

# Chapitre : 4 Types évolués

Yves Guidet pour Édugroupe

V1.4.8 May 3, 2017

# Types évolués (structurés)

Nous allons maintenant étudier un certain nombre de types, chaînes de caractères, listes, *tuples*, dictionnaires, ainsi que les *structures de contrôle* permettant leur manipulation.

- ▶ Généralités

```
s = "Bonjour"  
s = 'Bonjour'  
s = "Je m'appelle Yves"  
s = 'Il lui a dit : "Bonjour"'
```

- ▶ non modifiables !
- ▶ caractères spéciaux et *raw strings*
- ▶ formats *cf infra*

# Les chaînes de caractères ne sont pas modifiables

```
>>> s = 'yves'
>>> s[0] = 'Y'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

On verra plus loin comment faire.

Il ne s'agit pas d'un nouveau type de chaîne, mais d'une notation pratique pour les constantes ; cette notation sera souvent utilisée lors de la manipulation d'*expressions régulières*.

Considérons :

```
>>> s = "bonjour\n"
```

```
>>> print s
```

```
bonjour
```

```
>>>
```

La chaîne contient un `\n` et on passe bien à la ligne au moment du *print*.

## *raw strings*(suite)

Si maintenant on ne **veut pas** que cet `\n` soit interprété, il faut protéger le `\` par un deuxième que l'on met devant (je suis clair ?).

```
>>> s = "bonjour\\n"
>>> print s
bonjour\n
>>>
```

Ce n'est pas toujours pratique, et c'est là qu'interviennent nos *raw strings* :

```
>>> s = r"bonjour\n"
>>> print s
bonjour\n
>>>
```

```
>>> print 'Bonjour %s' % "Python"
Bonjour Python
```

On trouvera un autre exemple dans Swinnen  
(<http://inforef.be/swi/python.htm>) p138.  
Utiliser plutôt la méthode *str.format()*, plus récente.

Un nouveau format est apparu récemment, avec des accolades, voir ici <http://tinyurl.com/czcyyw3>.

```
format(...)  
    format(value[, format_spec]) -> string
```

Returns value.\_\_format\_\_(format\_spec)  
format\_spec defaults to ""



## format : un exemple

```
parents, babies = (1, 1)
while babies < 100:
    print 'This generation has {0} babies'.format(babies)
    parents, babies = (babies, parents + babies)
```

On pourrait enlever le *0*.

## format : un autre exemple avec enumerate

```
friends = ['john', 'pat', 'gary', 'michael']  
for i, name in enumerate(friends):  
    print "iteration {iteration} is {name}".format(iteration=i, name=name)
```

# Formats : un exemple récapitulatif

Reprenons (on a supprimé quelques lignes) :

```
4 nom = 'Yves'
5
6 age = 63
7
10 print "{} a {} ans" .format(nom, age)
11
12 print "{0} a {1} ans, il ne fait pas son âge, {0}"
.format(nom, age)
13
14
15 print "{n} a {a} ans, il ne fait pas son âge, {n}"
.format(n=nom, a=age)
```

J'ai connu des gens tellement déformés par Fortran ou C qu'ils n'hésitent pas à écrire :

```
for i in range(len(l)):  
    process(i, l[i])
```

Si vous avez absolument besoin d'un indice de boucle, écrivez :

```
for (i, li) in enumerate(l):  
    process(i, li)
```

c'est plus idiomatique et plus efficace.

## ► encodage

```
>>> uni=u'je vais être encodée'.encode('iso-8859-15')
>>> uni
'je vais \xeatre encod\xe9e'
>>> type(uni)
<type 'str'>
```

# Unicode (suite)

## ► décodage

```
>>> classik="je vais être décodée"
>>> classik.decode('iso-8859-1')
u'je vais \xc3\xaatre d\xc3\xa9cod\xc3\xa9e'
>>> print classik
je vais être décodée
>>> unicode(classik, 'iso-8859-1')
u'je vais \xc3\xaatre d\xc3\xa9cod\xc3\xa9e'
```

Noter que *unicode* est une fonction intégrée, pas une méthode.

## boucle *for* (strings)

Nous n'avons pas encore parlé des structures de contrôle, mais cela nous aidera à y voir plus clair.

La boucle *for* **énumère**. Dans les cas des chaînes, cela donne :

```
>>> s = "Antoine"
>>> for c in s:
...     print c
...
A
n
t
o
i
n
e
>>>
```

## boucle *for* (strings) (suite)

Avec des caractères accentués, tout va mal :

```
>>> s = "G  rard"
>>> for c in s:
...     print c
...
G
?
r
a
r
d
```



## boucle *for* (strings) (suite et fin)

Avec une chaîne Unicode, tout va bien.

```
>>> s = u"G  rard"
>>> for c in s:
...     print c
...
G
  
r
a
r
d
>>>
```

# La fonction *len* et ses pièges

Dans l'exemple du précédent transparent, on pourra vérifier que la fonction *len* donne un résultat correct avec la chaîne *uni*, mais pas avec la chaîne *str*.

Voir aussi Swinnen : - <http://inforef.be/swi/python.htm> - l'exemple avec René et Georges.

- ▶ comme les listes

```
>>> moi='yves'  
>>> moiAussi= 'Y' + moi[1:]  
>>> moiAussi  
'Yves'  
>>>
```

On reviendra sur ces « tranches » lors de l'étude des listes.

## « triples quotes »

```
un_texte = """Ce texte peut comporter des "doubles quotes",  
ou des 'simples quotes'.  
Les caractères d'échappement comme \\n (retour  
à la ligne) ou \\t (tabulation) doivent  
commencer par un anti-slash."""
```

- ▶ + : opérateur de concaténation (C++, Java, ...)

```
>>> a = "Hello"  
>>> b = "Python"  
>>> print a + b  
HelloPython
```

- ▶ \* : opérateur de répétition

```
>>> s='aïe'  
>>> print 5*s  
aïeaïeaïeaïeaïe  
>>>
```

- ▶ `int()` : conversion en entier. Provoque une erreur si cela n'est pas possible.
- ▶ `long()` : conversion en long.
- ▶ `str()` permet de transformer la plupart des variables d'un autre type en chaînes de caractères.
- ▶ `float()` : permet la transformation en flottant.

# La méthode *split*

On a aperçu cette fonction dans Saint.py.

Elle découpe la chaîne en une liste de chaînes, le premier argument étant un séparateur.

```
>>> s = 'Yves aime Python'
>>> s.split ()
['Yves', 'aime', 'Python']
>>> s.split ('e')
['Yv', 's aim', ' Python']
```

Sans argument, on utilise un blanc. La méthode *join* «< recolle »> tout. Il existe surtout `re.split()` beaucoup plus puissante.

Il s'agit bien d'une *méthode* de *str* ; attention elle n'insère pas de blanc par défaut.

```
>>> s = ''
>>> s.join(('Python', 'est', 'grand'))
'Pythonestgrand'
```

Autre essai :

```
>>> ' '.join(['join', 'est', 'tordu'])
'join est tordu'
```

Ici j'ai  $\ll$  joint  $\gg$  un *tuple*, ce peut être n'importe quel *énérable* :

Help on method\_descriptor in str:

```
str.join = join(...)
  S.join(iterable) -> string
```

Return a string which is the concatenation of the strings in the iterable. The separator between elements is S.



# Plus sur les string

Voir ici : <http://docs.python.org/2/library/string.html>.  
En particulier on aura besoin plus loin de *replace* :

```
str.replace = replace(...)
    S.replace(old, new[, count]) -> string
```

Return a copy of string S with all occurrences of substring old replaced by new. If the optional argument count is given, only the first count occurrences are replaced.

Mais ce sera beaucoup plus drôle avec les *expressions régulières*. Voir quand même *index* est ses acolytes dans ENI, ou dans la doc Python sur le web.

Conversions : écrire un programme qui demande à l'utilisateur la saisie de a et b et affiche la somme de a et de b.

Conversions (bis) : écrire un programme qui demande à l'utilisateur son année de naissance et qui affiche son âge. L'année courante sera « codée en dur » dans une variable.

## exercice futur (première personne)

- ▶ écrire un script demandant un verbe en -er ou -ir et le conjuguant à la première personne du futur

```
python futur.py  
Verbe ? programmer  
je programmerai
```

# Et maintenant ?

Avant de nous lancer dans listes, tuples et dictionnaires, nous avons besoin de faire une digression sur les structures de contrôle (alternatives et boucles) et pour cela nous devons en savoir un peu plus sur ce que Python entend par vrai ou faux.

- ▶ les opérateurs logiques
  - ▶  $X \text{ or } Y$  (on n'évalue que ce qui est nécessaire)
  - ▶  $X \text{ and } Y$  (idem)
  - ▶  $\text{not } X$

Attention la négation n'est pas  $\ll ! \gg$  comme en C ou en shell.

# Opérateurs de comparaison

- ▶ sur tous les types de base

<

>

<=

>=

== égal

!= différent

<> idem

X is Y : X et Y représentent le même objet.

X is not Y : X et Y ne représentent pas le même objet

- ▶ Enchaînement les opérateurs :  $X < Y < Z$

Si l'on compare 1953 et '1953', on pourrait s'attendre à une erreur de typage, mais il n'en est rien !

```
>>> 1953 != '1953'
```

```
True
```

```
>>> 5 < 'cinq'
```

```
True
```

En fait un numérique est toujours inférieur à une *str*.

```
>>> a=1
>>> b=a
>>> a is b
True
>>> a=9
>>> a is b
False
>>> a=b
>>> a is b
True
>>> del b
>>> a
1
```



## is (suite et fin)

Encore plus fort :

```
>>> from math import *
>>> type(cos)
<type 'builtin_function_or_method'>
>>> f = cos
>>> type(f)
<type 'builtin_function_or_method'>
>>> f(0)
1.0
>>> f is cos
True
```

- ▶ Suite d'instructions ; alignement sur la même tabulation.
- ▶ if, while et for ; fonctions.

Rappelons qu'il n'y a pas d'accolades en Python, l'indentation décide de tout, les exemples concrets arrivent !

Dans les transparents suivants nous allons étudier :

- ▶ if, while, for
- ▶ break, pass

Commençons par *pass*, qui ne fait ... rien (comme passer au poker).

```
if condition:
    pass
else:
    instruction ...
```

La structure de contrôle *if* (alternative) :

- ▶ simple

```
a = 11
if a > 10 :
    print "a est plus grand que dix"
```

Bien sûr il existe une forme avec *else* :

```
if a > 10 :  
    print "a est plus grand que dix"  
else:  
    print "a n'est pas plus grand que dix"
```

Et comme dans la plupart des langages de script il y a un *else ... if* appelé en Python *elif* :

```
if a > 10 :  
    print "a est plus grand que dix"  
elif a == 10:  
    print "a est égal à dix"  
else:  
    print "a est plus petit que dix"
```

## Exercice << température >>

- ▶ Ecrire un script demandant la température extérieure et affichant << chaud >> ou << froid >> si elle est supérieure ou inférieure à 20 degrés.
- ▶ Dans l'exercice précédent, ajouter une tranche << parfait >> entre 20 et 25 degrés.



La plus classique des structures de contrôle répétitives.

```
i = 0
while i<5:
    i += 1
    print i,
```

Noter la curieuse virgule. Elle signifie simplement qu'on ne passe pas à la ligne après le *print*.

En Python3, il faut écrire :

```
print(i, end=' ')
```

Parcourt une liste (ou un *tuple*, une chaîne, les lignes d'un fichier, les clés d'un dictionnaire) :

```
>>> for x in ['maths', 'physique', 'programmation']:
...     print x, " ? J'aime"
...
maths ? J'aime
physique ? J'aime
programmation ? J'aime
```

La fonction *range* génère une liste d'entiers.

```
for i in range(5):  
    if i==3:  
        break  
    print i
```

Avec le *break* peu de chance d'arriver à 4 ...

En savoir plus : *help (range)* : voir aussi plus bas.

```
maListe = [ 29, 11, 1953 ]  
maListe[ 2 ] = 1972  
maListe.append( 75 )  
maListe += ( 75, )  
  
for item in maListe:  
    print item
```

- Remarque : *idem* pour les caractères d'une chaîne

# l'opérateur in

```
>>> maListe = [ 29, 11, 1953 ]
>>> 29 in maListe
True
>>> x=1948
>>> x in maListe
False
>>>
```

## *len* et *range* : deux fonctions utiles

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(15,21)
[15, 16, 17, 18, 19, 20]
>>> range(0,1000,100)
[0, 100, 200, 300, 400, 500, 600, 700, 800, 900]
>>> range(0,-10,-1)
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>>

>>> langages = ['Python', 'Perl', 'Java']
>>> len(langages)
3
```

- ▶ on part de la fin !

```
>>> langages = ['C', 'Perl', 'Java', 'shell']
>>> langages[-4]
'C'
>>> langages[-2]
'Java'
>>> langages[-1]
'shell'
>>>
```

## listes : tranches

```
>>> langages = ['C', 'Perl', 'Java', 'shell' , 'Ruby']
>>> langages[0:2]
['C', 'Perl']
>>> langages[0:3]
['C', 'Perl', 'Java']
>>> langages[0:]
['C', 'Perl', 'Java', 'shell', 'Ruby']
>>> langages[:]
['C', 'Perl', 'Java', 'shell', 'Ruby']
>>> langages[1:]
['Perl', 'Java', 'shell', 'Ruby']
>>> langages[1:-1]
['Perl', 'Java', 'shell']
>>>
```

On peut même ajouter un pas : voir dans la littérature.



# Listes : append, extend, insert

On ira lire avec profit : <http://tinyurl.com/ovzamnx> ainsi que le manuel habituel

<http://docs.python.org/2/tutorial/datastructures.html>

On pourra ainsi utiliser les listes pour implémenter des piles ou des files.

# Tuples

- ▶ genre de liste
- ▶ éléments *non modifiables*

Prononcer *touple* (<http://tinyurl.com/oyqf6qu>)

```
>>> t=("Yves", 29, 11)
```

```
>>> t
```

```
('Yves', 29, 11)
```

```
>>> type(t)
```

```
<type 'tuple'>
```

```
>>> t[0]
```

```
'Yves'
```

```
>>> t[0] = 'Yves'
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
TypeError: 'tuple' object does not support item assignment
```

► concaténation

```
>>> t += (1953)
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
TypeError: can only concatenate tuple (not "int") to tuple
```

```
>>> t += (1953,)
```

# Tuples (fin)

## ► parcours d'un tuple

```
>>> for i in t:  
...     print i  
...  
Yves  
29  
11  
1953  
>>>
```

- hashes, maps ...

```
>>> monDico = { "lundi" : "monday", "mardi" : "wednesday"}  
>>> monDico ["mardi"]  
'wednesday'  
>>> monDico ["mardi"] = 'tuesday'  
>>> monDico  
{'mardi': 'tuesday', 'lundi': 'monday'}
```

## dictionnaires (suite)

- comment itérer sur les clés ?

```
>>> for item in monDico.keys():  
...     print monDico [item]  
...  
tuesday  
monday  
>>>  
monDico.values()  
['tuesday', 'monday']
```

La méthode *items()* retourne l'ensemble des couples.

En fait on peut faire l'économie de l'appel à la méthode *keys()* et écrire :

```
>>> for item in monDico:
```

Un **gros** piège : il n'y a aucun ordre sur les éléments (ni sur les

Réalisés par *in* et *del*, comme ici :

```
>>> monDico = { "lundi" : "monday", "mardi" : "wednesday"}
>>> 'lundi' in monDico
True
>>> del monDico['lundi']
>>> 'lundi' in monDico
False
```

# l'exception KeyError

```
>>> monDico ["jeudi"]  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
KeyError: 'jeudi'  
>>>
```



- ▶ Ecrire un script lisant un jour de la semaine au clavier et le traduisant en anglais.
- ▶ reprendre la conjugaison au futur, mais à toutes les personnes
- ▶ et au présent ?

Il convient :

- ▶ de déplacer ces transparents vers le chapitre « Modules »
- ▶ de le compléter (en attendant on consultera mypdf.pdf)

Il ne s'agit pas à proprement parler d'un type mais d'un module permettant de manipuler des chaînes, comme on le fait en Unix (*grep* et *sed*).

# Les expressions régulières

- ▶ Unix : vi, sed, grep, egrep, sed, ...
- ▶ Perl

# Avant d'aller plus loin

Pour info motif est la traduction de « *pattern* ».

# Rappel sur les 'raw strings'

```
>>> exp = '\bprint\b'  
>>> print exp  
print  
>>> exp = '\\bprint\\b'  
>>> print exp  
\\bprint\\b  
>>> exp = r'\bprint\b'  
>>> print exp  
\bprint\b
```

```
>>> import re
>>> p = re.compile ('[ei]r$')
>>> ligne = 'programmer'
>>> print p.search (ligne)
<_sre.SRE_Match object at 0xb7e906e8>
>>> ligne = 'joindre'
>>> p.search (ligne)
>>> print p.search (ligne)
None
```

- ▶ None -> False (dans un *if*)
- ▶ voir aussi findall (transparent suivant).

```
>>> motif = re.compile ('[A-Z]')  
  
>>> motif.findall ('Yves adore Python')  
['Y', 'P']
```



# Substitution

```
>>> import re
>>> x = re.sub (r'..$', '', 'programmer')
>>> x
'programm'
>>>
```

- ▶ Ecrire un script cherchant le mot `print` sur l'entrée standard, et imprimant les lignes trouvées avec leur numéro.
- ▶ (un genre de `grep -wn`)

Hint : *for lines in sys.stdin*

Dans l'exercice « futur », utiliser les regexps pour :

- ▶ tester que le verbe est d'un des bons groupes (on quittera par `sys.exit` ou on lèvera une exception)
- ▶ ne pas écrire « je » mais j' quand le radical commence par une voyelle

# Exercice re + fichier

- ▶ Ecrire un script cherchant le mot print dans un fichier, et imprimant les lignes trouvées avec leur numero.
- ▶ (un genre de grep -wn)

A faire quand on aura vu les E/S sur fichier :(