>>> 1 != 2
True
>>> 2 > 1
True
>>> 1 < 2
True
>>> 2 >= 1
True
>>> 1 <= 2
True
>>> (1 == 2) or (2 == 2)
True
>>> (1 == 1) and (2 == 2)
True
>>> not (1 == 2)
True
>>> (1 == 1) and not (1 == 2)
True
>>>
```

Ne pas confondre avec l'arithmétique binaire :

```
>>> 1 & 2 # and bit a bit
0
>>> 1 | 2 # or bit a bit
3
>>> 1 ^ 3 # xor bit a bit
2
>>> ~1 # Complement a 1
-2
>>>
```

Les types de base : scalaires

- `None`
  - Utilisé comme un marqueur spécial, un indicateur

```
1 >>> None
2 >>> n = None
3 >>> n
4 >>> print n
5 None
6 >>> print None
7 None
8 >>>
```

Les types de base : conteneurs

- Chaînes de caractères
  - str, ', " et """
  - Les \

```
>>> ''
''
>>> str()
''
>>> "Le python c'est bon !"
"Le python c'est bon !"
>>> str("Le python c'est bon !")
"Le python c'est bon !"
>>> 'Le python c\'est bon !'
"Le python c'est bon !"
>>> 'J\'ai dit : "Le python c\'est bon !"'
'J\'ai dit : "Le python c\'est bon !"'
>>> "J'ai dit : \"Le python c'est bon !\""
'J\'ai dit : "Le python c\'est bon !"'
>>> print "J'ai dit : \"Le python c'est bon !\""
J'ai dit : "Le python c'est bon !"
>>>
```

```
>>> s = "Et je passe
  File "<stdin>", line 1
    s = "Et je passe
                    ^
SyntaxError: EOL while scanning single-quoted string
>>> print "Et je passe\
... a la ligne"
"Et je passea la ligne"
>>> print "Et je passe\n\
... a la ligne"
Et je passe
a la ligne
>>> print r"Et je passe\n\
... a la ligne"
Et je passe\n\
a la ligne
>>> print """
... Et 'voici'
... environ "quatre" lignes
... """

Et 'voici'
environ "quatre" lignes

>>>
```

Les types de base : conteneurs

- Chaînes de caractères
  - Manipulations simples (*, +, [...], index négatifs)
  - Modifications impossibles
  - Pratiquement tout peut être converti en str (fonction `str`)
  - Variantes `unicode` et `bytes`

```
>>> str()
''
>>> ''
''
>>> 'a'
'a'
>>> "Le python c'est bon ! :)"
"Le python c'est bon ! :)"
>>> str("Le python c'est bon ! :)")
"Le python c'est bon ! :)"
>>> "Blah" * 3
'BlahBlahBlah'
>>> s = "Le python c'est bon ! :)"
>>> len(s)
24
>>> s[0]
'L'
>>> s[1]
'e'
>>> s[-1] # Le 1er a partir de la fin
')'
>>> s[-2] # Le 2eme a partir de la fin
':'
>>> s[3] = 'X' # Impossible avec des chaines
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> s[42] # Idem avec -42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> s + " C'est bin vrai ca !"
"Le python c'est bon ! :) C'est bin vrai ca !"
>>> print s # s pas modifiee
Le python c'est bon ! :)
>>> s + str([1, 2, 3]) # any -> str
"Le python c'est bon ! :)[1, 2, 3]"
>>>
```

Les types de base : conteneurs

- Les listes
  - Ça ressemble à str
  - Listes hétérogènes
  - On peut les modifier (index, slices, `del`) : type ”*mutable*”

```
>>> list()
[]
>>> []
[]
>>> [1]
[1]
>>> [1, 'abc', list(), 4]
[1, 'abc', [], 4]
>>> list([1, 42])
[1, 42]
>>> list("Mouf") # Attention ! str -> list
['M', 'o', 'u', 'f']
>>> [1,2] * 5
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
>>> l = [1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> len(l)
9
>>> l[0]
1
>>> l[1]
2
>>> l[-1]
9
>>> l[-2]
8
>>> l[3] = 42 # Possible avec des listes
>>> l[4] = [42, 43, 44] # sous-liste
>>> l
[1, 2, 3, 42, [42, 43, 44], 6, 7, 8, 9]
>>> del l[5] # Suppression !
>>> l
[1, 2, 3, 42, [42, 43, 44], 7, 8, 9]
>>> l[42] # Idem index -42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> l + [10, 11]
[1, 2, 3, 42, [42, 43, 44], 7, 8, 9, 10, 11]
>>> print l # l pas modifiee
[1, 2, 3, 42, [42, 43, 44], 7, 8, 9]
>>> l + list("Mouf")
[1, 2, 3, 42, [42, 43, 44], 7, 8, 9, 'M', 'o', 'u', 'f']
>>>
```

Les types de base : conteneurs

- Tuples
  - Des listes hétérogènes non modifiables (immutable)
  - Attention à la syntaxe du singleton !
  - Plus compactes en mémoire

Pratique aussi pour l'affectation par paquets :

```
>>> a, b, c = 1, 2, 3 # Idem (1, 2, 3)
>>> a, b, c = [1, 2, 3] # Liste
>>> a, b, c = "ABC"
```

```
>>> tuple()
()
>>> ()
()
>>> (1,) ## Virgule NECESSAIRE !
(1,)
>>> (1) # Sinon ce n'est qu'un entier !
1
>>> (1, 'abc', tuple(), 4)
(1, 'abc', (), 4)
>>> tuple((1,42))
(1, 42)
>>> tuple('abc') # Attention ! str -> tuple
('a', 'b', 'c')
>>> (1,2)*5
(1, 2, 1, 2, 1, 2, 1, 2, 1, 2)
>>> t = (1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> len(t)
9
>>> t[0]
1
>>> t[1]
2
>>> t[-1]
9
>>> t[-2]
8
>>> t[3] = 42 # Impossible avec des tuples
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> print t
(1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> t[42] # Idem index -42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: tuple index out of range
>>> t + (10, 11)
(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)
>>> print t # t pas modifiee
(1, 2, 3, 4, 5, 6, 7, 8, 9)
>>> t + tuple("Mouf") # str -> tuple
(1, 2, 3, 4, 5, 6, 7, 8, 9, 'M', 'o', 'u', 'f')
>>> tuple([4,5,6]) # list -> tuple
(4, 5, 6)
>>> list((7,8,9))  # tuple -> list
[7, 8, 9]
>>>
```

Les types de base : slices

- Notion de *slice* pour les types séquence str/list/tuple
  - `seq[a:b:s]` : **copie** des éléments de seq dont l'index est dans [a,b[, par pas de n (peut être négatif). Par défaut :
    - a = 0
    - b = fin de liste
    - s = 1
  - Une slice peut être vide : `seq[a:a]`
  - Utilisées à gauche de '=' → modif de la séq., pour des séq. *mutable* seulement (listes) !

```
>>> # Valable sur str/tuple/list :
... l = [0,1,2,3,4,5,6,7,8,9]
>>> l[3:8]
[3, 4, 5, 6, 7]
>>> l[3:8:2]
[3, 5, 7]
>>> l[:5]
[0, 1, 2, 3, 4]
>>> l[5:]
[5, 6, 7, 8, 9]
>>> l[:5:2]
[0, 2, 4]
>>> l[5::2]
[5, 7, 9]
>>> l[::2]
[0, 2, 4, 6, 8]
>>> extrait = l[3:8]
>>> extrait[2] = 42
>>> (extrait, l) # l pas modifiee
([3, 4, 42, 6, 7],
 [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> COPIE = l[:] # Idem l[::]
>>> COPIE
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> COPIE = list(l) # Meme chose
>>> COPIE
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> # Mofification de slices
... # Valable sur des listes seulement
... l[1:2] = list("any len")
>>> l
[0, 'a', 'n', 'y', ' ', 'l', 'e', 'n', 2, 3,
 4, 5, 6, 7, 8, 9]
>>> l[0:0] = [21,22,23] # Insertion en tete !
>>> l
[21, 22, 23, 0, 'a', 'n', 'y', ' ', 'l', 'e',
 'n', 2, 3, 4, 5, 6, 7, 8, 9]
>>> l[3:8:2] = [11,12,13] # len = 3 seulement
>>> l
[21, 22, 23, 11, 'a', 12, 'y', 13, 'l', 'e',
 'n', 2, 3, 4, 5, 6, 7, 8, 9]
>>>
```

Les types de base : conteneurs

- Type ”ensemble”, utiles pour opérations ensemblistes
  - Python >= 2.4 (`from sets import Set as set` sinon)
  - Non ordonné (≠ STL)
  - Hétérogène
  - Pas d'opérateur ”[]”/slice. API davantage ”objet”

```
>>> set()
set([])
>>> set((1,)) ## A partir d'un iterable
set([1])
>>> set('abc') # Attention !
set(['a', 'c', 'b'])
>>> set((1, 'abc', tuple(), 4))
set([1, 'abc', (), 4])
>>> s = set((1, 2, 3, 4, 5, 6, 7, 8, 9))
>>> len(s)
9
>>> s & set((3, 4, 42))
set([3, 4])
>>> s | set((3, 4, 42))
set([1, 2, 3, 4, 5, 6, 7, 8, 9, 42])
>>> s - set((8, 9))
set([1, 2, 3, 4, 5, 6, 7])
>>> s ^ set((7, 'nouveau'))
set(['nouveau', 1, 2, 3, 4, 5, 6, 8, 9])
>>>
```

Les types de base : conteneurs

- Dictionnaires
  - Correspondance clef (unique) → valeur
  - Central à tout le langage !
  - Non ordonné (≠ STL)
  - Hétérogène

```
>>> dict()
{}
>>> {}
{}
>>> {'clef': 'valeur'}
{'clef': 'valeur'}
>>> dict(clef = 'valeur') # Variadique
{'clef': 'valeur'}
>>> {1: 'toto', 2: 3, 'xyz': 55}
{1: 'toto', 2: 3, 'xyz': 55}
>>> dict([(1,'toto'), (2,3), ('xyz',55)])
{1: 'toto', 2: 3, 'xyz': 55}
>>> dict(("ab", "cd", "ef")) # Attention !
{'a': 'b', 'c': 'd', 'e': 'f'}
>>> d = dict(a=1, b=2, c=3)
>>> len(d)
3
>>> d['a']
1
>>> d['b']
2
>>> d['MOUF']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'MOUF'
>>> d['a'] = 42
>>> d
{'a': 42, 'c': 3, 'b': 2}
>>> d['MOUF'] = 'Hello'
>>> d
{'a': 42, 'c': 3, 'b': 2, 'MOUF': 'Hello'}
>>> del d['MOUF'] # Suppression
>>> d
{'a': 42, 'c': 3, 'b': 2}
>>>
```

Les types de base : conteneurs

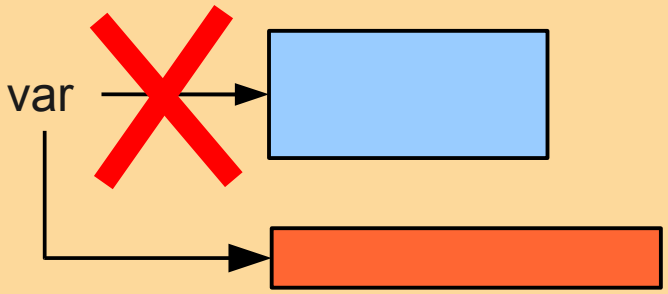
- Prédicats `in` / `not in`
  - Pour tous les conteneurs standards
  - Optimisés seulement pour `set()` et `dict()`

```
>>> s = "abcdefghijk"
>>> 'c' in s
True
>>> 'l' not in s
True
>>> lst = [1, 2, 3, 4]
>>> 1 in lst
True
>>> 5 not in lst
True
>>> t = (1, 2, 3, 4)
>>> 1 in t
True
>>> 5 not in t
True
>>> s = set((1, 2, 3, 4))
>>> 1 in s          # Optimise
True
>>> 5 not in s      # Optimise
True
>>> d = {'c1': 'v1', 'c2': 'v2'}
>>> 'c1' in d       # Optimise
True
>>> 'toto' not in d # Optimise
True
>>>
```

En python, (presque) tout est une référence (pointeur)

- Une variable pointe vers un objet
  - L'objet pointé a un type, mais la variable peut pointer vers n'importe quel objet
    - Une même variable pourra successivement pointer vers différents types d'objets

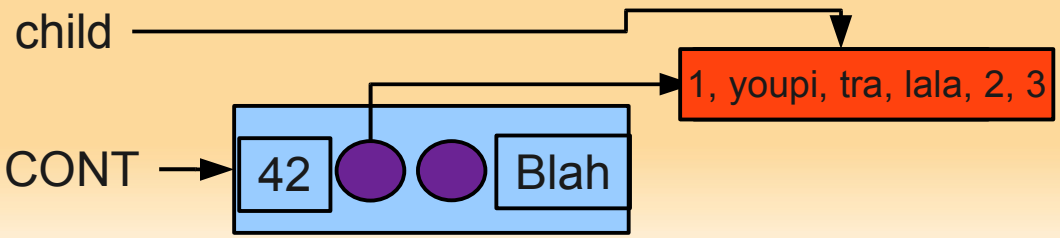
```
1 >>> VAR = 42 # Un entier
2 >>> VAR = "Python c'est bon" # Une chaine
3 >>> VAR = dict(a="b") # Un dictionnaire
4 >>>
```



En python, (presque) tout est une référence (pointeur)

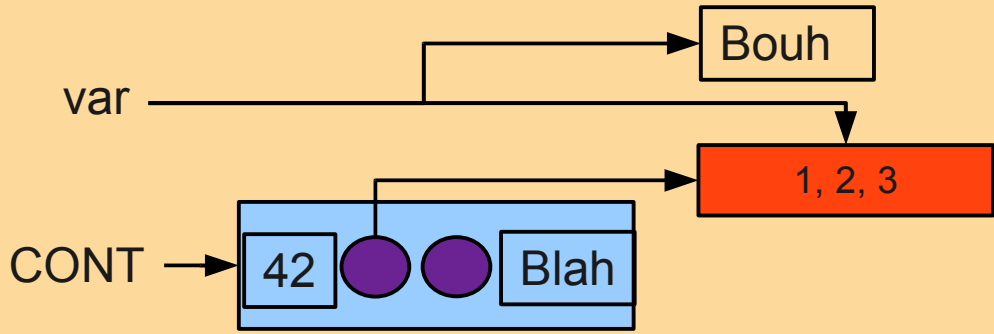
- Les conteneurs contiennent scalaires et références (pointeurs) vers d'autres objets
  - Quand on modifie les objets référencés, le conteneur le reflète

```
>>> CONT = [42, [1,2,3], [4,5,6], "Blah"]
>>> child = CONT[1]
>>> # On change le contenu de child
... child[1:1] = ["Youpi", "Tra", "Lala"]
>>> # On verifie que CONT reflete le chgt :
... CONT
[42, [1, 'Youpi', 'Tra', 'Lala', 2, 3], [4, 5, 6], 'Blah']
>>> # La meme chose en moins "surprenant" :
... CONT[2][1:1] = ["Monty", "Python"]
>>> CONT
[42, [1, 'Youpi', 'Tra', 'Lala', 2, 3], [4, 'Monty', 'Python', 5, 6], 'Blah']
>>>
```



Note : ci-dessous, on ne modifie pas le Contenu des objets référencés. On ne Modifie qu'une variable :

```
>>> CONT = [42, [1,2,3], [4,5,6], "Blah"]
>>> var = CONT[0]
>>> var = "Bouh"
>>> # CONT pas modifie
... CONT
[42, [1, 2, 3], [4, 5, 6], 'Blah']
>>> var = CONT[1]
>>> var = "Bouh bis"
>>> # CONT pas modifie
... CONT
[42, [1, 2, 3], [4, 5, 6], 'Blah']
>>> var = CONT[3]
>>> var = "Bouh ter"
>>> # CONT pas modifie
... CONT
[42, [1, 2, 3], [4, 5, 6], 'Blah']
>>>
```

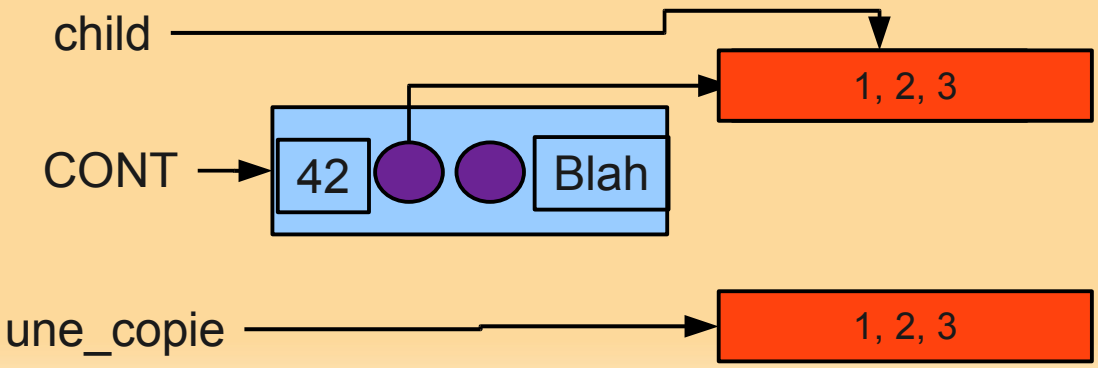


Utiliser le module "`copy`" si on veut dupliquer des objets afin de travailler sur des références distinctes

Prédicats `is`, `is not`, `==`, `!=`

- Égalité usuelle (contenu, ...):
  - `==` et `!=`
  - *(Méthodes `__eq__` et `__neq__`, cf. POO)*
- Même objet référencé ?
  - `is` et `is not`

```
1  >>> CONT = [42, [1,2,3], [4,5,6], "Blah"]
2  >>> child = CONT[1]
3  >>> CONT[1] is child
4  True
5  >>> child is CONT[1]
6  True
7  >>> child == CONT[1]
8  True
9  >>> CONT[0] is not child
10 True
11 >>> child is not CONT[0]
12 True
13 >>> child != CONT[0]
14 True
15 >>> une_copie = child[:]
16 >>> une_copie == child
17 True
18 >>> une_copie is not child
19 True
20 >>>
```



Commençons à programmer !

- Premier exemple de base
  - Assignement multiple
  - `while` *`prédicat`*`:...`
  - **Indentation !**

```
>>> a, b = 1, 1
>>> while a < 10:
...         print 'factorielle', a-1, '->', b
...         a, b = a+1, a*b
...
factorielle 0 -> 1
factorielle 1 -> 1
factorielle 2 -> 2
factorielle 3 -> 6
factorielle 4 -> 24
factorielle 5 -> 120
factorielle 6 -> 720
factorielle 7 -> 5040
factorielle 8 -> 40320
>>>
```

```
>>> a = 1
>>> # Et si j'oublie d'indenter ?...
... while a < 10:
... a = a + 1
  File "<stdin>", line 3
    a = a + 1
    ^
IndentationError: expected an indented block
>>>
```

```
>>> a = 1
>>> # Je peux tout mettre 'en-ligne' :
... # Uniquement si le corps du while
... # tient sur une seule ligne :
... while a < 10: a = a + 1
...
>>>
```

Structures de contrôle : if

- Expressions "`if prédicat:...else/elif:...`":

```
>>> x = int(raw_input("Entrez un nombre entier : "))
Entrez un nombre entier : 42
>>> if x < 0:
...       x = 0
...       print 'Negatif -> Force a 0'
... elif x == 0:
...       print 'Zero'
... elif x == 1:
...       print 'Unite'
... else:
...       print 'Positif'
...
Positif
>>>
```

Note : Pas de switch...case !

Structures de contrôle : prédicats implicites (conversion vers bool)

- Tous les objets sont implicitement convertibles en booléens :
  - `if une_valeur: ...` <=> une_valeur ≠ 0
  - `if une_liste: …` <=> une_liste n'est pas vide
  - `if un_tuple:...` <=> un_tuple n'est pas vide
  - `if un_ensemble:...` <=> un_ensemble n'est pas vide
  - `if un_dico: …` <=> un_dico n'est pas vide
  - `if un objet: …` <=> un_objet n'est pas `None`
- Idem avec if not... et while (not), etc.

Structures de contrôle : for

- Expressions "`for var in iterable:...`"
  - Itération sur un objet "*iterable*" (dont les conteneurs)
  - Iterable supposé pas modifié par les itérations
  - Itérateur = variable(s) locale(s) à la boucle

Cas habituel :

```
>>> a = ['youpi', 'tra', 'lala']
>>> for x in a:
...     print x, len(x)
...
youpi 5
tra 3
lala 4
>>>
```

L'itérateur peut être un tuple :

```
>>> for x,y in [('a', 1), ('b', 2), ('c', 3)]:
...     print "x*y:", (x*y)
...
x*y: a
x*y: bb
x*y: ccc
>>>
```

Si l'iterable doit être modifié, itérer sur une copie :

```
>>> a = ['youpi', 'tra', 'lala']
>>> for x in a[:]: # copie de la liste complete
...     if len(x) > 3: a.insert(0, (x, len(x)))
...
>>> a
[('lala', 4), ('youpi', 5), 'youpi', 'tra', 'lala']
>>>
```

Structures de contrôle

- Souvent : utilisation de `range` ou `xrange` avec `for`
  - `x?range(a,b,n)` = entiers [a,b[par pas de n (>0 ou <0)
- `continue`, `break` comme en C
- `else`: sortie de boucle sans break
- `pass`: ne fait rien, mais necessaire pour que la syntaxe reste correcte

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...             if n % x == 0:
...                     print n, '=', x, '*', n/x
...                     break # Termine le for interne
...     else: # Pas de break en cours de route
...             # pas de diviseur de n
...             print n, 'est un nombre premier'
...
2 est un nombre premier
3 est un nombre premier
4 = 2 * 2
5 est un nombre premier
6 = 2 * 3
7 est un nombre premier
8 = 2 * 4
9 = 3 * 3
>>>
```

```
>>> while True:
...     pass # Attend un Ctrl-C
... else: # Quand la condition du while devient False
...     # IMPOSSIBLE
...     print 'True devient False ?!!!'
...
^CTraceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyboardInterrupt
>>>
```

`print`, opérateur `%`

- En python 2.x, "`print`" est une expression du langage à part entière :
  - Ne suit pas la syntaxe d'appel de fonction usuelle
  - La virgule seule en fin d'expression = pas de retour chariot

```
1 >>> # Comment faire pour afficher plusieurs elements ?...
2 >>> print "Bonjour" ; print "Python"
3 Bonjour
4 Python
5 >>> print "Bonjour", ; print "Python"
6 Bonjour Python
7 >>> print "Bonjour", "Python"
8 Bonjour Python
9 >>>
```

Note : pas forcément besoin d'utiliser print en mode interactif

`print`, opérateur `%`

- Opérateur `%`:
  - Créé une nouvelle chaîne à partir d'un format (chaîne) et d'arguments (iterable)
  - Syntaxe du format proche (~ identique) à `printf()` du C
  - Utilisé souvent avec `print`, mais très pratique pour tout le reste (méthode `__str__, ...`)

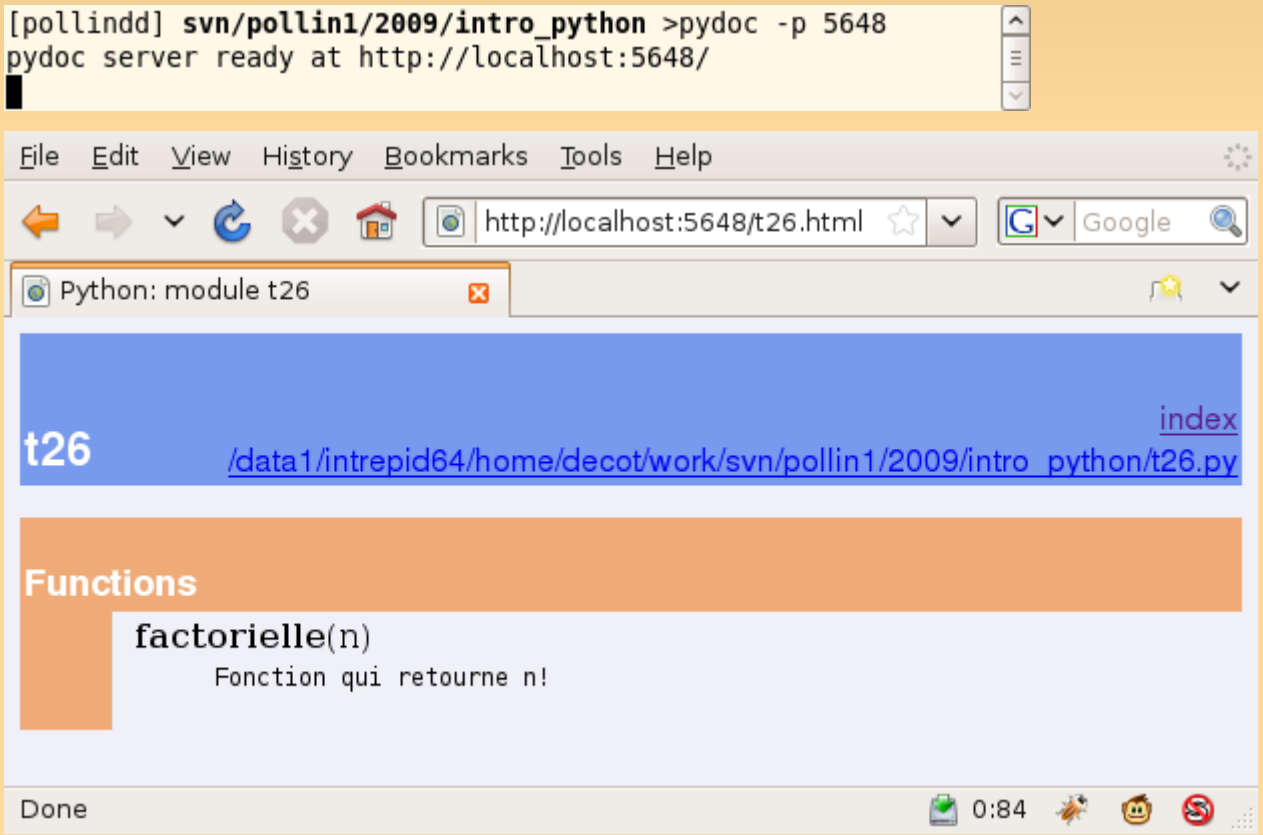
```
1 >>> print "Il %s %02d:%02d et %.5fs" % ("est", 10, 35, 42.123456789)
2 Il est 10:35 et 42.12346s
3 >>> s = "Le %s c'%s bon !" % ('Python', 'est')
4 >>> print s
5 Le Python c'est bon !
6 >>>
```

Fonctions

```
>>> def factorielle(n):
...     """Fonction qui retourne n!""" # docstring
...     a, b = 1, 1
...     while a <= n:
...         a, b = a+1, a*b
...     return b
...
>>> factorielle(8)
40320
>>> help(factorielle)
# Affiche l'aide (docstring) pour factorielle
>>>
```

Un peu d'aide ?

- Fonction intégrée `help()`
- Programme pydoc :



Fonctions

- Une fonction qui ne retourne rien retourne... `None`

```
>>> def prepend_to_list(lst, elt):
...     """Rajoute elt au debut de la liste lst"""
...     lst[0:0] = [elt]
...
>>> zeliste = [1,2,3]
>>> prepend_to_list(zeliste, 4)
>>> print prepend_to_list(zeliste, 5)
None
>>> zeliste
[5, 4, 1, 2, 3]
>>>
```

Fonctions : passage d'arguments

- Passage d'arguments par valeur/référence
  - Seuls `bool/int/long/float/None` sont passés par valeur
  - Tout le reste : par référence
    - Quand on passe un tel argument à une fonction qui le modifie, il reste modifié à la sortie de la fonction

```
>>> def passe_par_valeur(entier):
...     entier += 1 # Pas de '++' en python !
...
>>> def passe_par_reference(lst):
...     lst[0:0] = ['modifie']
...
>>> x = 42
>>> passe_par_valeur(x)
>>> print x
42
>>> l = [1, 2, 3]
>>> passe_par_reference(l)
>>> print l
['modifie', 1, 2, 3]
>>>
```

Visibilité des variables (scoping)

- Définit où une variable est visible dans le code
- Règles assez intuitives en python :
  - Un environnement ”global” au module
    - Dynamique (≠ java, C, C++, …) → attention aux écrasements
  - Un environnement ”local” propre à chaque appel de fonction
    - Contient les arguments de la fonction
    - Pour modifier l'environnement global depuis une fonction : mot clef ”`global`”
  - L'itérateur d'un `for` n'est visible que dans le `for` (idem avec exceptions et `with`)
  - Pas d'autres notions de ”blocs”
- Bien différent des règles classiques C/C++ !

Scoping

- Un environnement est un dictionnaire (clefs = noms de ”symboles”, valeur = ref vers objet associé)
  - Faire pointer une variable ailleurs ≠ changer le contenu de l'objet pointé
  - Il y a une différence entre changer l'environnement (noms de variables → objets pointés), et modifier le contenu des objets pointés

Modif environnements :

```
>>> def surcharge_env_local():
...     VAR = 'surcharge locale'
...     print "VAR fin surcharge locale :", VAR
...
>>> def modif_env_global():
...     # On suppose l'existence d'une globale VAR
...     global VAR
...     VAR = 'modif globale'
...     print "VAR fin modif globale:", VAR
...
>>> # On peut declarer VAR apres la def de fonctions
... # qui l'utilisent (global env est dynamique)
... VAR = ["Le", "Python", "c'est", "bon"]
>>> VAR
['Le', 'Python', "c'est", 'bon']
>>> # Seul l'env local va etre affecte :
... surcharge_env_local()
VAR fin surcharge locale : surcharge locale
>>> # On verifie que VAR globale n'a pas change :
... VAR
['Le', 'Python', "c'est", 'bon']
>>> #
... # Maintenant on change bien l'env global
... modif_env_global()
VAR fin modif globale: modif globale
>>> # On verifie que la variable globale pointe
... # vers le nouvel objet :
... VAR
'modif globale'
>>>
```

Modif objets pointés :

```
>>> def acces_globale_lecture():
...     print "On affiche VAR:", VAR
...
>>> def acces_globale_modif():
...     VAR[0:0] = ["OUI"]
...     print "On vient de changer VAR"
...
>>> VAR = ["Le", "Python", "c'est", "bon"]
>>> VAR
['Le', 'Python', "c'est", 'bon']
>>> # Dans la suite, l'env global n'est pas
... # modifie. Seul l'objet vers lequel VAR
... # pointe est considere :
... acces_globale_lecture()
On affiche VAR: ['Le', 'Python', "c'est", 'bon']
>>> acces_globale_modif()
On vient de changer VAR
>>> # On verifie que VAR a effectivement
... # ete modifiee :
... VAR
['OUI', 'Le', 'Python', "c'est", 'bon']
>>>
```

Scoping

- Hierarchie environnement de boucle > environnement local > environnement global

```
>>> def modifie_l_globale():
...     lst[0:0] = ["Modif globale"]
...
>>> def modifie_l_arg(lst):
...     lst[0:0] = ["Modif argument"]
...
>>> lst = [1, 2, 3]
>>> modifie_l_globale()
>>> lst
['Modif globale', 1, 2, 3]
>>> modifie_l_arg(["mon", "argument"])
>>> lst
['Modif globale', 1, 2, 3]
>>>
```

Pas de variables locales à des while/if/... :

```
>>> def est_impair(entier):
...     # Pas besoin de declarer resultat
...     if entier % 2 == 0:
...         resultat = False
...     else:
...         resultat = True
...     return resultat
...
>>> est_impair(5)
True
>>>
```

Autres détails sur les fonctions

- Fonctions récursives possibles
- Fonctions vues comme des valeurs (fonctionnel)
- Fonctions anonymes pour la concision : `lambda`
- Fonctions à nombre d'arguments variables (*variadiques*), arguments nommés, par défaut, etc.

```
>>> def factorielle(n):
...     if n == 1:
...         return 1
...     else:
...         return n*factorielle(n-1)
...
>>> factorielle(8)
40320
>>>
```

Récursives

```
>>> def renvoie_42():
...     return 42
...
>>> renvoie_42()
42
>>> # Et si on oublie les parentheses...
... renvoie_42
<function renvoie_42 at 0x7f49c6604488>
>>>
```

Une fonction vue comme une valeur

```
>>> def my_reduce(f, lst, initial = 0):
...     """f: fonction tmp,element -> new_tmp"""
...     resultat = initial
...     for elt in lst:
...         resultat = f(resultat, elt)
...     return resultat
...
>>> def my_f(tmp,elt):
...     return tmp+elt
...
>>> my_reduce(my_f, [1,2,3])
6
>>> # La meme chose avec lambda + initial a 42:
... my_reduce(lambda x,y: x+y, [1,2,3], 42)
48
>>>
```

Fonction lambda

```
>>> max(1,2,3)
3
>>> max(1,2,3,4)
4
>>> my_reduce(initial = 0,
...           lst = [1,2,3],
...           f = max)
3
>>>
```

Variadiques

Params nommés

Les ”list comprehensions” : des ”fonctions” spéciales

- Le programmeur python est parfois fainéant
  - Plutôt que d'écrire :

```
>>> def mult2(lst):
...     resultat = []
...     for x in lst:
...         resultat.append(x*2)
...     return resultat
...
>>> mult2([1,2,3])
[2, 4, 6]
>>>
```

  - Il utilise une *list comprehension*:

```
>>> def mult2(lst):
...     return [ x*2 for x in lst]
...
>>>
```

  - Ou parfois il préfère map/lambda :

```
>>> def mult2(lst):
...     return map(lambda x: x*2, lst)
...
>>>
```

  - Mais les list comprehensions sont assez puissantes:

```
>>> lst = range(10)
>>> [x/2 for x in lst if x%2==0]
[0, 1, 2, 3, 4]
>>>
```

Programmation objet

- Un objet = encapsulation + héritage
  - Encapsulation : Collection d'attributs + méthodes
  - Héritage (support de l'héritage multiple... avec précaution)
- Une méthode d'instance = fonction normale, mais 1er arg. = l'objet lui-même (nom "`self`" souvent)
  - Notation pointée pour l'accès aux attributs/méthodes
  - Constructeur = méthode `__init__`
  - Pas de destructeur (ou presque)
- Restriction d'accès primitif (`private` en C++)
- Attributs/méthodes de classes (`static` en C++) possible

Notre première classe !

```
>>> class MaClasse:
...     """Documentation pour MaClasse"""
...     def __init__(self, nom):
...         """Initialise l'objet"""
...         self.compteur = 0
...         self.nom      = nom
...     def incremente(self, a1=1):
...         """Incremente le compteur de a1"""
...         self.compteur += a1
...     # Methode 'protected' :
...     def __methode_protegee(self):
...         pass # Ne fait rien
...
>>> objet1 = MaClasse("Mon objet 1")
>>> objet1
<__main__.MaClasse instance at 0x7f78187843f8>
>>> objet1.incremente(42)
>>> objet1.nom
'Mon objet 1'
>>> objet1.compteur
42
>>> objet1.incremente(1234)
>>> objet1.compteur
1276
>>> objet1.__methode_protegee() # Interdit
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: MaClasse instance has no attribute '__methode_protegee'
>>>
```

Note : Constructeur sans argument `__init__(self)` →

```
objet = MaClass() # Toujours des parenthèses
```

Notes utiles

- Pour un objet obj instanciant MaClasse :

```
obj.methode(params...)
```

  - Est équivalent à :

```
MaClasse.methode(obj, params...)
```

- On peut accéder aux méthodes/attributs protégés depuis l'extérieur :

```
obj._MaClass__methode_protegee(params...)
```

Un peu d'héritage

- Class Fils(Parent)...
- Un fils peut appeler les méthodes de ses parents
  - Par exemple : `__init__`
- Pour l'héritage multiple :

`class F(P1,P2,P3):...`

```
>>> class Parent:
...     def __init__(self):
...         print "Bonjour du parent !"
...     def methode(self):
...         print "Methode du parent"
...
>>> class Fils1(Parent):
...     def __init__(self):
...         Parent.__init__(self) # Courant
...         print "Bonjour du fils !"
...     def methode(self):
...         print "Methode du fils"
...
>>> class Fils2(Parent):
...     pass
...
>>> class Fils3(Parent):
...     def methode(self):
...         print "Debut methode Fils3..."
...         Parent.methode(self)
...         print "Fin methode Fils3."
...
>>> parent = Parent()
Bonjour du parent !
>>> parent.methode()
Methode du parent
>>> fils1 = Fils1()
Bonjour du parent !
Bonjour du fils !
>>> fils1.methode()
Methode du fils
>>> fils2 = Fils2()
Bonjour du parent !
>>> fils2.methode()
Methode du parent
>>> fils3 = Fils3()
Bonjour du parent !
>>> fils3.methode()
Debut methode Fils3...
Methode du parent
Fin methode Fils3.
>>>
```

Dissection d'objets, reflexivité

- Un objet *s'apparente* à un dictionnaire
  - Clefs : Nom des attributs/méthodes
  - Valeurs : Attributs proprement dits, code des méthodes
  - On peut lister/ajouter/supprimer les méthodes/attributs depuis l'extérieur
    - Fonction `dir()` pour lister un objet
    - Fonctions `setattr`/`getattr` pour accéder aux méthodes/attributs
  - Au coeur de la sérialisation python (`pickle`). Permet aussi du RPC facile !

```
>>> class Toto:
...     def salut(self):
...         print "Salut"
...
>>> toto = Toto()
>>> toto.salut()
Salut
>>> dir(toto)
['__doc__', '__module__', 'salut']
>>> toto.blah = "Bonjour"
>>> dir(toto)
['__doc__', '__module__', 'blah', 'salut']
>>> toto.blah
'Bonjour'
>>> del toto.blah # Suppr
>>> dir(toto)
['__doc__', '__module__', 'salut']
>>> # Reference a la methode salut :
... ref_salut = toto.salut
>>> ref_salut()
Salut
>>> # Resolution par nom :
... methode_salut = getattr(toto, "salut")
>>> methode_salut()
Salut
>>> # Nouveaux attributs
... setattr(toto, "blah2", 42)
>>> dir(toto)
['__doc__', '__module__', 'blah2', 'salut']
>>> toto.blah2
42
>>>
```

Types de base : API objet

- Les listes sont des objets. Méthodes usuelles ci-contre
- Autres API pour les autres types standards (str, dict, ...)
  - Voir la ref python http://docs.python.org/library/
- En python >= 2.4, on peut hériter des types de base !

```
>>> lst = [1,2,3]
>>> lst.append(4) ; print lst
[1, 2, 3, 4]
>>> lst.extend([5,6,7,5,5]) ; print lst
[1, 2, 3, 4, 5, 6, 7, 5, 5]
>>> lst.insert(6,42) ; print lst
[1, 2, 3, 4, 5, 6, 42, 7, 5, 5]
>>> lst.remove(42) ; print lst
[1, 2, 3, 4, 5, 6, 7, 5, 5]
>>> lst.pop() ; print lst
5 [1, 2, 3, 4, 5, 6, 7, 5]
>>> print lst.index(5)
4
>>> print lst.count(5)
2
>>> lst.sort() ; print lst
[1, 2, 3, 4, 5, 5, 6, 7]
>>> lst.reverse() ; print lst
[7, 6, 5, 5, 4, 3, 2, 1]
>>>
```

Exceptions

- Comme en C++, les exceptions sont des classes normales
  - Préférer dériver des exceptions standards
- Utilisation similaire au C++

```
raise ExceptionType(params...)
…
try:
    ...code...
except ...:
    ...code... # Handler d'exception
else:
    ...code... # Pas d'exception
finally: # python >= 2.5
    ...code... # Dans tous les cas
```

- Affichage d'un *traceback* quand pas captées

```
>>> class MyExceptionType1(Exception):
...     pass
...
>>> class MyExceptionType2(Exception):
...     pass
...
>>> class MyExceptionType3(Exception):
...     pass
...
>>> class MyExceptionType4(Exception):
...     pass
...
>>> def f():
...     print "f leve une exception..."
...     raise MyExceptionType2("Bang !")
...
>>> try:
...     print "Avant f()"
...     f()
...     print "Fin du try sans exception"
... except MyExceptionType1:
...     print "Exception type 1"
...     raise
... except MyExceptionType2, ex:
...     # L'instance est : ex
...     print "Exception type 2:", ex
... except (MyExceptionType3,
...         MyExceptionType4), ex:
...     print "Exception type 3 ou 4:", ex
...     raise ex
... except Exception:
...     print "Standard exception"
...     raise
... except:
...     print "Strange exception..."
...     raise
... else:
...     print "Pas d'exception !"
... finally:
...     # python >= 2.5 seulement
...     print "Finally : dans tous les cas"
...
Avant f()
f leve une exception...
Exception type 2: Bang !
Finally : dans tous les cas
>>>
```

Interfaces standards

- Programmation par contrat : convention tacite en python 2.x, notion officielle en 3.x
  - Interfaces spécifiées pour accès dictionnaire, liste, etc.

Exemple : interface iterator (__iter__ / next)

```
>>> class Reverse:
...     "Permet de lire une chaine a l'envers"
...     def __init__(self, data):
...         self.data = data
...         self.index = len(data)
...     def __iter__(self):
...         return self
...     def next(self):
...         if self.index == 0:
...             raise StopIteration
...         self.index = self.index - 1
...         return self.data[self.index]
...
>>> for char in Reverse('Python'):
...     print "a l'envers:", char
...
a l'envers: n
a l'envers: o
a l'envers: h
a l'envers: t
a l'envers: y
a l'envers: P
>>>
```

Méthodes spéciales

http://docs.python.org/reference/datamodel.html

- Conversion vers str: `__str__` (”human readable”)
  - Version debug : `__repr__`
- Taille : `__len__`
- Comparaison : `__cmp__` ou `__eq__`/`__lt__`,...
- ”Destructeur” : `__del__`

```
1  class Panier:
2      def __init__(self, fruits):
3          self.__fruits = fruits
4      def __str__(self):
5          return "Panier de %d fruits: %s" \
6              % (len(self.__fruits),
7                 self.__fruits)
8      def __repr__(self):
9          return "Panier(%s)" \
10             % self.__fruits
11     def __len__(self):
12         return len(self.__fruits)
13     def __eq__(self, other): # op. '=='
14         print "Operator eq"
15         return self.__fruits == other.__fruits
16     def __del__(self):
17         print "dtor %s" % self
18
19 def test():
20     panier = Panier(["pomme", "poire"])
21     print str(panier)
22     print panier
23     print "Voici %s" % panier
24     print repr(panier)
25     print len(panier)
26     print panier == panier
27     print "Fin du test"
28     # dtor appele a la fin
29
30 test()
31 print "Fin du programme"
```

```
Panier de 2 fruits: ['pomme', 'poire']
Panier de 2 fruits: ['pomme', 'poire']
Voici Panier de 2 fruits: ['pomme', 'poire']
Panier(['pomme', 'poire'])
2
Operator eq
True
Fin du test
dtor Panier de 2 fruits: ['pomme', 'poire']
Fin du programme
```

Les classes sont aussi des objets

- On peut par exemple les passer à des fonctions
  - Exemple : pattern ”factory” facile en python
  - Exemple : fonction `isinstance()` pour vérifier si un objet est d'un type donné (ou un héritier)

```
>>> def factory_1_param(cls, param):
...     return cls(param)
...
>>> class Toto:
...     def __init__(self, param):
...         self.nom = param
...     def __str__(self):
...         return "Objet Toto '%s'" % self.nom
...
>>> objToto = factory_1_param(Toto, "une instance")
>>> print repr(objToto)
<__main__.Toto instance at 0x7f405191b710>
>>> print objToto
Objet Toto 'une instance'
>>> # objToto est-il de type Toto ?
... isinstance(objToto, Toto)
True
>>> # objToto est-il de type int ?
... isinstance(objToto, int)
False
>>>
```

Intermédiaires entre fonction et itérateur : les générateurs

- Générateur : fonction qui se souvient de son état d'un appel sur l'autre (~ coroutine)
  - `return` est remplacé par `yield`
- `yield` est obligatoirement dans la fonction générateur (pas dans une sous-fonction)

```
>>> def reverse(data):
...     for index in range(len(data)-1, -1, -1):
...         yield data[index]
...
>>> for char in reverse('Python'):
...     print "gen reverse : ", char
...
gen reverse :  n
gen reverse :  o
gen reverse :  h
gen reverse :  t
gen reverse :  y
gen reverse :  P
>>>
```

Exemple classique : xrange vs range

```
>>> for i in range(10000000):
...     pass
...
>>>
```

Créé une liste de 10M entiers : 346Mo de RAM sur mon test

```
>>> for i in xrange(10000000):
...     pass
...
>>>
```

Créé un générateur : quelques octets de RAM

Bibliothèque d'itérateurs pratiques : package `itertools`

Les ”propriétés”

- Comme les ”beans” java
  - Des attributs quand on y accède
  - Des méthodes `get`/`set` quand on les implémente
- La classe doit hériter de `object`

```
>>> class MaClasse(object):
...     def __init__(self, blah):
...         self.__blah = blah
...     def __get_blah(self):
...         print "Get de blah !"
...         return self.__blah
...     def __set_blah(self, blah):
...         print "Set blah a %s !" % blah
...         self.__blah = blah
...     def __del_blah(self):
...         print "Del de blah"
...         del self.__blah
...     # Definition de la propriete "blah" :
...     blah = property(__get_blah, __set_blah,
...                     __del_blah, "Doc blah")
...
>>> instance = MaClasse(42)
>>> print instance.blah
Get de blah ! 42
>>> instance.blah = 42
Set blah a 42 !
>>> help(instance) # Affiche "Doc blah" pour blah
>>> del instance.blah
Del de blah
>>>
```

Bilan

- Types standards :
  - Scalaires (None/bool/int/long/float) : opérateurs classiques. Entiers à précision illimitée
  - Types complexes :
    - Modifiables : list, dict (non ordonnés), set (non ordonnés)
    - Non-modifiables : str, tuple
- Fonctions : docstrings, variadiques, paramètres par défaut, nommés. Construction lambda, scoping intuitif, passage par référence des types complexes
- Objets : pas de destructeur, introspecition possible, pattern ”factory” facile, les classes sont des objets, interfaces conventionnelles list/dict/iter/..., méthodes spéciales `__str__`, `__cmp__`, …, propriétés
  - Exceptions : proche du C++
- Générateurs : expression `yield`. Préférer `xrange`

Non évoqués

- Implémentation des fonctions variadiques
- Décorateurs (`@staticmethod`, `@property`, ...)
- Attributs et méthodes de classe (`static` en C++)

Programmation modulaire

- Un module python = un fichier `.py`
- Utiliser un autre module :
  - `import autre_module, encore_un_autre_module..`
    - Le ”symbole” `toto` de `autre_module` peut alors être accédé en tant que `autre_module.toto`
  - `from autre_module import toto, titi`
    - Le ”symbole” `toto` de `autre_module` peut alors être accédé en tant que ”`toto`” directement, idem pour `titi`
    - Les autres symboles de `autre_module` ne sont pas importés, à moins d'utiliser d'autres `import`
  - `from autre_module import *`
    - Idem, mais avec `toto`, `titi`, ..., et **tous** les symboles publiés par `autre_module`
    - Tentant mais … **à déconseiller** ! (perd l'intérêt de la notion de *namespace*)

Exemples avec ”import”
(par ordre de préférence personnelle)

`import autre_module`

```
>>> import sys
>>> sys.prefix
'/usr'
>>> sys.path
['', '/usr/lib/python2.6', ...]
>>> sys.version
'2.6.2 (release26-maint, Apr 19 2009, 01:58:18) \n[GCC 4.3.3]'
>>>
```

`from autre_module import symboles...`

```
>>> from sys import prefix, version
>>> prefix
'/usr'
>>> path
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'path' is not defined
>>> sys.path
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'sys' is not defined
>>> version
'2.6.2 (release26-maint, Apr 19 2009, 01:58:18) \n[GCC 4.3.3]'
>>>
```

`from autre_module import *`

```
>>> # Fortement deconseille !
... from sys import *
>>> prefix
'/usr'
>>> path
['', '/usr/lib/python2.6', ...]
>>> version
'2.6.2 (release26-maint, Apr 19 2009, 01:58:18) \n[GCC 4.3.3]'
>>> platform
'linux2'
>>> byteorder
'little'
>>>
```

Import et __name__ == ”__main__”

- Quand on importe un module, il est interprété
  - Si il contient du code, il est exécuté !
  - Si on ne veut exécuter du code que quand on exécute le module et pas quand on l'importe :

```
if __name__ == ”__main__”:
    ...code...
```

  - Bon pour des tests unitaires sans douleur !
    - Au passage : la construction `assert` est votre amie !

```python
import tst_mod2

def fonction1():
    return 42

def fonction2():
    return tst_mod2.fonction3()

print "Blah blah... Initialisation mod1."

if __name__ == "__main__":
    print "test mod1..."
    assert 42 == fonction1()
```

```python
import tst_mod1

def fonction3():
    return "f3"

def fonction4():
    return tst_mod1.fonction1()

print "Blah blah... Initialisation mod2."

if __name__ == "__main__":
    print "test mod2..."
    assert "f3" == fonction3()
```

```
[babasse] svn/pollin1/2009/intro_python>python ./tst_mod1.py
Blah blah... Initialisation mod1.
Blah blah... Initialisation mod2.
Blah blah... Initialisation mod1.
test mod1...
[babasse] svn/pollin1/2009/intro_python>python ./tst_mod2.py
Blah blah... Initialisation mod2.
Blah blah... Initialisation mod1.
Blah blah... Initialisation mod2.
test mod2...
[babasse] svn/pollin1/2009/intro_python>
```

Symboles d'un module

- Pour les connaître :
  - `import autre_module ; print dir(autre_module)`
  - Cas particulier : `dir()` pour connaître l'environnement du module courant
  - Et pour l'aide, c'est `help()` ou le programme `pydoc` !
- Un module peut forcer la liste des symboles qu'il exporte pour `from lemodule import *`:
  - `__all__ = ['symbole1', 'symbole2', …]`

Packages

- But : regrouper les modules afin de hiérarchiser l'espace de nommage (namespace)
  - Pouvoir écrire :

    ```
    import pkg.subpkg.subsubpkg.module
    ```

    Pour utiliser `toto` de `module`, on doit ensuite écrire :

    ```
    pkg.subpkg.subsubpkg.module.toto
    ```

  - On préfère donc en général :

    ```
    from pkg.subpkg.subsubpkg import module
    ```

    toto s'accède ensuite par : `module.toto`

  - Toute autre combinaison possible :

    `from pkg import subpkg` puis `subpkg.subsubpkg.mdule.toto`

Packages et répertoires

- Un package = un répertoire contenant un fichier `__init__.py`
  - `__init__.py` exécuté à chaque import

```
.
|-- pkg
|   |-- __init__.py
|   |-- pkg_mod1.py
|   |-- pkg_mod2.py
|   |-- subpkg
|   |   |-- __init__.py
|   |   |-- subpkg_mod1.py
|   |   |-- subsubpkg
|   |   |   |-- __init__.py
|   |   |   |-- autre_module.py
|   |   |   `-- module.py
|   |   `-- subsubpkg2
|   |       |-- __init__.py
|   |       `-- module.py
|   `-- subpkg2
|       |-- __init__.py
|       `-- subpkg2_mod1.py
`-- pkg2
    |-- __init__.py
    |-- mod.py
    `-- subpkg
        |-- __init__.py
        `-- mod.py
```

Packages et modules standards dans
`<prefix>/lib/pythonX.Y/site-packages`

Chemin vers modules externes via
La variable d'environnement
`PYTONPATH`

Import ... as ...

- On peut *renommer* des modules ou symboles importés :

```
import cPickle as pickle
```

  - `cPickle.loads()` s'utilise alors en tant que `pickle.loads()`

- Autre exemple :

```
from sys import prefix as PREFIX
```

  - `sys.prefix` s'utilise alors en tant que `PREFIX`

- Plus compliqué :

```
from bidule.machin.chose import truc as LeTruc
```

Exemples CMS

- `import FWCore.ParameterSet.Config as cms`
  - L'objet **ou** module "`Config`" s'appelle "`cms`"
- `from Configuration.Examples.RecoExample_GlobalTag_cfg import process`
  - L'objet ou module "`RecoExample...`" s'appelle "`process`"
- `from IOMC.GeneratorInterface.data import pythiaDefault_cff`
- `import IOMC.GeneratorInterface.pythiaDefault_cff as defaults`
- `from SimCalorimetry.HcalSimProducers.hcalSimParameters_cfi import *`

Quelques détails sur les modules

- On peut importer des modules n'importe où
  - Pas seulement en tête de fichier → même dans un bloc `if` !
  - Utile pour limiter la ”pollution” de l'environnement global
- Un module est un objet
  - On peut les manipuler dans des fonctions
  - Il y a une version ”programmatique” de import !
    - `import imp ; imp.load_source(”toto.py”)`

Modules standards principaux

- `__builtins__` (importé par défaut)
  - Fournit les fonctions standards `dir, help, min, max, reduce, list, int, long, dict, isinstance, map, enumerate, …`
  - Pour ouvrir des fichiers : `open()`
- `sys` : accès à l'environnement d'exécution python + arguments du programme (`sys.argv`)
- `os` et `os.path` : intéraction avec le système, manipulation de chemin de fichiers
- `optparse`: parsing d'options ligne de commande
- `doctest` : tests unitaires via les docstrings

Autres modules utiles

- `re` : expressions rationnelles (regex)
- `itertools`: générateurs pratiques
- `pickle/cPickle` : sérialisation des objets (tous ?)
- `copy`: copie récursive de structures
- `urllib2` : accès HTTP, FTP, ...
- `SimpleHTTPServer` : comme son nom l'indique
- `xmlrpclib/SimpleXMLRPCServer` : XML-RPC en 5 minutes

serveur

```
1  import xmlrpclib
2  from SimpleXMLRPCServer import SimpleXMLRPCServer
3
4  def is_even(n):
5      return n%2 == 0
6
7  server = SimpleXMLRPCServer(("", 8000))
8  print "Listening on port 8000..."
9  server.register_function(is_even, "is_even")
10 server.serve_forever()
```

client

```
1 import xmlrpclib
2
3 proxy = xmlrpclib.ServerProxy("http://localhost:8000/")
4 print "3 is even: %s" % str(proxy.is_even(3))
5 print "100 is even: %s" % str(proxy.is_even(100))
```

- `sqlalchemy`: bases de données relationnelles... du *ruby on rails* à la sauce python !
- `scipy, numpy, matplotlib, pyROOT, ...` : Sciences, plots
- `PyQt, PyQwt, pyGTK, wxPython, ...` : GUI

Quelques bonnes pratiques (suggestions personnelles)

- Des commentaires !
  - Documentation (+ conventions doxygen c'est mieux !)
  - Annotations de type pour les prototypes de fonctions (python >= 2.6)
- Docstrings à tous les étages
  - Fonctions/méthodes
  - Classes
  - Modules
  - `doctest` pour les tests unitaires quand ça a un sens

Un exemple complet

```python
"""
Un module qui contient la definition de la fonction eratosthene,
implementation du script du meme nom.
"""

def eratosthene(nmax):
    """
    Renvoie le crible d'eratosthene jusqu'a nmax.

    Tests unitaires :

    >>> eratosthene(-1)
    Traceback (most recent call last):
    ...
    TypeError: nmax doit etre >= 0
    >>> eratosthene(0)
    []
    >>> eratosthene(1)
    [1]
    >>> eratosthene(2)
    [1, 2]
    >>> eratosthene(127)
    [1, 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41,\
 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97, 101,\
 103, 107, 109, 113, 127]
    >>> eratosthene(8.2)
    [1, 2, 3, 5, 7]
    """

    # Le resultat
    crible = []

    # Test arguments
    if nmax < 0:
        raise TypeError("nmax doit etre >= 0")

    # Cas particulier du zero
    if nmax == 0:
        return []

    # On balaye les entiers jusqu'a nmax
    for entier in xrange(2, int(nmax)+1):
        est_premier = True

        # On balaye le crible
        for untest in crible:
            if untest**2 > entier: # Puissance 2
                # Pas besoin d'aller au-dela
                # Entier est bel et bien premier
                break
            elif entier % untest == 0:
                # untest divise entier
                # on passe a l'entier suivant
                est_premier = False
                break

        # Si personne dans le cribe ne le divise...
        if est_premier:
            crible.append(entier)

    # On rajoute 1 artificiellement a la liste
    return [1] + crible


def _unit_tests():
    """Tests unitaires automatiques"""
    import doctest
    doctest.testmod()


if __name__ == "__main__":
    # On lance les tests unitaires
    _unit_tests()
    # Utiliser l'option -v pour les details
```

Un exemple complet

```python
from eratosthene import eratosthene


# Des fonctions qui utilisent eratoshene :
# ...d'autres morceaux de code ici...
# Voila

if __name__ == "__main__":
    import sys
    from optparse import OptionParser

    # Parse les arguments de la ligne de commande
    parser = OptionParser(usage = "%prog entier_max")
    parser.add_option("-d", "--doc",
                      help   = "Documentation de la fonction eratosthene",
                      action = "store_true")
    (options, args) = parser.parse_args()

    # Reagit a l'option -d/--doc
    if options.doc:
        help(eratosthene)
        # Termine le programme (exit code 0)
        sys.exit(0)

    # S'assure qu'il y a bien 1 seul argument
    if len(args) != 1:
        parser.error("Parametre invalide")

    # Appelle la fonction et affuche le reultat sous forme jolie
    premiers = eratosthene(int(sys.argv[1]))
    print "%d nombres premiers jusqu'a %s :" % (len(premiers), sys.argv[1])
    for p in premiers: print "  %5d" % p
```

Plus d'infos

- Tutoriel pour commencer :
  - http://docs.python.org/tutorial/index.html (ou ces slides...)
- Doc bibliothèque standard :
  - http://docs.python.org/library/index.html
- Le reste de la doc officielle :
  - http://docs.python.org/
- Autres librairies (futur standard ?) :
  - http://pypi.python.org (python cheeseshop)
- Bouts de code, exemples :
  - http://code.activestate.com/recipes/langs/python/ (python cookbook)

Note : Installer une librairie

- En général, fournies avec un script `setup.py` :
  - `python ./setup.py build`
  - `python ./setup.py install # Eventuellement --prefix=...`
  - Plus d'infos : http://docs.python.org/install/index.html
- Créer votre `setup.py` :
  - http://docs.python.org/distutils/index.html
- Ou utiliser des `.egg` !

Note : debugger du python

- Moins de choses ”bizarres” qu'en C/C++ !
  - En général : des erreurs bêtes d'algorithmique, faciles à comprendre
- Tests unitaires avec `doctest`
- Utiliser des `assert` aux endroits critiques
- Ajouter des `print` jusqu'à comprendre
- Utiliser le module ”`pdb`” (`python -m pdb`...)
  - Intéraction similaire à `gdb`
- Fuites de mémoires étranges :
  - `import gc` + regarder de près `gc.garbage`
- Utiliser `valgrind` avec les suppressions python

Alors ?...

- Que fait :

```
1 def f(l):
2     return reduce(lambda x,y: x*(y+1)/2, \
3                   [x+y+1 for x,y in enumerate(l)], 1)
```

```
1  >>> f(range(1))
2  1
3  >>> f(range(2))
4  2
5  >>> f(range(3))
6  6
7  >>> f(range(4))
8  24
9  >>> f(range(5))
10 120
11 >>> f(range(6))
12 720
13 >>> f(range(7))
14 5040
15 >>> f(range(8))
16 40320
17 >>>
```

C'est `factorielle()` codée avec les pieds !

Ça utilise visiblement des listes. Essayons avec des range()...

D'abord, que fait enumerate ?...

```
1 >>> list(enumerate(['a', 'b', 'c', 'c']))
2 [(0, 'a'), (1, 'b'), (2, 'c'), (3, 'c')]
3 >>>
```

En essayant de traduire à la main :

```
1  def f(l):
2      resultat = 1
3      for x,y in enumerate(l):
4          resultat = resultat * ((x+y+1)+1)/2
5          # En fait : x == y puisqu'on utilise
6          # des range() en entree !
7          assert x == y
8          # Donc on a en realite :
9          #   resultat = resultat * (2*y+2)/2
10         # cad :
11         #   resultat = resultat * (y+1)
12     return resultat
```

The Zen of Python (Tim Peters)

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!