Introduction à la programmation en Python

Release 0.1

Christian Poli

December 01, 2015

CHAPTER

ONE

LICENCE

CHAPTER

TWO

INTRODUCTION

Cette introduction au langage Python s'adresse à un public pratiquant déjà autre langage de programmation (Java, C/C++, perl etc.).

Tout en s'appuyant sur les points communs entre Python et les autres langages, on se propose ici de faire ressortir les spécificités de Python et d'introduire, aussi tôt que possible au cours de l'exposé, les idiomes et les bonnes pratiques permettant d'écrire des programmes lisibles et efficaces, correspondant à l'esprit de ce langage.

Connaître déjà un langage de programmation est un atout certain pour en apprendre un nouveau, au moindre coût.

Par contre, la tentation peut être grande (surtout quand on est obligé d'aller vite) d'écrire du code dans le nouveau langage en faisant confiance au réflexes acquises avec les langages déjà pratiqués.

Le bon, tout comme le mauvais coté de cette pratique est qu'en général ça marche: on peut écrire du Python opérationnel qui ressemble à du C ou du Perl mais il sera probablement inefficace, inélégant et difficile à maintenir et à partager. Car même sans être ancré dans un paradigme particulier, Python dispose de ses propres idiomes.

Ces idiomes sont souvent présentés comme des "concepts avancés", à l'intention des développeurs avertis.

Ce n'est pas cette vision de l'apprentissage de Python qui est développée ici : au contraire, on part du principe que les idiomes doivent être introduits en même temps que les concepts de base afin de faire adopter les bons réflexes avant que les mauvaises habitudes s'installent.

Cette introduction ne se veut pas exhaustive

Elle ne se substitue pas à la documentation de référence et sa lecture ne dispense pas de la consulter.

Comme il arrive souvent dans la pratique, les notions présentées ici ne découlent pas les unes des autres au sens unique. Par exemple, il est difficile d'expliquer les types de base sans évoquer les fonctions et réciproquement. Afin de briser des tels cycles, il est souvent utile d'introduire une notion en deux temps: d'une manière succincte, voire incomplète d'abord et de manière détaillée ultérieurement.

C'est pour cette raison, entre autres, que la pratique l'un autre langage facilite grandement l'utilisation efficace de ce support.

2.1 Python en quelques phrases

Les langages sont souvent classifiés en fonction de leux expressivité et de leur indépendance par rapport à la couche matérielle. Ainsi on parle de langages de :

- **"bas niveau"** expressivité faible, grande dépendance par rapport à l'architecture de la machine (assembleur,...)
- **"haut niveau"** bonne expressivité mais une certaine dépendance rapport à la couche matérielle subsiste (la majeure partie des langages)
- **"très haut niveau"** langages expressifs, offrant des mécanismes d'abstraction avancées et l'indépendance par rapport aux contraintes de l'architecture de la machine.

Selon cette classification, Python fait partie des langages dits de "très haut niveau" car:

- Il est expressif, modulaire, multi-paradigme (impératif, objet, fonctionnel)
- Encourage l'écriture d'un code de qualité, lisible et compact (indentation obligatoire, mécanismes de documentation intégrés au langage)
- Il prend en charge les aspects "bas niveau" de la programmation, disposant d'un ramasse miettes (*garbage collector*), et de types prédéfinis de haut niveau (entiers de taille arbitraire, listes, dictionnaires, ensembles etc.)
- Met en oeuvre des mécanismes favorisant l'abstraction : décorateurs, générateurs (détaillés dans la suite du présent document).

Python est également:

- Disponible sous licence open source (**Python Software Foundation License**), compatible avec la **GPL** pour les versions >=2.0 de Python
- Existe sur différents OS, supporte bien l'intégration avec d'autres langages et fait l'objet de plusieurs implémentations:
  - **Cpython:** implémentation de référence (en **C**), la plus répandue
  - **Pypy:** projet *R&D* visant une implémentation haute performance
  - **Jython:** implémentation exécutable sur la machine virtuelle **Java(TM)**
  - **IronPython:** implémentation pour **.NET**
  - **Stackless Python:** variante de **Python**, implémentant des **micro-threads**
- Son typage est fort, mais dynamique (**duck typing**)

Duck typing

Un objet est du "bon type" dans un contexte donné s'il dispose des propriétés nécessaires et non pas en fonction de son "étiquette":

"Quand je vois un oiseau marcher comme un canard, nager comme un canard et cancaner comme un canard, j'appelle cet oiseau un canard" **James Whitcomb Riley** (1849-1916)

- Les usages de Python sont diverses:
  - Programmation système: utilisation de Python en tant que "super-shell"
  - Calcul numérique (numpy, scipy)
  - Développement réseau et web (twisted, django, flask, pyramid)
- Son évolution est gérée par la Python Software Fundation (PSF)
- Les normes du langage sont élaborées sous forme de **Python Enhancement Proposals**, ou **PEPs**
- Il existe une association francophone (AFPY) très active

2.2 L'esprit Python

Tim Peters, un des pionniers du langage, a formulé un ensemble d'aphorismes qui définissent, selon certains, de manière humoristique, cet esprit.

Et, comme l'humour est parfois l'expression d'une sagesse, ces 19 aphorismes, regroupés sous le nom de "The Zen of Python", ont fait l'objet de la **PEP 20** (PEP: document normatif, acronyme de *Python Enhancement Proposals*)

The Zen of python est même accessible dans l'interpréteur interactif :

```
>>> import this
The Zen of Python, by 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!
>>>
```

2.3 Une courte histoire de Python

Le développement de Python a commencé en décembre 1989 au CWI (Pays Bas) et son auteur est *Guido van Rossum*, mais la première publication du code a lieu en février 1991 (sous le numéro de version 0.9.0).

Ont suivi :

- Python 1.0 - janvier 1994
- Python 2.0 - octobre 2000

Depuis 2008 on assiste à la coexistence de deux versions : la version 3 fait son apparition alors que la version 2 continue d'évoluer jusqu'en 2010. La 2.7 est encore présente sur le terrain malgré les progrès réels de portage des principales bibliothèques et frameworks en V3.

- Python 2.6 - octobre 2008
- Python 2.7 - juillet 2010
- Python 3.0 - décembre 2008
  - Python 3.1 - juin 2009
  - Python 3.2 - février 2011
  - Python 3.3 - septembre 2012
  - Python 3.4 - mars 2014
  - Python 3.5 - septembre 2015

Pour prendre en compte cette situation de fait, le présent support met en perspective les deux versions.

2.4 Python et les autres langages

Python présente des points communs avec les langages de programmation les plus populaires. On va mettre en perspective ces points communs ainsi que les différences pour deux raisons:

- Tirer profit autant que possible des expériences acquises, éventuellement, dans les autres langages pour mieux appréhender le nouveau.
- Donner un aperçu rapide des avantages/inconvénients d'utiliser Python plutôt qu'un autre langage dans des situations connues.

2.4.1 Python et Java

Malgré leurs syntaxes très différentes, les similarités entre les deux langages sont importantes mais avec quelques nuances.

- **Compilation :** Les deux langages sont compilés vers un "bytecode", mais la compilation de Python (contrairement à celle de Java) n'est pas distincte de l'exécution. Entre autres, il n'y a pas de notion de "directive de compilation" dans Python.
- **Typage :** Les deux langages sont fortement typés, seulement: le typage dans Python est dynamique. Contrairement à une idée reçue, Python n'est pas un langage non typé!
- **Affectation, passage d'arguments, retour des fonctions :** Les deux langages implémentent la sémantique par *référence uniforme*, mais :
  - partiellement en Java, où les types numériques disposent également d'une représentation alternative, non-objet, pour les entiers et les flottants
  - intégralement dans Python où tout est objet de première classe (y compris les entiers, les fonctions etc.)
- **Gestion de la mémoire :** les deux disposent d'un garbage collector (ramassage de miettes)
- **Modèle objet :**
  - Le point commun: tout hérite d'une racine commune dans les deux langages (la classe *object*) avec les exceptions précitées pour les types numériques, pour ce qui concerne Java
  - Python supporte l'héritage multiple, contrairement à Java.
  - Pas de notion d'interface en Python (mais des extensions implémentant ce concept existent)
  - Pas d'implémentation de l'encapsulation dans le modèle objet de Python, mais une simple convention de nommage (les attributs et les méthodes préfixés par "__" (deux "_") sont "privés", en quelque sorte)
  - Python supporte aussi bien les méthodes statiques (comme Java) que les méthodes de classe dans l'esprit de *SmallTalk*
- **Bibliothèque standard :** les deux langages disposent de bibliothèques standard riches

2.4.2 Python et C++

Les similarités entre Python et C++ sont moins nombreuses, mais elles méritent d'être soulignées :

- Les deux langages sont multi-paradigme
- Python, tout comme le C++, supporte la surcharge des opérateurs, à l'exception de celle de l'opérateur d'affectation (non supportée par Python).
- Les deux langages supportent l'héritage multiple.

2.4.3 Python et Bash

- Les deux langages ont en commun la possibilité d'être utilisés en mode interactif
- Certains aspects de la syntaxe de *Python* rappellent le *Bash* et le langages de script en général : par exemple, les instructions Python se terminent par un retour à la ligne, comme les instructions *Bash*.
- Python, comme Bash, exécute les instructions présentes dans un script au chargement, sans faire intervenir une fonction spéciale, comme la fonction "main" présente en C/C++)

2.5 Découverte de l'interpréteur, premiers programmes

Comme dans bien d'autres langages, les concepts de base de Python dépendent parfois les uns des autres, de manière cyclique.

Pour cette raison, avant d'aller plus loin, on va introduire provisoirement, de manière informelle et simplifiée, quelques notions indispensables, sachant que sur la plupart de ces notions on va revenir d'une manière plus rigoureuse dans la suite de ce document.

Dans l'immédiat on parlera de **variable**, **expression** et de **fonction** avec les significations rencontrées dans la pratique de tout autre langage. Dans les chapitres suivants ces notions seront précisées pour le cas de Python.

Un **module**, désignera dans ce chapitre un fichier contenant des fonctions ayant vocation à être utilisées localement (appels dans le même fichier) ou dans un autre contexte, après chargement (ou importation) par l'interpréteur du dit module. C'est une notion qui sera également traitée plus en détail par la suite.

Le commentaire en Python est une chaîne de caractères tenant sur une ligne et préfixée par un dièse.

```
#ligne complète de commentaire
a = 1 #une affectation: commentaire en fin de ligne
```

NB: Le commentaire multi-ligne n'existe pas en Python

2.5.1 Exécution interactive

Afin de permettre le travail en mode interactif, l'interpréteur est prévu d'une boucle lecture-évaluation ("read-eval" loop). Sous Linux il suffit d'exécuter la commande "python" sans arguments pour accéder au mode interactif:

```
$ python
Python 2.7.10 (default, Sep 24 2015, 17:50:09)
[GCC 5.1.1 20150618 (Red Hat 5.1.1-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

En mode interactif une expression est évaluée après le retour à la ligne. Si elle n'apparaît pas dans une affectation (à droite du signe "="), le résultat de l'évaluation est affiché.

```
>>> 1+2
3
>>>
```

NB: En réalité, quand une expression apparaît sans affectation explicite, une affectation implicite du résultat à la variable _ se produit:

```
>>> 1+2
3
>>>
>>> _
3
>>>
```

Avec **ipython** on bénéficiera de l'auto-complétion et on aura accès aux commandes système. Dans l'exemple, un premier essai avec l'incontournable "Hello world!"

```
$ ipython
Python 2.7.10 (default, Sep 24 2015, 17:50:09)
Type "copyright", "credits" or "license" for more information.

IPython 2.4.1 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.
```

```
In [1]: print "Hello world!"
Hello world!

In [2]:
```

A l'heure actuelle (fin 2015) les principales distributions Linux intègrent toujours Python 2.7.x préinstallé mais proposent en option les packages python3 et ipython3 qui peuvent cohabiter avec la version par défaut.

```
$ python3
Python 3.4.2 (default, Jul  9 2015, 17:24:30)
[GCC 5.1.1 20150618 (Red Hat 5.1.1-4)] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Dans l'exemple on rencontre une première incompatibilité entre les versions 2 et 3 de Python ; l'instruction "print" devient une fonction dans la version 3:

```
$ ipython3
Python 3.4.3 |Anaconda 2.3.0 (64-bit)| (default, Oct 19 2015, 21:52:17)
Type "copyright", "credits" or "license" for more information.

IPython 3.2.0 -- An enhanced Interactive Python.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://anaconda.org
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: print "Hello world!"
File "<ipython-input-1-59c9fe2caa70>", line 1
print "Hello world!"
                   ^
SyntaxError: Missing parentheses in call to 'print'

In [2]: print("Hello world!")
Hello world!
```

2.5.2 Le rôle de l'indentation

L'indentation détermine les blocs d'instructions, au même titre que les accolades en Java et C/C++. Par exemple, une tentative d'écrire une boucle sans indentation se soldera par une erreur :

```
>>> #tentative d'écrire une boucle sans indentation:
... for x in [1,2,3]:
... print(x)
File "<stdin>", line 3
 print(x)
     ^
IndentationError: expected an indented block

>>> #...et avec indentation:
... for x in [1,2,3]:
...     print(x)
...
1
2
3
>>>
```

NB: pour terminer une instruction multi-ligne en mode interactif, saisir une ligne vide

2.5.3 Exécution à partir d'un fichier source

Le code source contenu dans un fichier est exécuté s'il est présent sur la ligne de commande invoquant l'interpréteur :

```
$ cat hw.py
print("Hello World!")
$ python3 hw.py
Hello World!
$
```

Sous Linux, ou tout autre environnement "UNIX-like", on peut rendre le fichier directement exécutable (sans avoir à le passer en argument à l'exécutable *python*):

Pour cela, la première ligne du fichier doit être:

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

Le droits sur le fichiers devront être positionnés en conséquence, par exemple, pour un droit d'exécution réservé au propriétaire du fichier:

```
chmod u+x sayhello.py
```

CHAPTER

THREE

QUELQUES LIENS UTILES

Il y a beaucoup d'information utiles sur https://www.python.org/documentation/ et en particulier:

- The Python Language Reference:
  - version 2 : http://docs.python.org/2/reference/
  - version 3 : http://docs.python.org/3/reference/
- Les tutoriels et les FAQs de la "Python Software Foundation" :
  - version 2 : http://docs.python.org/2/tutorial/
  - version 3 : http://docs.python.org/3/tutorial/

Un cours introductif en français :

- http://perso.limsi.fr/pointal/python:courspython3

"Think Python" ou "How to Think Like a Computer Scientist" by Allen B. Downey :

- http://www.greenteapress.com/thinkpython/thinkpython.html

Les ressources didactiques Gérard Swinnen :

- http://inforef.be/swi/python.htm

L'Association Francophone Python :

- http://www.afpy.org/

"Dive Into Python 3" by Mark Pilgrim :

- http://www.diveintopython3.net/

Des nombreuses ressources aussi sur :

- http://pythonbooks.revolunet.com/

Pour découvrir les bibliothèques pour le calcul scientifique:

- Scipy Lecture Notes (One document to learn numerics, science, and data with Python) http://www.scipy-lectures.org/
- Formation à Python scientifique - ENS Paris http://python-prepa.github.io/index.html

CHAPTER

FOUR

LES ÉLÉMENTS DU LANGAGE

4.1 Une vue d'ensemble

L'objectif de cette section est d'introduire de manière succincte et parfois intuitive certaines notions qui seront précisées dans les chapitres suivants.

4.1.1 Les objets

A ce stade on va se contenter de décrire les objets *Python* d'une manière très simplifiée et souvent partielle, description sur laquelle on va revenir ultérieurement pour la parfaire.

On va considérer pour l'instant l'objet comme une entité ayant une structure composé de:

- données accessibles par des noms appelés **attributs** (Notation: *objet.attribut*)
- comportements propres qui sont des fonctions attachées à l'objet appelés **méthodes** (exemple d'appel d'une méthode: `"abc".upper()`)
- un identifiant unique
- un type (ou une classe)

Quelques objets...

Pour illustration, un aperçu rapide et très partiel de quelques types de base (qui seront expliqués plus en détail dans un chapitre dédié).

Un entier:

```
>>> id(1) # possède un id
9357408
>>> type(1) # possède un type
<class 'int'>
>>> dir(1) # la primitive dir() fournit la liste des attributs et de méthodes
['__abs__', '__add__', '__and__', '__class__', '__cmp__', '__coerce__', '__delattr__', '__div__',
```

Une chaîne de caractères:

```
>>> id('abc') # possède un id
140219035744176
>>> type('abc') # possède un type
<type 'str'>
>>> dir('abc') # attributs, méthodes
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__
```

Une liste:

Pour l'instant, on va se contenter d'envisager la liste comme un tableau d'objets (pas forcément tous du même type).

NB: Les listes font partie d'une "famille" de types de base appelées **séquences** et qui seront détaillées plus loin.

```
>>> id([1,'abc']) # possède un id
30386656
>>> type([1,'abc']) # possède un type
<type 'list'>
>>> dir([1,'abc']) # attributs, méthodes
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__delslice__', '__doc__',
```

Une fonction est aussi un objet:

```
>>> def say_hello():
...     print("Hello!")
...
>>> id(say_hello)
30495040
>>> type(say_hello)
<type 'function'>
>>> dir(say_hello)
['__call__', '__class__', '__closure__', '__code__', '__defaults__', '__delattr__', '__dict__', '
```

4.1.2 La référence uniforme en Python

En Python, tout est objet de "première classe", du simple entier jusqu'à la fonction.

Note: Un objet de première classe ("first class object") est un objet qui peut être affecté à une variable, passé en paramètre à une fonction ou retourné par une fonction (cf. Raphael Finkel)

Les langages de programmation peuvent manipuler les données de deux manières:

- par valeur (par exemple, les entiers dans le langage **C**)
- par référence (Java, C++).

En Python, toute donnée, même très simple, étant représentée sous forme d'objet, sa manipulation se fait toujours par référence, jamais par valeur, ce qui correspond à la sémantique de **référence uniforme**.

4.1.3 Les variables

En Python, une variable n'a pas de correspondant physique (comme les variables en *C*, qui sont des locations mémoire). Les variables *Python* sont juste des **noms**, des **identifiants** attribués à des objets. Ainsi, une variable:

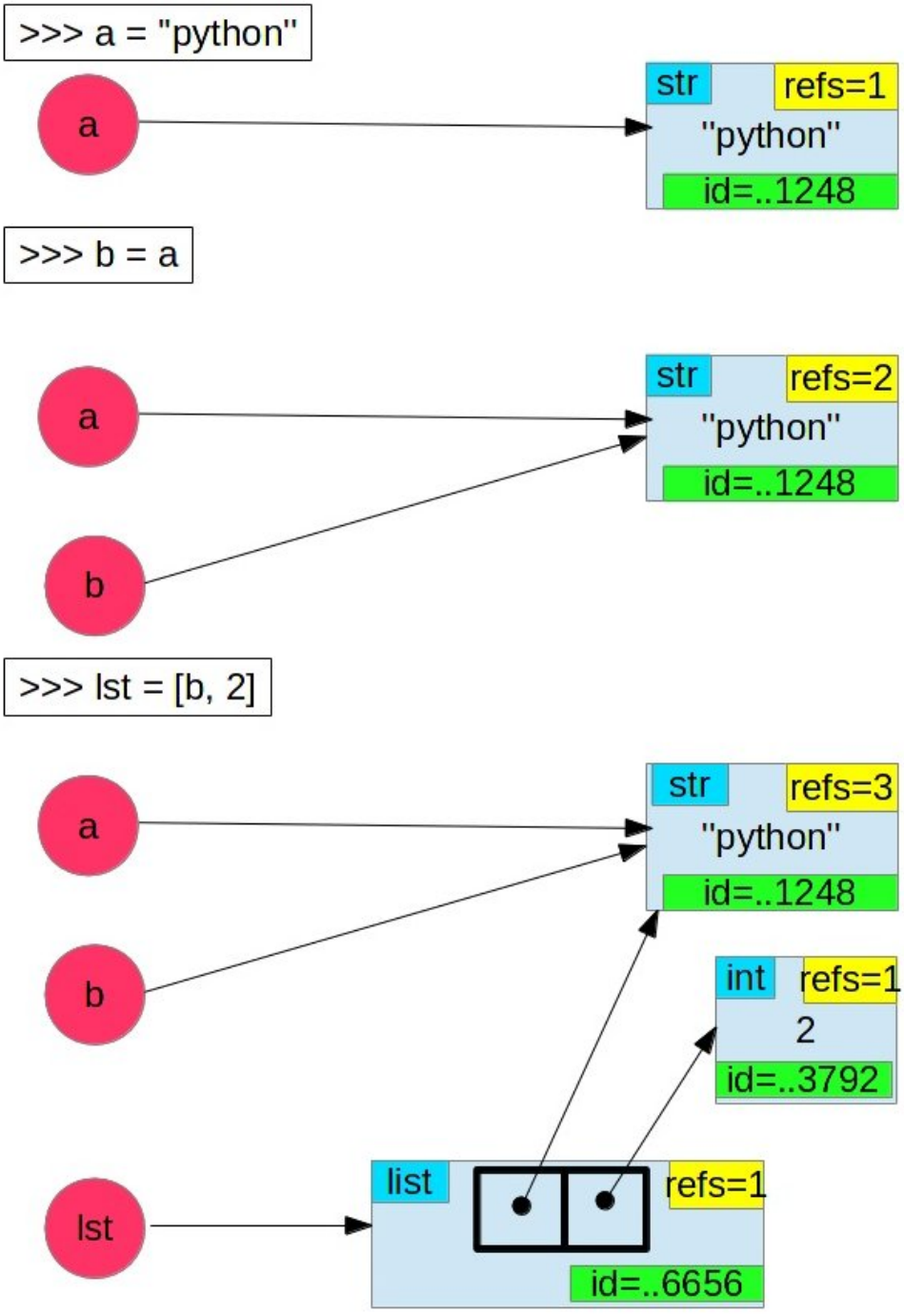
- Ne fait pas l'objet d'une déclaration
- Ne désigne pas une adresse mémoire comme en *C/C++*
- N'est pas un objet (mais juste un nom attribué à un objet)
- N'a pas de type (mais l'objet qu'elle désigne en a un, **toujours** !)

Illustration:

A la suite des opérations suivantes, l'objet (chaîne de caractères) 'python' sera référencé 3 fois:

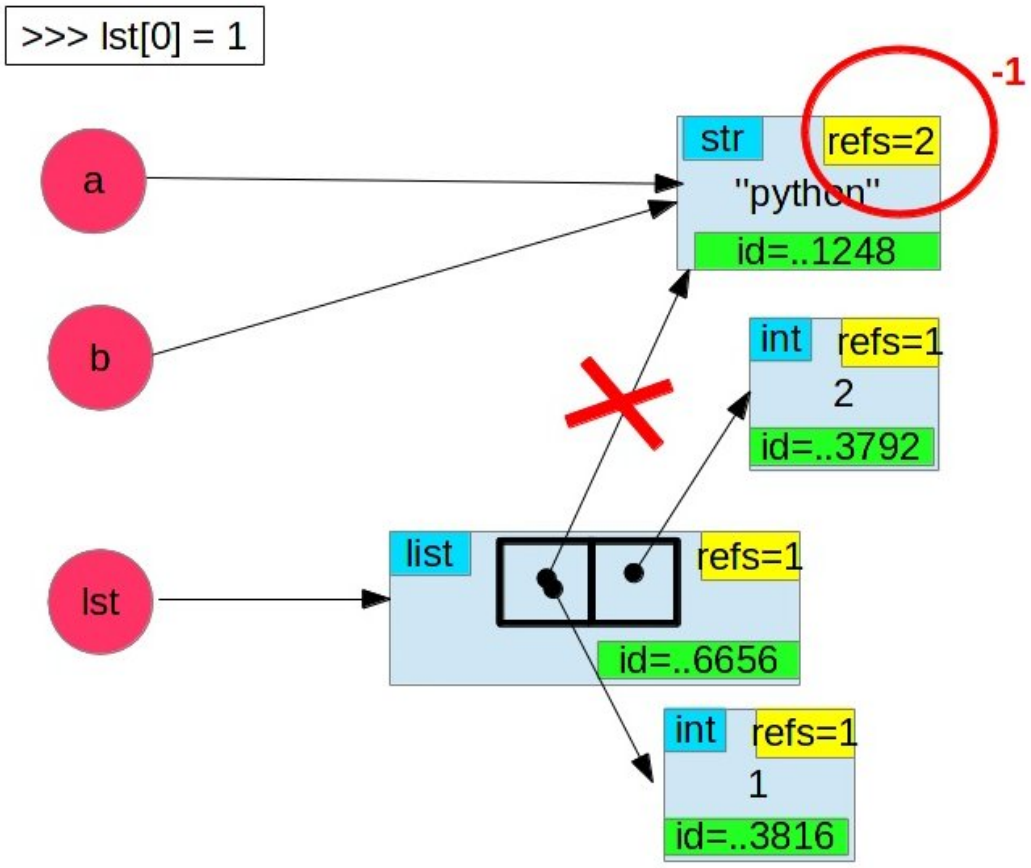
```
>>> a='python'
>>> id(a)
140219035541248
>>> id('python')
140219035541248
>>> b=a
>>> id(b)
140219035541248
>>> lst=[b,2]
```

```
>>> id(lst)
30386656
>>> id(2)
25253792
>>> id(lst[0])
140219035541248
>>> id(lst[1])
25253792
```



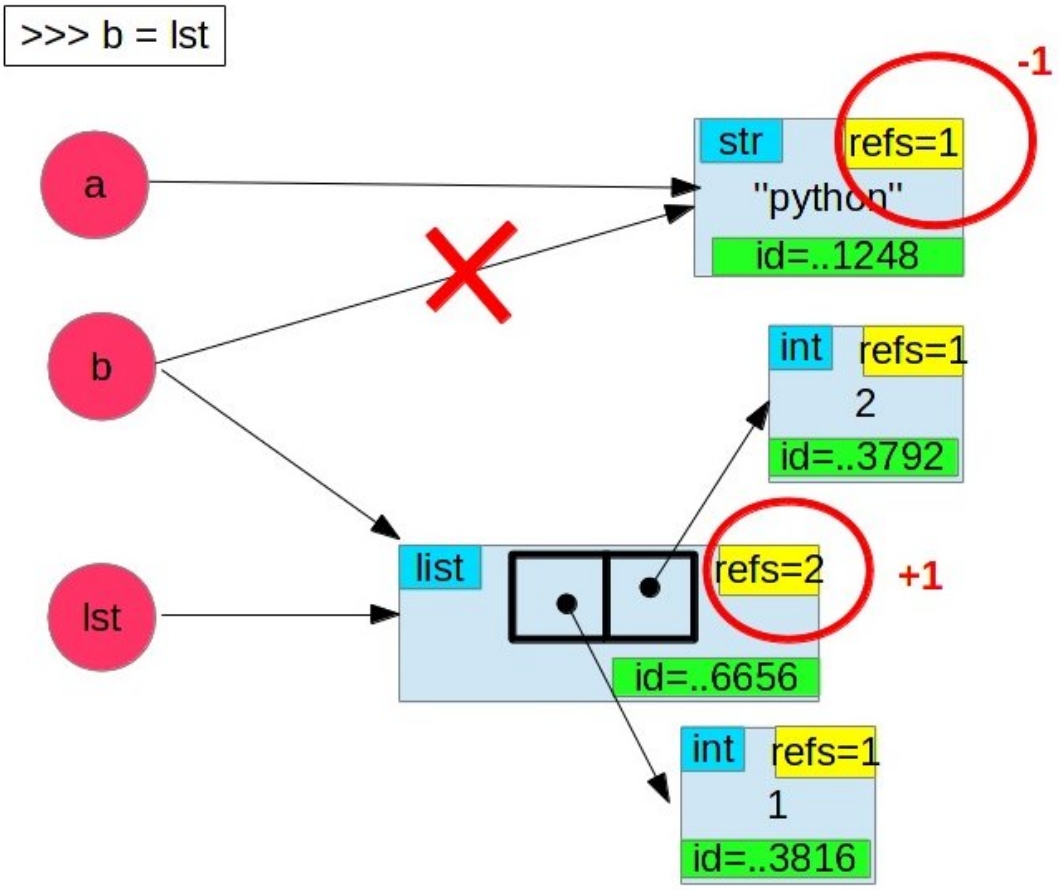
Par la suite, l'objet va "perdre" une référence:

```
>>> lst[0]=1
```



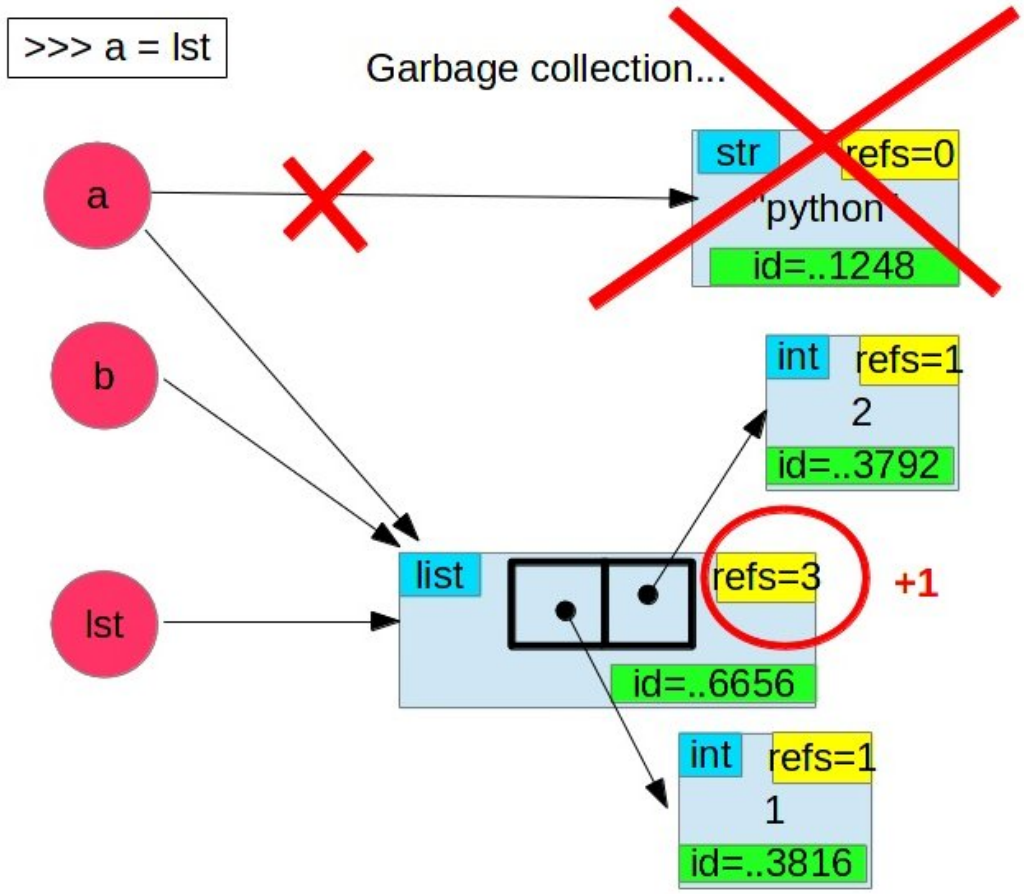
Et encore une:

```
>>> b=lst
```



Avec la perte de la dernière référence, l'objet sera purgé par le "garbage collector" (ramassage des miettes):

```
>>> a=lst
```



Lexicalement, les noms des variables en Python sont des identifiants soumis, globalement, aux mêmes règles qu'en *C*, *C++* ou *Java*:

identifiant := (lettre|_)(lettre|chiffre|_)*

Avertissement

A partir de la version 3 de *Python*, toutes les lettres Unicode peuvent entrer dans la composition d'un identifiant, par exemple les lettres accentuées. Utiliser des caractères non-ASCII dans les identifiants est une pratique à éviter, car elle est source confusion...

Python 2.x:

```
$ python
Python 2.7.10 (default, Sep 24 2015, 17:50:09)
[GCC 5.1.1 20150618 (Red Hat 5.1.1-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> à_éviter=0
File "<stdin>", line 1
à_éviter=0
^
SyntaxError: invalid syntax
>>>
```

Python 3.x:

```
$ python3
Python 3Python 3.4.3 |Anaconda 2.3.0 (64-bit)| (default, Oct 19 2015, 21:52:17)
[GCC 4.4.7 20120313 (Red Hat 4.4.7-1)] on linux
```

```
Type "help", "copyright", "credits" or "license" for more information.
>>>  à_éviter=0
 >>> id(à_éviter)
 140544516702304
 >>> à_éviter
 0
```

Pour qu'un identifiant soit un nom correct de variable il ne doit pas faire partie de la liste de mots clés du langage:

```
>>> and=4
    File "<stdin>", line 1
      and=4
        ^
 SyntaxError: invalid syntax
 >>> True=0
 File "<stdin>", line 1
 SyntaxError: assignment to keyword
```

NB: La portée des variables sera discutée dans la section sur les fonctions.

4.1.4 Les opérateurs

Il s'agit ici d'une vue d'ensemble des opérateurs comme entités syntaxiques permettant de construire des expressions. Leur sémantique dépend du type des opérandes et elle sera développée par la suite, au cas par cas, en même temps que les types prédéfinis. La possibilité de redéfinir les opérateurs sera présentée en même temps que le modèle objet.

Opérateurs de base (+,-,*,**,/,//,%)

On les associe spontanément aux opérations mathématiques, mais certains d'entre eux peuvent être associés, avec des sémantiques adaptées, à des types non numériques. En plus, comme on le verra plus tard, ils sont redéfinissables par des types applicatifs (classes).

```
>>> 1+2
3
>>> "abc"+"xyz"
'abcxyz'
>>> "abc"*3
'abcabcabc'
>>> 2-1
1
>>> 2*2
4
>>> 3/4
0
>>> 5/3
1
>>> 5.0/3
1.6666666666666667
>>>
```

NB: Ces opérateurs peuvent apparaître dans des "affectations augmentées" sous la forme *variable operateur*= *expression* (Ex: *x* += *5*, *x* -= *7*, etc.)

```
>>> x=1
>>> x+=5
>>> x
6
```

Opérateurs relationnels

Il sont les mêmes que dans la plupart des langages, avec une syntaxe proche de celle du langage *C*:

Symbole	Opération
<	inférieur
<=	inférieur ou égal
>	supérieur
>=	supérieur ou égal
==, is	égal (par convention *is* est à utiliser avec *None*)
!=	différent

NB: L'opérateur **<>** (l'équivalent de **!=**) n'est plus supporté en version 3 et son utilisation en version 2 est déconseillée.

Opérateurs logiques

En Python, les expressions suivante sont logiquement fausses: 0,'','',[],(),{}, False

Les opérateurs logiques sont: **and**, **or**, **not**

Les fonctions : any(c), all(c) appliquent **or** et **and** aux elements du conteneur **c** (par exemple, les éléments d'une liste):

```
>>> any([0,"",5])
True
>>> any([0,"",{}])
False
>>> all([0,"",5])
False
>>>
```

En dehors des connecteurs logiques **and**, **or**, **not** présentés ici il existe des opérateurs logiques "bit à bit" applicables aux entiers. Il seront présentés dans la section destinées aux types numériques.

4.1.5 Les expressions

Les expressions en Python ne diffèrent pas beaucoup de celles de la plupart des langages de programmation les plus courants.

De manière très informelle, il s'agit du combinaison cohérente d'objets, variables, appels de fonctions et méthodes et opérateurs.

Pour une définition formelle voir : http://docs.python.org/3/reference/expressions.html

4.1.6 Les instructions

Comme dans d'autres langages on peut parler d'instructions simples et d'instructions composées.

Les instructions simples

Incluent :

- les expressions (saisies en mode interactif, appel de fonction dont la valeur de retour n'est pas utilisée)
- les affectations:
  - simples (=)
  - multiples (=)
  - augmentées (+=, −=, ...)

- opérations introduites par un mot clé : assert, pass, del, return, ...

Le plus souvent une instruction simple s'écrit sur une seule ligne mais on peut écrire plusieurs instructions par ligne séparées par **;** (pratique généralement déconseillée, sauf quand son utilité est évidente).

Affectations multiples:

Python permet d'affecter en parallèle plusieurs variables. Cela est pratique si on veut intervertir les valeurs entre deux variables ou faire des permutations:

```
>>> a = 1
>>> b = 2
>>> c = 3
>>> a, b, c = b, c, a
>>> a
2
>>> b
3
>>> c
1
>>>
```

NB: Cette écriture fait intervenir le type **tuple** qui sera évoqué plus loin.

Une autre écriture qu'on peut qualifier d'affectation multiple:

```
>>> a = b = 3
>>> a
3
>>> b
3
>>>
```

Pour en savoir plus: http://docs.python.org/3/reference/simple_stmts.html

Les instructions composées

Il s'agit des instructions contenant à leur tour d'autres instructions regroupées en blocs.

Elles contrôlent le flux de l'exécution des instructions qu'elles contiennent, c'est pourquoi on les appellent aussi **structures de contrôle**.

En Python, comme dans d'autres langages, on va trouver:

- des structures alternatives
- des structures répétitives
- des structures pour le traitement des exceptions

avec quelques spécificités qui seront traitées dans une section dédiée.

4.2 Les entrées/sorties standard

4.2.1 La sortie standard

Python 2.x dispose d'une instruction **print** dédiée, avec une syntaxe rappelant vaguement celle des langages *shell* :

print expr $_1$, expr $_2$, ..., expr $_n$

```
$ python
Python 2.7.10 (default, Sep 24 2015, 17:50:09)
[GCC 5.1.1 20150618 (Red Hat 5.1.1-4)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print "aaa", "bbb", "ccc"
aaa bbb ccc
```

Cette instruction a été remplacée dans Python 3 avec une fonction portant le même nom. La nouvelle fonction est plus puissante que l'instruction qu'elle remplace:

> def print(*args, sep=' ', end='\n', file=None)

Les paramètres nommés:

- **sep** représente le séparateur entre arguments
- **end** représente le caractère de fin de ligne. Si on ne souhaite pas de retour à la ligne il suffit de positionner end='', chose impossible avec l'instruction **print** de la version 2.
- **file** représente le dispositif de sortie (la sortie standard, par défaut)

Pour en savoir plus: http://www.python.org/dev/peps/pep-3105/

Note: On peut disposer de la fonction print() de **Python 3.x** dans un script **Python 2.x** en exécutant au début du script l'instruction :

```
from __future__ import print_function
```

Cette démarche, qui sera expliquée plus tard, a été adoptée pour l'élaboration de ce support chaque fois qu'on a voulu qu'un exemple soit compatible avec les deux versions de Python.

4.2.2 L'entrée standard

Pour lire l'entrée standard il faut utiliser:

- raw_input([messg])=>str en Python 2.x
- input([messg])=>str en Python 3.x

Avertissement

Python 2 dispose également d'une fonction appelée *input()* qui évalue à la volée le texte en entrée. Cette fonctionnalité, peu utilisée et potentiellement dangereuse en termes de sécurité a été supprimée dans la nouvelle version 3.x. et l'ancienne fonction *raw_input()* a été renommée *input()*.

Conclusion: En Python 2 il est déconseillé d'utiliser la fonction *input()*. Utiliser *raw_input()* à la place et convertir le résultat si nécessaire.

La fonction affiche le message et restitue l'entrée clavier sous forme de chaîne.

Exemple en Python 2.7.x

```
$ cat echo2.py
#!/usr/bin/env python

inp = raw_input("?: ")
print inp
$ ./echo2.py
?: ça va?
ça va?
```

Exemple en Python 3.4.x

```
$ cat echo3.py
#!/usr/bin/env python3

inp = input("?: ")
print(inp)
$ ./echo3.py
?: ça va?
ça va?
```

4.3 Les structures de contrôle

4.3.1 Structures alternatives (if-elif-else)

```
if <expr-0>:
    <statement>
    # ...
    <statement>
elif <expr-1>: #optionnel
    <statement>
    # ...
    <statement>
# autres blocs elif optionnels ...
elif <expr-i>:
    <statement>
    # ...
    <statement>
else: #optionnel
    <statement>
    # ...
    <statement>
```

Les blocs sont définis par l'indentation. Pour illustration, un exemple (artificiellement compliqué), qui implique deux niveaux de **if-elif-else** :

```
x = int(input("x="))
y = int(input("y="))
if x > 0:
    if y > 0:
        print("1er quadrant (+,+)")
    elif y < 0:
        print("4ème quadrant (+,-)")
    else:
        print("abscisse, x strictement positif")
elif x < 0:
    if y > 0:
        print("2ème quadrant (+,-)")
    elif y < 0:
        print("3ème quadrant (-,-)")
    else:
        print("abscisse, x strictement négatif")
else:
    if y > 0:
        print("ordonnée, y strictement positif")
    elif y < 0:
        print("ordonnée, y strictement négatif")
    else:
        print("origine")
```

Dans certains cas, si les blocs d'instructions contenus dans un **if-elif-else** ne contiennent que des instructions simples, **et si cela améliore la lisibilité**, on peut écrire de manière plus compacte:

```
x = int(input("x="))
if x > 0: print("x strictement positif:");mod = x
elif x < 0: print("x strictement négatif"); mod = -x
else: print("x == 0"); mod = 0
print("module:"+str(mod))
```

Expressions ternaires

Des nombreux langages (C/C++, Java, php,...) acceptent les expressions dites "ternaires" qui sont de la forme `test ? expression1 : expression2`. Python permet d'écrire la même chose avec une syntaxe différente : `expression1 if test else expression2`

Exemple:

```
>>> x = -10
>>> mod_x = x if x > 0 else -x
>>> mod_x
10
>>>
```

4.3.2 Exceptions

Contrairement aux erreurs de syntaxe qui sont détectées avant le début de l'exécution, les anomalies détectées pendant l'exécution activent des mécanismes spécifiques qui modifient le cours de l'exécution, qui ne peut pas se dérouler normalement.

On va parler dans ce cas d'**exceptions** :

```
>>> "Python"+3
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

Les informations affichées sont:

- le contenu de la pile d'exécution au moment de la levée de l'exception
- le type de l'exception levée
- une description du problème rencontré

Au déclenchement d'une exception, le cours de l'exécution change (c'est pourquoi on a choisi de traiter la gestion des exceptions avec les structures de contrôle). L'exécution de la séquence courante s'arrête et les différentes strates de l'exécution sont remontées jusqu'à :

- la rencontre d'un bloc **try-except** qui traite l'exception levée
- la sortie du programme, en absence du dit bloc

Pour gérer les exceptions, un programme a la possibilité de:

1. arrêter la propagation d'une exception par un bloc **try-except**
2. lever des exceptions propres ou laisser remonter des exceptions préalablement "capturées" (par l'instruction *raise*)
3. définir ses propres types d'exceptions

Le bloc try-except-finally

C'est une construction de la forme:

```
try:
    ... # code à surveiller
    ... # susceptible de lever une exception
except Exception_1 as instance_1:
    ... # s'exécute si une exception
    ... de type Exception_1 est levée
except Exception_2 as instance_2:
    ... # s'exécute si une exception
    ... de type Exception_2 est levée
... # autres blocs except, éventuellement
else:
    ... # si présent, s'exécute si aucune exception n'a été levée
finally:
    ... # si présent, s'exécute toujours (s'il y a exception ou pas)
```

NB: Après un bloc **try** , au moins un bloc **except** *ou* le bloc **finally** doit être présent. On peut donc avoir plusieurs cas:

- try-except
- try-finally
- try-except-finally

Traitement des exceptions

Le bloc *except* peut prendre plusieurs formes en fonction du traitement envisagé:

```
try:
    ...
except:
    print("Erreur...mais on ne sait pas laquelle...")

try:
    ...
except Exception:
    print("Erreur...mais on ne sait pas laquelle...")

try:
    ...
except Exception as err: # toutes les erreurs sont traitées
    print(type(err)) # mais l'objet err apporte + d'informations par son type
    print(err.args) # et éventuellement par son attribut args


try:
    ...
except TypeError [as err]:
    ... # seulement les exceptions TypeError seront traitées
    ... # toutes les autres vont remonter

try:
    ...
except TypeError [as err]: # un type d'exception par bloc
    ...
except ZeroDivisionError [as err]:
    ...

try:
    ...
except (TypeError, ZeroDivisionError) [as err]:
    ... # plusieurs types exceptions traitées en un seul bloc
```

Lever des exceptions

On utilisera l'instruction *raise*:

```
x = int(input("x="))
if x < 0:
    raise ValueError("x must be positive")
print("x is positive")
```

Les arguments du constructeur de l'exception levée par *raise* se retrouvent dans l'attribut *args* de l'objet récupérée par *except* :

```
>>> try:
...     # simule un traitement levant une exception
...     raise ValueError("arg1","arg2")
... except ValueError as obj:
...     print(obj.args)
...
('arg1', 'arg2')
>>>
```

NB: Un bloc *except* peut laisser remonter une exception après avoir exploité l'information fournie. Par exemple, un serveur web peut utiliser le bloc *except* pour récupérer et écrire l'erreur dans les fichiers de journalisation (logs) et en même temps la laisser remonter pour que les couches supérieures de l'application qui pourraient utiliser aussi (pour la communiquer à l'interface utilisateur, par exemple) :

```
try:
    "café".encode("ascii") # simule un traitement provoquant une anomalie
except UnicodeEncodeError as err:
    write_log(err)
    raise
```

Définir des exceptions

Les exceptions sont des classes héritant directement ou indirectement de la classe prédéfinie *Exception*. Le chapitre sur le modèle objet de Python expliquera en détail comment définir ses propres classes.

Néanmoins, il convient de préciser que les exceptions prédéfinies sont déjà nombreuses et elles couvrent la plupart des situations pratiques. Plus de détails : http://docs.python.org/3.4/library/exceptions.html#exception-hierarchy

4.3.3 La structure with

Cette structure est une application directe de **try-except-finally**. Utilisée beaucoup pour la manipulation des fichiers, elle garantit la libération de la ressource même si l'opération se passe mal. Ainsi, la structure:

```
>>> with open("/tmp/fich.txt","w") as fd:
...     fd.write("some text")
```

qui permet l'ouverture d'un fichier suivie d'une opération d'écriture peut être résumée, (en simplifiant un peu...), par la séquence:

```
>>> fd = open("/tmp/fich.txt","w")
>>> try:
...     fd.write("some text")
... finally:
...     fd.close()
```

L'utilisateur peut définir ses propres objets *"with compatibles"* appelés "gestionnaires de context" (**context managers**), à condition de définir deux méthodes "magiques" (qui n'ont pas vocation à être appelées dans le programme, car en principe destinées aux mécanismes internes de l'interpréteur):

- *__enter__()*

- *__exit__()*

Ces deux méthodes participent au protocole suivant, qui a pour but de garantir que si *__enter__()* s'exécute avec succès alors *__exit__()* sera toujours exécutée, même en cas d'exception survenue à l'exécution du bloc:

```
with expression [as var]:
    code
```

1. *expression* est évaluée, le résultat étant un objet **context manager**
2. la méthode *__enter__()* est invoquée (sur l'objet **context manager**). Elle peut effectuer, par exemple, l'ouverture d'un fichier, socket, connexion DB etc. Le retour de *__enter__()* est affecté à la variable **var** si elle est fournie (i.e. la partie "[as var]" est présente).
3. le **code** est exécuté dans un bloc **try-except-finally**
4. la méthode **__exit__()** est appelée. En cas d'exception levée à l'exécution de *code* (3) l'appel contiendra trois informations décrivant l'exception: type, valeur, traceback. Dans ce cas, *__exit__()* peut retourner:
   - **False** pour provoquer la propagation de l'exception par l'appelant
   - **True** pour empêcher la propagation de l'exception

Note: A partir de **Python 3.1**, l'instruction **with** peut déclarer plusieurs contextes:

```
with expression1 [as var1], expression2 [as var2], ...:
    code
```

Ce qui est équivalent à:

```
with expression1 [as var1]:
    with expression2 [as var2]:
       ....
       code
```

4.3.4 Structure répétitives (while-else, for-else)

La structure while-else

L'instruction **while**, ressemble beaucoup à son homonyme dans les autres langages :

```
while condition:
   instruction 1
   instruction 2
   ...
else: # optionnel
   ...
   ...
```

Le bloc (*instruction 1, instruction 2, ...*) s'exécute tant que la *condition* reste vraie:

```
>>> x=0
>>> while x < 5:
...     print(x)
...     x += 1
...
0
1
2
3
4
>>>
```

Python dispose, comme d'autres langages, de deux instructions permettant le contrôle de l'exécution des boucles, à utiliser dans le bloc **while** :

- **break** permettant la sortie de la boucle avant la fin (i.e. avant que la condition ne devienne fausse)
- **continue** qui permet de passer directement à l'itération suivante

La vraie spécificité est le bloc **else** qui préserve la même sémantique que dans un **if-else**, autrement dit, il s'exécute quand la condition est fausse. Dans un *while* la condition est nécessairement fausse à la sortie de la boucle, si la sortie se fait **normalement**, sans recours à l'instruction **break** .

Illustration:

```
to_find = int(input("find:"))
x = 0
while x <= 10:
    if x == to_find:
        print("found")
        break
    x += 1
else:
    print("not found")
```

La structure for-in

La forme générale :

```
for element in iterable:
   ...
   ...
else: # optionnel
   ...
   ...
```

Propriétés en commun avec l'instruction **while** :

- la possibilité d'utiliser les instructions *break* et *continue*
- le bloc optionnel **else** avec une sémantique équivalente

On appelle **"itérable"** un objet qui permet l'accès séquentiel à un ensemble d'autres objets qui le composent ou qu'il crée à la demande, autrement dit un objet qui supporte l'itération.

C'est le cas de tous les objets **conteneurs**, autrement dit des objets contenant d'autres objets (listes, chaînes etc.), mais pas seulement (plus de détails dans la section sur le **générateurs**).

- Un objet **itérable** doit implémenter la méthode magique *__iter__()*. Cette méthode sera exécutée à chaque invocation de la primitive *iter()* pour retourner un **itérateur** qui est une sorte d"'objet-curseur".
- Un **itérateur** doit implémenter :
  - *next()* (renommée en **__next__()** en Python 3) qui renvoie à chaque appel l'élément "suivant", s'il y en a un, ou lève l'exception **StopIteration** dans le cas contraire.

Note: Accessoirement, les itérateurs implémentent également *__iter__()* qui renvoie l'itérateur lui même (seulement pour assurer un comportement homogène entre itérateurs et itérables)

Illustration:

```
>>> l=[1,2,3]
>>> l.__iter__
<method-wrapper '__iter__' of list object at 0x7f4020da0830>
>>> i = iter(l)
>>> next(i)
1
```

```
>>> next(i)
2
>>> next(i)
3
>>> next(i)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
StopIteration
>>>
```

NB: Dans l'exemple précédent *next(i)* est une primitive du langage (la même dans les deux versions) qui se chargera de l'appel de *i.next()* (Python 2.x) ou de *i.__next__()* (Python 3.x).

Ainsi, l'instruction:

```
for element in iterable:
   ... # bloc
   ... # d'instructions
```

est équivalente à:

```
i = iter(iterable)
while True:
   try:
      element = next(i)
   except StopIteration:
      break
   ... # bloc
   ... # d'instructions
```

Exemple basique:

```
>>> for elt in [1,2,3]:
...     print(elt)
...
1
2
3
>>>
```

Note: Python ne possède pas d'instruction équivalente à `for(i=0;i<max;i++){\...}` présente dans le langage *C* et dans bien d'autres. Dans la section suivante on va voir un idiome qui permet une écriture équivalente.

Exercice

```
"""
fichier: loop.py

Ecrire un petit programme qui, en utilisant une boucle infinie:

a) demande la saisie d'un entier compris dans [-10,10]

b) convertit le résultat de la saisie avec int(saisie) et
   traite l'exception ValueError en cas de saisie incorrecte
   en affichant un message approprié suivi du retour à (a).
   Sinon:

c) vérifie que la valeur saisie est dans l'intervale  [-10,10] et affiche
   un message approprié dans le cas contraire, suivi du retour à (a).
Sinon:

d) La liste lst contient plusieurs sous-listes à 2 éléments définissant des
```

```
   intervales. Vérifier l'appartenance de la valeur saisie à un des intervales.
   Affichez un message approprié en cas de succès, un autre en cas d'échec,
   puis retour à (a) . Utiliser la clause else du for pour traiter le cas de la
   recherche infructueuse

Si vous êtes en evironnement Unix/Linux, rendre le fichier de votre programme
exécutable directement en ligne de commande.
"""

L = [[-8, -5], [-3, -2], [3, 5], [6, 8]]

"""
fichier: loop_sol.py
"""
from __future__ import print_function
import sys

L = [[-8, -5], [-3, -2], [3, 5], [6, 8]]
prompt = "Get an integer in [-10,10]"
while True:
    num_str = input(prompt) if sys.version_info.major==3 else raw_input(prompt)
    try:
        num = int(num_str)
    except ValueError:
        print(num_str, " is not a number")
        continue
    if num < -10 or num > 10:
        print(num, " not in  [-10,10]")
        continue
    for interval in L:
        if num >= interval[0] and num <= interval[1]:
            print(num, " found in ", interval)
            break
    else:
        print("interval not found for ", num)
    print("next loop")
```

Une écriture légèrement différente, plus “pythonique”:

```
"""
fichier: loop_sol2.py
"""
from __future__ import print_function
import sys

L = [[-8, -5], [-3, -2], [3, 5], [6, 8]]
prompt = "Get an integer in [-10,10]"
while True:
    num_str = input(prompt) if sys.version_info.major==3 else raw_input(prompt)
    try:
        num = int(num_str)
    except ValueError:
        print(num_str, "is not a number")
        continue
    if num < -10 or num > 10:
        print(num, "not in  [-10,10]")
        continue
    for min_, max_ in L:
        if num >= min_ and num <= max_:
            print(num, "found in", min_, max_)
            break
    else:
        print("interval not found for ", num)
```

```
    print("next loop")
```

4.4 Le types de base

Il s'agit des types prédéfinis dans Python et intégrés dans la syntaxe du langage:

- types numériques
- conteneurs (types itérables)
- types des constantes nommées
- le type **file**

4.4.1 Les types numériques

Permettent de représenter les nombres entiers, réels (virgule flottante), complexes et décimaux.

Les entiers et les réels ont beaucoup de points communs avec leurs équivalents en **C** .

Les entiers

En **Python 2** il y a deux types entiers signés:

- Les **int** sont représentés sur 32 ou 64 bits selon le processeur, avec la même représentation que le type *int* du langage **C** , la plage de représentation étant [-2 ^{arch}, 2 ^{arch} - 1] avec arch = 32-1 ou arch = 64-1 (le *-1* étant dû au bit de signe).
- Les **long** sont utilisés pour représenter les entiers situés à l'extérieur de cet intervalle. Leur plage de représentation est illimitée.

L'interpréteur choisit automatiquement le bon type pour représenter les entiers:

```
>>> # Python 2.7.x
>>> type(1)
<type 'int'>
>>> type(9223372036854775808) # i.e. 2 ** 63
<type 'long'>
>>> type(-9223372036854775808) # i.e. -(2 ** 63)
<type 'int'>
>>> type(9223372036854775807)
<type 'int'>
```

En **Python 3** il y a un type unique pour les entiers:

```
>>> # Python 3.4.x
>>> type(1)
<class 'int'>
>>> type(9223372036854775808)
<class 'int'>
```

Avertissement

Des différences de comportement existent à ce niveau entre les versions 2 et 3 du langage:

```
>>> # Python 2.7
...
>>> isinstance(1, int)
True
>>> isinstance(2**63, int)
```

```
False
>>> isinstance(2**63, long)
True
>>>

>>> # Python 3.4
...
>>> isinstance(1, int)
True
>>> isinstance(2**63, int)
True
>>> isinstance(2**63, long)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'long' is not defined
>>>
```

Comme d'autres langages, Python supporte plusieurs notations pour les entiers, pratiquement les mêmes que celles de C/C++ et Java :

Notation	Préfixe	Exemple
Décimale	sans	165
Hexadécimale	0x	0xA5
Octale	0o	0o245
Binaire	0b	0b10100101

Les réels

Les réels, au sens mathématique, n'existent pas dans les langages de programmation. Ils sont exprimés par des nombres dits "à virgule flottante" ou **flottants** (cf. IEEE754) qui représentent une approximation rationnelle. Il existe deux formes standarisées de flottants : simple précision (32 bits) et double précision (64 bits).

Python implémente seulement les flottants en double précision.

Les décimaux

Permettent de s'affranchir des problèmes de perte de précision inhérents au flottants pour représenter des nombres rationnels. Leur utilisation nécessite l'importation du module **decimal** .

Les nombres complexes

Les nombres complexes sont représentés par une paire de flottants pour les parties réelle et imaginaire. La partie imaginaire est suffixée par la lettre **j** ou **J** .

```
>>> c=3+4j
>>> c
(3+4j)
>>> type(c)
<type 'complex'>
>>> c/2
(1.5+2j)
>>> c.real
3.0
>>> c.imag
4.0
>>> abs(c)
5.0
>>>
```

Les opérateurs arithmétiques, par ordre croissante de leur priorité sont :

Opérateurs	Forme	Commentaire
+, -,	binaire, infixe	même chose qu'en C/C++, Java etc.
*, /, //, %	binaire, infixe	particularité: deux opérateurs de division: / et //, op.modulo: %
+x, -x, ~	unaire, préfixe	+/- signe d'une expression, ~ négation binaire (bit à bit)
x ** y	binaire, infixe	x^y

Note: Pourquoi deux opérateurs de division? Initialement, Python disposait du seul opérateur de division, **/**, au comportement identique à celui du C:

- division entière si les deux opérandes sont entiers
- division réelle si au moins un des opérandes est réel

Malgré sa compatibilité avec le C, ce fonctionnement n'a pas eu l'adhésion de la communauté et il a été décidé depuis longtemps de le changer dans la version 3 de Python avec une division réelle, non tronquée, y compris quand les deux opérandes sont des entiers.

Pour préparer ce changement important, la version 2.2 a introduit l'opérateur // qui fait une division tronquée de manière homogène, pour tous les types numériques, l'opérateur / restant inchangé en attendant la version 3.

Pour résumer:

Opérateur	Python 2.x	Python 3.x
/	**compatible C/C++/Java** : division entière si les 2 opérandes sont entiers, réelle si au moins un des deux est réel	Division réelle, même si le deux opérandes sont entiers
//	Division tronquée, même si les opérandes sont réels	Comportement inchangé entre les versions 2.x et 3.x

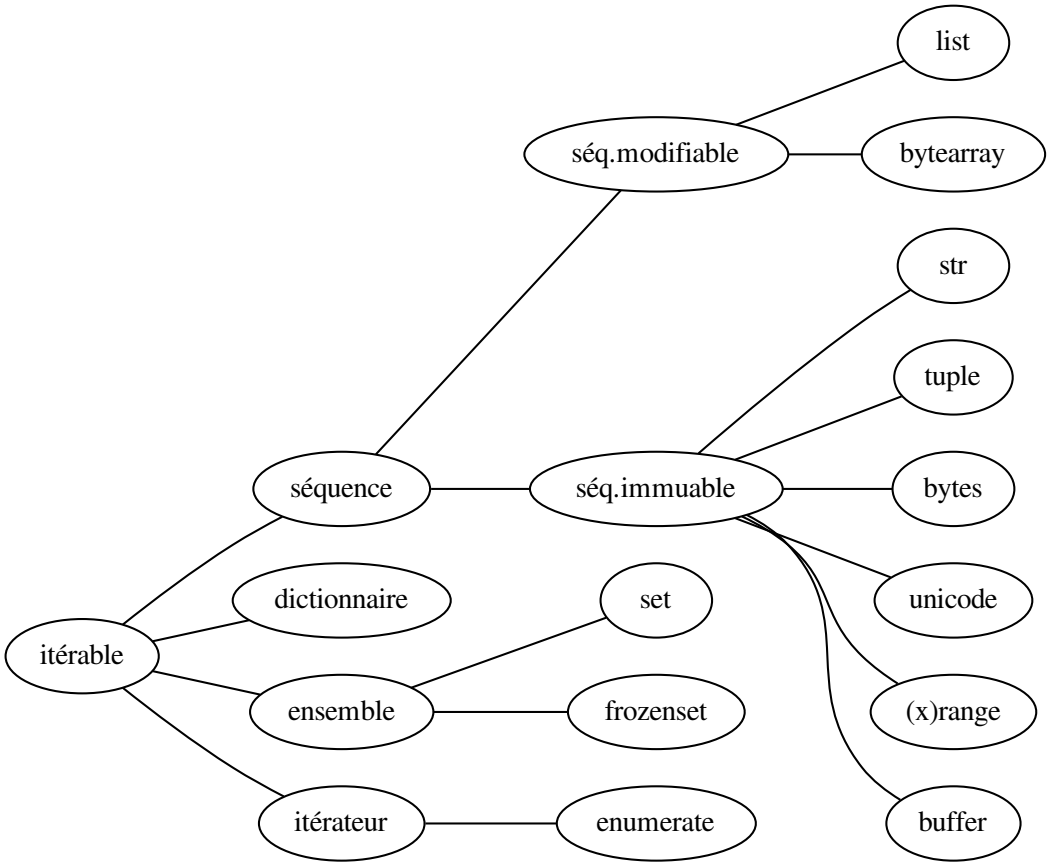
4.4.2 Les itérables prédéfinis

Il s'agit d'objets prédéfinis qui sont itérables au sens de l'explication fournie dans la section sur l'instruction **for-in**. Ils sont, pour la plupart, des **conteneurs**, autrement dit des objets qui ont vocation à contenir d'autres objets (listes, ensembles, etc.).

Certains itérables prédéfinis sont plus "opaques": ils ne contiennent pas physiquement d'autres objets mais ils les créent au fur et à mesure de la demande (range/xrange).

Les itérables fournissent au moins les fonctionnalités suivantes:

- accès aux objets contenus un par un, autrement dit ils sont utilisables par l'instruction **for** (*for elt in conteneur:...*)
- test d'appartenance d'un objet au conteneur, utilisable par l'instruction **if** (*if elt in conteneur*)
- nombre d'éléments du conteneur (ou sa longueur, i.e. **len(conteneur)**). Exception: *enumerate*
- le plus petit et le plus grand élément du conteneur (si une relation d'ordre existe entre les éléments). Optionellement, la relation d'ordre peut être définie par une fonction particulière (plus de détails au chapitre sur les **fonctions**)



Les séquences, les dictionnaires et les ensembles, ainsi que leur sous-types concrets sont des containers spécialisés, proposant des fonctionnalités spécifiques en plus de celles déjà énumérées.

4.4.3 Les séquences

On appelle **séquence** une collection dont les éléments sont ordonnés et indexés.

Pour faire une parallèle avec d'autres langages, les tableaux du **C** ainsi que les objets de type **std::Vector** en **C++** et **java.util.Vector** en **Java** sont des séquences.

La convention de numérotation des index, héritée du **C**, fait que pour une séquence de taille *N* les index prennent des valeurs entre *0* et *N-1*.

En plus des fonctionnalités communes à tous les conteneurs, les séquences proposent deux propriétés spécifiques:

- l'accès direct aux éléments à travers leur index:
  - avec des index positifs: 1er élément: seq[0], dernier élément: seq[taille-1]
  - avec des index négatifs: 1er élément: seq[-taille], dernier élément: seq[-1] (**NB:** index $_{[-]}$ = index $_{[+]}$ - taille)
- la possibilité d'extraire des tranches d'une séquence donnée (slicing)
  - seq[i:j] avec i=premier **dans** la tranche, j=premier **dehors**
  - seq[i:j:p] où p est le **pas** qui peut être positif ou négatif

```
>>> s="Python"
>>> s
'Python'
>>> s[1:3]
'yt'
>>> s[1:-1]
'ytho'
>>> s[1:5:2]
'yh'
>>> s[5:1:-2]
'nh'
>>>
```

NB: i, j et p peuvent être absents:

- i absent => La seule manière de sélectionner le **premier** élément de la séquence quand le pas est **négatif**
- j absent => La seule manière de sélectionner le **dernier** élément de la séquence quand le pas est **positif**
- p absent => pas = 1

NB(2): Dans le slicing les index hors limites sont tronqués implicitement, sans message d'erreur:

```
>>> s[-100:100]
'Python'
>>> s[100:-100:-1]
'nohtyP'
>>>
```

Par contre, une tentative d'accès indexé à une position inexistante provoquera une erreur:

```
>>> s[100]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>>
```

Exercice

```
"""
fichier: slicing.py

A partir de la chaine de caractères s, extraire, en utilisant le slicing
a) la chaîne 8jJ
b) la chaîne contenant toutes les majuscules
c) la chaîne contenant tous les chiffres, par odre décroissant
d) la chaîne initiale inversée
"""
s = 'aA0bB1cC2dD3eE4fF5gG6hH7iI8jJ9'

"""
fichier: slicing_sol.py
"""
#a
s[-4:-1]
#b
s[1::3]
#c
s[::-3]
#d
s[::-1]
```

Toutes les séquences supportent les méthodes:

- *seq.index(elt,[first,[last]])* # index de la première occurence de *elt*
- *seq.count(elt)* # nombre d'occurences de *elt*

Note: Toutes les séquences peuvent être triées avec la fonction native **sorted(seq,...)** qui sera détaillée au chapitre **fonctions**.

Selon leur capacité à être modifiées ou pas, on va parler de séquences **modifiables** respectivement ou non-modifiables (ou **immuables**) .

Les séquences modifiables (list et bytearray)

Il y a seulement deux types de séquences modifiables, les **list** et les **bytearray**. Les opérations supplémentaires supportées par les séquences modifiables sont:

- modification d'une position ou d'une plage de positions
- ajout/suppression d'un élément ou d'une (sous)séquence

Ainsi, toutes les expressions de sélection d'un élément ou d'une sous-séquence pourront apparaître en partie gauche d'une affectation ou en argument de l'instruction de suppression **del**

Les modifications peuvent se faire également à travers les méthodes natives suivantes :

- *seq.append(x)* # ajout d'un élément à la fin de la séquence
- *seq.extend(seq2)* # extension de seq avec seq2
- *seq.insert(i,x)* # insertion de l'élément x en position i
- *seq.pop(i)* # retourne et enlève l'élément en position i
- *seq.remove(x)* # supprime la première occurrence de x dans seq
- *seq.reverse()* # inverse l'ordre des éléments
- *seq.sort(...)* # tri (sera détaillée, au chapitre **fonctions**)

Note: La différence entre *seq.sort(...)* et *sorted(seq,...)* est que *seq.sort(...)* modifie *seq* et renvoie *None* pendant que *sorted(seq,...)* laisse *seq* inchangé et renvoie une nouvelle séquence, triée.

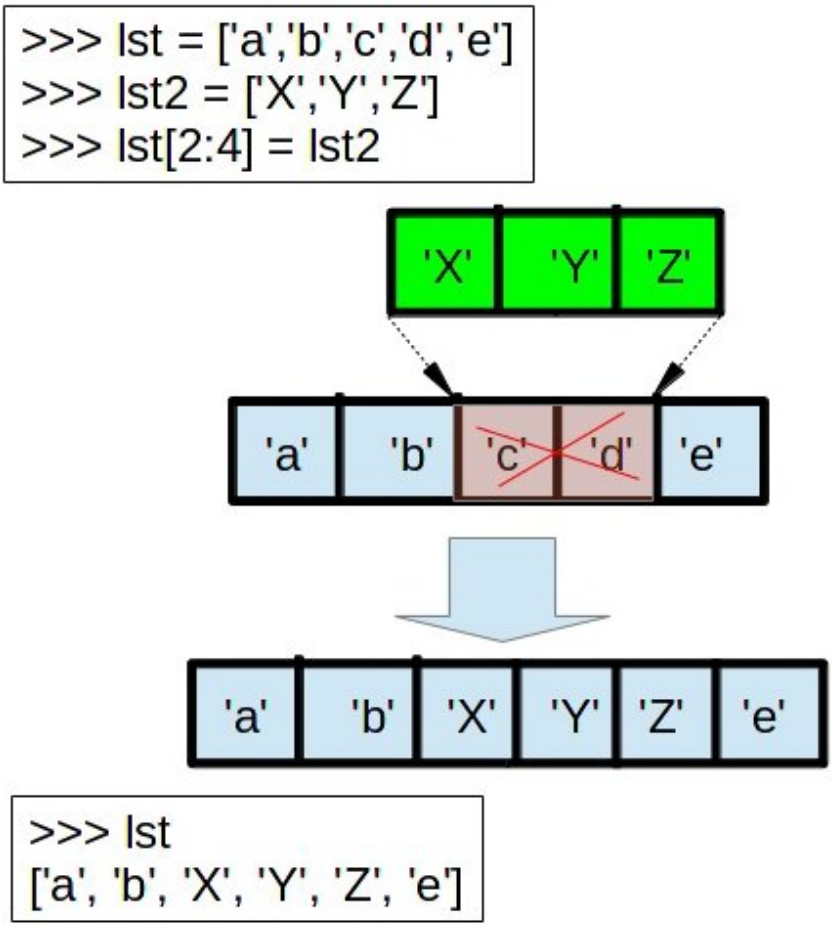
Illustration avec le type **list**, qui porte assez mal son nom (son comportement est plus proche des vecteurs que des listes (chaînées) présentes dans d'autres langages:

Affectation :

```
>>> lst
['P', 'y', 't', 'h', 'o', 'n']
>>> lst[3]="H"
>>> lst
['P', 'y', 't', 'H', 'o', 'n']
>>> lst[1:4] = ["l","a","t"]
>>> lst
['P', 'l', 'a', 't', 'o', 'n']
>>>
```

NB: En cas d'affectation avec slicing, la tranche substituée peut avoir une taille différente de celle la liste substitution:

```
>>> lst = ['a','b','c','d','e']
>>> lst2 = ['X','Y','Z']
>>> lst[2:4] = lst2
>>> lst
['a', 'b', 'X', 'Y', 'Z', 'e']
```



Délétion :

```
>>> lst
['P', 'l', 'a', 't', 'o', 'n']
>>> del lst[3:-1]
>>> lst
['P', 'l', 'a', 'n']
>>>
```

Primitives :

```
>>> lst = list("Monty")
>>> lst.append(" ")
>>> lst
['M', 'o', 'n', 't', 'y', ' ']
>>> lst2 = list("Python")
>>> lst.extend(lst2)
>>> lst
['M', 'o', 'n', 't', 'y', ' ', 'P', 'y', 't', 'h', 'o', 'n']
>>>
>>> lst2
['P', 'y', 't', 'h', 'o', 'n']
>>> lst2.insert(2,"*")
>>> lst2
['P', 'y', '*', 't', 'h', 'o', 'n']
>>> lst2.pop(2)
'*'
>>> lst2
```

```
['P', 'y', 't', 'h', 'o', 'n']
>>>
>>> lst2.remove("h")
>>> lst2
['P', 'y', 't', 'o', 'n']
>>>
>>> lst2.reverse()
>>> lst2
['n', 'o', 't', 'y', 'P']
>>>
>>> lst2.sort()
>>> lst2
['P', 'n', 'o', 't', 'y']
>>>
```

Astuce: Pour obtenir une copie (superficielle) d'une liste **lst**, on peut utiliser le slicing: **lst[:]**

Les listes en intension (ou compréhension)

Un problème récurent en programmation est la construction d'une liste à partir d'une liste existante (ou de tout autre ensemble d'objets existant). On souhaite souvent obtenir :

1. une nouvelle liste de la même taille que la liste en entrée (Ex: la liste des carrés d'une liste d'entiers fournie)
2. une liste avec filtrage (Ex: liste des carrés des entiers impaires se trouvant dans la liste fournie)

Une construction de ces listes avec une boucle **for-in** classique est, bien sûr, possible.

Premier cas:

```
>>> result=[]
>>> for e in [1,2,3]:
...     result.append(e**2)
...
>>> result
[1, 4, 9]
>>>
```

Deuxième cas:

```
>>> result=[]
>>> for e in [1,2,3,4,5]:
...     if e % 2:
...             result.append(e**2)
...
>>> result
[1, 9, 25]
>>>
```

Python propose en plus, une écriture plus concise, plus élégante aussi : c'est la construction de listes dites "en compréhension" ou "en intension" (antonyme du mot "extension") inspirée des notations utilisées dans la théorie des ensembles.

Avec cette notation, le premier cas (sans filtre) s'écrit ainsi:

```
>>> [e**2 for e in [1,2,3]]
[1, 4, 9]
>>>
```

et le deuxième (avec filtre):

```
>>> [e**2 for e in [1,2,3,4,5] if e%2]
[1, 9, 25]
>>>
```

L'écriture de boucles imbriquées est également possible.

L'écriture classique pour obtenir un pseudo produit cartésien ("pseudo", car les listes ne sont pas des ensembles au sens mathématique):

```
>>> result=[]
>>> for c in ["a","b","c"]:
...     for n in [1,2,3]:
...             result.append([n,c])
...
>>> result
[[1, 'a'], [2, 'a'], [3, 'a'], [1, 'b'], [2, 'b'], [3, 'b'], [1, 'c'], [2, 'c'], [3, 'c']]
>>>
```

Devient:

```
>>> [[n,c] for c in ["a","b","c"] for n in [1,2,3]]
[[1, 'a'], [2, 'a'], [3, 'a'], [1, 'b'], [2, 'b'], [3, 'b'], [1, 'c'], [2, 'c'], [3, 'c']]
>>>
```

Exercice

```
"""
fichier: comprehension.py

Soient les listes L1 et L2.
NB: les deux listes contiennent seulement des éléments uniques

En utilisant des listes en intension construire:
a) l'intersection des deux listes (considérées comme des ensembles)
b) la différence ensembliste L1 - L2
c) Le sous-ensemble du produit cartésien L1 X L2 défini ainsi:
   {[e1, e2]|e1 appartenant à L1, e2 appartenant à L2, tq e1>=e2}
"""
L1 = [1, 2, 3, 4, 5, 6]
L2 = [4, 5, 6, 7, 8, 9]
```

```
"""
fichier: comprehension_sol.py
"""
from comprehension import L1, L2
# a) l'intersection
intersection = [elt for elt in L1 if elt in L2]

# b) L1 - L2
L1_L2 = [elt for elt in L1 if elt not in L2]

# c) sous ensemble de L1 x L2 tq e1 >= e2
sub_prod = [[e1, e2] for e1 in L1 for e2 in L2 if e1 >= e2]
```

Les séquences immuables

En Python, les types suivants correspondent à cette classification:

- Le type **tuple** (déjà évoqué avec l'affectation multiple)
- Les types **str**, **bytes** et **unicode** sont liés à la représentation des caractères et seront traités ensemble.
- Le type **buffer/memoryview** donne accès à l'image mémoire d'un objet (supportant un protocole particulier) pour un traitement efficace des gros volumes de données. Il relève d'une utilisation avancée et ne sera pas traité ici.

- Le type **xrange/range** est une séquence de valeurs entières contiguës dans une plage donnée. Son utilité (par rapport à une liste) est l'économie de mémoire (dans les instructions **for** par exemple)

Les tuples

Sont des tableaux très simples et assez similaires aux tableaux du **C** (par exemple: `int int_var[5] = {1,3,5};`, mais leurs éléments peuvent être hétérogènes. Ils peuvent se définir de deux manières:

- en énumérant leurs éléments, séparés par des virgules (souvent, mais pas toujours, entre parenthèses pour éviter les ambiguïtés syntaxiques)
- en passant une autre itérable (une liste, par exemple) en argument au constructeur *tuple()*

```
>>> 1,"a",[3,4]
(1, 'a', [3, 4])
>>> (1,"a",[3,4])
(1, 'a', [3, 4])
>>> tuple([1,"a",[3,4]])
(1, 'a', [3, 4])
>>>
```

Note: Syntaxiquement, ce ne sont pas les parenthèses qui définissent le tuple, mais la virgule utilisée comme séparateur entre ses éléments. Dans un tuple à un seul élément l'unique élément doit être suivi d'une virgule :

```
>>> (1) #ceci n'est pas un tuple
1
>>> (1,) #ceci est un tuple
(1,)
>>> 1, #ceci aussi est un tuple
(1,)
>>>
```

Une fois créé, le tuple est immuable mais il peut contenir des éléments modifiables:

```
>>> t=(1,"a",[3,4])
>>> t[2] = 9
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> t[2][1]=9
>>> t
(1, 'a', [3, 9])
>>>
```

Une séquence contenant des variables, et en particulier un tuple peut être lvalue dans une affectation:

```
>>> [a, b] = ["x", "y"] # ou ("x", "y") ou "xy"
>>> a
'x'
>>> b
'y'
>>> (a, b) = ["x", "y"]
>>> a
'x'
>>> b
'y'
```

Les parenthèses n'étant pas obligatoires, on peut écrire:

```
>>> a, b = ("x", "y")
>>> a
'x'
```

```
>>> b
'y'
```

ou:

```
>>> a, b = "x", "y"
>>> a
'x'
>>> b
'y'
```

qu'on a appelé précédemment "affectation multiple".

Les chaînes de caractères (str, mais aussi bytes et unicode)

On est obligé d'évoquer trois types (*str*, *bytes* et *unicode*) pour expliquer le propriétés des chaînes de caractères en Python.

Il convient de préciser que c'est le type **str** qui représente, par défaut, les chaînes de caractères en Python. Les deux autres, *bytes* et *unicode*, ont joué et jouent encore un rôle de transition, pour aider à la prise en charge des mutations que le type **str** a subi entre les versions 2 et 3 du langage.

Dans les explications qui vont suivre il s'agit du type **str** implicitement. Les références ponctuelles aux deux autres types seront explicites.

Les chaînes de caractères sont des séquences destinées à manipuler du texte. En Python, plusieurs syntaxes sont acceptées. Un peut utiliser les simples et les doubles quotes :

```
>>> print('un texte')
un texte
>>> print("un texte")
un texte
```

On peut écrire une chaîne sur plusieurs lignes (si elle est trop longue, par exemple). Cette écriture est sans impact sur le contenu de la chaîne (elle n'introduit pas des retours à la ligne) :

```
>>> print("si un texte est trop long \
... on peut l'écrire \
... sur plusieurs lignes")
si un texte est trop long on peut l'écrire sur plusieurs lignes
```

On peut introduire des retours à la ligne de deux manières:

- avec la séquence d'échappement **\n** (norme *POSIX*)

  ```
  >>> print("première ligne \n deuxième ligne")
  première ligne
  deuxième ligne
  >>> print("première ligne \n\
  ... deuxième ligne")
  première ligne
  deuxième ligne
  ```

- Avec des triples quotes (spécifique Python):

  ```
  >>> print("""
  ... première ligne
  ... deuxième ligne
  ... """)

  première ligne
  deuxième ligne
  ```

Dans certaines situations, l'interprétation des séquences d'échappement (\n, \r etc.) n'est pas souhaitable, par exemple, dans les expressions rationnelles (regular expressions). Dans ce cas, on peut utiliser les chaînes brutes (raw strings), introduites par le préfixe **"r"** :

```
>>> print(r"première ligne \n deuxième ligne")
première ligne \n deuxième ligne
```

Les chaînes sont des instances la classe **str** et elles supportent toutes les opérations sur les séquences non modifiables (indexing, slicing, etc.)

Elles supportent également des opérations spécifiques implémentés en tant que fonctions ou accessibles en tant que méthodes et opérateurs sur la classe **str**.

Les opérateurs supportés par les chaînes sont:

- **+** concaténation (Exemple : `"abc" + "xyz"`)
- ***** multiplication (Une répétition, en réalité. Exemple : `"abc" * 3`)
- **%** opérateur d'interpolation, expliqué en détail plus loin

Deux fonctions spécifiques, *chr()* et *ord()*, particulièrement importantes car elles font le lien entre un caractère et le code (ASCII ou Unicode, selon le cas) correspondant:

```
>>> chr(65)
'A'
>>> ord('A')
65
```

Les méthodes sur les chaînes sont nombreuses, illustration en utilisant l'autocomplétion (déclenchée par la tabulation) dans **ipython** pour voir leur liste exhaustive:

```
In [1]: s="abc"

In [2]: s.
s.capitalize  s.endswith    s.isalnum     s.istitle     s.lstrip      s.rjust       s.splitlines
s.center      s.expandtabs  s.isalpha     s.isupper     s.partition   s.rpartition  s.startswith
s.count       s.find        s.isdigit     s.join        s.replace     s.rsplit      s.strip
s.decode      s.format      s.islower     s.ljust       s.rfind       s.rstrip      s.swapcase
s.encode      s.index       s.isspace     s.lower       s.rindex      s.split       s.title
```

Exemples:

- quelques méthodes de test (retournant un booléen) :

  ```
  >>> s="abc"
  >>> s.islower()
  True
  >>> s.isupper()
  False
  >>> s.startswith("ab")
  True
  >>> s.startswith("x")
  False
  ```

- quelques méthodes produisant une chaîne à partir de la chaîne donnée:

  ```
  >>> s.upper()
  'ABC'
  >>> s.title()
  'Abc'
  ```

Pour savoir plus sur ces méthodes, consulter: http://docs.python.org/2/library/stdtypes.html#string-methods

Exercice

```
"""
fichier: lcd.py

Ecrire un programme qui demande la saisie d'un entier
(sur plusieurs chiffres) et qui affiche le numéro saisi
à la manière des affichages LCD en utilisant la liste
'LCD' fournie plus bas.
NB: Si besoin, on pourra utiliser la primitive
range(n) qui retourne la liste [0,1,...,n-1]
"""

LCD = [["+--+",
        "|  |",
        "+  +",
        "|  |",
        "+--+"],
       ["+",
        "|",
        "+",
        "|",
        "+"],
       ["+--+",
        "   |",
        "+--+",
        "|   ",
        "+--+"],
       ["+--+",
        "   |",
        "+--+",
        "   |",
        "+--+"],
       ["+   ",
        "|   ",
        "+--+",
        "   |",
        "   +"],
       ["+--+",
        "|   ",
        "+--+",
        "   |",
        "+--+"],
       ["+--+",
        "|   ",
        "+--+",
        "|  |",
        "+--+"],
       ["+--+",
        "   |",
        "   +",
        "   |",
        "   +"],
       ["+--+",
        "|  |",
        "+--+",
        "|  |",
        "+--+"],
       ["+--+",
        "|  |",
        "+--+",
        "   |",
        "+--+"]]
```

```
"""
fichier: lcd_sol.py
"""

from __future__ import print_function
from lcd import LCD # i.e. la liste de l'ennoncé
import sys

prompt = "Saisir un numéro: "
str_number = input(prompt) if sys.version_info.major==3 else raw_input(prompt)

print(str_number)

for ln in range(5):
    line = ""
    for n in str_number:
        d = LCD[int(n)]
        line += " "
        line += d[ln]
    print(line)
```

Le formatage des chaînes de caractères

Depuis la version **2.6** les chaînes peuvent être formatées en utilisant la méthode *format()* et un "mini-langage" puissant de description des formats. Précédemment, le formatage était fait grâce à l'opérateur d'interpolation, expliqué plus loin.

Le spécificateur de substitution le plus simple est **{}** :

```
>>> "Langage: {}, version: {}".format("Python",3)
'Langage: Python, version: 3'
```

Dans ce cas, l'ordre des substitutions dans la chaîne est identique à l'ordre des arguments de l'appel.

On peut (ré)utiliser les substitutions dans un ordre arbitraire en numérotant les spécificateurs:

```
>>> "Que choisir: {2} {1} ou {2} {0} ?".\
... format(3, 2, 'Python')
'Que choisir: Python 2 ou Python 3 ?'
```

Si un des arguments de *format()* est une **liste**, les spécificateurs peuvent accéder ses éléments de cette liste par leur **index** :

```
>>> versions=[2, 3]
>>> "Que choisir: {1} {0[0]} ou {1} {0[1]} ?".\
... format(versions,'Python')
'Que choisir: Python 2 ou Python 3 ?'
```

Si un des arguments de *format()* est un **dictionnaire**, les spécificateurs peuvent accéder ses éléments de cette liste par leur **clé** :

```
>>> versions={'old_v':2, 'new_v':3}
>>> "Que choisir: {1} {0[old_v]} ou {1} {0[new_v]} ?".\
... format(versions,'Python')
'Que choisir: Python 2 ou Python 3 ?'
```

Les spécificateurs peuvent accéder aux attributs d'un objet ou d'un module:

```
>>> import sys
>>> sys.version_info.major
2
>>> sys.version_info.minor
7
>>> sys.version_info.micro
3
```

```
>>> "{1} {0.major}.{0.minor}.{0.micro}".\
... format(sys.version_info,"Python")
'Python 2.7.3'
```

Dans les exemples précédents, les arguments de *format()* sont identifiés par leur position à l'appel, mais on peut également utiliser des arguments nommés.

Les exemples précédents peuvent être réécrits ainsi:

```
>>> "Langage: {langage}, version: {version}".\
... format(langage="Python",version=3)
'Langage: Python, version: 3'

>>> "Que choisir: {lang} {old_v} ou {lang} {new_v} ?".\
... format(new_v=3, old_v=2, lang='Python')
'Que choisir: Python 2 ou Python 3 ?'

>>> versions=[2, 3]
>>> "Que choisir: {lang} {v_list[0]} ou {lang} {v_list[1]} ?".\
... format(v_list=versions,lang='Python')
'Que choisir: Python 2 ou Python 3 ?'

>>> versions={'old_v':2, 'new_v':3}
>>> "Que choisir: {lang} {v_dict[old_v]} ou {lang} {v_dict[new_v]} ?".\
... format(v_dict=versions,lang='Python')
'Que choisir: Python 2 ou Python 3 ?'

>>> import sys
>>> "{lang} {v_info.major}.{v_info.minor}.{v_info.micro}".\
... format(v_info=sys.version_info,lang="Python")
'Python 2.7.3'
```

NB: Si on doit générer des accolades, il suffit de les doubler:

```
>>> "Ensemble vide: {{}},1 elt:{{{0}}}, 2 elts:{{{0},{1}}}".\
... format('a','b')
'Ensemble vide: {},1 elt:{a}, 2 elts:{a,b}'
```

NB: Dans les cas précédents, les formats n'étant pas spécifiés, les arguments sont convertis en **str** avec la conversion par défaut.

En général les syntaxes suivantes sont correctes:

- **{}** : (voir les cas précédents)
- **{id}** : (voir les cas précédents)
- **{id!conversion}}**
- **{id:format}**
- **{id!conversion:format}}**

Les conversions

Trois types de conversions sont possibles:

- **s** : force la conversion en chaîne de caractères (**str**) qui est la conversion par défaut
- **r** : force la forme "représentative"
- **a** : (Python 3 seulement) force la forme "représentative" en utilisant seulement les caractères ASCII

```
>>> import datetime
>>> d = datetime.datetime(2014,12,1)
>>> d # affiche la forme représentative
datetime.datetime(2014, 12, 1, 0, 0)
```

```
>>> str(d)
'2014-12-01 00:00:00'
>>> "{0} || {0!s} || {0!r}".format(d)
'2014-12-01 00:00:00 || 2014-12-01 00:00:00 || datetime.datetime(2014, 12, 1, 0, 0)'
```

Les formatage proprement dit

Pour les chaînes de caractères il s'agit de définir:

- caractère de remplissage
- alignement (à gauche, à droite, au centre)
- largeur minimum du champ

Plus spécifiquement, pour les nombres il s'agit de préciser:

- la base de numération pour les entiers (binaire, octal, décimal, hexadécimal ...)
- la précision pour les flottants (6 par défaut)
- la notation pour les flottants (exponentielle, décimale)
- le traitement du signe

La syntaxe générale du descripteur est: *[[fill]align][sign][#][0][width][,][.precision][type]* avec :

- **fill** : tout caractère
- **align** :
  - **'<'** : à gauche
  - **'>'** : à droite
  - **'^'** : au centre
  - **'='** : pour les nombres, le remplissage (fill) se fera entre le signe et le nombre
- **sign** :
  - **'+'** : toujours afficher le signe, '+' ou '-'
  - **'-'** : afficher que le '-', le '+' étant implicite (comportement par défaut)
  - **' '** : un espace sera utilisé à la place du '+' pour les nombres positifs
- **#** : indique si les entiers doivent être préfixés par *0b*, *0o*, *0x*, selon leur base
- **0** : remplissage avec des zéros pour les nombres (raccourci équivalent à fill:'0' et align:'=')
- **width** : largeur minimum du champ
- **precision** : pour les flottants
- **type** :
  - **s** : chaîne de caractères (par défaut, omis d'habitude)
  - **d,i** : entier (en décimal)
  - **b** : entier en binaire
  - **o** : entier en octal
  - **x,X** : entier en hexadécimal
  - **e,E** : flottant (notation exponentielle)
  - **f,F** : flottant (notation décimale)
  - **g,G** format général selon la grandeur du nombre, appliquera un de deux types précédents pour obtenir la représentation la plus compacte

Pour en savoir plus : https://docs.python.org/3/library/string.html#format-specification-mini-language

Exemples:

Alignements :

::

```
>>> '|{:<20}|{:^20}|{:>20}|'.format('gauche', 'centre', 'droite')
'|gauche              |       centre       |              droite|'
>>> '|{:_<20}|{:=^20}|{:->20}|'.format('gauche', 'centre', 'droite')
'|gauche______________|=======centre=======|--------------droite|'
```

Affichage de flottants:

```
>>> import math
>>> math.pi
3.141592653589793
>>> '|{0:<20.2f}|{0:^+20.3f}|{0:>20.4f}|'.format(math.pi)
'|3.14                |       +3.142       |              3.1416|'
```

L'opérateur d'interpolation (%)

Note: L'utilisation de cet opérateur n'est plus préconisée, même s'il est toujours présent dans Python 3. Le nouveau standard est la méthode `format()`, présentée précédemment.

Il rappelle la fonction **sprintf()** du **C** et il est de la forme:

format % substitution

- La partie **format** est une chaîne contenant 0..n spécificateurs de substitution
- La partie **substitution** peut contenir:
  - un tuple de valeurs ou un dictionnaire d'une taille **N** égale au nombre de spécificateurs si N!=1
  - un objet d'un autre type pour une substitution unique si N==1

La syntaxe d'un spécificateur de substitution dans la partie *format* est **%**[kfmpl]**T**:

Seuls sont obligatoires le **%** initial et le **T** final qui représente le type de conversion encodé par une lettre:

- **s** chaîne de caractères
- **d,i** entier (en décimal)
- **o** entier en octal
- **x,X** entier en hexadécimal
- **e,E** flottant (notation exponentielle)
- **f,F** flottant (notation décimale simple)
- ...

Exemple:

```
>>> "Langage: %s, version: %d" % ("Python", 3)
'Langage: Python, version: 3'
>>>
```

Le même exemple avec des spécificateurs nommés :

```
"Langage: %(langage)s, version: %(version)d" % {"langage":"Python", "version":3}
'Langage: Python, version: 3'
```

Les options ([kfmpl]), qui peuvent apparaître entre le **%** et le **T**:

- **k** la clé indiquant le nom du spécificateur entre parenthèses (cf. exemple précédent avec des spécificateurs nommés et dictionnaire)

- **f** paramétrage pour certaines conversions comme, par exemple, justification à gauche, remplissage avec des zéros etc. (détails ici: http://docs.python.org/2/library/stdtypes.html#string-formatting-operations).
- **m** taille minimum du résultat
- **p** précision (entier précédé par un point)
- **l** peut être l,h ou L. Ce paramètre est ignoré (gardé juste pour la compatibilité avec le **C**)

Exemple:

```
>>> import math
>>> math.pi
3.141592653589793
>>> "PI: %10.2f" % math.pi
'PI:       3.14'
```

Plus de détails sur l'ensemble des options **[kfmpl]** ainsi que la liste exhaustive de types de conversion sont accessibles ici: http://docs.python.org/2/library/stdtypes.html#string-formatting-operations

NB: En Python, il n'existe pas de type "caractère". Un élément d'une chaîne est à son tour une chaîne à un seul élément:

```
>>> type("foo")
<type 'str'>
>>> s="foo"
>>> type(s)
<type 'str'>
>>> s[1]
'o'
>>> type(s[1])
<type 'str'>
>>> s[1][0]
'o'
```

Note: Mais qu'est-ce qu'un caractère ?

- Pour l'utilisateur, il s'agit d'une lettre, d'un symbole, d'un contrôle
- Pour un langage de programmation, un caractère à été longtemps un octet, un code *ASCII*. Avec l'internationalisation, les choses ont changé...

A ses débuts, et jusqu'à une date récente, *Python* a représenté les caractères par des octets encodés en *ASCII*, comme bien d'autres langages (n'oublions pas qu'en **C** le type **char** a la taille d'un octet!). Le type **str** a été, avant la version 3, une séquence d'octets. Cette approche a montré ses limites assez tôt et la version 2.0 introduisait déjà un nouveau type, appelé **unicode** ...

Note: Un peu d'histoire...

- Au départ (1963), il y a eu *ASCII* (American Standard Code for Information Interchange):
  - 128 codes (2^7 car encodage sur 7 bits seulement)
  - bijection "caractère <=> octet"
  - jeu de caractères limité aux besoins de l'anglais (pas de caractères accentués)
- Plus tard (1987), une première tentative d'internationalisation, la norme IS0/CEI 8859:
  - 256 codes (2^8)
  - sur-ensemble *ASCII*
  - déclinée en plusieurs jeux de caractères nationaux, dont *ISO 8859-1* (ou *Latin-1*) pour l'Europe de l'Ouest.
  - (+) bijection "caractère <=> octet" préservée

  - (-) impossible de combiner plusieurs jeux de caractères dans un même document (par exemple : un texte en français avec des citations en grec)
  - (-) pas de solution pour les jeux de caractères riches (chinois, par exemple)
- 1990 *Unicode*, standard qui définit un jeu de caractères unique
  - 1114112 codes (de 0x00 à 0x10FFFF)
  - couvrant la plupart des alphabets connus, symboles mathématiques etc.(http://www.unicode.org/charts/)
  - abandon de la représentation "octet", au profit du "code point" (point code), numéro unique pour chaque caractère, implémenté dans les langages comme un **int**
  - reste un sur-ensemble *ASCII* (malgré l'abandon de la représentation octet)
  - bijection "caractère <=> code point"

La représentation des caractères par des points code ne pose pas de problème tant qu'il n'y a pas d'opération d'entrée-sortie. Par contre, les points code, qui sont des entiers, ne peuvent pas être utilisés tels quels dans les opérations d'entrée-sortie pour au moins deux raisons: .. la représentation interne des points code n'est pas standardisée. Par exemple, un langage peut très bien les représenter avec des entiers 64 bits, un autre avec des int 32 bits.

- la représentation en mémoire des points code peut varier en fonction :
  - des choix d'implémentation, les points code étant simplement des entiers
  - de l'architecture physique (*little endian* ou *big endian*)
- tous les protocoles et *APIs* d'entrée/sortie sont conçus pour manipuler des octets.

Que faire (pour travailler en *Unicode*) **?**

- décoder les entrées (octets = > Unicode)
- encoder les sorties (Unicode => octets)

Les encodages **UTF** (Unicode Transformation Format) se déclinent en UTF-8, UTF-16 et UTF-32

UTF	Format	Commentaires
UTF-8	variable (1-4 octets)	Encodage par défaut dans Python 3.x
UTF-16	variable (1-2 mots de 16 bits)	Utilisé par Windows et Java
UTF-32	fixe (32 bits)	Non supportée dans Python

Remarque: Les fichiers *UTF** peuvent contenir au début une signature de 4 octets appelée **BOM** (Byte Order Mark). Non obligatoire pour *UTF-8*, elle permet de déterminer l'ordre des octets à la lecture d'un fichier UTF-16 ou UTF-32 car ces deux formats dépendent de l'*endianité*.

Pour en savoir plus sur *Unicode* et *UTF*: http://www.unicode.org/faq/utf_bom.html

Le détail de l'encodage *UTF-8*:

Octets	Bits codants	Format
1	7	0xxxxxxx (*ASCII*)
2	11	110xxxxx 10xxxxxx
3	16	1110xxxx 10xxxxxx 10xxxxxx
4	21	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Le traitement des chaînes de caractères est un point majeur de divergence entre les versions 2 et 3 de Python.

Les chaînes de caractères en Python 2

- l'encodage par défaut est l'*ASCII*:

```
Python 2.7.3 (default, Sep 26 2013, 20:03:06)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.getdefaultencoding()
'ascii'
```

- l'encodage par défaut ne s'applique pas pour l'entrée et pour la sortie standard, qui utilisent l'encodage de votre environnement (la variable `$LANG` sous Linux). Dans les environnements Linux récents, *LANG=UTF-8* par défaut.

```
Python 2.7.3 (default, Sep 26 2013, 20:03:06)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import sys
>>> sys.stdin.encoding
'UTF-8'
>>> sys.stdout.encoding
'UTF-8'
```

- Le type **str** est une séquence d'octets. En complément, depuis la version 2.0, on dispose du type **unicode** pour gérer les caractères *non-ASCII*.

Cette implémentation n'est pas sans conséquences :

- Les chaînes contenant des caractères non-ASCII sont plus longues que le nombre de symboles représentés:

```
>>> s = "Café"
>>> len(s)
5
>>> s
'Caf\xc3\xa9'
```

- Il n'y a pas de bijection entre chaque caractère et une position dans la séquence

Explication: La longueur affichée est de 5 au lieu de 4 car l'entrée standard utilise l'encodage du sytème d'exploitation qui est (dans ce cas) *UTF-8*. Le caractère **é** est encodé par les deux octets `\xc3\xa9`. Sur un système basé sur un autre encodage, le résultat pourrait être différent...

Pour rétablir la bijection *caractère* <=> *donnée* il faut décoder la chaîne pour obtenir une séquence *Unicode*. Cette opération nous réserve une nouvelle surprise:

```
>>> s="Café"
>>> s.decode()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position 3: ordinal not in range(128)
```

Enfin, en utilisant le bon format, *UTF-8*, on obtient :

```
>>> u=s.decode('UTF-8')
>>> u
u'Caf\xe9'
>>> type(u)
<type 'unicode'>
>>> print u
Café
>>> len(u)
4
```

Remarque: Un format de décodage inadapté ne produit pas forcément une erreur mais le résultat sera décevant:

```
>>> s="Café" # chaîne UTF-8
>>> u=s.decode("iso-8859-1") #mauvais format
>>> len(u)
```

```
5
>>> print u
CafÃ©
```

On peut concaténer objets unicode et objets str (le type de l'objet résultat sera unicode):

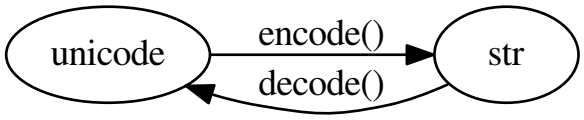
```
>>> s="Café"
>>> s2=" concert"
>>> u=s.decode("UTF-8")
>>> print u
Café
>>> print s2
concert
>>> u2=u+s2
>>> print u2
Café concert
>>> type(u2)
<type 'unicode'>
```

Le passage de *unicode* à *str* se fait par *encodage*, toujours en indiquant le bon format:

```
>>> u2
u'Caf\xe9 concert'
>>> print u2
Café concert
>>> u2.encode()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character u'\xe9' in position 3: ordinal not in ra
>>> u2.encode('UTF-8')
'Caf\xc3\xa9 concert'
>>> print 'Caf\xc3\xa9 concert'
Café concert
```

Conseil pour le traitement des chaînes d'entrée-sortie:

- **décoder** le plus tôt possible pour passer à **unicode**
- faire les traitements en **unicode**
- **encoder** le résultat le plus tard possible (pour la sortie)



En Python 2, le type **bytes** est un simple alias pour **str**

Les chaînes de caractères en Python 3

Le traitement des chaînes de caractères, avec l'adoption du standard *Unicode* pour le type *str*, est un des grands progrès de Python 3, mais les structures de données de la version précédente sont bouleversées:

Type	Python 2	Python 3
str	séquence d'octets	adopte la structure **unicode** de la V2
uni-code	séquence d'entiers (points code)	disparaît car devenu inutile
bytes	simple alias pour str	comportement inchangé mais ce n'est plus un alias, mais un type à part entière

De plus, l'encodage par défaut n'est plus l'*ASCII*, mais **UTF-8**.

```
Python 3.2.3 (default, Sep 25 2013, 18:22:43)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> s="Café"
>>> s
'Café'
>>> len(s)
4
```

Les opérations d'entrée sortie sont facilitées, surtout quand l'encodage 'UTF-8' est utilisé. On n'est pas complètement à l'abri des problèmes évoqué précédemment, car le format 'UTF-8', même s'il est de plus en plus répandu, il n'est pas unique. Des logiciels produisant des fichiers ISO-8859 sont encore en service, le format UTF-16 est utilisé dans certains environnements.

4.4.4 Les dictionnaires

Les dictionnaires sont des collections de couples clé-valeur donnant un accès efficace à chaque valeur à travers sa clé correspondante. Propriétés:

- Chaque clé est unique et non modifiable.
- Les éléments ne sont pas ordonnés
- Les valeurs associés aux clé peuvent être de n'importe quel type, modifiable ou pas
- Les dictionnaires sont des objets modifiables: on peut ajouter ou retirer des éléments, modifier les valeurs associées aux clés.

Syntaxiquement, l'élément générique est de la forme *<clé>:<valeur>*. Les éléments sont écrits entre accolades et séparés par des virgules:

```
>>> account = {'sn':'Poli','gn':'Christian','login':'cpoli','uid':1000,'gid':1000,'groups':[1005,
>>> account
{'sn': 'Poli', 'uid': 1000, 'gid': 1000, 'groups': [1005, 1007], 'login': 'cpoli', 'gn': 'Christi
```

Une écriture alternative utilisant la fonction native *dict()* (qui est un constructeur) :

```
>>> account = dict(sn='Poli',gn='Christian',login='cpoli',uid=1000, gid=1000, groups=[1005,1007])
>>> account
{'sn': 'Poli', 'gid': 1000, 'uid': 1000, 'groups': [1005, 1007], 'login': 'cpoli', 'gn': 'Christi
```

Remarque: Cette écriture alternative est réservée aux situations où les clés peuvent s'écrire sous forme d'identifiants et respectent donc la syntaxe des noms des variables.

Note: Pour garantir la cohérence des dictionnaires, seulement les objets immuables peuvent servir de clés.

L'implémentation des dictionnaires est basée sur des "hash tables" (tableaux associatifs) et les objets faisant office de clés doivent disposer d'une fonction de hashage, implémentée seulement par les objets immuables. Ainsi, par exemple, une chaîne de caractères peut servir de clé mais une liste ne le peut pas car:

```
>>> hash("Café")
7319500004332556263
>>> hash([1,2,3])
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Accès aux valeurs:

La syntaxe rappelle celle utilisée pour accéder aux valeurs d'une séquence modifiable, les index étant remplacés par des clés:

```
>>> account = {'sn':'Poli','gn':'Christian','login':'cpoli','uid':1000,'gid':1000,'groups':[1005,
>>> account['sn']
'Poli'
>>> account['login']='christian'
>>> account
{'sn': 'Poli', 'gn': 'Christian', 'gid': 1000, 'groups': [1005, 1007], 'login': 'christian', 'uid
>>> account['office']=14
>>> account
{'sn': 'Poli', 'gn': 'Christian', 'office': 14, 'gid': 1000, 'groups': [1005, 1007], 'login': 'ch
```

Bien sûr, le slicing n'existe pas, les clés n'étant pas ordonnées.

Suppressions:

```
>>> account = {'sn':'Poli','gn':'Christian','login':'cpoli','uid':1000,'gid':1000,'groups':[1005,
>>> del account['groups']
>>> account
{'sn': 'Poli', 'gn': 'Christian', 'gid': 1000, 'login': 'cpoli', 'uid': 1000}
>>> account.clear()
>>> account
{}
```

En dehors des fonctionnalités communes à tous les conteneurs, les dictionnaires disposent de plusieurs méthodes spécifiques intéressantes:

Méthode	Description
keys()	fournit les clés du dictionnaire
values()	fournit les valeurs associées aux clés
items()	fournit les couples (clé,valeur)
get(cle,defaut)	fournit la valeur associée à la clé si la clé est présente, sinon renvoie defaut
has_key(cle)	teste la présence d'une clé (**supprimée dans la version 3**)
copy()	copie superficielle

Note: Le résultat retourné par les méthodes keys(), values(), items() est:

- une liste en Python 2.x
- un itérateur en Python 3.x

Les 3 méthodes retournant des itérateurs ont été introduites comme implémentations alternatives, plus performantes, des méthodes *keys()*, *values()*,*items()* dans **Python 2.2** sous les noms *iterkeys()*, *itervalues()*, *iteritems()*.

A partir de **Python 3.0** les méthodes *iter...()* ont remplacé les méthodes les implémentations anciennes, retournant des listes.

```
>>> account = {'sn':'Poli','gn':'Christian','login':'cpoli','uid':1000,'gid':1000,'groups':[1005,
>>> account.keys()
['sn', 'gn', 'gid', 'groups', 'login', 'uid']
>>> account.values()
['Poli', 'Christian', 1000, [1005, 1007], 'cpoli', 1000]
>>> account.items()
[('sn', 'Poli'), ('gn', 'Christian'), ('gid', 1000), ('groups', [1005, 1007]), ('login', 'cpoli'),
>>> account.has_key("gid")
True
>>> account.has_key("email")
False
```

```
>>> account.get("email","inconnu")
'inconnu'
```

Question de style...

Ecrire:

```
if 'login' in account:
    val = account['login']
```

au lieu de :

```
if account.has_key('login'):
    val = account['login']
```

ou de:

```
if 'login' in account.keys():
    val = account['login']
```

Ecrire:

```
val = account.get("login","anonymous")
```

au lieu de :

```
if 'login' in account:
    val = account['login']
else:
    val = "anonymous"
```

Dans un traitement nécessitant aussi bien la présence des clés que celles des valeurs associées, éviter d'écrire (même si ce n'est pas incorrect):

```
for key in account:
    print("{}: {}".format(key, account[key]))
```

écrire plutôt:

```
for key,val in account.items():
    print("{}: {}".format(key, val))
```

Dictionnaires et listes:

La méthode *items()*, employée dans l'exemple précédent, fournit une liste de tuples correspondant aux paires clé-valeur du dictionnaire:

```
>>> account
{'uid': 1000, 'gid': 1000, 'sn': 'Poli', 'groups': [1005, 1007], 'gn': 'Christian', 'login': 'cpo
>>> account.items()
[('uid', 1000), ('gid', 1000), ('sn', 'Poli'), ('groups', [1005, 1007]), ('gn', 'Christian'), ('l
```

On peut également réunir les éléments de deux ou plusieurs sequences distinctes en une séquence (liste en Python2, itérateur en Python 3) de tuples avec la fonction *zip()*:

```
>>> account_keys
['uid', 'gid', 'sn', 'groups', 'gn', 'login']
>>> account_values
[1000, 1000, 'Poli', [1005, 1007], 'Christian', 'cpoli']
>>> zip(account_keys,account_values)
[('uid', 1000), ('gid', 1000), ('sn', 'Poli'), ('groups', [1005, 1007]), ('gn', 'Christian'), ('l
```

A partir d'une liste de paires on peut construire un dictionnaire:

```
>>> zipped
[('uid', 1000), ('gid', 1000), ('sn', 'Poli'), ('groups', [1005, 1007]), ('gn', 'Christian'), ('l
>>> dict(zipped)
{'uid': 1000, 'gid': 1000, 'sn': 'Poli', 'groups': [1005, 1007], 'gn': 'Christian', 'login': 'cpo
```

Les dictionnaires en intension, c'est possible :

```
>>> {x:ord(x) for x in "Python"}
{'h': 104, 'o': 111, 'n': 110, 'P': 80, 't': 116, 'y': 121}
```

Exercice

```
"""
fichier: week_dict.py

Soient les deux variables, week_fr_en et week_fr:

- La variable week_fr_en contient le dictionnaire français=>anglais pour les
  jours de la semaine.

- La liste week_fr contient les noms des jours de la semaine en français,
  dans le bon ordre

a) En utilisant les listes en intension, construire à partir des deux
   collections précédentes la liste week_en des noms des jours de la
   semaine en anglais dans le bon ordre.

b) Ecrire une programme interactif qui demande la saisie d'un jour de la
   semaine. Le mot saisi est recherché  dans week_fr, puis, en cas d'échec,
   dans week_en (utiliser les méthodes prédéfinies pour les séquences).
   Si le mot est trouvé dans une des deux listes, afficher la position dans
   la liste et la langue de la liste.
   En cas de recherche infructueuse lèver une exception ValueError.

c) En utilisant week_fr, écrire une boucle for..in qui affiche les jours de la
   semaine,numérotés, un par ligne, sous la forme:

   0 - lundi
   1 - mardi
   ....

d) Modifier le programme pour une numérotation à partir de 1:
   1 - lundi
   2 - mardi
   .....

e) à partir des collections précédentes (fournies et/ou construites, au choix)
   construire le dictionnaire inversé week_en_fr (plusieurs solutions)
"""

week_fr_en = { "jeudi":"thursday", "vendredi":"friday", "samedi":"saturday",
               "lundi":"monday", "mardi":"tuesday", "mercredi":"wednesday",
               "dimanche":"sunday" }

week_fr = ["lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi",
           "dimanche"]

"""
fichier: week_dict_sol.py
"""

from __future__ import print_function
```

```
import sys

week_fr_en = {"jeudi":"thursday", "vendredi":"friday", "samedi":"saturday",
              "lundi":"monday", "mardi":"tuesday", "mercredi":"wednesday",
              "dimanche":"sunday"}

week_fr = ["lundi", "mardi", "mercredi",
           "jeudi", "vendredi", "samedi","dimanche"]

#
# a
#

week_en = [week_fr_en[fr] for fr in week_fr]

#
# b
#
prompt = "jour:"
while True:
    day = input(prompt) if sys.version_info[0]==3 else raw_input(prompt)
    if day in week_fr:
        print("FR:position", week_fr.index(day))
    elif day in week_en:
        print("EN:position", week_en.index(day))
    else:
        raise ValueError("Not found")

#
# c
#

# .....

#
# d
#

# .....

#
#e 1
#

dict(zip(week_en,week_fr))

#
#e 2
#

{en:fr for fr,en in week_fr_en.items()}
```

Une solution légèrement différente:

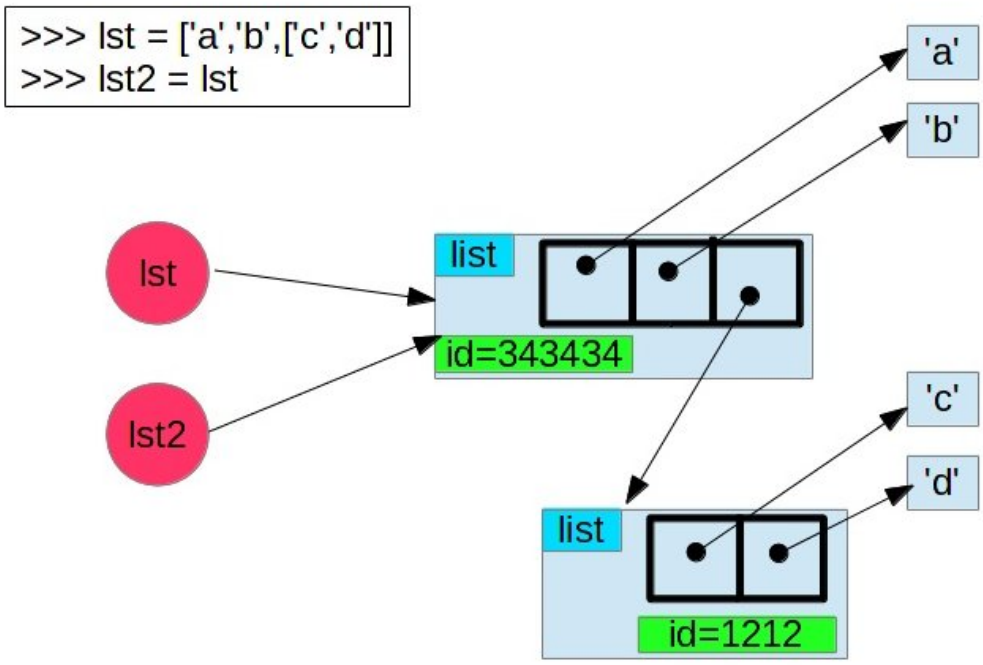
```
"""
fichier: week_dict_sol2.py
"""
from __future__ import print_function
import sys

week_fr_en = {"jeudi":"thursday", "vendredi":"friday", "samedi":"saturday",
              "lundi":"monday", "mardi":"tuesday", "mercredi":"wednesday",
              "dimanche":"sunday"}
```

```
week_fr = ["lundi", "mardi", "mercredi",
           "jeudi", "vendredi", "samedi","dimanche"]

#
# a
#

week_en = [week_fr_en[fr] for fr in week_fr]

#
# b
#
prompt = "jour:"
while True:
    day = input(prompt) if sys.version_info[0]==3 else raw_input(prompt)
    try:
        print("FR:position", week_fr.index(day))
    except ValueError:
        try:
            print("EN:position", week_en.index(day))
        except ValueError:
            raise ValueError('not found')

#
# c
#

# .....

#
# d
#

# .....

#
#e 1
#

dict(zip(week_en,week_fr))

#
#e 2
#

{en:fr for fr,en in week_fr_en.items()}
```

4.4.5 Objets partagés, copie d'objets

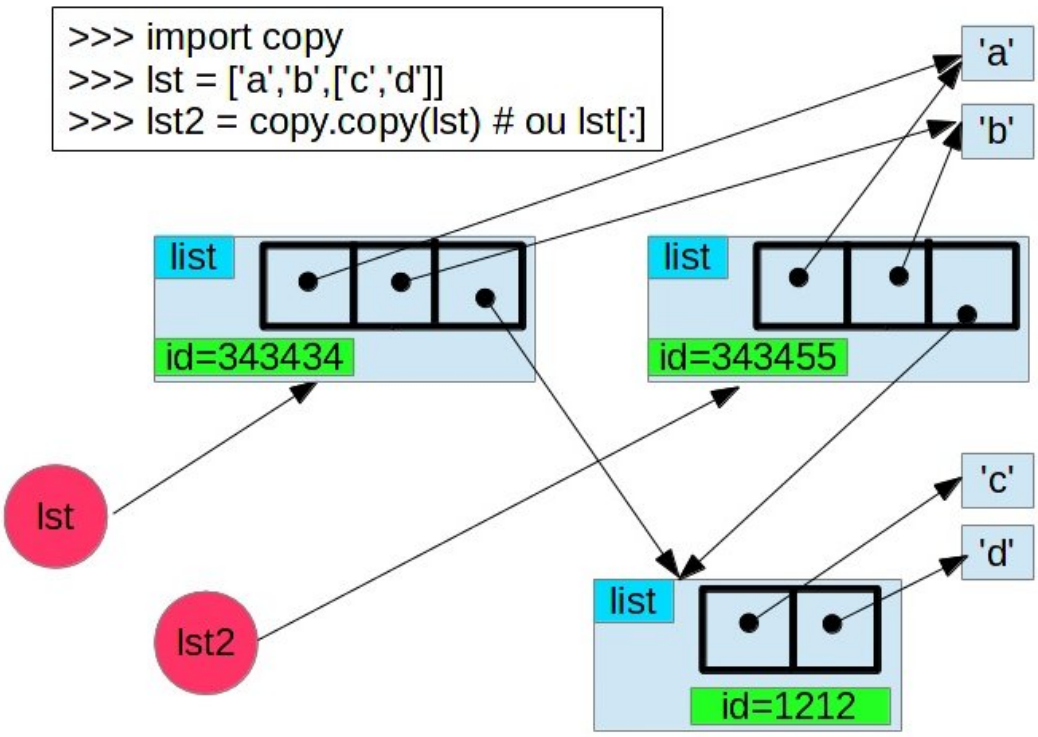
Par défaut, les objets référencés plusieurs fois sont partagés, ce qui nécessite quelques précautions dans le cas des objets modifiables (listes, dictionnaires etc).

Illustration:



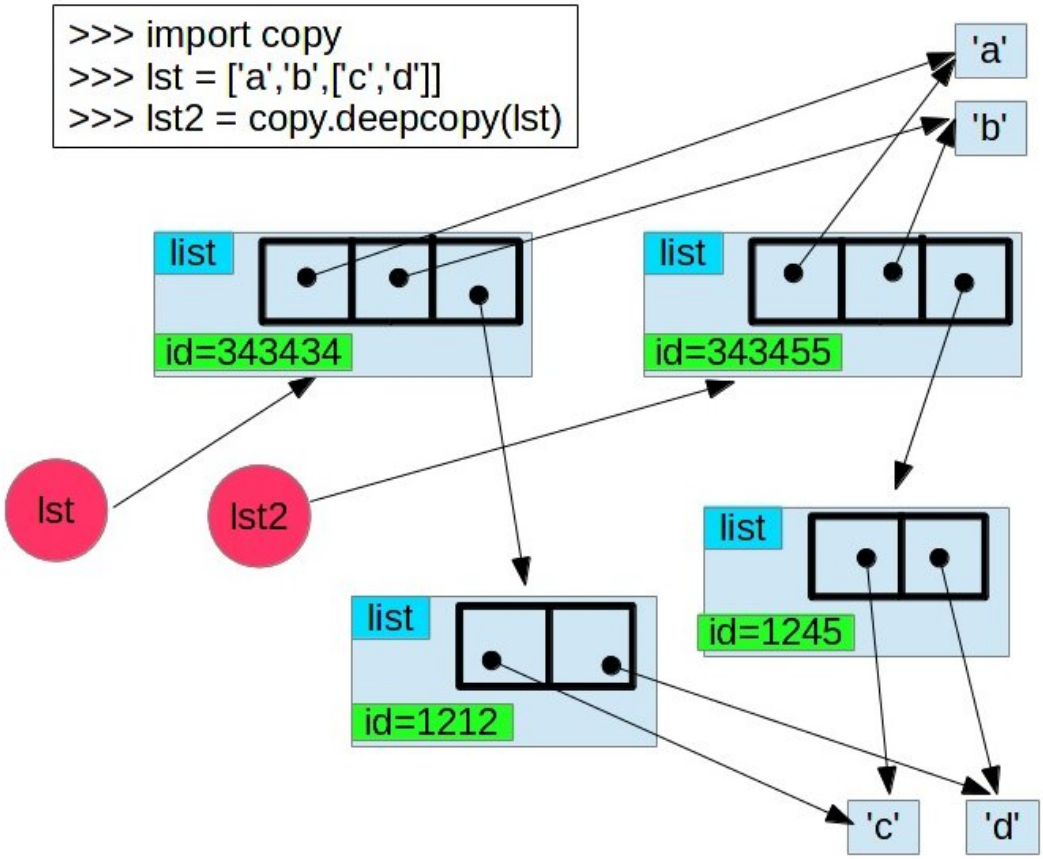

```
>>> lst = ['a','b',['c','d']]
>>> lst2 = lst
>>> lst[1] = 'X'
>>> lst2 # lst2 est affectée par l'op.précédente
['a', 'X', ['c', 'd']]
```

Si ce comportement n'est pas celui souhaité, une solution imparfaite à ce problème est la copie superficielle (shallow copy). Ainsi, l'objet de premier niveau est dupliqué, mais pas les objets qu'il référence. Dans l'illustration suivante, la liste de premier niveau est repliquée, mais pas "la liste dans la liste":




```
>>> lst = ['a','b',['c','d']]
>>> lst2 = lst[:]
>>> lst[1] = 'X'
>>> lst2 # lst2 n'est pas affecté par l'op. précédente
['a', 'b', ['c', 'd']]
>>> lst[2][1] = 'Y'
>>> lst2 # maintenant si...
['a', 'b', ['c', 'Y']]
```

Une solution radicale est la copie profonde, qui s'applique récursivement aux objets référencés tant que ces références concernent des objets modifiables:




```
>>> import copy
>>> lst = ['a','b',['c','d']]
>>> lst2 = copy.deepcopy(lst)
>>> lst[1] = 'X'
>>> lst2
['a', 'b', ['c', 'd']]
>>> lst[2][1] = 'Y'
>>> lst2
['a', 'b', ['c', 'd']]
```

Note: On peut personnaliser la copie des objets sur une classe en définissant les méthodes **__copy__**() et **__deepcopy__**()

4.4.6 Les ensembles

Il s'agit d'objets représentant la notion d'ensemble au sens mathématique:

- les éléments sont uniques
- il n'y a pas de notion d'ordre entre les éléments.

Concrètement, les ensembles dans Python sont représentés par deux types:

- **set** : ensembles modifiables
- **frozenset** : ensembles immuables

Les éléments d'un ensemble (set ou frozenset) sont des objets hashables donc immuables.

On peut ajouter/retirer des éléments sur un ensemble *set* après sa création, mais pas sur un ensemble *frozenset*. L'intérêt de ce dernier et d'entre hashable à son tour, donc utilisable comme clé dans un dictionnaire ou comme élément d'un autre ensemble.

```
>>> {1, 'a'}  # premier format (à privilégier)
set(['a', 1])
>>> set([1,"a"]) # création avec le constructeur
set(['a', 1])
>>> set([1,"a",[6,7]])
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
>>> set1 = set([1,"a"])
>>> set1.add("b")
>>> set1
set(['a', 1, 'b'])
>>> set1.remove("b")
>>> set1
set(['a', 1])
>>> set2=set(["x",set1])
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
>>> frozen = frozenset(['a', 1, 'b'])
>>> set2=set(["x",frozen])
>>> set2
set(['x', frozenset(['a', 1, 'b'])])
>>> frozen.add("y")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
>>>
```

Le type range/xrange

Le type appelé **range** en *Python 3*, pré-existant sous le nom **xrange** en *Python 2.x (x>=6)* est un type de séquence très simple supportant seulement:

- l'itération
- la fonction *len()*
- l'accès indexé

NB: La documentation Python classe ce type parmi les séquences car il permet l'accès indexé. Néanmoins, il ne supporte pas des fonctionnalités communes à toutes les autres séquences, comme le slicing.

Son rôle est de fournir une séquence de valeurs entières contiguës dans une plage donnée et il est destiné à la construction des boucles "for" équivalentes à `for(i=0;i<max;i++){\...}`, possibles en *C* et les langages "C-like" pour lesquelles Python ne propose pas de syntaxe dédiée. Ainsi, en Python on va écrire:

```
>>> # Python 3
>>> max=3
>>> for e in range(max):
...     print(e)
...
0
1
2
>>>

>>> # Python 2.7.x
>>> max=3
>>> for e in xrange(max):
```

```
...     print(e)
...
0
1
2
>>>
```

En Python 2.7, **xrange** coexiste avec la fonction "historique" *range()* qui renvoie une liste:

```
>>> range(5)
[0, 1, 2, 3, 4]
>>>
```

Elle assure le même fonctionnement d'une boucle que *xrange()*:

```
>>> max=3
>>> for e in range(max):
...     print(e)
...
0
1
2
>>>
```

Pourquoi avoir introduit *xrange*? Pour une consommation mémoire constante par rapport au nombre d'itérations.

NB: Dans la version 3 de Python la fonction *range()* a été réimplémentée sur le principe du type *xrange* de la version 2. Le type *xrange* n'existe plus dans la nouvelle version.

Le type enumerate

Tout comme (x)range, cet itérateur est important par les possibilités qu'il offre dans l'écriture des boucles.

Scénario: Supposons que, pour la liste ['a', 'b', 'c', d'], il est demandé d'afficher chaque élément précédé par sa position dans la liste:

```
0 a
1 b
2 c
3 d
```

Une écriture possible est:

```
i=0
for e in ['a','b','c','d']:
    print("{} {}".format(i,e))
    i += 1
```

Ou encore:

```
lst = ['a','b','c','d']
for i in range(len(lst)):
    print("{} {}".format(i,lst[i]))
    i += 1
```

mais cette écriture n'est pas très élégante...

C'est ici que *enumerate* intervient:

```
for i,e in enumerate(['a','b','c','d']):
    print("{} {}".format(i,e))
```

Explication: un objet *enumerate* produit à chaque itération un tuple *(position, item)* :

```
>>> e=enumerate(['a','b','c','d'])
>>> type(e)
<type 'enumerate'>
>>> e.next()
(0, 'a')
>>> e.next()
(1, 'b')
>>> e.next()
(2, 'c')
>>> e.next()
(3, 'd')
>>> e.next()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Sur l'exemple précédent, si on souhaite numéroter les éléments à partir de 1, on peut écrire:

```
for i,e in enumerate(['a','b','c','d'], 1):
    print("{} {}".format(i,e))
```

```
1 a
2 b
3 c
4 d
```

Exercice

```
"""
fichier: week_dict2.py

En utilisant les listes week_fr et week_en fournies:

a) Ecrire une programme interactif qui demande la saisie d'au moins 3 lettres.
   En utilisant la méthode startwith() retrouver l'élément qui commence avec
   les groupe de lettres saisies dans week_fr, puis, en cas d'échec, dans
   week_en (utiliser, si c'est pertinent, la clause "else" pour les boucles).
   Si le mot est trouvé dans une des deux listes, afficher la position dans
   la liste, le mot trouvé et la langue de la liste.
   En cas de recherche infructueuse lèver une exception ValueError.


b) En utilisant week_fr, écrire une boucle for..in qui affiche les jours de la
   semaine, un par ligne, sous la forme:

   0 - lundi
   1 - mardi
   ....

c) Modifier le programme pour une numérotation à partir de 1:
   1 - lundi
   2 - mardi
   .....
"""

week_fr = ["lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi",
           "dimanche"]
week_en = ["monday", "tuesday", "wednesday", "thursday", "friday", "saturday",
           "sunday"]
```

```
"""
fichier: week_dict2_sol.py
"""
from __future__ import print_function
import sys

week_fr = ["lundi", "mardi", "mercredi",
           "jeudi", "vendredi", "samedi","dimanche"]
week_en = ["monday", "tuesday", "wednesday", "thursday", "friday", "saturday",
           "sunday"]

#
# b
#
prompt = "jour(3 lettres min.):"
while True:
    inp = input(prompt) if sys.version_info[0]==3 else raw_input(prompt)
    if len(inp)<3:
        print("saisir 3 lettres minimum SVP")
        continue
    for pos,day in enumerate(week_fr):
        if day.startswith(inp):
            print("FR", day, "position:", pos)
            break
    else:
        for pos,day in enumerate(week_en):
            if day.startswith(inp):
                print("EN", day, "position:", pos)
                break
        else:
            raise ValueError("Not found")


#
# c
#

for i,day in enumerate(week_fr):
    print("{} - {}".format(i,day))

#
# d
#

for i,day in enumerate(week_fr,1):
    print("{} - {}".format(i,day))
```

4.4.7 Les types des constantes nommées

Ils sont destinés à introduire les constantes spéciales du langage:

Les constantes booléennes

Les deux catégories logiques de **vrai** et de **faux** sont représentées par les deux objets de type **bool** : **True** et **False**. Ils ont la même valeur logique que les *int* **1** et **0** tout en ayant leur identité propre:

```
>>> type(True)
<type 'bool'>
>>>
>>> id(True)
```

```
8916816
>>> id(1)
27633592
>>> True==1
True
>>> True==2
False
```

La constante None

Elle désigne l'absence de valeur (équivalent du **null** dans d'autres langages) et est définie par un type dont *None* est le nom est la seule instance.

```
>>> type(None)
<type 'NoneType'>
>>>
```

D'autres constantes spéciales, moins utilisées sont implémentées de la même manière: *NotImplemented* , *Ellipsis* etc.

Le type file

Sans surprise, il intervient dans les opérations sur des fichiers. Il a été brièvement évoqué avec l'instruction **with** et sera détaillé dans le chapitre sur la bibliothèque standard.

On va juste souligner ici son comportement en tant que type itérable:

```
$ cat abc.txt
aaa
bbb
ccc

>>> with open('abc.txt') as f:
...     for line in f:
...             print(line)
...
aaa

bbb

ccc
```

NB: On notera que, même si le constructeur `file()` existe, il est préférable d'utiliser la primitive `open()` pour les créations de fichiers.

4.5 Les conventions de codage

Les conventions de codage en Python ont été synthétisées dans une *"PEP"* (*Python Enhancement Proposals*), probablement une des *PEPs* les plus connues, la **PEP8** (auteurs : Guido van Rossum, Barry Warsaw et Nick Coghlan)

La *PEP8* est disponible en ligne et c'est un document important, qui mérite une lecture intégrale : http://legacy.python.org/dev/peps/pep-0008/

Dans l'immédiat, on va se contenter d'un rapide survol:

- utiliser 4 espaces pour l'indentation (la tabulation à 8 espaces tolérée pour les codes anciens, mais ne pas combiner espaces et tabulations!)
- Longueur des lignes < 80 caractères

- Eviter les espaces dans les situations suivantes:
  - avant les ”,” ”:” :
    - ne pas écrire *a , b* mais plutôt *a, b*
    - ne pas écrire *{‘a’ :1, ...}* mais plutôt *{‘a’: 1, ...}*
  - après les ‘(‘, ‘[’, ‘{‘ :
    - ne pas écrire *[a, b]*, *(a, b)*, *{ a, b }* mais plutôt *[a, b]*, *(a, b)*, *{a, b}*
  - entre un nom de variable et ‘(‘, ‘[’:
    - ne pas écrire *func ()*, *map [’id’]* mais plutôt *func()*, *map[’id’]*
- Entourer d’un espace de chaque coté les opérateurs binaires dans les expressions (une exception: ‘=’ dans les arguments d’appel des méthodes):
  - on va écrire (espaces avant/après ‘=’ et ‘==’):

```
var = 1 # affectation
if var == 1:
   pass
```

  - mais on va écrire (sans espaces avant/après ‘=’):

```
'{langage},{version}'.format(langage='Python', version=3)
```

- Si on utilise des opérateurs de priorité différentes dans une expression, prévoir des espaces (seulement) autour de opérateurs de priorité moindre. Exemple `a*x + b*y`

NB: Ne pas hésiter à utiliser l’utilitaire **pylint** pour auditer son code par rapport aux préconisations de la PEP8:

```
$pylint fichier.py
```

4.5.1 Les noms des variables

La **PEP8** cite les styles suivants, sans les préconiser tous pour autant:

- b (single lowercase letter)
- B (single uppercase letter)
- lowercase
- lower_case_with_underscores
- UPPERCASE
- UPPER_CASE_WITH_UNDERSCORES
- CapitalizedWords (CapWords, CamelCase)
- mixedCase
- Capitalized_Words_With_Underscores (ugly!)

4.5.2 Convention générales de nommage

- Noms à éviter: en règle générale, les variables à une seule lettre et en particulier les lettres ‘l’ (L minuscule) et ‘O’ (‘o’ majuscule) et I (‘i’ majuscule)
- variables désignant des constantes sont nommées en lettre majuscules avec des soulignées. Exemple MAX_HEIGHT
- une variable définie au niveau du module commençant avec un “_” est “locale” au module. Elle ne sera pas importée par `from module import *`

- une variable se termine par un "_" pour éviter la collision avec un mot clé du langage. Exemples : min_, max_

Le prochains chapitres vont apporter plus des précision sur les conventions de nommage spécifiques aux notions qui seront introduites.

4.6 Exercice

```
"""
fichier: coauteurs.py

La liste suivante contient des dictionnaires de la forme {'author': nom,
'id': publication_id}, i.e. un auteur, une publication (relation N x N entre
auteurs et publications).

A partir de cette liste construisez la structure de votre choix qui représente
toutes les paires d'auteurs ayant publié ensemble associé au nombre de
publications en commun, i.e. f(auteur1,auteur2, nombre-pub-communes).

La structure choisie doit garantir l'unicité des résultats.

"""

lst = [{'author': 'A', 'id': 4},
       {'author': 'A', 'id': 12},
       {'author': 'A', 'id': 7},
       {'author': 'A', 'id': 15},
       {'author': 'B', 'id': 4},
       {'author': 'B', 'id': 21},
       {'author': 'B', 'id': 22},
       {'author': 'B', 'id': 7},
       {'author': 'B', 'id': 12},
       {'author': 'C', 'id': 21},
       {'author': 'C', 'id': 12},
       {'author': 'C', 'id': 15},
       {'author': 'D', 'id': 22},
       {'author': 'D', 'id': 7},
       {'author': 'D', 'id': 21},
       {'author': 'D', 'id': 17},
       {'author': 'E', 'id': 15},
       {'author': 'E', 'id': 17},
       {'author': 'E', 'id': 4}]

"""
fichier: coauteurs_sol.py
"""
from __future__ import print_function

lst = [{'author': 'A', 'id': 4},
       {'author': 'A', 'id': 12},
       {'author': 'A', 'id': 7},
       {'author': 'A', 'id': 15},
       {'author': 'B', 'id': 4},
       {'author': 'B', 'id': 21},
       {'author': 'B', 'id': 22},
       {'author': 'B', 'id': 7},
       {'author': 'B', 'id': 12},
       {'author': 'C', 'id': 21},
       {'author': 'C', 'id': 12},
       {'author': 'C', 'id': 15},
       {'author': 'D', 'id': 22},
```

```
        {'author': 'D', 'id': 7},
        {'author': 'D', 'id': 21},
        {'author': 'D', 'id': 17},
        {'author': 'E', 'id': 15},
        {'author': 'E', 'id': 17},
        {'author': 'E', 'id': 4}]

dict_article = {}

for entry in lst:
    id = entry.get('id')
    author = entry.get('author')
    dict_article[id] = dict_article.get(id,[])
    dict_article[id].append(author)
print(dict_article)

result = {}

for entry in lst:
    id = entry.get('id')
    author = entry.get('author')
    for coauthor in dict_article.get(id):
        if coauthor == author: continue
        pair = frozenset((author, coauthor))
        result[pair] = result.get(pair,set())
        result[pair].add(id)
print("#####################################################")
print(result)
```

CHAPTER

FIVE

MODULES ET PAQUETS

- Modules
- Paquets (ou packages)

5.1 Modules

Un module est un fichier contenant des instructions Python.

```
#!/usr/bin/env python
"""
Module: 'simplemod'
"""

def do_something():
    print("do something")

def _do_something_else():
    print("do something else")



if __name__ == "__main__":
    print("runs as a script")
    do_something()
else:
    print("module {} was imported".format(__name__))
```

Il peut être utilisé de deux manières:

- en tant que script

  ```
  python simplemod.py
  runs as a script
  do something
  ```

- chargé avec la directive *import*

  ```
  >>> import simplemod
  module simplemod was imported
  >>> __doc__
  >>> help(simplemod)

  >>> simplemod.__doc__
  "\nModule: 'simplemod'\n"
  >>> simplemod.do_something()
  do something
  >>> simplemod._do_something_else()
  ```

```
    do something else
    >>>
```

On notera la possibilité de documenter un module à l'aide d'une chaîne de caractères en début du fichier, appelée une *docstring* . La docstring ast accessible dans l'attribut `__doc__` .

NB: Les noms contenus dans le module importé (de cette façon) doivent être préfixés par le nom du module pour être référencés (dans le contexte de l'*import*).

Rappel: Pour rendre *simplemod.py* directement exécutable (en environnement *Linux* ou *Unix*) il faut:

- rajouter `#!/usr/bin/env python` comme première ligne du fichier
- attribuer des droits d'exécution sur le fichier `chmod u+x simplemod.py`

Au chargement, un objet de type *module* est créé, les instructions contenues dans le fichier sont exécutées et les noms résultant de l'exécution (variables, classes, fonctions) sont placés dans un espace de noms défini par l'objet-module.

Cet objet-module définit l'attribut **__name__** contenant:

1. "__main__" si le module est chargé en tant que script
2. le nom du fichier (sans l'extension ".py") s'il s'agit d'un import

Si le corps du module commence avec une chaîne de caractères (entre quotes simples, doubles, triples etc.) celle-ci sera interprétée comme une "docstring".

Remarques:

- le nom du fichier doit satisfaire les règles imposés pour les noms des variables en Python. La **PEP 8** préconise l'utilisation de noms courts, les _ étant "tolérés".
- un même module peut être utilisé aussi bien en tant que script ou par importation. Par exemple, une bibliothèque ayant vocation à être importée peut contenir une suite de tests à exécuter un mode script. Dans ce cas, c'est une bonne idée de prévoir une exécution conditionnelle de la partie du code réservée au mode script:

```
if __name__ == "__main__":
   ...
   ...
```

Au chargement, un module est cherché dans trois endroits:

1. dans le répertoire courant
2. dans les endroits définis dans la variable d'environnement PYTHONPATH
3. dans le répertoire d'installation de l'interpréteur

NB: Le contenu de la variable d'environnement **PYTHONPATH** est accessible à travers l'attribut *path* du module *sys*:

```
>>> import sys
>>> sys.path
['', '/usr/local/lib/python2.7/dist-packages/django_http_proxy-0.3.2-py2.7.egg', '/usr/local/lib/p
>>>
```

La fonction *dir()* révèle d'autres attributs importants:

```
>>> dir(simplemod)
['__builtins__', '__doc__', '__file__', '__name__', '__package__', '_do_something_else', 'do_somet
```

- '__builtins__' est le dictionnaire des noms prédéfinis
- '__file__' est le nom du fichier du module

5.1.1 Le cache

Lors du chargement d'un module, l'interpréteur vérifie la présence d'un fichier précompilé ayant l'extension ".pyc":

- si le fichier existe et il est valide alors il est chargé directement en évitant une nouvelle compilation à partir du code source.
- si le fichier est absent, corrompu et obsolète, le code source est compilé et chargé en mémoire et le fichier correspondant ".pyc" est généré.

L'emplacement du fichier ".pyc" est:

- dans le répertoire courant en Python 2.x
- dans le répertoire ./__pycache__ en Python 3.x

5.1.2 Importation dans le contexte courant

Avec l'instruction from...import... on peut importer sélectivement des objets d'un module dans le contexte courant en s'affranchissant de l'utilisation du préfixe à l'appel:

```
>>> from simplemod import do_something, _do_something_else
module simplemod is imported
>>> do_something()
do something
>>> _do_something_else()
do something else
>>>
```

On peut aussi importer **tous** les noms (**sauf** ceux préfixés par un _) d'un module dans le contexte courant :

```
>>> from simplemod import *
module simplemod is imported
>>> do_something()
do something
>>> _do_something_else()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name '_do_something_else' is not defined
>>>
```

Attention

La pratique consistant à importer tous les noms d'un module dans un autre contexte est **déconseillée** (sauf cas particuliers) car:

- elle nuit à la lisibilité du code
- il y a risque d'écrasement de fonctions prédéfinies pas leurs homonymes éventuelles

---

5.1.3 Recharger un module

En mode interactif on peut avoir besoin de recharger un module après l'avoir modifié, mais une ré-exécution de l'import ne va pas recharger le module. On peut avoir recours à la fonction *reload()*. Néanmoins cela peut s'avérer insatisfaisant dans certains cas, étant donné que les instances des classes rechargées déjà présentes dans le contexte d'exécution ne seront pas impactées.

5.2 Paquets

Grâce au paquets (packages) on peut créer des hiérarchies de modules en organisant les fichiers correspondants dans des répertoires à plusieurs niveaux. Lors de l'importation, un module est identifié par son nom précédé par les noms successifs des répertoires le contenant, à partir de la racine du paquet, séparés par des points.

Pour illustration, la structure "fichiers" d'un paquet (correspondant à un *framework* imaginaire):

```
framework
|-- controller
|   |-- admurls.py
|   |-- appurls1.py
|   |-- appurls2.py
|   `-- __init__.py
|-- __init__.py
|-- model
|   |-- admin.py
|   |-- appmodel1.py
|   |-- appmodel2.py
|   `-- __init__.py
`-- view
    |-- admviews.py
    |-- appviews1.py
    |-- appviews2.py
    `-- __init__.py
```

Pour qu'un répertoire soit reconnu par l'interpréteur comme membre du paquet, il doit contenir un fichier *__init__.py*, éventuellement vide.

Exemples d'importations:

```
>>> import framework.controller.admurls
>>> import framework.model.appmodel1
>>>
```

NB: Le fichier *__init__.py*, quand il n'est pas vide, il peut contenir l'attribut spécial *__all__* et/ou tout autre code utile au processus d'initialisation (création de fonctions, constantes etc.)

5.2.1 Importer tous les modules d'un paquet

La structure `from ... import *` peut être utilisée pour importer "tous" les modules d'un paquet. Seulement, par "tous" on entend **seulement** les noms listés par la variable **__all__** si elle est présente dans *__init__.py*. Les autres fichiers ".py" présents dans le répertoire seront ignorés. En absence de la variable **__all__** aucun module ne sera importé:

```
#
# framework/model/__init__.py
#

__all__ = ['admin','appmodel1']
```

```
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__']
>>> from framework.model import *
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'admin', 'appmodel1']
```

Dans l'exemple, on constate que *appmodel2*, module pourtant bien présent dans *model*, n'a pas été importé car absent de *__all__*.

Néanmoins, il existe une situation où `from ... import *` peut importer des modules non référencés dans *__all__*. C'est le cas ou les modules en question on fait préalablement l'objet d'importations individuelles:

```
>>> from framework.view import *
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__'] # normal, __init__.py étant vide
>>> import framework.view.admviews
>>> import framework.view.appviews1
>>> import framework.view.appviews2
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'framework']
>>> from framework.view import *
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'admviews', 'appviews1', 'appviews2', 'fra
>>>
```

5.2.2 Références intra-paquet

A l'intérieur d'un paquet on peut utiliser la notation **.** pour référencer le répertoire courant et **..** pour le répertoire parent.

Par exemple, `framework.model.appmodel2` peut importer les modules se situant au même niveau que lui en utilisant la notation **".":**

```
#
# framework.model.appmodel2
#

from . import * # i.e. 'admin', 'appmodel1'
```

Le résultat:

```
>>> import framework.model.appmodel2
>>> dir(framework.model.appmodel2)
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'admin', 'appmodel1']
>>>
```

Un autre exemple, `framework.controller.appurls1` importe les modules se situant dans le répertoire *model* en utilisant **..**:

```
#
# framework.controller.appurls2
#

from ..model import * # i.e.

>>> import framework.controller.appurls1
>>> dir(framework.controller.appurls1)
['__builtins__', '__doc__', '__file__', '__name__', '__package__', 'admin', 'appmodel1']
>>>
```

5.3 Conventions de nommage

- pour les modules: noms courts, en minuscules. Les underscores sont tolérés pour une meilleure lisibilité.
- pour les packages: noms courts, en minuscules, mais l'utilisation des underscores est déconseillée.

NB: Garder à l'esprit que les nom de modules et de packages correspondent à des noms de fichiers et répertoires qui peuvent être soummis à des contraintes spécifiques de certains OS.

CHAPTER

SIX

LES FONCTIONS

Au cours des chapitres précédents on a été amené à appeler des fonctions prédéfinies (built-in).

Dans ce chapitre on va expliquer comment définir ses propres fonctions et on va détailler l'ensemble des mécanismes intervenant dans la définition et dans l'appel d'une fonction. On va également revenir sur le concept de variable pour introduire la notion de *portée*.

6.1 Une vue d'ensemble

6.1.1 Une fonction très simple

```
def say_hi(name):
    print("Hi {}!".format(name))
```

Premières remarques:

- Une fonction est définie avec *def*
- *def* est une **instruction** (et non pas une déclaration, directive etc.) qui, entre autres, opère une **affectation** de variable.
- Le nom introduit avec *def*, le mot *say_hi*, est une *variable* comme une autre, référençant un objet de type *function*.

```
>>> def say_hi(name):
...     print("Hi {}!".format(name))
...
>>> say_hi('Mom')
Hi Mom!
>>> say_hi # est une variable
<function say_hi at 0x7f23515985f0>
```

6.1.2 L'instruction return

En Python, comme en C/C++ et Java, une fonction retourne **toujours** une valeur. En Python, en absence de l'instruction *return* ou en cas d'appel de l'instruction *return* sans argument, la valeur de retour est **None**.

```
>>> res # au départ, n'existe pas
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'res' is not defined
>>> res = say_hi('Mom')
Hi Mom!
>>> res # existe bien maintenant (plus d'erreur)!
>>>
>>> res==None
True
```

L'instruction *return* peut être suivie d'une expression qui sera évaluée pour calculer la valeur de retour:

```
def cube(x):
    return x**3

>>> cube(2)
8
>>>
```

Astuce: Si la valeur de retour est un tuple, le résultat retourné peut être récupéré directement dans plusieurs variables (qui forment un tuple aussi) :

```
>>> def f():
...     # ...
...     # ...
...     return 5, "abc", [5,6]
...
>>> x, y, z = f()
>>> x
5
>>> y
'abc'
>>> z
[5, 6]
>>>
```

6.1.3 Les fonctions imbriquées

En Python les définitions des fonctions peuvent s'imbriquer à volonté:

```
>>> def cube(x):
...     def square(t):
...         return t**2
...     return x*square(x)
...
>>> cube(2)
8
>>>
```

Dans les chapitres suivants on va rencontrer des utilisations plus pertinentes de cette propriété très puissante...

6.2 La portée des variables

On a vu précédemment que définir une variable revient à associer une référence d'objet à un nom. Une variable peut donc être définie à l'intérieur d'une fonction comme en dehors de toute fonction. Selon l'endroit de leur définition, la "visibilité" des variables, appelée couramment **portée** (*scope* en anglais) n'est pas la même.

Python 2.x implémente 2 niveaux de portée:

- *local*, qui est celui de la fonction, i.e. du bloc *def*. Il est accessible via la primitive *locals()*
- *global*, celui du module, i.e. du fichier, accessible via la primitive *globals()*.

```
>>> def f(arg):
...     x = arg
...     print(locals())
...
>>> f(5)
{'x': 5, 'arg': 5}
>>> globals()
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', 'f': <function f at 0
>>>
```

NB : Des noms identiques, définis dans des modules distincts, peuvent coexister dans un même contexte, sans conflit, car ils seront préfixés par le nom de leur module de définition, après importation.

Note: En *Python*, contrairement à *C++* et *Java*, la portée locale est relative **exclusivement** au contexte de la fonction. Il n'y a pas de portée locale liée à un autre type de bloc d'instructions (comme c'est le cas pour les boucles *while* ou *for* en *Java* et *C++*), sauf pour deux cas très particuliers :

- la variable de liste en intension (seulement depuis Python 3)
- la variable du bloc *except* (introduite par *as*)

Autrement dit, une affectation de variable faite à l'intérieur d'une boucle *while* ou *for* dans une fonction sera visible dans le corps de la fonction même après la sortie de la boucle. L'exemple suivant illustre ce comportement avec une boucle *for* dans une fonction:

```
def f():
    x = 33
    for i in range(1,3):
        y = i
        x = i+1
    # end for
    print("values in the context of f(): x={}, y={}, i={}".format(x,y,i))

>>> x = 5
>>> f()
values in f() context: x=3, y=2, i=2
>>> x
5
```

On constate que les variables *x*, *y* et *i* conservent les modifications intervenues à l'intérieur de la boucle *for* après la sortie du bloc.

En plus, *y* et *i* apparaissent pour la première fois dans le *for*, et pourtant elles sont présentes dans le contexte de la fonction, après la sortie de la boucle.

6.2.1 Que se passe-t-il en cas de conflit?

Des conflits de nom de variable peuvent apparaître entre le niveau local et global. Plusieurs situations peuvent se présenter:

- **référencement:** la variable globale est accédée en lecture à l'intérieur de la fonction:

```
>>> x=1
>>> def f():
...     print(x)
...
>>> f()
1
>>>
```

- **affectation:** la variable globale est "cachée" (silencieusement) par son homonyme locale:

```
>>> x=1
>>> def f():
...     x = 2
...     print("x={}".format(x))
...
>>> f()
x=2
>>> x
1
>>>
```

- **référencement suivi d'affectation** situation plus contrariante pour l'interpréteur (mais réaction logique):

```
>>> x=1
>>> def f():
...     print("x={}".format(x))
...     x = 2
...
>>> f()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 2, in f
UnboundLocalError: local variable 'x' referenced before assignment
>>>
```

6.2.2 L'instruction global

La sémantique de cette instruction rappelle les déclarations de variables des autres langages.

Appelée à l'intérieur d'une fonction, elle prends la forme `global var1[,var2][,...]` et elle indique à l'interpréteur que les variables var1, var2, ... font partie du contexte global:

```
>>> x=1
>>> def f():
...     global x
...     x=2
...
>>> x
1
>>> f()
>>> x
2
>>>
```

NB: En complément de l'instruction *global*, Python 3 introduit l'instruction *nonlocal* qui sera expliquée par la suite.

6.2.3 La résolution des noms

Pour toute variable qui n'est pas **global** ou **nonlocal**, les règles suivantes s'appliquent:

- L'**affectation** de variable à l'intérieur d'une fonction **F** se fait dans le contexte local de la fonction
- Toute **référence** à une variable à l'intérieur de **F** donnera lieu à une recherche dans plusieurs endroits, dans l'ordre suivant (la recherche s'arrête au premier succès):
  1. Le contexte local de **F**
  2. Si **F** est définie à l'intérieur d'une autre fonction **G** on va chercher la variable dans le contexte local de **G**. En cas d'échec la recherche continuera en remontant les contextes des fonctions englobantes, s'il y en a.
  3. Le contexte global du module
  4. Le contexte des noms prédéfinis (built-ins)

NB:

- Si la variable est **global**, la recherche commencera à l'étape **3**
- Les paramètres d'une fonction font partie du contexte local de celle ci.

Illustration:

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

def fun():
    #--------------------------
    y = 22
    def fun2():
        #----------------------
        z = 333
        print("x={}".format(x))
        print("y={}".format(y))
        print("z={}".format(z))
        print("True={}".format(True))
        #----------------------
    fun2()
    #--------------------------

if __name__ == "__main__":
    fun()
```

```
>>> fun()
x=1
y=22
z=333
True=True
>>>
```

On voit que :

- **z** provient du contexte local de fun2()
- **y** provient du contexte local de fun() (qui contient fun2())
- **x** provient du contexte global
- **True** est une variable du contexte built-in

Problème:

Supposons qu'on veut modifier l'exemple précédent pour que la fonction **fun2()** modifie **y** dans le contexte de **fun()** mais pas dans le contexte global (car pour le niveau global il suffirait de déclarer **global y** à l'intérieur de **fun2()**)

Réponse:

- **Python 2.x:** il n'y a pas de solution
- **Python 3.x:** il suffit de déclarer **nonlocal y**:

Illustration avec l'exemple précédent modifié (on a tracé **y** à tous les niveaux et éliminé les variables **x** et **z**):

```
y = 2

def fun():
    #--------------------------
    y = 22
    def fun2():
        nonlocal y
        #----------------------
        print("fun2(): y={} [avant affectation]".format(y))
        #----------------------
        y = 99
        #----------------------
        print("fun2(): y={} [apres affectation]".format(y))
        #----------------------
    fun2()
```

```
    print("fun(): y={}".format(y))
    #---------------------------

if __name__ == "__main__":
    fun()
    print("global: y={}".format(y))
```

```
$ python3 resolution_nonlocal.py
fun2(): y=22 [avant affectation]
fun2(): y=99 [apres affectation]
fun(): y=99
global: y=2
```

6.2.4 L'instruction nonlocal

Cette instruction à été introduite avec la version 3 pour répondre au besoin illustré précédemment.

La résolution des noms, aussi bien pour l'affectation que pour le référencement, se fait suivant l'étape 2 (seulement) du processus décrit précédemment pour le référencement des variables locales.

Ainsi, si la variable n'est pas trouvée dans un des blocs **def** qui englobent la fonction contenant la déclaration **nonlocal**, une erreur de syntaxe se produit (même si la variable existe dans le contexte global!)

6.3 Signatures des fonctions

La signature d'une fonction est composée du nom de la fonction suivi, entre parenthèses, par l'ensemble de ses paramètres (ensemble qui peut être vide). Elle représente la première partie de l'instruction **def** et est suivie par **":"**.

La forme générale d'une signature est:

def **nom_fonction(p1, p2, ...)** *:*

les paramètres *p1*, *p2*,... pouvant prendre plusieurs formes.

Vocabulaire

On va appeler :

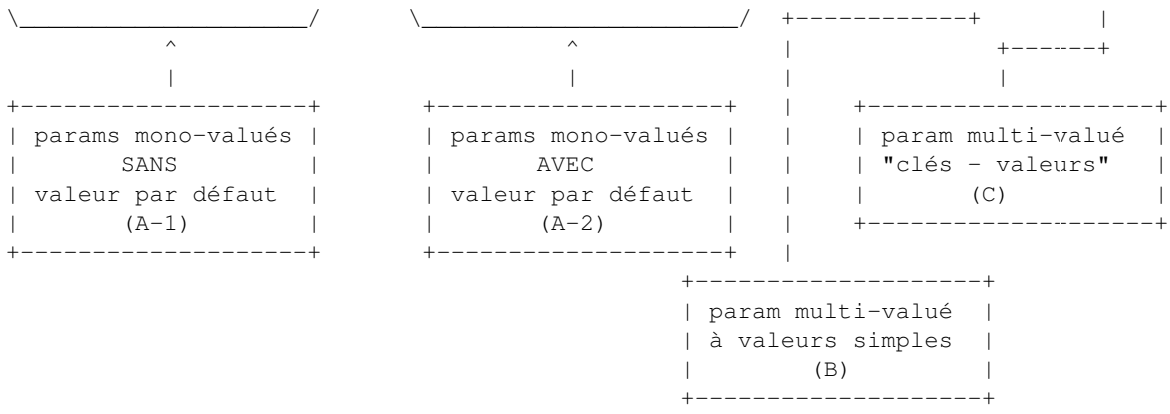
- **"paramètres"** les entités nommées (*p1*, *p2*, ...), faisant partie de la **signature** de la fonction.
- **"arguments"** les valeurs concrètes (références, en fait) passées à la fonction lors des **appels** .
- **"paramètres mono-valués"** les paramètres recevant un seul argument (maximum) lors des appels de la fonction.
- **"paramètres multi-valués"** les paramètres pouvant recevoir un nombre indéterminé (0..N) d'arguments lors des appels de la fonction.

NB : On a utilisé des noms de la forme *arg_x*, *args*, *kwargs* pour nommer des paramètres afin de mieux identifier leur rôle.

On va voir que, dans le cas le plus général, **il n'y a pas de bijection paramètre<=>argument**.

Dans une signature de fonction, trois catégories de paramètres peuvent être présentes, toutes optionnelles :

```
#
# définition de la fonction avec ses paramètres
#
def func(arg_1, arg_2, ..., arg_i-1, arg_i=def_val_i, ..., arg_n=def_val_n, *args, **kwargs):
   ...     ^                   ^      ^                    ^               ^        ^
   ...     |                   |      |                    |               |        |
```



- **0..n** paramètres mono-valués **(A)** , notés dans la suite **arg_1**,..., **arg_n**, qui peuvent prendre deux formes :
  - *variable* : à l'appel, cette variable **(A-1)** recevra un argument, qui devra **toujours** être fourni.
  - *variable = valeur_par_defaut* : à l'appel, l'argument correspondant au paramètre **(A-2)** peut manquer, et dans ce cas la valeur par défaut sera utilisée. Cette notation (*nom=valeur*) est connue sous le nom de "syntaxe **keyword**" en jargon Python. Dans la suite, on va l'appeler parfois "syntaxe **clé-valeur**".

    NB : Tous les paramètres suivant un paramètre à valeur par défaut doivent avoir aussi une valeur par défaut.
- **0..1** paramètre multi-valué à valeurs simples **(B)** : reçoit, à l'appel, un nombre arbitraire (0..*) d'arguments sous la forme d'un tuple (noté ***args** dans la suite), l'ordre des arguments à l'appel étant respectée dans le tuple.
- **0..1** paramètre multi-valué à valeurs nommées **(C)** (syntaxe *clé-valeur*): reçoit, à l'appel, un nombre arbitraire (0..*) d'arguments "clé-valeur" sous la forme d'un dictionnaire (noté ****kwargs** dans la suite)

Si présents, ces trois types de paramètres apparaissent dans la signature dans cet ordre:

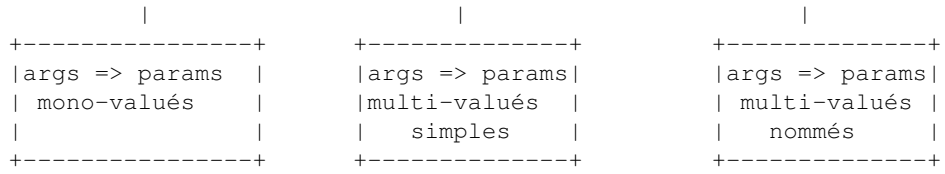
```
def func(arg_1, ..., arg_n, *args, **kwargs):
   ...
```

Les seuls appels de fonctions permettant aux trois catégories de paramètres de recevoir des arguments sont ceux qui:

- fournissent les arguments destinés aux paramètres mono-valués sous forme de valeurs simples (non-nommées) dans l'ordre des paramètres. On va parler d'**arguments positionnels** (on va voir que ce n'est pas la seule manière pour passer des arguments à une fonction).
- fournissent à l'appel **tous** les arguments correspondants aux paramètres mono-valués (pas d'utilisation des valeurs par défaut)

Dans cette situation, la correspondance entre les paramètres de la signature et les arguments à l'appel se présente ainsi :

```
#
# correspondance paramètres <=> arguments
#
def func(arg_1, arg_2, ..., arg_n, *args, **kwargs):
   ...     ^       ^          ^      ^      ^
   ...     |       |          |      |      |
   ...     |       |          |      |      +----------------------+
           |       |          |      |                             |
...        |       |          |      +---+--------+     +---------+-------------+
#          |       |          |      |   |        |     |         |             |
# appel :  |       |          |      |   |        |     |         |             |
#          |       |          |      |   |        |     |         |             |
var = func(val_1, val_2, ..., val_n, t1, t2, ..., t_m, k_1=v_1, k_2=v_2, ..., k_p=v_p)
           \_____________________/   \_____________/   \_____________________________/
                     ^                     ^                       ^
```



6.3.1 Les paramètres mono-valués

Représentés précédemment par *arg_1,...,arg_n*. On rappelle qu'un paramètre mono-valué peut se présenter sous une des deux formes:

- Paramètre simple (*variable*) : à l'appel, il référencera un argument, qui devra **toujours** être présent.
- Paramètre clé-valeur (*variable=valeur_par_défaut*) : à l'appel, l'argument correspondant au paramètre peut manquer, et dans ce cas la valeur par défaut sera utilisée.

On va rappeler également qu'un paramètre à valeur par défaut ne peut pas être suivi, dans la signature, par un paramètre simple (sans valeur par défaut).

Exemple

Une fonction de division avec une option de division tronquée:

```
def div(a, b, trunc=False):
    if trunc:
        return a // b
    return float(a) / b # en V3, inutile de convertir a en float
```

```
>>> div(5,12)
0.4166666666666667
>>> div(5,12,False)
0.4166666666666667
>>> div(5,12,True)
```

Attention

Les valeurs par défaut ne sont pas évaluées à chaque appel de la fonction, mais une seule fois, à l'appel de l'instruction **def** (qui se produit en règle générale une seule fois, au chargement d'un module, pour les fonctions non-imbriquées)

```
>>> k = 7
>>> def f(x=k+3):
...     print(x)
...
>>> f()
10
>>> k=15
>>> f()
10
>>>
```

Une précaution particulière s'impose quand les valeurs par défaut sont des objets modifiables.

Exemple à ne pas suivre :

Supposons que la fonction *create_user()* est destiné à la création de comptes informatiques (dans cette implémentation factice, elle retourne simplement un dictionnaire):

- Par défaut, en absence d'argument *groups* explicite, l'utilisateur créé sera associé au groupe 'lambda', le groupe de "tout le monde".
- Si l'argument is_admin est vrai, l'utilisateur sera associé en plus au groupe 'admin' qui donne des privilèges élevés. Ce groupe sera ajouté à la liste des groupes, qu'elle soit implicite ou explicite.

Avec l'implémentation suivante, les choses ne se passent pas comme prévu:

```
def create_user(surname, given_name, is_admin, groups=['lambda']):
    """
    - par défaut, tout le monde appartient au groupe ''lambda'
    - les admins sont aussi dans le groupe 'admin' qui donne des
    privilèges élevés
    """
    if is_admin: # c'est un admin
        groups.append('admin') # ajout du groupe 'admin'
    return {'sn': surname, 'gn': given_name, 'groups': groups}

>>> create_user('Poli','Christian',False) # utilisateur sans privilèges
{'gn': 'Christian', 'sn': 'Poli', 'groups': ['lambda']}
>>> create_user('Grand','Gourou',True) # création d'un utilisateur admin
{'gn': 'Gourou', 'sn': 'Grand', 'groups': ['lambda', 'admin']}
>>> create_user('Lagaffe','Gaston',False) # oups! lambda avec privileges admin!
{'gn': 'Gaston', 'sn': 'Lagaffe', 'groups': ['lambda', 'admin']}
```

Explication:

- le premier appel se passe comme prévu, un utilisateur sans privilèges est créé
- le deuxième appel produit également le résultat attendu (l'utilisateur créé, administrateur, a les bons groupes) mais la liste *groups* est altérée.
- le troisième appel donne un résultat erroné (alors que les arguments sont corrects) et potentiellement dangereux: un utilisateur 'lambda' se retrouve avec des privilèges 'admin'

Arguments positionnels et arguments nommés (*keyword args*)

Dans les exemples précédents les arguments à l'appel sont associés à leur paramètres correspondants par leur position à l'appel. On va parler dans ce cas d'arguments **positionnels** (*non-keyword args* en jargon pythonique). Une autre possibilité est d'exprimer les arguments sous la forme *nom=valeur* (*keyword args*) où *nom* est le nom du paramètre dans la signature. Pour ce deuxième type d'appel, l'ordre des arguments est arbitraire.

Un peut utiliser librement arguments positionnels et arguments nommés au sein d'un même appel, à condition de **ne jamais faire succéder un argument positionnel à un argument nommé** :

```
>>> div(12,5,trunc=True)
2
>>> div(12,trunc=True,b=5)
2
>>> div(trunc=True,b=5,a=12)
2
>>> div(b=5,a=12,True)
File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
>>>
```

6.3.2 Le paramètre multi-valué à valeurs simples (ou "positionnelles")

Ce paramètre optionnel succède au dernier paramètre mono-valué dans la signature (s'il y en a).

Il est unique et son nom est précédé par une étoile ("*").

Il permet d'inclure dans un appel, à la suite des arguments positionnels correspondants aux paramètres mono-valués, un nombre arbitraire d'arguments simples (positionnels aussi) qui seront accessibles, dans le bon ordre, en tant que membres d'un tuple désigné par la variable dont le nom est précédée par "*":

```
def f(a,b,c=0,*args):
    print("a:{}".format(a))
    print("b:{}".format(b))
```

```
    print("c:{}".format(c))
    print("args:{}".format(args))

>>> f(1,2)
a:1
b:2
c:0
args:()
>>> f(1,2,4)
a:1
b:2
c:4
args:()
>>> f(1,2,4,10,11,12)
a:1
b:2
c:4
args:(10, 11, 12)
>>> f(b=1,a=2,4,10,11)
File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg
>>>
```

Note: Pour pouvoir utiliser ce paramètre multi-valué il faut:

- Utiliser uniquement des arguments positionnels pour renseigner les paramètres mono-valués qui le précèdent.
- Fournir des valeurs pour **tous** les paramètres mono-valués, même pour ceux pourvus de valeurs par défaut.

NB: Ce paramètre est similaire au paramètre dit “elliptique” (...) présent dans la signature de la fonction **printf** et des autres fonctions dites “multivariadiques”, présentes dans certaines bibliothèques du langage **C** .

6.3.3 Le paramètre multi-valué à valeurs nommées

Ce paramètre optionnel et quand il est présent, il est toujours le dernier paramètre de la signature.

Il est unique et son nom est précédé par deux étoiles (“**”).

Il permet d’inclure dans un appel un nombre arbitraire d’arguments “nom=valeur” (keyword args).

Les noms des variables correspondant aux paramètres mono-valués ne peuvent pas servir de clé pour ces arguments.

```
def f(a,b,c=0,*args, **kwargs):
    print("a:{}".format(a))
    print("b:{}".format(b))
    print("c:{}".format(c))
    print("args:{}".format(args))
    print("kwargs:{}".format(kwargs))

>>> f(1,2,3,[],4,5,x=11,y=12)
a:1
b:2
c:3
args:([], 4, 5)
kwargs:{'y': 12, 'x': 11}
>>> f(x=55,b=6,y=2,a=2)
a:2
b:6
c:0
args:()
```

```
kwargs:{'y': 2, 'x': 55}
>>>
```

Dans le dernier exemple on voit que les arguments “nom=valeur” correspondant au paramètres mono-valués se mélangent sans restriction avec les arguments “nom=valeur” du paramètre `kwargs`.

NB: La fonction dict(**kwargs), déjà utilisée précédemment pour définir des dictionnaires, est un exemple d’utilisation de ce paramètre.

6.3.4 Les *var et **var dans les appels de fonctions

Les préfixes * et ** sont utilisés aussi dans les appels de fonctions:

- Avec *iter_var on peut passer les éléments contenus dans un itérable quelconque sous forme d’arguments individuels à une fonction appelée
- Avec **dict_var on peut faire la même chose avec un dictionnaire (les arguments seront de la forme nom=valeur)

NB: par *iter_var* on désigne ici toute variable référençant un itérable et par *dict_var* toute autre référençant un dictionnaire

```
def f(a,b,c,d):
    """
    prend exactement 4 args
    """
    print("f({},{},{},{})".format(a,b,c,d))


def f_call():
    """
    tente (sans succès) d'appeler f() an lui passant une liste
    """
    t = range(4)
    f(t)


def f_call2():
    """
    appel correct
    """
    t = range(4)
    f(*t)
```

Sans surprise, l’appel suivant de *f_call()* va échouer:

```
>>> f_call()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 3, in f_call
TypeError: f() takes exactly 4 arguments (1 given)
```

En revanche, l’appel de *f_call2()* fonctionnera correctement:

```
>>> f_call2()
f(0,1,2,3)
>>>
```

Les expressions *nom **nom (appelées “varargs” en jargon Python) sont utiles pour construire des appels de fonctions dynamiquement, par programme.

Exercice

```
"""
fichier: print_table.py

Ecrire une fonction qui affiche des tableaux avec print.
Les arguments positionnels donnent les noms des colonnes.
L'appel de la fonction comporte autant d'arguments positionnels qu'il y a
des colonnes, mais la signature de la fonction doit imposer qu'il y ait
au moins une colonne.

Après les noms des colonnes, un nombre arbitraire de lignes du tableau sont
fournies sous forme d'arguments nommés id=[v1, v2,..]
La clé de chaque argument nommé va alimenter la première cellule de la ligne,
les éléments de la liste associé rempliront les cellules suivantes

Par exemple, pour obtenir ce tableau:

+--------------------------------+
|Col1      |Col2      |Col3      |
+--------------------------------+
|yy        |789       |234       |
|xx        |123       |456       |
+--------------------------------+

On doit appeler:

print_table('Col1', 'Col2', 'Col3', xx=[123,456], yy=[789,234])

Pour simplifier, on va considérer:
1) la largeur des colonnes comme étant fixe (par exemple 10)
2) les arguments nommés comme étant valides (bon nombre de valeurs,
   pas de dépassement).

"""

"""
fichier: print_table_sol.py
"""
def print_table(first_hdr, *args, **kwargs):
    """
    Imprime un tableau de valeurs
    """
    nb_cols = len(args) + 1 # à cause du  first_hdr
    size_col = 10
    nb_pipes = nb_cols -1 # barres verticales entre cols (sans les extremités)
    nb_minus = size_col*nb_cols + nb_pipes
    # ligne de "-", sans les extremités:
    minus_line = nb_minus * '-'
    # avec des "+" aux extremités:
    horizontal_line = '+{}+'.format(minus_line)
    # i.e. horizontal_line: +--------------------------------+
    row_template = '|{:<10}'*nb_cols # i.e. |{:10}|{:10}|{:10}....
    row_template +='|' # i.e. |{:10}|{:10}...|
    print(horizontal_line)
    print(row_template.format(first_hdr, *args)) #print headers
    print(horizontal_line)
    for key, row in kwargs.items():
        print(row_template.format(key, *row))
    print(horizontal_line)

if __name__ == '__main__':
    print_table('Col1', 'Col2', 'Col3', xx=[123,456], yy=[789,234])
```

6.4 La documentation des fonctions

La syntaxe des fonctions Python prévoit un élément dédié à leur documentation. Cet élément, appelée **docstring** est une chaîne de caractères placée après la signature et avant le corps de la fonction:

```
def div(a, b, trunc=False):
    """
    Cette fonction divise a par b,....
    """
    if trunc:
        return a // b
    return float(a) / b # en V3, inutile de convertir a en float
```

Tous le formats string sont acceptés, le plus courant étant le format multi-ligne (triple quote).

La docstring est accessible par programme à travers l'attribut *__doc__* ou à travers la fonction *help()*:

```
>>> div.__doc__
'\n    Cette fonction divise a par b,....\n    '
>>> help(div)

Help on function div in module docstring:

div(a, b, trunc=False)
   Cette fonction divise a par b,....
(END)
```

6.5 Fonctions anonymes (lambda)

En dehors des fonctions créées avec l'instruction **def** qui ont toujours un nom et une implémentation définie par un bloc d'instructions, il existe également la possibilité de définir à la volée, pour des traitements très simples, des fonctions implémentées par une simple expression. Elles ont la forme :

lambda arg_1, arg_2,...arg_n **:** expression

Illustration :

```
>>> moyenne = lambda a,b: (a+b)/2.0
>>> type(moyenne)
<type 'function'>
>>> moyenne(5,6)
5.5
>>> (lambda a,b: (a+b)/2.0)(5,6) # même chose avec évaluation directe
5.5
>>>
>>> module = lambda x: x if x>0 else -x
>>> module(-5)
5
>>> module(7)
7
>>> (lambda x: x if x>0 else -x)(-5) # même chose avec évaluation directe
5
>>> (lambda x: x if x>0 else -x)(7)
7
>>>
```

Important

L'écriture `moyenne = lambda a,b: (a+b)/2.0` a été utilisée dans l'illustration précédente seulement pour "prouver" que *moyenne* est bien une fonction.

> Dans le développement, les écritures de la forme `foo = lambda x,y:...` sont bannies par la **PEP8** et sans intérêt pratique. Si on a besoin d'une "fonction-expression", non-anonyme, l'écriture préconisée (pour l'illustration précédente) est `def moyenne(a, b):  return (a+b)/2.0`.

Différences entre les structures **lambda** et **def**:

- **lambda** est une expression alors que **def** est une instruction
- **lambda** renvoie un objet-fonction alors que **def** affecte l'objet-fonction à un nom
- **def** définit une fonction par un bloc d'instructions, **lambda** par une simple expression (pas d'instruction dans une fonction lambda, donc pas de *return* explicite)

Quand utiliser des fonctions anonymes

Quand cela améliore la lisibilité du code. En règle générale, une fonction classique trop simple est un moindre mal qu'une fonction anonyme trop complexe :)

6.6 Conventions de nommage

La **PEP8** préconise l'utilisation des noms en minuscules pour les fonctions, si nécessaire avec des underscores pour une meilleure lisibilité. L'écriture "mixedMode" est tolérée dans les contextes ou elle est déjà présente, pour assurer la compatibilité.

6.7 Le tri revisité

6.7.1 Rappel

- Tout itérable peut être trié
- les **séquences modifiables** (*list*, *bytearray*) disposent d'une méthode **sort()** qui trie la sequence en la modifiant et retourne **None**
- **tous les itérables** peuvent être triées avec la fonction **sorted()** qui ne modifie pas l'objet à trier et qui retourne un nouvel objet qui est le résultat du tri.

```
>>> L = [5,2,1,8,0]
>>> L.sort()
>>> L
[0, 1, 2, 5, 8]
>>> set_ = {5,2,1,8,0} # ceci n'est pas une séquence...
>>> sorted(set_)
[0, 1, 2, 5, 8]
>>>
```

6.7.2 Le tri en Python 2.x

```
# Signatures:

sorted(iterable[, cmp[, key[, reverse]]])

sequence.sort([cmp[, key[, reverse]]])
```

- *cmp:* fonction de comparaison "à l'ancienne", à 2 paramètres (disons x et y) qui retourne:
  - une valeur négative si x<y
  - zéro si x==y

  - une valeur positive si x>y

- *key:* fonction à un seul paramètre qui retourne la clé à comparer pour chaque item
- *reverse:* booléen

6.7.3 Illustration:

```
us_presidents=[
    {'gn':'George','sn':'Washington'},
    {'gn':'John','sn':'Adams'},
    {'gn':'Thomas','sn':'Jefferson'}]

def cmp_sn(p1, p2):
    if p1['sn'] > p2['sn']:
        return 1
    elif p1['sn'] < p2['sn']:
        return -1
    return 0

>>> # tri avec cmp=cmp_sn
>>> sorted(us_presidents,cmp=cmp_sn)
[{'gn': 'John', 'sn': 'Adams'}, {'gn': 'Thomas', 'sn': 'Jefferson'}, {'gn': 'George', 'sn': 'Wash
>>> # même chose avec une fonction lambda (assez illisible...)
>>> sorted(us_presidents,cmp=lambda p1, p2: 1 if p1['sn'] > p2['sn'] else -1 if  p1['sn'] < p2['sn
[{'gn': 'John', 'sn': 'Adams'}, {'gn': 'Thomas', 'sn': 'Jefferson'}, {'gn': 'George', 'sn': 'Wash
>>> # key + lambda (bien adaptée car très simple)
>>> sorted(us_presidents,key=lambda p: p['sn'])
[{'gn': 'John', 'sn': 'Adams'}, {'gn': 'Thomas', 'sn': 'Jefferson'}, {'gn': 'George', 'sn': 'Wash
>>> # même chose avec reverse
>>> sorted(us_presidents,key=lambda p: p['sn'], reverse=True)
[{'gn': 'George', 'sn': 'Washington'}, {'gn': 'Thomas', 'sn': 'Jefferson'}, {'gn': 'John', 'sn':
>>>
```

6.7.4 Le tri en Python 3.x

```
# Signatures:

sorted(iterable[, key][, reverse])

sequence.sorted([key][, reverse])
```

Ce qui change:

- les fonctions **cmp** ne sont plus supportées. La bibliothèque standard fournit la fonction **functools.cmp_to_key()** qui est un outil de transition entre les versions.
- les paramètres optionnels (key, reverse) exigent des arguments keyword à l'appel

```
>>> import functools
>>> sorted(us_presidents, key=functools.cmp_to_key(cmp_sn))
[{'gn': 'John', 'sn': 'Adams'}, {'gn': 'Thomas', 'sn': 'Jefferson'}, {'gn': 'George', 'sn': 'Wash
>>>
```

Pour en savoir plus: https://docs.python.org/3/howto/sorting.html#the-old-way-using-the-cmp-parameter

CHAPTER

SEVEN

CONCEPTS AVANCÉS

7.1 Les fermetures (closures)

Rappel: En Python, les fonctions sont des objets de "premier ordre". Entre autres, un objet-fonction peut être passé en argument ou être retourné par une autre fonction :

```
>>> def fa():
...     x = 123
...     print("je suis fa(), je renvoie fb")
...     def fb():
...             print("je suis fb() et je connais x={}".format(x))
...     return fb
...
>>> func_fb = fa()
je suis fa(), je renvoie fb
>>> func_fb()
je suis fb() et je connais x=123
>>>
```

Dans cet exemple:

1. la fonction *fa()* est appelée et elle retourne l'objet-fonction *fb* qui est affecté à la variable *func_fb*. A ce stade, *fb* n'a pas été exécutée, mais seulement définie (par *def*).
2. ensuite, en exécutant *func_fb()* (par l'application de *()* à la variable *func_fb*) la fonction *fb* s'exécute enfin.

De manière informelle, le terme de **fermeture** (ou *closure* en anglais) désigne une fonction (considérée en tant qu'objet) accomapgnée de son contexte de définition. Ce contexte est représenté par l'ensemble des valeurs des variables non locales à la fonction, référencées dans son code, au moment de sa définition.

Autrement dit, dans l'exemple précédent, *func_fb* (qui référence l'objet-fonction *fb*, retourné lors de l'appel de *fa()*) se "rappelle" que `x = 123` alors que cette information provient du contexte d'exécution de *fa()*, terminé depuis.

NB: On va parler de *fermeture* seulement pour les fonctions définies à l'intérieur d'autres fonctions, les seules disposant d'un contexte non local).

Pour une illustration plus complète, voici une utilisation possible des fonctions imbriquées et du principe de fermeture.

Scénario : Dans une organisation, les utilisateurs peuvent avoir plusieurs profils prédéfinis, chaque profil définissant une liste de groups et une liste de protocoles autorisés. On souhaite disposer de plusieurs fonctions permettant de créer des comptes utilisateur, une fonction pour chaque profil.

Solution : Nous allons définir un "générateur de profils" concrétisé par la fonction **make_profile()** qui a le rôle de "fabriquer" les fonctions créatrices de comptes souhaitées (implémentation factice qui renvoie un dictionnaire).

A chaque appel *make_profile(groups, protocols)* produira une fonction capable de créer des utilisateurs ayant un profil donné :

```
def make_profile(groups, protocols):
    """
    retourne des fonctions permettant de créer des comptes utilisateur
    avec un profil donné
    """
    organisation = 'Example.org'
    def add_user(login, cn):
        """
        dans une vraie implémentation, crée le compte
        ici, retourne les données de création sous forme de dictionnaire
        """
        return dict(login=login, cn=cn, org=organisation, groups=groups,
                protocols=protocols)
    return add_user
```

On va utiliser cette fonction pour "fabriquer" des fonctions utilisables pour créer des comptes :

```
>>> add_guest = make_profile(groups=['guests'],protocols=['http'])
>>> add_staff = make_profile(groups=['staff'],protocols=['http','imap','smtp'])
>>> add_admin = make_profile(groups=['adm', 'staff'],protocols=['http','imap',\
... 'smtp','ssh'])
```

Utilisation:

```
>>> add_guest('alex','Alex Terrieur')
{'org': 'Example.org', 'login': 'alex', 'protocols': ['http'], 'groups': ['guests'], 'cn': 'Alex
>>> add_staff('alain','Alain Terrieur')
{'org': 'Example.org', 'login': 'alain', 'protocols': ['http', 'imap', 'smtp'], 'groups': ['staff
>>> add_admin('cpoli','Christian Poli')
{'org': 'Example.org', 'login': 'cpoli', 'protocols': ['http', 'imap', 'smtp', 'ssh'], 'groups':
```

Question légitime: Comment les fonctions créées (*add_guest()*, *add_staff()*, *add_admin()*) ont mémorisé les valeurs de *groups*, *protocols* et *organisation* alors que ces variables ne font pas partie de leurs contextes locaux respectifs, mais du contexte d'une fonction englobante (*make_profile()*) dont l'exécution est terminée et, par conséquent, son contexte aussi?

C'est ici que la notion de **fermeture** intervient. Si on regarde de près l'objet *add_guest* par exemple:

```
>>> dir(add_guest)
['__call__', '__class__', '__closure__', '__code__', '__defaults__', '__delattr__', '__dict__', '
```

on observe la présence d'un attribut magique nommé **__closure__**.

Cet attribut contient:

```
>>> add_guest.__closure__
(<cell at 0x1156d00: str object at 0x1159090>, <cell at 0x1156e18: list object at 0x1129b90>, <ce
```

et encore:

```
>>> add_guest.__closure__[0].cell_contents
'Example.org'
>>> add_guest.__closure__[1].cell_contents
['guests']
>>> add_guest.__closure__[2].cell_contents
['http']
```

On y reconnaît les objets référencés par les variables *organisation*, *groups* et *protocols* (on obtiendra des résultats similaires pour *add_staff* et *add_admin*).

L'attribut *__closure__* conserve ainsi une copie partielle du contexte non local des fonctions imbriquées, faite à l'instant précis de leur définition. Cette partie de contexte conservée dans *__closure__* concerne les objets référencés dans le corps de la fonction et définis dans la (ou les) fonction(s) englobante(s).

7.1.1 Exercice

```
"""
fichier: polynom.py

Ecrire une fonction polynom() ayant la signature:

def polynom(degree, a=0, b=0, c=0, d=0):
    ....

Cette fonction retourne un objet-fonction destiné au calcul de la valeur d'un
polynome de degré <degree> {0,...3} et de coéfficients a, [b,[ c,[ d]]] selon
le degré choisi.

Exemple d'utilisation:
Pour construire la fonction correspondante à : P2(x) = 2 *x^2 + x + 1
on appelle:

>>> p2 = polynom(degree=2, a=2, b=1, c=1)

>>> type(p2)
<type 'function'>

Ensuite, pour x = 2:

>>> p2(2)
11

Ensuite, pour x = 3:

>>> p2(3)
22

Pour faire simple, on ne va pas s'occuper de la validité des arguments d'appel.

Dans ces conditions, comme exercice de style, essayez d'écrire la fonction sans
utiliser des structures de contrôle.

"""
```

```
"""
fichier: polynom_sol.py
"""

def polynom(degree, a=0, b=0, c=0, d=0):
    """
    Polynom factory
    """
    def p_0(x): return a
    def p_1(x): return a*x+b
    def p_2(x): return a*x**2 + b*x +c
    def p_3(x): return a*x**3 + b*x**2 +c*x+d
    f_list = [p_0, p_1, p_2, p_3]
    return f_list[degree]
```

Une autre solution:

```
"""
fichier: polynom_sol2.py
"""

def polynom(degree, a=0, b=0, c=0, d=0):
    """
```

```
    Polynom factory
    """
    f_list = [
        lambda: a,
        lambda x: a*x+b,
        lambda x: a*x**2 + b*x +c,
        lambda x: a*x**3 + b*x**2 +c*x+d
        ]
    return f_list[degree]
```

7.2 Les décorateurs de fonctions

Il s'agit d'une technique simple et surtout non-intrusive permettant d'enrichir le comportement d'une fonction donnée sans modifier son code.

L'enrichissement du comportement se traduit par le rajout de nouvelles actions qui seront exécutées avant et/ou après l'exécution de la fonction elle même.

On va appeler **"fonction décorée"** la fonction dont le comportement sera enrichi et **"décoration"** la fonction apportant les nouveaux comportements.

Enfin, on appelera **"décorateur"** la fonction qui fait l'assemblage des deux fonctions précédentes.

Les décorateurs de fonctions sont applications directes du mécanisme de **fermeture** :

```
def simple_decorator(to_decorate):
    """
    Cette fonction est le décorateur. Elle va faire l'assemblage
    entre la décoration et la fonction
    à décorer. Elle contient la définition
    de la fonction "décoration"
    """
    def simple_wrapper(*args, **kwargs):
        """
        Cette fonction constitue
        la décoration
        """
        print("action avant {}()".format(to_decorate.__name__))
        ret = to_decorate(*args, **kwargs) # "to_decorate" fait partie de la fermeture
        print("action après {}()".format(to_decorate.__name__))
        return ret
    return simple_wrapper



def do_something(x,y):
    """
    une méthode à décorer
    """
    print("fait qq chose (x={},y={})".format(x,y))

def do_something_else(a,b,c):
    """
    une autre méthode à décorer
    """
    print("fait qq chose d'autre (a={},b={},c={})".format(a,b,c))

>>> # exécution sans décoration:
>>> do_something(1,2)
fait qq chose (x=1,y=2)
>>> do_something_else("g","r","e")
```

```
fait qq chose d'autre (a=g,b=r,c=e)
>>>
```

L'application du décorateur aux deux fonctions:

```
>>> do_something=simple_decorator(do_something)
>>> do_something_else=simple_decorator(do_something_else)
```

Exécution après décoration:

```
>>> do_something(1,2)
action avant do_something()
fait qq chose (x=1,y=2)
action après do_something()
>>> do_something_else("g","r","e")
action avant do_something_else()
fait qq chose d'autre (a=g,b=r,c=e)
action après do_something_else()
>>>
```

Python propose une syntaxe simplifiée pour appliquer un décorateur:

```
def simple_decorator(to_decorate):
    """
    Cette fonction est le décorateur. Elle va faire l'assemblage
    entre la décoration et la fonction
    à décorer. Elle contient la définition
    de la fonction "décoration"
    """
    def simple_wrapper(*args, **kwargs):
        """
        Cette fonction constitue
        la décoration
        """
        print("action avant {}()".format(to_decorate.__name__))
        ret = to_decorate(*args, **kwargs) # "to_decorate" fait partie de la fermeture
        print("action après {}()".format(to_decorate.__name__))
        return ret
    return simple_wrapper


@simple_decorator
def do_something(x,y):
    """
    une méthode à décorer
    """
    print("fait qq chose (x={},y={})".format(x,y))

@simple_decorator
def do_something_else(a,b,c):
    """
    une autre méthode à décorer
    """
    print("fait qq chose d'autre (a={},b={},c={})".format(a,b,c))
```

Le décorateur présenté ici illustre bien le principe, mais il est encore imparfait. Par exemple, la *docstring* de la fonction décorée n'est pas correcte:

```
>>> do_something.__name__
'simple_wrapper'
>>> do_something.__doc__
'\n        Cette fonction constitue\n        la décoration\n        '
>>>
```

Pour y remédier, la bibliothèque standard propose un outil qui copie les informations nécessaires de la fonction initiale vers la fonction décorée. Le décorateur initial, revu et corrigé, devient:

```
from functools import wraps

def simple_decorator(to_decorate):
    """
    Cette fonction est le décorateur. Elle va faire l'assemblage
    entre la décoration et la fonction
    à décorer. Elle contient la définition
    de la fonction "décoration"
    """
    @wraps(to_decorate)
    def simple_wrapper(*args, **kwargs):
        """
        Cette fonction constitue
        la décoration
        """
        print("action avant {}()".format(to_decorate.__name__))
        ret = to_decorate(*args, **kwargs) # "to_decorate" fait partie de la fermeture
        print("action après {}()".format(to_decorate.__name__))
        return ret
    return simple_wrapper
```

7.2.1 Exercice

```
"""
fichier: lib_decorator.py

Une bibliothèque logicielle fournit des fonctions pour le calcul des surfaces
de quelques figures géométriques.

On souhaite adapter la bibliothèque pour l'apprentissage du calcul des surfaces.

L'élève doit effectuer le calcul de la surface lui même avant d'appeler
la fonction.

Le fonction appelée :
 - demande la saisie de la valeur préalablement calculée par l'élève
 - effectue le même calcul
 - affiche les deux résultats pour vérification

Exemple d'exécution:


>>> rectangle_area(3,4)
Your value ?12
Your value: 12, calculated value: 12
12
>>>


A l'origine, la bibliothèque n'a pas été conçue pour ça. Pour l'adapter,
on va éviter les solutions naïves qui consistent à:
 - (N1) modifier les fonctions existantes (voir triangle_area_modified
   par rapport à triangle_area)
 - (N2) envelopper les fonctions de la bibliothèque par des fonctions dédiées
   (voir triangle_area_wrapper pour triangle_area)

Les deux solutions sont non satisfaisantes (pourquoi?) alors:

Il est demandé d'écrire un décorateur unique pour répondre à cette demande.
```

```
NB: On ne va pas s'occuper de détails (docstring etc.)


"""
#
# La bibliothèque
#

def triangle_area(base, height):
    "triangle area"
    return base * height * 0.5

def rectangle_area(base, height):
    "rectangle area"
    return base * height

def trapezoid_area(base, base2, height):
    "trapezoid area"
    return 0.5 * (base + base2) * height

import math

def circle_area(ray):
    "circle area"
    return math.pi * ray ** 2

# ...
# ...


#
# PS: Les solutions naïves - A EVITER!
# (elle sont là juste à titre d'exemple négatif!)
#

def triangle_area_modified(base, height):
    """
    N1) solution naïve I
    triangle_area() modifiée pour intégrer le dialogue
    """
    resp = input("Your value ?")
    calc = base * height * 0.5
    print("Your value: {}, calculated value: {}".format(resp, calc))
    return calc

def triangle_area_wrapper(base, height):
    """
    N2) solution naïve II
    wrapper dédié à triangle_area()
    """
    resp = input("Your value ?")
    calc = triangle_area(base, height)
    print("Your value: {}, calculated value: {}".format(resp, calc))
    return calc

"""
fichier: lib_decorator_sol.py
"""
import sys

def simple_dialog(to_decorate):
    """
```

```
    Cette fonction est le décorateur. Elle va faire l'assemblage
    entre la décoration et la fonction
    à décorer. Elle contient la définition
    de la fonction "décoration"
    """
    def simple_wrapper(*args, **kwargs):
        """
        Cette fonction constitue
        la décoration
        """
        question = "Your answer is?"
        resp = input(question) if sys.version_info.major==3 else raw_input(question)
        calc = to_decorate(*args, **kwargs)
        # NB: "to_decorate" fait partie de la fermeture!
        print("Your answer: {}, calculated value: {}".format(resp, calc))
        return calc
    return simple_wrapper

@simple_dialog
def triangle_area(base, height):
    "triangle area"
    return base * height * 0.5

@simple_dialog
def rectangle_area(base, height):
    "rectangle area"
    return base * height

@simple_dialog
def trapezoid_area(base, base2, height):
    "trapezoid area"
    return 0.5 * (base + base2) * height

import math

@simple_dialog
def circle_area(ray):
    "circle area"
    return math.pi * ray ** 2

if __name__ == '__main__':
    base = 5
    height = 4
    print("Compute the area of a triangle (given: base={}, height={})".format(base, height))
    triangle_area(base, height)
```

7.2.2 Exercice (suite)

```
"""
fichier: lib_decorator2.py

Parfois il peut être intéressant de pouvoir passer des paramètres à un
décorateur.

Dans l'exemple précédent créer un décorateur permettant de personnaliser le
message d'accueil.

En absence d'un message personnalisé, on va afficher un message par défaut.
NB: Pour simplifier, on va considérer que le décorateur sera appliqué
systématiquement avec l'opérateur"()" de la manière suivante:
"""
```

```
#
# avec message personnalisé:
#

@dialog_with_args(question="Triangle area?")
def triangle_area(base, height):
    "..."
    pass # ...

#
# sans message personnalisé:
#

@dialog_with_args()
def triangle_area(base, height):
    "..."
    pass # ...

"""
fichier: lib_decorator_sol2.py
"""
import sys
from functools import wraps

def dialog_with_args(question="Your value ?"):
    """
    cette fonction permet d'introduire la variable 'question' dans la fermeture
    du décorateur
    """
    def simple_dialog(to_decorate):
        """
        Cette fonction est le décorateur. Elle va faire l'assemblage
        entre la décoration et la fonction
        à décorer. Elle contient la définition
        de la fonction "décoration"
        """
        @wraps(to_decorate)
        def simple_wrapper(*args, **kwargs):
            """
            Cette fonction constitue
            la décoration
            """
            resp = input(question) if sys.version_info.major==3 else raw_input(question)
            calc = to_decorate(*args, **kwargs)
            # NB: "to_decorate" fait partie de la fermeture
            print("Your value: {}, calculated value: {}".format(resp, calc))
            return calc
        return simple_wrapper
    return simple_dialog

@dialog_with_args(question="Triangle area is?")
def triangle_area(base, height):
    "triangle area"
    return base * height * 0.5

@dialog_with_args()
def rectangle_area(base, height):
    "rectangle area"
    return base * height

@dialog_with_args()
def trapezoid_area(base, base2, height):
    "trapezoid area"
```

```
    return 0.5 * (base + base2) * height


import math

@dialog_with_args()
def circle_area(ray):
    "circle area"
    return math.pi * ray ** 2

if __name__ == '__main__':
    base = 5
    height = 4
    print("Compute the area of a triangle (given: base={}, height={})".format(base, height))
    triangle_area(base, height)
```

7.2.3 Exercice (suite)

```
"""
fichier: lib_decorator3.py

Comment faire pour éviter l'opérateur "()" quand on n'a pas d'option de décoration à passer?
"""

#
# avec message personnalisé et "()":
#

@dialog_with_args(question="Triangle area?")
def triangle_area(base, height):
    ...

#
# sans message personnalisé et SANS "()":
#

@dialog_with_args
def triangle_area(base, height):
    ...

"""
fichier: lib_decorator_sol3.py
"""
import sys
from functools import wraps

def dialog_with_args(to_decorate=None, question="Your answer is?"):
    """
    cette fonction permet d'introduire la variable 'question' dans la fermeture
    du décorateur
    """
    def simple_dialog(to_decorate):
        """
        Cette fonction est le décorateur. Elle va faire l'assemblage
        entre la décoration et la fonction
        à décorer. Elle contient la définition
        de la fonction "décoration"
        """
        @wraps(to_decorate)
        def simple_wrapper(*args, **kwargs):
            """
```

```
            Cette fonction constitue
            la décoration
            """
            resp = input(question) if sys.version_info.major==3 else raw_input(question)
            calc = to_decorate(*args, **kwargs)
            # NB: "to_decorate" fait partie de la fermeture
            print("Your answer: {}, calculated value: {}".format(resp, calc))
            return calc
        return simple_wrapper
    if to_decorate is None:
        # appel avec options, du style:
        # @dialog_with_args(question="Triangle area?")
        # En clair:
        # triangle_area = dialog_with_args(question="Triangle area?")(triangle_area)
        # dialog_with_args() n'est pas appelée en tant que décorateur mais en tant que
        # fonction retournant le décorateur donc:
        return simple_dialog
    # appel sans options, du style:
    # @dialog_with_args
    # en clair triangle_area = dialog_with_args(triangle_area)
    # dialog_with_args() est  appelée en tant que décorateur!
    # elle doit donc relayer l'appel au vrai décorateur
    # et retourner le résultat
    return simple_dialog(to_decorate)

@dialog_with_args(question="Triangle area is?")
def triangle_area(base, height):
    "triangle area"
    return base * height * 0.5

@dialog_with_args
def rectangle_area(base, height):
    "rectangle area"
    return base * height

@dialog_with_args
def trapezoid_area(base, base2, height):
    "trapezoid area"
    return 0.5 * (base + base2) * height


import math

@dialog_with_args
def circle_area(ray):
    "circle area"
    return math.pi * ray ** 2

if __name__ == '__main__':
    base = 5
    height = 4
    print("Compute the area of a triangle (given: base={}, height={})".format(base, height))
    triangle_area(base, height)
    base = 7
    height = 6
    print("Compute the area of a rectangle (given: base={}, height={})".format(base, height))
    rectangle_area(base, height)
```

7.3 Les générateurs

On a déjà évoqué précédemment le fait que les objets *itérables* sont soit des *conteneurs*, autrement dit des objets qui ont vocation à contenir d'autres objets (listes, ensembles, etc.) soit des objets plus "opaques" (comme range/xrange), qui ne contiennent pas physiquement d'autres objets, mais qui les créent au fur et à mesure de la demande, de manière "paresseuse".

Les générateurs sont des itérateurs (un itérateur peut être vu comme un itérable à usage unique!)

Les avantages des générateurs par rapport aux conteneurs classiques:

- Consommation mémoire réduite et indépendante du nombre d'occurences
- Solution en cas d'impossibilité d'utiliser les conteneurs (quand le nombre d'occurrences n'est pas prévisible à l'avance).

Syntaxiquement, un générateur se définit avec l'instruction **def**, comme une fonction classique.

Sur la forme, la particularité d'une fonction-générateur par rapport à une fonction classique est l'usage de l'instruction *yield*.

Sur le fond, la différence entre une fonction classique et une fonction-générateur est plus subtile :

- la fonction classique construit un résultat unique (par exemple, une liste de valeurs) qu'elle retourne d'un coup (avec l'instruction *return*)
- la fonction-générateur (où "générateur", tout court), au lieu de construire une liste d'éléments, produit et délivre les éléments un par un, à la demande, par l'instruction *yield*. Après chaque valeur délivrée, le générateur suspende son exécution tout en conservant l'état courant (les valeurs courantes des variables locales) lui permettant de reprendre l'exécution lors de la prochaine demande de nouvelle valeur.

Comme tout **itérateur**, un générateur implémente le protocole d'itération, déjà discuté précédemment.

On rappelle que, pour supporter ce protocole, un objet doit:

- implémenter la méthode *next()* (renommée *__next__()* en Python3)
- lever de l'exception *StopIteration* à la fin du processus.
- implémenter, la méthode *__iter__()* qui renvoie l'objet lui-même (alors qu'un itérable renvoie à chaque appel un itérateur "frais")

L'exécution de l'instruction *return* (explicite ou implicite) par une fonction-générateur provoque la levée de l'exception *StopIteration*.

Pour illustration, un exemple simple de générateur, produisant les premiers *n* entiers au cube:

```
def gen_cube_n(n):
    """
    1**3, 2**3, ...n**3
    """
    i = 0
    while i < n:
        i += 1
        yield i ** 3

>>> gen = gen_cube_n(3)
>>> type(gen)
<type 'generator'>
>>> next(gen)
1
>>> next(gen)
8
>>> next(gen)
27
>>> next(gen)
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
StopIteration
>>> for e in gen_cube_n(4):
...     print(e)
...
1
8
27
64
>>>
```

Le même exemple en version “sans fin” :

```
def gen_cube_endless():
    """
    1**3, 2**3, ...(sans fin)
    """
    i = 0
    while True:
        i += 1
        yield i ** 3
```

```
>>> gen = gen_cube_endless()
>>> next(gen)
1
>>> next(gen)
8
>>> next(gen)
27
>>> next(gen)
64
```

7.3.1 Les générateurs “expression”

Les expressions faisant office de générateurs ont une syntaxe très proche des listes en intension, les parenthèses prenant la place des crochets englobants:

```
>>> g= (i**3 for i in range(4))
>>> type(g)
<class 'generator'>
>>> next(g)
0
>>> next(g)
1
>>> next(g)
8
>>> next(g)
27
>>> next(g)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
StopIteration
>>>
```

7.3.2 Exercice

```
"""
fichier: math_serie.py

La fonction suivante calcule une liste avec les termes
```

```
d'une suite définie par une relation de récurence.
Modifier cette fonction pour en faire un générateur.
a) avec  n_max fixe
b) sans limitation
"""

def serie(n_max):
    """
    U0=-5 et Un+1=Un + 2*n+9 pour n<n_max .
    """
    u = -5
    res = [u]
    for n in xrange(n_max):
        u = u + 2 * n +9
        res.append(u)
    return res

"""
fichier: math_serie_sol.py
"""

def serie(n_max):
    """
    U0=-5 et Un+1=Un + 2*n+9 pour n<n_max .
    """
    u = -5
    yield u
    for n in xrange(n_max):
        u = u + 2*n +9
        yield u

def serie2():
    """
    U0=-5 et Un+1=Un + 2*n+9 pour n illimité
    """
    u = -5
    n = 0
    yield u
    while True:
        u = u + 2*n +9
        n += 1
        yield u
```

7.4 Fonctions récursives

En Python, comme dans d'autres langages, une fonction peut s'appeler elle-même, directement (récursion simple) ou indirectement (récursion croisée : *a()* appelle *b()* qui appelle *a()*, par exemple).

```
def fact(n):
    """
    fonction factorielle
    """
    return 1 if n==0 else n*fact(n-1)
```

```
>>> fact(4)
24
>>>
```

Contrairement à d'autres langages qui accordent une place importante à la récursion (OCaml, Prolog), Python n'optimise pas la récursion terminale (tail recursion). La profondeur de la récursion est configurée par défaut à une valeur assez contraignante :

```
 >>> sys.getrecursionlimit()
 1000

 >>> fact(1001)
File "recursion.py", line 5, in fact
  return 1 if n==0 else n*fact(n-1)
....
File "recursion.py", line 5, in fact
  return 1 if n==0 else n*fact(n-1)
RuntimeError: maximum recursion depth exceeded in comparison
>>>
```

7.5 Python et la programmation fonctionnelle

7.5.1 Rappel

La programmation fonctionnelle est une paradigme qui aborde le calcul en tant qu'évaluation des fonctions sans effet de bord, sur le modèle des fonctions mathématiques.

Corollaire: dans une telle approche, les changements d'état, les modifications de données sont bannis.

Les avantages revendiquées par ce paradigme sont, principalement:

- Prouvabilité formelle des programmes
- Décomposabilité/Composabilité des programmes
- Testing et mise au point facilités

Langage multi-paradigme, Python contient des éléments de langage propices à la programmation fonctionnelle :

- les fonctions sont des objets de première classe
- il offre des mécanismes avancés pour le traitement des listes et des itérables en général:
  - listes en intension et générateurs-expression
  - une large palette de fonctions sur les itérateurs (module **itertools**)
  - les primitives "historiques" de la programmation fonctionnelle *map()*, *reduce()*, *filter()*
  - d'autres primitives sur les itérables: *all()*, *any()*, *min()*, *max()*
- il permet la définition de fonctions d'ordre supérieur (fonctions qui manipulent d'autres fonctions, comme les décorateurs)

Pourtant, Python n'est pas un langage fonctionnel "pur" car :

- il n'encourage pas l'utilisation de la récursivité
- par défaut, il n'empêche pas les effets de bord (sauf dans les fonctions anonymes) ni les données modifiables

Les trois fonctions "historiques" de la programmation fonctionnelle (*map*, *filter*, *reduce*) sont présentes dans le langage, mais, dans la pratique les deux premières perdent du terrain en faveur des listes en intension et des générateurs-expression

- **map(f, iterable1, iterable2,...)** : applique la fonction *f* sur chaque item résultat d'une itération:
  - le nombre d'arguments de la fonction doit correspondre avec le nombre d'itérables à traiter
  - les itérables seront traités parallèlement : la fonction recevra en argument, lors d'une itération *i*, le i-ème élément de chaque
  - le processus s'arrête lorsque la fin de l'itérable le plus court est atteinte
- **filter(f, iterable_in)** : applique la fonction *f* à tous les éléments de *iterable_in* et renvoie un nouvel itérable construit à partir des éléments du premier pour lesquels *f* renvoie *True*

- **reduce(f, iterable_in[,init)** : applique la fonction *f* (à deux arguments) à tous les items de *iterable_in* de manière cumulative, de gauche à droite:
  - pour [1,2,3,4] le résultat sera équivalent à `f(f(f(1,2),3),4)`
  - Si init est fourni, il sera utilisé dans le processus comme l'élément le plus à gauche. pour init=99 l'équivalent en terme de résultat sera `f(f(f(f(99,1),2),3),4)`
  - **NB:** En version 3, la fonction *reduce* a été déplacée dans le module *functools* (cf. http://docs.python.org/3.0/whatsnew/3.0.html)

Le même document WHATSNEW préconise l'utilisation des listes en intension à la place de *map* et *filter* quand le résultat souhaité est une liste (car en version 3 ces deux fonctions fournissent des itérateurs en sortie)

NB : Les fonctions *map()* et *filter()* renvoient une liste en version 2 et un itérateur en version 3.

7.5.2 La fonction *map*

```
lst=[
    {'gn':'George','sn':'Washington'},
    {'gn':'John','sn':'Adams'},
    {'gn':'Thomas','sn':'Jefferson'}
    ]

def full_name(entry):
    """
    nom prénom
    """
    return "{0[gn]} {0[sn]}".format(entry)

>>> res = map(full_name,lst); list(res)
['George Washington', 'John Adams', 'Thomas Jefferson']
>>> # même traitement avec des listes en intension et générateurs-expression:
>>> [full_name(e) for e in lst]
['George Washington', 'John Adams', 'Thomas Jefferson']
>>> res = (full_name(e) for e in lst); list(res)
['George Washington', 'John Adams', 'Thomas Jefferson']
>>>
```

Les fonctions anonymes ont le mérite de ne jamais provoquer des effets de bord, ce qui est une exigence de première importance dans la programmation fonctionnelle:

```
>>> res = map(lambda x: x['gn']+' '+x['sn'], lst); list(res)
['George Washington', 'John Adams', 'Thomas Jefferson']
>>> # même traitement avec des listes en intension (pas besoin de lambda):
>>> [e['gn']+' '+e['sn']for e in lst]
['George Washington', 'John Adams', 'Thomas Jefferson']
>>> # générateur-expression
>>> res = (e['gn']+' '+e['sn']for e in lst); list(res)
['George Washington', 'John Adams', 'Thomas Jefferson']
>>>
```

Remarque: *map()* permet de travailler en parallèle sur plusieurs itérables, alors qu'avec les listes en intension on doit recourir à une astuce, l'utilisation de **zip**():

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

lst1 = range(1,6)
lst2 = range(7,12)

>>> res = map(prod, lst1, lst2); list(res)
[7, 16, 27, 40, 55]
>>> # avec des listes en intension:
```

```
>>> [prod(x,y) for (x,y) in  zip(lst1, lst2)]
[7, 16, 27, 40, 55]
>>> # ou encore:
>>> [x*y for (x,y) in  zip(lst1, lst2)]
[7, 16, 27, 40, 55]
>>> # générateur-expression
>>> res = (prod(x,y) for (x,y) in  zip(lst1, lst2)); list(res)
[7, 16, 27, 40, 55]
>>>
```

7.5.3 La fonction *filter*

```
>>> lst = list(range(-5,8)); lst
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7]
>>> res = filter(lambda x: x>0, lst); list(res)
[1, 2, 3, 4, 5, 6, 7]
>>> # avec des listes en intension:
>>> [x for x in lst if x>0]
[1, 2, 3, 4, 5, 6, 7]
>>>
```

7.5.4 La fonction *reduce*

```
>>> list(range(1,5))
[1, 2, 3, 4]
>>> def sum(x,y):
...     return x+y
...
>>> reduce(sum, range(1,5))
10
>>> # factorielle de 5:
>>> from functools import reduce
>>> reduce(lambda x,y:x*y, range(1,6))
120
>>>
```

7.5.5 Le module *itertools*

itertools fournit des itérables très utiles, en programmation fonctionnelle ou impérative, illustré ici sur un exemple. Soit le générateur suivant implémentant une série:

```
def series():
    """
    U0=-5 et Un+1=Un + 2*n+9 pour n illimité
    """
    u = -5
    n = 0
    yield u
    while True:
        u = u + 2*n +9
        n += 1
        yield u
```

On souhaite implémenter un itérable “paresseux” qui fournit à chaque itération, le terme courant de la série et la somme des éléments déjà générés

```
>>> from itertools import accumulate, tee
>>> crt, acc = tee(series())
```

```
>>> crt_acc = zip(crt, accumulate(acc))
>>> for res, _ in zip(crt_acc, range(5)):
...     print(res)
...
(-5, -5)
(4, -1)
(15, 14)
(28, 42)
(43, 85)
>>>
```

CHAPTER

EIGHT

L'APPROCHE OBJET

En langage commun on entend par *classe* une catégorie qui permet de désigner un ensemble d'objets ayant des propriétés en commun.

Dans les langages orientés objets (LOO) la notion de **classe** désigne l'implémentation d'un **type**.

Question légitime: Pourquoi "implémenter" des types alors que les types existent, sans les classes, dans d'autres familles de langages de programmation?

Réponse:

- l'approche procédurale classigue désigne par **type** une structure de données (seulement) et opère ainsi une séparation conceptuelle données - traitements
- le paradigme objet remet en cause cette séparation et étend la notion de **type** qui devient un ensemble cohérent de données et de traitements

La classe implémente le type (qui est juste une spécification, une "interface") et elle permet de créer, par instanciation, des objets qui ont:

- une structure de données commune
- des comportements en commun

Note: Dans la pratique, les deux termes (classe et type) sont utilisés souvent comme de synonymes.

Dans la terminologie des *LOO* on va parler d'*attributs* pour désigner les champs de la structure de données d'un objet et de *méthodes* pour désigner les comportements associés à l'objet (qui sont des fonctions avec quelques spécificités). Les comportements propres aux objets (l'exécution des *méthodes*) sont déclenchés par le mécanisme dit d'"'envoi de message" déjà utilisé dans les chapitres précédents pour les types prédéfinis, par exemple:

```
>>> "abc".upper()
'ABC'
>>> s="abc"
>>> s.upper()
'ABC'
>>>
```

L'approche objet revendique l'aptitude de favoriser la réutilisation du code grâce au mécanisme d'héritage combiné avec le polymorphisme.

Généralement on considère que les trois piliers de l'approche objet sont:

- l'héritage
- le polymorphisme
- l'encapsulation

Dans la suite de ce chapitre on va voir dans quelle mesure ces concepts sont représentés dans Python et de quelle manière.

8.1 Classes et instances

Une classe représente un **type** dans le sens décrit précédemment mais aussi une "fabrique" d'objets appartenant à ce type pour lesquels la classe elle-même constitue le *modèle*.

Le processus de création d'un objet à partir d'une classe s'appelle *instanciation* et l'objet résultant de ce processus s'appelle *instance* de ladite classe.

Chaque instance d'une classe est un objet ayant une identité propre mais possédant les attributs et les méthodes de sa classe.

Concrètement, en Python, une classe est un regroupement de variables et de fonctions dans un espace de nommage qui lui est propre.

Elle est créée par l'instruction **class** qui est (tout comme *def*) une instruction d'affectation particulière.

Tout comme l'instruction *def*, **class** contient un bloc d'instructions, mais contrairement à *def*, les instructions membres du bloc sont immédiatement exécutées, dans la foulée.

Normalement, les instructions du bloc sont des affectations de variables et des définitions de fonctions (méthodes) par *def* (qui est, on le rappelle, une forme d'affectation également). Tous les noms ainsi créés feront partie de l'espace de nommage portant le nom de la classe et désigneront ses attributs et ses méthodes.

En faisant, pour l'instant, abstraction de la notion d'héritage, la classe la plus simple (sans attributs, sans méthodes) s'écrit:

```
>>> class Simple:
...     pass #instruction qui ne fait rien
...
>>> id(Simple)
29608288
```

Pour instancier une classe, on fait appel à son *constructeur* qui est une fonction du même nom que la classe:

```
>>> s = Simple()
>>> id(s)
29913552
>>> s2 = Simple()
>>> id(s2)
29913616
>>>
```

Les objets *s* et *s2* sont des *instances* de la classe *Simple*. Ils ont des identités propres.

8.2 Les attributs

```
class Point:
    x = 100.0
    y = 100.0
```

Contrairement à *C++* et *Java*, la séparation *attribut de classe/attribut d'instance* n'est pas "étanche". Ainsi, l'attribut, défini par une affectation dans le corps de la classe, est accessible aussi bien sur la classe elle même que sur ses instances:

```
>>> Point.x
100.0
>>> Point.y
100.0
>>> p = Point()
>>> p.x
100.0
>>> p.y
100.0
```

Le mécanisme de résolution pour les variables, présenté au chapitre sur les fonctions, s'applique également entre la classe et ses instances. Tout comme dans les fonctions imbriquées, le contexte de la classe se comporte comme étant nonlocal par rapport au contexte de ses instances.

La résolution (en référencement) de *p.x* se fait dans l'ordre suivant:

1. dans le contexte de **p**
2. dans le contexte de la classe de **p**, (en cas d'échec à l'étape précédente)
3. en cas de nouvel échec le processus continuera en utilisant les mécanismes d'héritage décrits plus loin.

Ainsi, une nouvelle affectation de *x* au niveau de l'instance n'affectera pas la valeur de *x* au niveau de la classe:

```
>>> p.x = 33.0
>>> p.x
33.0
>>> Point.x # valeur inchangée:
100.0
>>>
```

L'affectation de *x* se fera dans le contexte local (celui de l'instance) et va "cacher" la définition non-locale, existante au niveau de la classe.

Réciproquement, après une affectation de *x* au niveau de l'instance, un changement de valeur (nouvelle affectation) de *x* au niveau de la classe sera sans effets sur l'instance ayant déjà affecté *x* localement:

```
>>> Point.x = 150.0
>>> p.x
33.0
>>>
```

Évidemment, les choses sont différentes pour l'attribut **y** que l'instance n'a pas modifié localement:

```
>>> p.y
100.0
>>> Point.y  = 150.0
>>> p.y
150.0
>>>
```

Mise en garde

Les objets Python acceptent de nouveaux attributs de manière "ad-hoc", ce qui peut faire passer sous silence une erreur de nom d'attribut.

```
>>> class Foo: pass
...
>>> f= Foo()
>>> f.x=4
>>> f.y=5
>>> f.z=6
>>> f.x
4
>>> f.y
5
>>> f.z
6
>>>
```

8.3 Les méthodes

Appelées aussi *fonctions membres*, elles sont des fonctions définies dans l'espace de nom de la classe, mais qui ont vocation à être utilisées au niveau des instances, par le biais du mécanisme d'envoi de messages.

Afin que chaque méthode puisse accéder au contexte de l'instance concernée par l'appel (contexte qui ne lui est pas accessible par le mécanisme de résolution) l'interpréteur "injecte" à l'appel la référence de l'instance ayant reçu me message (syntaxiquement, l'entité se trouvant du coté gauche du point précédant le message) à chaque invocation de la fonction. Afin de recevoir cette instance, la signature de chaque méthode doit contenir en première position une variable. Par convention, cette variable doit s'appeler **toujours** *self*.

La variable "self" est l'équivalente de la variable "this", présente en *C++* et *Java*:

```
class Point:
    x = 100.0
    y = 100.0
    def move(self, dx, dy):
        self.x += dx
        self.y += dy

>>> p=Point()
>>> p.move(5,6)
>>> p.x
105.0
>>> p.y
106.0
>>> Point.x
100.0
>>> Point.y
100.0
>>>
```

8.4 La méthode __init__()

C'est une méthode spéciale ou "magique" ce qui signifie qu'elle n'a pas vocation à être appelée directement par le programme, mais indirectement, par des mécanismes du langage. Néanmoins la méthode *__init__()* représente une petite entorse à cette règle : elle peut être appelée par programme mais seulement dans la définition d'une autre méthode *__init__()*.

La méthode *__init__()*, si elle existe, est appelée à chaque instanciation, suite à l'invocation du constructeur de la classe (fonction du même nom que la classe). Elle peut avoir des paramètres (autres que *self*) et dans ce cas le constructeur pourra (et devra) être invoqué avec des arguments:

```
class Point:
    def move(self, dx, dy):
        self.x += dx
        self.y += dy
    def __init__(self, x, y):
        self.x = x
        self.y = y

>>> p = Point(7,8)
>>> p.x
7
>>> p.y
8
>>>
```

Note: Afin d'éviter les interférences non souhaitées entre les attributs de classe et ceux d'instance, il est recommandé d'utiliser la méthode __init__ pour initialiser **tous** les attributs "d'instance" (qui définissent l'état de chaque instance comme, par exemple, les attributs *x* et *y* de la classe *Point*) et de dédier les attributs définis au niveau de la classe aux informations non locales, concernant l'ensemble des instances de la classe (par exemple, un compteur d'instances de *Point* créées):

8.5 La documentation des classes

Les classes disposent, tout comme les méthodes, d'une "docstring" accessible par programme:

```
>>> class Simple:
...     """Une classe
...     très simple"""
...     pass
...
>>> Simple.__doc__
'Une classe\n\ttrès simple'
>>>
```

8.6 L'héritage

C'est un mécanisme permettant à une classe de s'attribuer les définitions des attributs et les méthodes provenant d'une ou plusieurs autres classes. L'attribution des dites définitions se fait sans recopie, par le lien particulier, appelé "lien d'héritage".

La classe héritière (ou fille) dispose des définitions héritées comme s'ils elles étaient définies localement. En fonction du nombre de classes héritées (appelées *classes parentes* ou *super-classes*) on va parler d'héritage simple ou multiple.

```
class Placemark(Point):
    def __init__(self, x, y, desc, icon):
        super(Placemark, self).__init__(x,y)
        self.description = desc
        self.icon = icon
    def show(self):
        """
        description of the placemark
        """
        return "Position ({0.x}, {0.y}):{0.icon} {0.description}".format(self)
```

Dans cette exemple, la classe *Placemark* (destinée à définir des points d'intérêt sur une carte ou un plan) hérite de *Point* pour les coordonnées. C'est une spécialisation de la classe *Point*, ou une "classe fille". A ce titre:

- elle bénéficie de la méthode *move()* sans avoir à la définir.
- utilise la méthode __init__() de *Point* dans sa propre implémentation, en évitant toute redondance.
- enfin, elle est une spécialisation de *Point*, car elle implémente une nouvelle méthode, *show()*, qui lui est propre.

```
>>> p = Placemark(3,4,"phone", "@")
>>> p.show()
'Position (3, 4):@ phone'
p.move(6,7)
>>> p.show()
'Position (9.0, 11.0):@ phone'
```

Note: La primitive *super(Class,self)* permet d'appeler une méthode (le plus souvent une homonyme) héritée, en excluant *Class* et sa descendance du processus de résolution (sinon, dans l'exemple précédent, en utilisant *self* directement, on aurait une récursion infinie). Typiquement, *Class* désigne la classe courante (la classe de définition de la méthode appelante, comme dans l'exemple précédent). La primitive `super()` fait partie du modèle objet "new style", détaillé au chapitre suivant.

8.7 Old style / New style classes

- Avant la version 2.2 de Python, le modèle objet du langage était dissocié de ses types de base.
- A partir de la 2.2, on a introduit les "new style classes" dans le but d'unifier à terme les types prédéfinis et les types utilisateur. Pour écrire des classes "new style" il suffit de les faire hériter (directement ou indirectement) de la classe prédéfinie **object** qui est aussi la racine de tous les types prédéfinis. Le modèle "old style" est supporté dans toutes les versions 2.x pour des raisons de compatibilité.
- A partir de la 3.0 le modèle "old style" est abandonné: toute classe sans super-classe explicite hérite d'office de la classe *object*:

Illustration :

- Python 2.7.x:

```
>>> class Simple: # Old Style
...     pass
...
>>> s = Simple()
>>> type(s)
<type 'instance'>
>>> s.__class__
<class __main__.Simple at 0x7f449397c0b8>
>>> dir(s)
['__doc__', '__module__']
>>>
>>> class Simple(object): # New style
...     pass
...
>>> s = Simple()
>>> type(s)
<class '__main__.Simple'>
>>> s.__class__
<class '__main__.Simple'>
>>> dir(s)
['__class__', '__delattr__', '__dict__', '__doc__', '__format__', '__getattribute__', '__hash
```

- Python 3.x:

```
>>> class Simple: # New style
...     pass
...
>>> s = Simple()
>>> type(s)
<class '__main__.Simple'>
>>> dir(s)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__', '__ge__', '__geta
>>>
```

Note: Dans l'exemple précédent, on a déjà utilisé des classes "new style" en faisant hériter *Point* de *object* car, sinon, l'utilisation de la primitive *super()* n'aurait pas été possible.

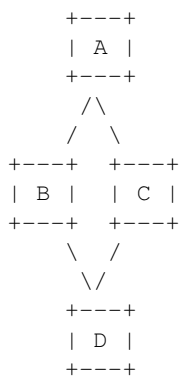
8.8 L'héritage multiple

L'héritage multiple existe dans les deux modèles ("old style" et "new style") mais les méthodes de résolution diffèrent:

- recherche en profondeur d'abord pour les classes "old style"
- résolution selon l'algorithme le linéarisation C3 (à partir de la version 2.3)

Le modèle “new style” permet de connaître l’ordre de résolution de l’interpréteur en appelant la méthode `mro()` (method resolution order).

Exemple pour le diagramme dit “en diamant”:




```
>>> class A(object): pass
...
>>> class B(A): pass
...
>>> class C(A): pass
...
>>> class D(B,C): pass
...
>>> D.mro()
[<class '__main__.D'>, <class '__main__.B'>, <class '__main__.C'>, <class '__main__.A'>, <type 'ob
```

Remarques:

- En **Python 3.x** la primitive `super()` peut être appelée sans arguments, avec un comportement par défaut identique à celui décrit précédemment.
- L’utilisation de la primitive `super()` dans la méthode `__init__()` en cas d’héritage multiple est problématique. Pour plus de détails voir http://www.artima.com/weblogs/viewpost.jsp?thread=281127

8.9 Polymorphisme

On parle de *polymorphisme* lorsque deux ou plusieurs méthodes, ayant des signatures identiques, sont définies sur des classes différentes. Dans la pratique, le polymorphisme est intéressant quand les dites classes sont reliées par des liens d’héritage.

En effet, le polymorphisme complète l’héritage au sens que :

- l’héritage permet à la classe de s’approprier l’ensemble des méthodes définies par ses super-classes.
- grâce au polymorphisme on peut redéfinir, ponctuellement, certaines méthodes héritées dans un but de *spécialisation*.

Pour mettre en perspective le polymorphisme de plusieurs *LOO* il est utile de souligner que:

- En C++, le mécanisme s’appelle aussi “liaison dynamique” et s’applique uniquement aux méthodes déclarées **virtual** ou **pure virtual**.
- En Java, toute méthode peut être spécialisée dans une sous-classe de sa classe d’origine, sauf si la dite méthode est déclarée **final**.
- En Python, les mécanismes de spécialisation s’appliquent toujours, sans exception.

Pour illustrer la complémentarité héritage - polymorphisme, prenons l’exemple d’un outil de dessin très rudimentaire, permettant de tracer le contour de certaines figures géométriques de tailles fixes avec un “motif” particulier (le motif étant un caractère) :

```
class Figure(object):
    def shape(self):
        """
        raise an error
        subclasses have to implement
        their own shape method
        """
        raise NotImplementedError
    def draw(self,pattern):
        template = self.shape()
        print(template.replace('.',pattern))

class Rectangle(Figure):
    def shape(self):
        return """
               ........
               .      .
               .      .
               .      .
               ........"""

class Triangle(Figure):
    def shape(self):
        return """
                    .
                   . .
                  .   .
                 .     .
                ........."""
```

La classe *Figure* est, en jargon "objet", une *classe abstraite* : son utilité n'est pas de produire des instances mais de faire hériter à ses classes filles (*Rectangle* et *Triangle*) des comportements communs, comme la méthode *draw()*, sans avoir à les réécrire.

Par contre, la méthode *shape()*, héritée aussi par *Rectangle* et *Triangle* a vocation à être redéfinie, sous peine d'obtenir une *NotImplementedError* à l'exécution.

Cette redéfinition de *shape()* par les classes filles est l'expression du **polymorphisme**.

Même si techniquement *Figure* reste instanciable, ses instances sont inutilisables:

```
>>> f=Figure()
>>> f.draw('*')
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 10, in draw
File "<stdin>", line 8, in shape
NotImplementedError
```

Par contre, ses sous-classes :

- bénéficient, grâce à l'héritage, de la méthode *draw()* qu'elles ne possèdent pas localement
- spécialisent la méthode *shape()* en fonction de leur propres besoins, grâce au polymorphisme

Illustration :

```
>>> r=Rectangle()
>>> r.draw('*')

            ********
            *      *
            *      *
            *      *
            ********
```

```
>>>
>>> t = Triangle()
>>> t.draw('+')

                +
               + +
              +   +
             +     +
            +++++++++
>>>
```

8.10 Encapsulation

Python ne dispose pas d'un mécanisme d'encapsulation au sens strict, comme C++ et Java. Il propose en échange une convention de nommage qui assure une forme de "pseudo-encapsulation":

- si le nom d'un attribut/méthode à l'intérieur d'une classe commence par __ (double underscore) et **ne** se termine **pas** par __ (par exemple: *__nom*) alors:
  - il sera accessible sous le nom *__nom* **seulement** dans le contexte de la classe et de ses instances
  - il sera accessible dans les autres contextes sous le nom *_Classe__nom*

```
class FrozenPoint(object):
    __counter = 0
    def getCounter(self):
        return self.__counter
    def __init__(self, x, y):
        self.__x = x
        self.__y = y
        type(self).__counter += 1
    def getX(self):
        return self.__x
    def getY(self):
        return self.__y
```

Les implémentations des méthodes accèdent les attributs sous leur vrai nom:

```
>>> p = FrozenPoint(5,6)
>>> p.getX()
5
>>> p.getY()
6
>>> p.getCounter()
1
```

Alors que les accès directs (avec les vrais noms) échouent s'ils sont faits à partir d'un contexte extérieur:

```
>>> p.__x
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'FrozenPoint' object has no attribute '__x'
```

Par contre, cette protection est contournable :

```
>>> p._FrozenPoint__x
5
>>>
```

8.10.1 La fonction property()

Dans les bonnes pratiques de la programmation objet en C++ et Java il est préconisé de contrôler l'accès aux attributs privés avec des méthodes dédiées, appelées parfois "accesseurs" (`setAttr(val)`, `val = getAttr()`, etc.). Cette pratique a des mérites mais elle nous prive de l'écriture plus lisible et concise que l'accès direct aux attributs confère.

Python arrive à concilier les deux aspects avec la primitive `property()` ayant la signature complète:

`attribute = property([getfun[, setfun[, delfun[, docstring]]]])`

- getfun(): méthode d'accès en lecture
- setfun(): méthode de mise à jour
- delfun(): suppression
- docstring : documentation

```
class Circle(object):
    def __init__(self,x, y, ray):
        self.x = x
        self.y = y
        self.__ray = ray # internal attribute
    def get_ray(self):
        return self.__ray
    def set_ray(self, ray):
        if ray <= 0:
            raise ValueError
        self.__ray = ray
    def del_ray(self):
        raise NotImplementedError
    ray = property(get_ray, set_ray, del_ray, "this is the ray attribute property")
```

```
>>> c = Circle(4,5,15)
>>> c.ray # consultation
15
>>> c.ray=-7 # tentative d'affectation avec une valeur illegale
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 10, in set_ray
ValueError
>>> c.ray=7 # affectation correcte
>>> c.ray
7
>>> del c.ray # suppression interdite
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 13, in del_ray
NotImplementedError
>>> Circle.ray.__doc__ # documentation
'this is the ray attribute property'
>>>
```

Une écriture sémantiquement équivalente, avec des décorateurs dédiés, est également possible:

```
class Circle(object):
    def __init__(self,x, y, ray):
        self.x = x
        self.y = y
        self.__ray = ray # internal attribute
    @property
    def ray(self):
        """this is the 'ray' property"""
        return self.__ray
```

```
    @ray.setter
    def ray(self, ray):
        if ray <= 0:
            raise ValueError
        self.__ray = ray
    @ray.deleter
    def ray(self):
        raise NotImplementedError
    @property
    def diameter(self):
        return self.__ray * 2
```

NB: Dans cette deuxième variante il y a un "intrus", appelé *diameter*. Il illustre la possibilité de définir des pseudo-attributs calculés, accessibles, bien sûr, en lecture seule:

```
>>> c.ray=7
>>> c.ray
7
>>> c.diameter
14
>>> c.ray=10
>>> c.diameter
20
>>>
>>> c.diameter=15
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>>
```

8.11 Agrégation d'objets

L'agrégation est une association asymétrique entre deux classe **A** et **B** qui exprime un rapport de type:

- **A** est composé de ... **B**
- **A** possède ... **B**

On va parler de **A** comme étant le contenant et de **B** comme contenu.

Dans la classe *Circle* de l'exemple précédent on a utilisé les coordonnées x et y pour désigner le centre. En utilisant l'agrégation avec la classe *Point* déjà évoquée on obtient:

```
class Circle(object):
    def __init__(self,x, y, ray):
        self.center = Point(x,y)
        self.__ray = ray
    # ...
```

Question

Pourquoi ne pas avoir fait hériter *Circle* de *Point* pour obtenir le même résultat ?

8.12 Méthodes de classe

Les méthodes déjà étudiées ne peuvent pas être appelées sans avoir créé préalablement une instance de leur classe de définition ou de l'une de ses sous-classes.

Pourtant, on peut avoir besoin définir des traitements qui concernent la classe indépendamment de ses instances.

Dans l'exemple suivant on souhaite afficher l'attribut *title*, qui est un attribut de classe par la méthode *banner()*, qui fait une mise en forme quelconque:

```
class A(object):
    title = "class A"
    def banner(self):
        print("*******")
        print(self.title)
        print("*******")
```

```
>>> A.banner()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unbound method banner() must be called with A instance as first argument (got nothing
>>> a=A()
>>> a.banner()
*******
class A
*******
```

Dans l'exemple précédent, pour pouvoir exécuter *banner()* on a dû créer une instance pour une utilisation détournée, ce qui est généralement une mauvaise idée. Pour résoudre ce problème, Python propose deux décorateurs apportant deux solutions différentes: **@staticmethod** et **@classmethod**

8.12.1 Le décorateur @staticmethod

Ce décorateur permet de définir une fonction "classique" dans l'espace de noms de la classe, rappelant le fonctionnement des méthodes statiques du *C++* et *Java*. La signature de la fonction ne contiendra pas le paramètre *self* :

```
class A(object):
    title = "class A"
    @staticmethod
    def banner():
        print("*******")
        print(A.title)
        print("*******")

class B(A):
    title = "class B"
```

La méthode est héritée, mais l'absence de *self* empêche une écriture "polymorphe" (prendre en compte l'attribut *title* de la classe sur laquelle la méthode *banner()* s'exécute):

```
>>> A.banner()
*******
class A
*******
>>> B.banner()
*******
class A
*******
>>>
```

8.12.2 Le décorateur @classmethod

Il apporte une solution aux limitations de @staticmethod. La méthode décorée ainsi contient un paramètre équivalent à *self* (appelé **par convention** *cls*) qui permet l'expression du polymorphisme:

```
class A(object):
    title = "class A"
```

```
    @classmethod
    def banner(cls):
        print("*******")
        print(cls.title)
        print("*******")

class B(A):
    title = "class B"
```

```
>>> A.banner()
*******
class A
*******
>>> B.banner()
*******
class B
*******
>>>
```

8.13 La surcharge des opérateurs

Tout comme C++ Python permet la surcharge des opérateurs à l'exception de celui d'affectation. Cette surcharge se fait en implémentant des méthodes spéciales ou "magiques". Les noms de ces méthodes commencent et se terminent par __ (double underscore):

- méthodes pour les opérateurs arithmétiques : __add__, __sub__, __mul__, __div__, __neg__, ...
- méthodes pour les opérateurs de comparaison : __eq__, __gt__, __lt__, ...
- méthodes pour les itérables : __iter__, __len__, ...
- autres méthodes __call__ pour (), __str__, ...

Sans redéfinition des comportements par défaut:

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Le résultat de la comparaison entre deux objets *Point* identiques est surprenant:

```
>>> p1 = Point(4,5)
>>> p2 = Point(4,5)
>>> p1 == p2
False
>>>
```

Et le résultat de l'impression pas très parlant:

```
>>> print(p1)
<__main__.Point object at 0x286e150>
>>>
```

Le remède :

- spécialisation de l'opérateur "==" (méthode *__eq__*)
- spécialisation de la méthode *__str__*

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y
    def __str__(self):
        return "Point({0.x}, {0.y})".format(self)
```

Les résultats sont différents:

```
>>> p1 = Point(4,5)
>>> p2 = Point(4,5)
>>> p1 == p2
True
>>> print(p1)
Point(4, 5)
>>>
```

8.14 Conventions de nommage

La **PEP8** préconise:

- Pour les **noms des classes**, la convention "CapWords" sera utilisée par défaut. La convention utilisée pour les fonctions peut s'appliquer pour certaines classes dont l'usage est basé principalement sur leur interface appelable (autrement dit, qui sont perçues par l'utilisateur comme des fonctions) **NB:** Cette convention ne s'applique pas aux builtins
- Les exceptions sont des classes (rappel) et leur nommage suit les mêmes règles.
- Toujours utiliser **self** en premier argument d'une méthode (d'instance) et **cls** en premier argument d'une méthode de classe (décorée @classmethod)
- les noms des méthodes et des attributs suivent la même convention que les fonctions En plus :
  - les noms commençant par un underscore sont considérés, par convention, comme "internes" (weak "internal use" indicator)
  - les noms commençant par deux underscores seront littéralement remplacés à la compilation comme montré précédemment

8.15 Exercice

```
fichier: chess.py

Dans le jeu d'échecs les cases sont identifiées par un couple lettre-chiffre,
la lettre étant dans la plage A-H et le chiffre dans la plage 1-8.

On souhaite implémenter (partiellement) un modèle d'objets pour représenter les
pièces du jeu d'échecs à partir de la classe "abstraite" Piece, fournie dans
l'énoncé.

a) En dérivant "Piece", implémentez la classe fille de votre choix (King,
Queen, Rook, Bishop etc.).

Trouver et implémenter la ou les méthode(s) manquante(s) pour pouvoir
l'instancier et exécuter la méthode move() sur l'instance. On va faire
abstraction de la présence d'autres pièces sur l'échiquier.

NB: si vous ne connaissez pas le mouvement des pièces sur l'équiquier, en voici
quelques exemples:

King: se déplace d'une case dans n'importe quelle direction
Queen: nombre indéterminé de cases dans n'importe quelle direction
```

```
Rook (la tour): nombre indéterminé de cases en ligne ou colonne
Bishop (le fou) : nombre indéterminé de cases en diagonale


Exemple d'exécution, après avoir implémenté la classe King:

>>> k = King('B',3,'black')
>>> k.show()
piece: <class 'echecs_sol.King'>
position: B3
>>> k.move('G',8)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "echecs_sol.py", line 22, in move
    raise ValueError('ilegal move')
ValueError: ilegal move
>>> k.move('C',2)
moved!
>>> k.show()
piece: <class 'echecs_sol.King'>
position: C2
>>>
"""
from __future__ import print_function

class Piece(object):
    """
    Abstract class Piece
    """
    @staticmethod
    def col_to_x(col):
        "a..h to 0..7 conversion"
        if col not in 'abcdefghABCDEFGH':
            raise ValueError('col not in abcdefghABCDEFGH')
        return ord(col.upper())-ord('A')
    @staticmethod
    def row_to_y(row):
        "1..8 to 0..7"
        if row < 1 or row > 8:
            raise ValueError('row not in 1..8')
        return row - 1
    def __init__(self, col, row, color):
        "constructor"
        self.__x = Piece.col_to_x(col)
        self.__y = Piece.row_to_y(row)
        if color not in ['black','white']:
            raise ValueError("color not in ['black','white']")
        self.color = color
    def move(self, new_col, new_row):
        "moves a piece on a new position"
        if not self.is_legal_move(new_col, new_row):
            raise ValueError('ilegal move')
        self.__x = Piece.col_to_x(new_col)
        self.__y = Piece.row_to_y(new_row)
        print("moved!")
    def is_legal_move(self, col, row):
        "checks if the move is legal"
        raise NotImplementedError('is_legal_move')
    def col(self):
        "get the column (as a letter)"
        return 'ABCDEFGH'[self.__x]
    def row(self):
        "get the row (as an 1..8 int)"
```

```
        return self.__y + 1
    def x(self):
        "get the comlumn (0..7 internal value)"
        return self.__x
    def y(self):
        "get the row (0..7 internal value)"
        return self.__y

    def show(self):
        "show the piece features"
        print("piece: {}".format(type(self)))
        print("position: {}{}".format(self.col(), self.row()))

fichier: chess_sol.py
"""
from chess import Piece

class King(Piece):
    "King chess piece"
    def is_legal_move(self, new_col, new_row):
        "checks if the move is legal"
        new_x = Piece.col_to_x(new_col)
        new_y = Piece.row_to_y(new_row)
        if abs(new_x - self.x()) >1 or abs(new_y - self.y())>1:
            return False
        return True

class Rook(Piece):
    "Rook chess piece"
    def is_legal_move(self, new_col, new_row):
        "checks if the move is legal"
        new_x = Piece.col_to_x(new_col)
        new_y = Piece.row_to_y(new_row)
        if new_x == self.x() or new_y == self.y():
            return True
        return False
```

8.15.1 Exercice (suite)

```
fichier: chess2.py

a) En utilisant @property, modifier la classe Piece pour implémenter les
attributs col, row, x et y en lecture seule.

Modifiez, si nécessaire, les autres méthodes sur Piece et sa (ses)
classe(s) fille(s).

Exemple d'utilisation:

>>> k = King('B',3,'black')
>>> k.x
1
>>> k.y
2
>>> k.x=5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute

b) redéfinir l'opérateur == sur Piece pour répondre True si les pièces
comparées sont du même type et ont la même couleur
```

```
(Ex: deux tours noires sont "égales")
c) redéfinir __str__ sur la classe fille implémentée pour un
affichage plus parlant
"""

fichier: chess_sol2.py
"""
from __future__ import print_function

class Piece(object):
    """
    Abstract class Piece
    """
    @staticmethod
    def col_to_x(col):
        "a..h to 0..7 conversion"
        if col not in 'abcdefghABCDEFGH':
            raise ValueError('col not in abcdefghABCDEFGH')
        return ord(col.upper())-ord('A')
    @staticmethod
    def row_to_y(row):
        "1..8 to 0..7"
        if row < 1 or row > 8:
            raise ValueError('row not in 1..8')
        return row - 1
    def __init__(self, col, row, color):
        "constructor"
        self.__x = Piece.col_to_x(col)
        self.__y = Piece.row_to_y(row)
        if color not in ['black','white']:
            raise ValueError("color not in ['black','white']")
        self.color = color
    def move(self, new_col, new_row):
        "moves a piece on a new position"
        if not self.is_legal_move(new_col, new_row):
            raise ValueError('ilegal move')
        self.__x = Piece.col_to_x(new_col)
        self.__y = Piece.row_to_y(new_row)
        print("moved!")
    def is_legal_move(self, col, row):
        "checks if the move is legal"
        raise NotImplementedError('is_legal_move')
    @property
    def col(self):
        "get the column (as a letter)"
        return 'ABCDEFGH'[self.__x]
    @property
    def row(self):
        "get the row (as an 1..8 int)"
        return self.__y + 1
    @property
    def x(self):
        "get the comlumn (0..7 internal value)"
        return self.__x
    @property
    def y(self):
        "get the row (0..7 internal value)"
        return self.__y
    def show(self):
        "show the piece features"
        print("piece: {}".format(type(self)))
        print("color: {}".format(self.color))
        print("position: {}{}".format(self.col, self.row))
```

```
    def __eq__(self, other):
        "equality of two chess pieces"
        if type(self) != type(other):
            return False
        return self.color == other.color


class King(Piece):
    "King chess piece"
    def is_legal_move(self, new_col, new_row):
        "checks if the move is legal"
        new_x = Piece.col_to_x(new_col)
        new_y = Piece.row_to_y(new_row)
        if abs(new_x - self.x) >1 or abs(new_y - self.y) > 1:
            return False
        return True
    def __str__(self):
        "object as a string"
        return "{} king  at {}{}".format(self.color, self.col, self.row)


class Rook(Piece):
    "Rook chess piece"
    def is_legal_move(self, new_col, new_row):
        "checks if the move is legal"
        new_x = Piece.col_to_x(new_col)
        new_y = Piece.row_to_y(new_row)
        if new_x == self.x or new_y == self.y:
            return True
        return False
    def __str__(self):
        "object as a string"
        return "{} rook at {}{}".format(self.color, self.col, self.row)
```

CHAPTER

NINE

LA BIBLIOTHÈQUE STANDARD EN BREF

Il s'agit d'un simple aperçu, partiel et partial, de quelques fonctions avec les paramètres les plus courants dans le but de donner un avant goût des possibilités de cette bibliothèque, très riche. Il est donc fortement conseillé de consulter constamment la documentation de référence pour une utilisation correcte de la bibliothèque standard dans le processus de développement.

9.1 La manipulation des fichiers

Le type **file** est un des types prédéfinis en Python, déjà évoqué précédemment pour illustrer la structure **with**. Pour ouvrir un fichier, on utilise le plus souvent la fonction prédéfinie *open()* à la place du constructeur *file()*. Il y a des différences notables entre les versions 2 et 3 au niveau de la signature de *open()*.

En Python 2.7.x:

```
open(name[, mode[, buffering]])
```

- name : nom du fichier (absolut ou relatif au répertoire courant)
- mode :
  - **r** : lecture (par défaut)
  - **w** : écriture (en début du fichier). **Avertissement:** Si le fichier existe le contenu courant sera perdu.
  - **a** : ajout (append) L'écriture se fera à la fin du fichier, s'il existe, sinon il sera créé.
  - **r+**, **w+**, **a+** : ouverture en mise à jour(lecture/écriture). **Avertissement:** tout comme le *w*, le *w+* tronque le fichier, le contenu existant sera perdu.
  - **b** : il s'additionne à un des flags précédents (ex: *rb*) et il signifie *binary* pour les systèmes qui en font la distinction par rapport au mode text.
- buffering : gestion de la zone mémoire "tampon":
  - 0 : pas de mémoire tampon
  - 1 : mode "ligne"
  - > 1 : taille du tampon (en octets)
  - < 0 : taille par défaut prévue par le système (et comportement par défaut de la fonction)

En Python 3.x:

```
open(file, mode='r', buffering=-1, encoding=None, errors=None, newline=None, closefd=True, opener=
```

- file : peut être un chemin (comme précédemment) ou un descripteur de fichier. Si *file* est un descripteur, le fichier correspondant sera fermé par *close()*, sauf si *closefd=False*
- mode :
  - **r**, **w**, **a**, **+** : avec les significations décrites précédemment
  - **x** création uniquement, erreur si le fichier existe déjà. *(nouveauté Python 3.3)*

  - **b** : signification nouvelle car il implique que le résultat rendu sera de type **bytes** alors que par défaut (mode "texte") le résultat est de type **str** ce qui signifie **unicode** en Python 3.x. En mode "texte", l'encodage sera:
    * celui prévu par *encoding* s'il est défini
    * celui fourni par `locale.getpreferredencoding(False)` dans le cas contraire
  - **t** mode "texte" (par défaut)
- encoding : encodage
- errors : peut contenir une des chaînes:
  - 'strict' provoque une *ValueError* en cas de caractère non supportée (valeur par défaut)
  - 'ignore' ignore les erreurs d'encodage avec le risque de perte de données
  - 'replace' remplace les caractères en erreur par un marqueur ('?')
  - 'surrogateescape' représente les octets non reconnus par un point code dans la plage U+DC80 - U+DCFF
  - 'xmlcharrefreplace' remplace les caractères non supportés par l'encodage avec l'entité XML *&#nn;* (en écriture seulement)
  - 'backslashreplace' remplace les caractères non supportés par l'encodage avec les séquence d'échapement nnn
- newline : valeurs possibles None, '', '\n', '\r', et '\r\n'
- closefd : déjà évoqué pour "file". Si "file" est un nom il est sans effet et il doit contenir la valeur par défaut.
- opener : fonction utilisateur destiné à l'ouverture du fichier *(nouveauté Python 3.3)*

Rappel:

```
>>> with open("/tmp/fich.txt","w") as fd:
...     fd.write("some text")
```

Est l'équivalent de:

```
>>> fd = open("/tmp/fich.txt","w")
>>> try:
...     fd.write("some text")
... finally:
...     fd.close()
```

9.1.1 Les autres méthodes sur les fichiers

- read(size) : *size* exprime la quantité de données à lire : s'il est négatif ou absent tout le fichier sera lu
- seek(offset,from_what) positionnement dans le fichier. *from_what* représente l'origine par rapport à laquelle l'offset est exprimé et peut prendre trois valeurs:
  - 0 : début du fichier
  - 1 : la position courante dans le fichier
  - 2 : fin du fichier
- tell() fournit la position courante dans le fichier

Exemple:

```
>>> f=open("/tmp/testfile.txt","r+")
>>> f.read()
'01234567890123456789\n'
>>> f.seek(3)
```

```
>>> f.read()
'34567890123456789\n'
>>> f.seek(-3,2)
>>> f.read()
'89\n'
>>> f.tell()
21
>>> f.read()
''
>>> f.seek(-3,2)
>>> f.write("xy")
>>> f.seek(5)
>>> f.write("ab")
>>> f.seek(0)
>>> f.read()
'01234ab78901234567xy\n'
>>>
```

9.2 Le module sys

C'est le module, déjà évoqué dans les chapitres précédents, joue principalement deux rôles:

- il fournit au programme les moyens de base d'interagir avec son environnement d'exécution (entrées/sorties, arguments de la ligne de commande)
- il donne au programme l'accès aux paramètres de l'interpréteur

Le script suivant affiche les arguments de la ligne de commande:

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

for pos, arg in enumerate(sys.argv):
    print("{}: {}".format(pos, arg))
```

```
$ ./print_argv.py 66 abc
0: ./print_argv.py
1: 66
2: abc
```

Quelques autres attributs et fonctions de ce module, illustrés par des exemples:

```
>>> import sys
>>> sys.executable
'/usr/bin/python'
>>> sys.platform
'linux2'
>>> sys.version
'2.7.3 (default, Sep 26 2013, 20:03:06) \n[GCC 4.6.3]'
>>> sys.getdefaultencoding()
'ascii'
>>> sys.modules # les modules chargés par l'interpréteur
{'copy_reg': <module 'copy_reg' from '/usr/lib/python2.7/copy_reg.pyc'>, ....}
>>> sys.path
['', '/usr/lib/python2.7', '/usr/lib/python2.7/plat-linux2', '/usr/lib/python2.7/lib-tk', '/usr/l
>>> sys.stdout.write("Un message")
Un message>>> sys.stdout.write("Un autre message \n")
Un autre message
>>> sys.stderr.write("Une erreur\n")
Une erreur
>>> sys.exit(0) # fin de l'exécution
```

9.3 L'interface avec le système d'exploitation

Il s'agit des fonctionnalités souvent équivalentes aux commandes système usuelles. Elles se retrouvent dans plusieurs modules dont **os**, **shutil** et **glob**

9.3.1 Le module os

Ce module contient un grand nombre de fonctions permettant l'interaction avec le système d'exploitation.

Avertissement

Il est vivement déconseillé d'importer ce module par `from os import *` car la fonction *os.open()* va se substituer à la primitive *open()*, avec un comportement différent. L'utilisation de l'importation simple `import os` évite ce problème.

Pour illustrer quelques unes des fonctionnalités de ce module, une mise en perspective d'opérations équivalentes réalisés en *shell* (bash) et en Python:

Accès à l'environnement (variable *PATH*) en *bash*:

```
$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/bin/X11:/usr/games
```

En Python:

```
>>> os.environ['PATH']
'/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/bin/X11:/usr/games'
```

Manipulation de répertoires en bash:

```
poli@host:~$ cd /tmp
poli@host:/tmp$ mkdir testpy
poli@host:/tmp$ cd testpy/
poli@host:/tmp/testpy$ pwd
/tmp/testpy
poli@host:/tmp/testpy$ cd ..
poli@host:/tmp$ pwd
/tmp
poli@host:/tmp$ rmdir testpy
poli@host:/tmp$
```

L'équivalent en Python:

```
>>> import os
>>> os.chdir('/tmp/') # 'cd /tmp'
>>> os.mkdir('testpy') # 'mkdir testpy'
>>> os.chdir('testpy') # 'cd testpy'
>>> os.getcwd()        # 'pwd'
'/tmp/testpy'
>>> os.chdir('..')     # 'cd ..'
>>> os.getcwd()        # 'pwd'
'/tmp'
>>> os.rmdir('testpy') # 'rmdir testpy'
```

Autres manipulations simples sur des fichiers en bash :

```
$ ls /tmp/
keyring-uCfKvw  pulse-2L9K88eMlGn7  RewriteLog.log    testpy.txt
plugtmp         pulse-PKdhtXMmr18n  ssh-lPLxInqU2454
$ mv /tmp/testpy.txt /tmp/testpy2.txt
$ ls /tmp/
keyring-uCfKvw  pulse-2L9K88eMlGn7  RewriteLog.log    testpy2.txt
```

```
plugtmp         pulse-PKdhtXMmr18n  ssh-lPLxInqU2454
$ rm /tmp/testpy2.txt
$ ls /tmp/
keyring-uCfKvw  pulse-2L9K88eMlGn7  RewriteLog.log
plugtmp         pulse-PKdhtXMmr18n  ssh-lPLxInqU2454
```

Et leur équivalent en Python:

```
>>> os.listdir("/tmp/") # 'ls /tmp/' ('ls' -A plus précisément)
['RewriteLog.log', 'testpy.txt', 'ssh-lPLxInqU2454', 'plugtmp', 'keyring-uCfKvw', 'pulse-PKdhtXMm
>>> os.rename("/tmp/testpy.txt", "/tmp/testpy2.txt") # 'mv /tmp/testpy.txt /tmp/testpy2.txt'
>>> os.listdir("/tmp/") # 'ls /tmp/'
['RewriteLog.log', 'ssh-lPLxInqU2454', 'plugtmp', 'keyring-uCfKvw', 'pulse-PKdhtXMmr18n', 'testpy
>>> os.remove("/tmp/testpy2.txt") # 'rm /tmp/testpy2.txt'
>>> os.listdir("/tmp/")
['RewriteLog.log', 'ssh-lPLxInqU2454', 'plugtmp', 'keyring-uCfKvw', 'pulse-PKdhtXMmr18n', 'pulse-
```

Autres fonctions utiles:

```
os.makedirs("dir1/dir2/.../cible")
```

L'effet de cette fonction est équivalent à celui de la commande (Linux/Unix) `mkdir -p ...` : elle crée le répertoire cible ainsi que tous les répertoires intermédiaires s'ils n'existent pas:

```
>>> os.path.exists("/tmp/")
True
>>> os.path.exists("/tmp/dir1")
False
>>> os.makedirs("/tmp/dir1/dir2/cible")
>>> os.path.exists("/tmp/dir1")
True
>>> os.path.exists("/tmp/dir1/dir2")
True
>>> os.path.exists("/tmp/dir1/dir2/cible")
True
>>>
```

```
os.walk(top, topdown=True, onerror=None, followlinks=False)
```

Cette fonction permet de parcourir une arborescence de fichiers ayant pour racine le répertoire *top*. Elle retourne un générateur qui fournit à chaque itération un triplet de valeurs contenant:

- le nom du répertoire courant
- la liste des sous-répertoires directs
- la liste des fichiers membres

Les options:

- topdown :
  - True : l'itération se fait à partir de la racine (valeur par défaut)
  - False: l'itération se fait à partir des "feuilles"
- onerror : None (par défaut) ou une fonction utilisateur qui sera appelée en cas d'erreur.
- followlinks : suivi des liens symboliques (si *True*). Peut provoquer une récursion infinie en cas de liens cycliques

Exemples:

- top-down :

```
>>> import os
>>> g=os.walk("/usr/local/lib/python2.7/dist-packages/")
>>> type(g)
```

```
<type 'generator'>
>>> next(g)
('/usr/local/lib/python2.7/dist-packages/', ['django_http_proxy-0.3.2-py2.7.egg', 'django_gua
>>> next(g)
('/usr/local/lib/python2.7/dist-packages/django_http_proxy-0.3.2-py2.7.egg', ['EGG-INFO', 'ht
>>> next(g)
('/usr/local/lib/python2.7/dist-packages/django_http_proxy-0.3.2-py2.7.egg/EGG-INFO', [], ['z
>>> # etc...
```

- bottom-up :

```
>>> g=os.walk("/usr/local/lib/python2.7/dist-packages/",topdown=False)
>>> next(g)
('/usr/local/lib/python2.7/dist-packages/django_http_proxy-0.3.2-py2.7.egg/EGG-INFO', [], ['z
>>> next(g)
('/usr/local/lib/python2.7/dist-packages/django_http_proxy-0.3.2-py2.7.egg/httpproxy', [], ['
>>>
```

A manipuler avec précaution

- remove(path) supprime le fichier
- rmdir(path) supprime le répertoire s'il est vide

9.3.2 Le module os.path

Ce module est dédié à la manipulation des chemins dans le système de fichiers.

```
>>> import os
>>> os.getcwd()
'/usr/lib/python2.7'
>>> import os.path
>>> os.path.abspath("code.py") # construit le chemin absolue en fonction de l'emplacement courant
'/usr/lib/python2.7/code.py'
>>> os.path.basename('/usr/lib/python2.7/code.py') # ~ basename cmd
'code.py'
>>> os.path.dirname('/usr/lib/python2.7/code.py') # ~ dirname cmd
'/usr/lib/python2.7'
>>> os.path.split('/usr/lib/python2.7/code.py') # (dirname, basename)
('/usr/lib/python2.7', 'code.py')
```

Les fonction suivantes permettent de tester le type d'objet (répertoire, fichier, lien etc.) désigné par un chemin:

```
>>> os.path.exists('/usr/lib/python2.7/code.py')
True
>>> os.path.exists('/usr/lib/python2.7/foobar.py')
False
>>> os.path.isfile('/usr/lib/python2.7/code.py')
True
>>> os.path.isdir('/usr/lib/python2.7/code.py')
False
>>> os.path.isdir('/usr/lib/python2.7')
True
>>> os.path.islink('/usr/lib/python2.7/code.py')
False
>>>
```

Pour la portabilité, il est préférable d'utiliser *os.path.join(...)* pour construire des chemins:

```
>>> os.path.join('/tmp','foo','bar')
'/tmp/foo/bar'
```

Attention

Si un chemin absolu est fourni, tous les éléments qui le précédent sont ignorés silencieusement:

```
>>> os.path.join('tmp','/foo','bar')
'/foo/bar'
>>> os.path.join('/tmp','/foo','bar')
'/foo/bar'
>>>
```

La fonction `os.path.normpath()` élimine les redondances pour ramener le chemin au format "canonique":

```
>>> os.path.normpath('/usr/../usr/lib//python2.7/./code.py')
'/usr/lib/python2.7/code.py'
>>>
```

NB: Sous Windows cette fonction transforme les / en \. Pour les OS "case insesitive" la fonction *os.path.normcase()* permet de passer les noms en minuscules.

9.3.3 Utilisation du caractère "joker"

La fonction *glob.glob()* permet de lister le contenu d'un répertoire avec un filtre:

```
>>> import glob
>>> glob.glob("/usr/lib/python2.7/b*py")
['/usr/lib/python2.7/binhex.py', '/usr/lib/python2.7/bdb.py', '/usr/lib/python2.7/base64.py', '/us
```

9.3.4 Des fonctions de plus haut niveau

Le module *shutil* fournit des fonctions de manipulation de fichiers (copie, suppression) d'un niveau d'abstraction plus proche des commandes système.

Fonction	Description
shutil.copy(src, dst)	Copie un fichier avec les permissions. *dst* peut être un répertoire
shutil.copymode(src, dst)	Copie les permissions **seules** entre *src* et *dst*
shutil.copystat(src, dst)	Copie les permissions+infos temps(dernier accès/modification)+flags
shutil.copy2(src, dst)	Équivalent copy() + copystat()

NB: shutil contient d'autres fonctions puissantes, permettant la copie et la suppression d'arborescences de fichiers, à manipuler avec précaution.

9.4 Les (sous)processus

Les situations les plus courantes nécessitant le lancement des nouveaux processus sont traitées par quelques fonctions du module **subprocess** :

```
subprocess.call(args, *, stdin=None, stdout=None, stderr=None, shell=False, timeout=None) # timeou
```

C'est une signature abrégée, le deuxième paramètre `*` symbolise des paramètres moins utilisés qui ne seront pas discutés ici.

- args : représente la commande à exécuter. Peut être une liste ou une chaîne de caractères
- stdin, stdout, stderr : redirection possible des entrées/sorties standard
- shell : si `shell=True` la commande sera exécutée par le *shell*

Avertissement

L'argument shell=True représente un risque important en terme de sécurité en particulier en présence d'entrées non fiables. Son utilisation est *vivement déconseillée*.

Cette fonction:

- exécute la commande exprimée par args
- attend la fin de l'exécution
- renvoie le code de retour système

```
>>> subprocess.call("ls")
controller  __init__.py  __init__.pyc  model  view
0
>>> subprocess.call(["ls", "view"])
admviews.py   appviews1.py   appviews1.pyc  appviews2.pyc  __init__.pyc
admviews.pyc  appviews1.py~  appviews2.py   __init__.py
0
>>> subprocess.call(["ls","-l", "/tmp/locked.txt"])
---------- 1 poli poli 0 mars  11 15:44 /tmp/locked.txt
0
>>> subprocess.call(["cat", "/tmp/locked.txt"])
cat: /tmp/locked.txt: Permission non accordée
1
>>>

subprocess.check_call(args, *, stdin=None, stdout=None, stderr=None, shell=False, timeout=None) #
```

Effectue le même traitement que *subprocess.call()* mais lève une exception si le code de retour est *!=0*

```
>>> subprocess.check_call(["cat", "/tmp/locked.txt"])
cat: /tmp/locked.txt: Permission non accordée
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/usr/lib/python2.7/subprocess.py", line 511, in check_call
raise CalledProcessError(retcode, cmd)
subprocess.CalledProcessError: Command '['cat', '/tmp/locked.txt']' returned non-zero exit status
>>>

subprocess.check_output(args, *, stdin=None, stdout=None, stderr=None, shell=False, timeout=None)
```

Effectue le même traitement que `subprocess.check_call()` mais retourne la sortie standard:

```
>>> result = subprocess.check_output(["echo", "Hello"])
>>> result
'Hello\n'
>>>
```

Note: Des utilisation avancées de ce module, qui dépassent le cadre de cette introduction, passent par l'utilisation directe de la classe **subprocess.Popen**

9.5 Le module platform

Comme son nom l'indique ce module fournit des informations sur la plateforme :

- Architecture : platform.architecture(executable='/usr/bin/...')
- Type de matériel : platform.machine()
- Nom réseau de l'hôte : platform.node()
- Chaîne de caractère contenant système, release, type de machine etc. : platform.platform()
- Nom du processeur : platform.processor()
- Infos système (nom,version,release) du système : platform.system(), platform.version(), platform.release()

- Infos sur l'interpréteur : python_build(), python_compiler(), python_version()

9.6 L'accès à Internet en http

Un module simple permettant de faire des requêtes *http* est *urllib2*. Pour une utilisation basique il suffit de passer une *url* en argument à la fonction *urlopen()* qui retournera un itérateur qui permet de récupérer le résultat, ligne par ligne:

```
>>> import urllib2
>>> i = urllib2.urlopen("http://example.org")
>>> next(i)
'<!doctype html>\n'
>>> next(i)
'<html>\n'
>>> next(i)
'<head>\n'
>>> # ...
```

On peut également récupérer le résultat d'un coup:

```
>>> urllib2.urlopen("http://example.org").read()
'<!doctype html>\n<html>\n<head>\n ...
```

9.7 Compression des données

Les formats courants sont supportés:

- zlib
- gzip
- bz2
- zipfile
- tarfile

Illustration, avec une belle citation de *Boby Lapointe* :

```
>>> import zlib
>>> citation = "Davantage d'avantages avantagent davantage."
>>> len(citation)
43
>>> zipped = zlib.compress(citation)
>>> len(zipped)
32
>>> zlib.decompress(zipped)
"Davantage d'avantages avantagent davantage."
>>>
```

9.8 ConfigParser et json

Configparser, renommé *configparser* en version 3, permet de créer et parser des fichiers de configuration de type **.ini**, **.cfg** etc., assez répandus dans tous les environnements.

Création d'un fichier de configuration (pour un logiciel imaginaire):

```
>>> import ConfigParser
>>> conf = ConfigParser.RawConfigParser()
>>> conf.add_section("ClientSide")
>>> conf.set("ClientSide","buffer",1000)
>>> conf.set("ClientSide","timeout",10)
>>> conf.add_section("ServerSide")
>>> conf.set("ServerSide","db","mysql")
>>> conf.set("ServerSide","logs","/var/log/mysrv")
>>> with open("myconf.cfg","w") as fd:
...     conf.write(fd)
```

Résultat :

```
$ cat myconf.cfg
[ClientSide]
buffer = 1000
timeout = 10

[ServerSide]
db = mysql
logs = /var/log/mysrv
```

Pour parser le fichier de configuration précédent:

```
>>> conf = ConfigParser.RawConfigParser()
>>> conf.read("myconf.cfg")
['myconf.cfg']
>>> conf.getint("ClientSide","buffer")
1000
>>> conf.get("ServerSide","db")
'mysql'
```

JSON, acronyme de *JavaScript Object Notation*, est un format d'échange très utilisé. Les structures de données *JavaScript* étant assez proches de celles de Python (listes, dictionnaires), l'utilisation de *JSON* est très intuitive;

```
>>> import json
>>> some_data = {'a':3, 'b':[5,'xyz']}
>>> json.dumps(some_data)
'{"a": 3, "b": [5, "xyz"]}'
>>> txt = json.dumps(some_data)
>>> type(txt)
<type 'str'>
>>> json.loads(txt)
{u'a': 3, u'b': [5, u'xyz']}
>>> data = json.loads(txt)
>>> type(data)
```

9.9 Mesure des performances

Le module *timeit* offre un moyen simple pour comparer des performances entre plusieurs implémentations d'une fonctionnalités, plusieurs approches d'un même problème etc.

Par exemple, on peut comparer le temps nécessaire pour la construction d'une liste de 100 éléments avec le temps pour construire un générateur équivalent:

```
>>> from timeit import Timer
>>> Timer('[e**2 for e in xrange(1,100)]').timeit()
9.104944229125977
>>> Timer('(e**2 for e in xrange(1,100))').timeit()
0.6118359565734863
```

...et le temps pour fabriquer la liste indirectement, en passant par la fabrication du dit générateur:

```
>>> Timer('list((e**2 for e in xrange(1,100)))').timeit()
10.48848009109497
```

9.10 Le module math

Fournit des fonctions de base en théorie des nombres, logarithmes, trigonométrie, conversions d'angles etc:

```
>>> import math
>>> math.tan(math.pi/4)
0.9999999999999999
>>> math.pi
3.141592653589793
>>> math.e
2.718281828459045
>>> math.log(math.e**3) # logarithme naturel (appel à 1 seul argument)
3.0
>>> math.log(1000,10) # 2 args, équivalent  log(x)/log(10)
2.9999999999999996
>>> math.log10(1000) # log décimal, plus précis que log(x,10)
3.0
>>> math.radians(90) # pi/2
1.5707963267948966
>>> math.degrees(math.pi/2)
90.0
>>> # etc.
...
```

9.11 Le module random

Fournit des générateurs de nombres pseudo-aléatoires.

```
>>> import random
>>> random.random()
0.28578533604922873
>>> random.randrange(16)
15
>>> random.randrange(16)
10
>>> random.choice("abc")
'c'
>>> random.choice("abc")
'a'
>>>
```

9.12 Les expressions rationnelles (ou régulières)

```
>>> re.findall(r'\bon[a-z]*', 'There should be one-- and preferably only one --obvious way to do i
['one', 'only', 'one']
>>> prog= re.compile(r'\bon[a-z]*') # avec un motif compilé
>>> prog.findall('There should be one-- and preferably only one --obvious way to do it. (Tim Pete
['one', 'only', 'one']
```

CHAPTER

TEN

LES TESTS

Plusieurs méthodes dites *agiles* préconisent une démarche de développement dirigée par les tests (“Test Driven Development” ou TDD) qui suppose l’écriture des scripts de test parallèlement à l’écriture du logiciel cible.

D’autres approches méthodologiques préconisent l’utilisation de tests unitaires au cours du processus de développement.

La mise en place de tout processus de développement intégrant les tests dans sa stratégie nécessite des outils d’automatisation des *tests unitaires*. La bibliothèque standard de Python dispose de deux modules dédiés à cette tache:

- *doctest* qui est destiné plus à maintenir une documentation de qualité qu’à écrire des vrais tests unitaires
- *unittest* outil beaucoup plus complet que le précédent, bien adapté pour la mise en place d’une politique de test.

10.1 Le module doctest

Est un outil qui va scanner un module et va exécuter les tests incorporés dans les docstrings des fonctions. Même s’il n’est pas complet en tant qu’outil de test, il a au moins trois mérites:

- permet d’enrichir la documentation
- permet de détecter des éventuelles divergences entre la documentation et le code
- il est très simple à mettre en place. Il suffit d’exécuter la fonction à tester dans l’interpréteur interactif et de copier/coller les lignes contenant l’appel ainsi que le résultat dans la docstring de la dite fonction.

```
def triangle_area(height,base):
    """ Computes the triangle area
    >>> triangle_area(5,7)
    17.5
    """
    return (height * base)/2.0
import doctest
if __name__ == "__main__":
    doctest.testmod()
```

Si on injecte une erreur dans le code (par exemple, on remplace par inadvertance l’opérateur **/** par **//**):

```
def triangle_area(height,base):
    """ Computes the triangle area
    >>> triangle_area(5,7)
    17.5
    """
    return (height * base)//2.0
import doctest
if __name__ == "__main__":
    doctest.testmod()
```

l'exécution va lever une exception:

```
$ python area_doctest_err.py
**********************************************************************
File "area_doctest_err.py", line 3, in __main__.triangle_area
Failed example:
triangle_area(5,7)
Expected:
     17.5
Got:
     17.0
**********************************************************************
1 items had failures:
1 of   1 in __main__.triangle_area
***Test Failed*** 1 failures.
```

10.2 Le module unittest

Est un outil plus complet, permettant de définir des jeux de test ayant une structure propre et maintenus dans des fichiers séparés du module testé. Il met en oeuvre quatre concepts:

- "test fixture" (banc de test) qui met en place l'environnement d'exécution. Cette action précède le test (création d'objets, connexions DB, répertoires etc).
- "test case" c'est une unité individuelle de test. Concrètement, c'est une sous-classe de la classe *TestCase* fournie par le module.
- "test suite" une collection composite de "test cases" et/ou "test suites"
- "test runner" dispositif pour organiser l'exécution des tests et la collecte/rendu des résultats. Il peut prendre plusieurs formes: interface graphique, ligne de commande etc.

Les sous-classes de *TestCase* héritent de plusieurs méthodes intéressantes:

- setUp() : appelée avant l'exécution de chaque méthode de test:
  - met en place l'environnement d'exécution préalable au test ("test fixture")
  - elle a vocation à être surchargée (par défaut elle ne fait rien)
- run() : lance les test de la classe (les méthodes dont le nom commence par **test_**
- TearDown() : fait le ménage après chaque test (réussi ou pas)

Les tests, (les méthodes **test_**) ont la possibilité d'utiliser dans leur implémentation plusieurs méthodes héritées utiles, au nom évocateur (liste non exhaustive):

- *assertEqual(first, second, msg=None)*
- *assertRaises(exceptionClass, callableObj, *args, **kwargs)*
- *assertTrue(expr, msg=None)*

Par exemple, si on se propose de tester le module *modpoint* :

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def move(self, dx, dy):
        self.x += float(dx)
        self.y += float(dy)
    def __eq__(self, other):
        return self.x == other.x and self.y == other.y
```

Un *test case* très simple peut se présenter ainsi :

```
import modpoint
import unittest


class TestPoint(unittest.TestCase):

    def setUp(self):
        self.point = modpoint.Point(4,5)

    def test_move(self):
        with self.assertRaises(ValueError):
            self.point.move("abc",4)
        self.point.move(3,4)
        self.assertEqual((self.point.x,self.point.y), (7,9))

    def test_equal(self):
        point2 =  modpoint.Point(4,5)
        self.assertTrue(self.point == point2)


if __name__ == '__main__':
    #unittest.main()
    suite = unittest.TestLoader().loadTestsFromTestCase(TestPoint)
    unittest.TextTestRunner(verbosity=2).run(suite)
```

On peut faire exécuter les tests avec:

```
if __name__ == '__main__':
    unittest.main()
```

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

Sinon, avec un “runner” plus évolué:

```
if __name__ == '__main__':
   suite = unittest.TestLoader().loadTestsFromTestCase(TestPoint)
   unittest.TextTestRunner(verbosity=2).run(suite)
```

```
$ python test_modpoint.py
test_equal (__main__.TestPoint) ... ok
test_move (__main__.TestPoint) ... ok

----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

NB: Par convention, les tests relatifs à un module (par exemple *foobar*) sont regroupé dans un fichier séparé appelé test_foobar.py

CHAPTER

ELEVEN

OUTILS DIVERS

11.1 L'installeur Pip

L'installation de nouveaux paquets peut se faire de plusieurs manières, entre autres:

1. en utilisant les paquets de sa distribution de système d'exploitation (Linux, par exemple)
2. en utilisant les paquets du PyPi (Python Package Index) https://pypi.python.org/pypi et l'installeur **pip**

Les deux approches présentent des avantages et des inconvénients et il ne s'agit pas ici de proposer un choix dans le cas d'une utilisation *standard* car ce choix peut dépendre de plusieurs facteurs (culture de l'équipe, contraintes de déploiement etc.).

Par contre dans la perspective de l'utilisation de **virtualenv** pour créer des environnements d'expérimentation (section suivante), la connaissance des fonctionnalités de *pip* est utile.

11.1.1 Installer pip

On a le choix entre:

- installer le paquet de la distribution (selon le système d'exploitation)
- L'archive source http://pypi.python.org/packages/source/p/pip/
  - télécharger le bonne version
  - décompresser, désarchiver
  - dans le répertoire pip-xxx (désarchivé) exécuter `python setup.py install`
- utiliser *easy_install*:

  ```
  easy_install pip
  ```

11.1.2 Installer un paquet avec *pip* :

```
$ pip install MonPaquet
```

11.1.3 Lister les paquets installés

```
$ pip freeze
Cheetah==2.4.4
CherryPy==3.2.2
Django==1.5.4
Fabric==1.3.2
.....
```

11.1.4 Installer une version avec contraintes

```
$ pip install 'MonPaquet<2.1'
```

On peut combiner plusieurs contraintes parmi **==, >=, >, <, <=**:

```
$ pip install 'MonPaquet>2.0,<2.1'
```

11.1.5 Mise à jour d'un paquet

```
$ pip install -U MonPaquet
```

11.1.6 Suppression d'un paquet

```
$ pip uninstall MonPaquet
```

11.2 Les environnements virtuels

Nous avons parfois besoin d'expérimenter des logiciels qui ont des dépendances particulières (par exemple, des versions très récentes et pas encore stables de certaines bibliothèques).

Plusieurs projets gérés en parallèle peuvent utiliser des bibliothèques incompatibles entre elles. Faire ces installations dans l'environnement standard peut être dangereux (risque de rendre le système instable) voire impossible.

Pour faire face à ce problème, une des solutions passe par la création des environnements virtuels. Même si Python3.3 possède une solution integrée (**venv**) la solution la plus répandue reste, pour l'instant (début 2014), **virtualenv** .

11.2.1 Installer virtualenv

On peut installer *virtualenv* à partir de la distribution système, par exemple, pour ubuntu:

```
$ sudo apt-get install python-virtualenv
```

ou avec pip:

```
$ sudo pip install virtualenv
```

ou, pour une installation dans son espace personnel:

```
$ pip install --user virtualenv
```

Ensuite, on peut créer des environnements virtuels (par exemple, dans un sous-répertoire dédié dans son répertoire $HOME):

```
$ pwd
/home/poli/virtual_envs
$ virtualenv virt_env_1
New python executable in virt_env_1/bin/python
Installing distribute.....................................................................
Installing pip.............done.
```

La commande précédente crée un environnement Python à part entière (ici, une vue partielle de l'arborescence de fichiers créée):

```
$ tree --charset ascii  -L 2 virt_env_1
virt_env_1
|-- bin
|   |-- activate
|   |-- activate.csh
|   |-- activate.fish
|   |-- activate_this.py
|   |-- easy_install
|   |-- easy_install-2.7
|   |-- pip
|   |-- pip-2.7
|   `-- python
|-- include
|   `-- python2.7 -> /usr/include/python2.7
|-- lib
|   `-- python2.7
`-- local
    |-- bin -> /home/poli/virtual_envs/virt_env_1/bin
    |-- include -> /home/poli/virtual_envs/virt_env_1/include
    `-- lib -> /home/poli/virtual_envs/virt_env_1/lib
```

Une fois créé, il faut activer l'environnement pour l'utiliser:

```
$ cd virt_env_1/
$ source bin/activate
(virt_env_1) $
```

NB: Le prompt modifié `(virt_env_1) $` indique l'environnement courant

Toute installation faite avec **pip** dans un environnement activé sera locale à cet environnement, sans interférence avec l'environnement standard.

L'interpréteur lancé dans cet environnement aura accès aux modules installés localement.

```
>>> import sys
>>> sys.path
['', '/home/poli/virtual_envs/virt_env_1/local/lib/python2.7/site-packages/distribute-0.6.24-py2.
>>>
```

Installation d'un module :

```
(virt_env_1) $ pip install web.py
Downloading/unpacking web.py
Downloading web.py-0.37.tar.gz (90Kb): 90Kb downloaded
Running setup.py egg_info for package web.py

Installing collected packages: web.py
Running setup.py install for web.py

Successfully installed web.py
Cleaning up...
```

Desactivation de l'environnement (pour revenir à l'environnement de départ):

```
(virt_env_1) $ deactivate
```

11.2.2 Options intéressantes *de virtualenv*

Niveau d'isolation par rapport au système:

- *–no-site-packages* : pas d'héritage des modules installés sur le système, sauf les modules standard (os,sys,...)

- *–system-site-packages* : tout module python installé sur le système sera disponible en local (y compris les installations ultérieures à la création de l'environnement virtuel). Bien sûr, tout module installé localement sera prioritaire par rapport à son homonyme présent sur le système, le cas échéant.

Choix de la version de Python:

Si plusieurs versions de Python sont disponibles sur le système on peut opter pour une des versions à la création de l'environnement:

```
$ virtualenv virt_env_2 -p /usr/bin/python3
```

11.3 Le débogueur pdb

Est un environnement interactif proposant les fonctionnalités classiques, utiles dans la mise au point du logiciel:

- mise en pause du programme
- inspection des variables et de la pile d'exécution
- exécution "pas à pas"
- définition de points d'arrêt
- ...

Dans la suite on utilisera le script suivant pour illustrer l'utilisation de pdb :

```
class DummyClass(object):
    def __init__(self,val):
        self.first_val = val

    def do_something(self):
        alphabet = "abcdefghijklmnopqrstuvxyzw"
        first = self.first_val;
        last = first + len(alphabet)
        alpha_range = list(range(first, last))
        alpha_zip=zip(alphabet, alpha_range)
        for tpl in alpha_zip:
            print("{0[0]}:{0[1]}".format(tpl))


if __name__ == "__main__":
    obj = DummyClass(514)
    obj.do_something()
```

Il y a trois manières pour lancer *pdb*:

- à partir de la ligne de commande
- dans une session interactive de l'interpréteur
- à partir du programme

11.3.1 pdb en ligne de commande

Il s'agit d'exécuter **pdb** en lui passant en argument le script à déboguer.

```
$ python -m pdb ./pdb_sample.py
> /home/poli/mypython-course/source/includes/pdb_sample.py(1)<module>()
-> class DummyClass(object):
(Pdb)
```

L'exécution de pdb aura comme effet le chargement du code source. L'exécution s'arrête sur la première instruction présente dans le fichier et *pdb* se met en attente des commandes utilisateur.

11.3.2 pdb lancé dans une session interactive

Il s'agit d'importer le module à déboguer ainsi que le module pdb ensuite exécuter `pdb.run()` :

```
>>> import pdb_sample
>>> import pdb
>>> pdb.run('pdb_sample.DummyClass(514).do_something()')
> <string>(1)<module>()
(Pdb)
```

Parmi les commandes les plus utilisées:

- h(elp) [cmd] : aide en ligne
- w(here) : affiche la pile d'exécution, de haut en bas. Une flèche indique le contexte courant (dans lequel on peut visualiser l'état des variables etc.)
- d(own) [cnt]/u(p) [cnt] : déplacements dans la pile
- b(reak) [([fich:]ligne | fonction) [, condition]] : point d'arrêt (retourne un numéro)
- tbreak [([fich:]ligne | fonction) [, condition]] : point d'arrêt temporaire (disparaît après avoir été déclenché une fois)
- cl(ear) [fich:ligne | num [num ...]] : effacer 1..n points d'arrêt
- enable/disable [num [num ...]] : activer désactiver point d'arrêt.
- ...

11.3.3 pdb lancé à partir du programme

Cette possibilité est utile quand on souhaite déclencher *pdb* le plus tard possible, par exemple en cas d'exécution longue avant l'apparition du problème à déboguer.

La mise en oeuvre se fait en appelant la fonction *pdb.set_trace()* :

```
import pdb
class DummyClass(object):
    def __init__(self,val):
        self.first_val = val

    def do_something(self):
        alphabet = "abcdefghijklmnopqrstuvxyzw"
        first = self.first_val;
        last = first + len(alphabet)
        alpha_range = list(range(first, last))
        alpha_zip=zip(alphabet, alpha_range)
        for tpl in alpha_zip:
            print("{0[0]}:{0[1]}".format(tpl))
            pdb.set_trace()

if __name__ == "__main__":
    obj = DummyClass(514)
    obj.do_something()
```

Il suffit d'exécuter le script par un appel normal:

```
$ python pdb_sample_set_trace.py
a:514
> /home/poli/mypython-course/source/includes/pdb_sample_set_trace.py(12)do_something()
-> for tpl in alpha_zip:
(Pdb)
```

11.3.4 débogue “post mortem”

Il s’agit de déboguer en programme qui s’est déjà terminé par une exception. Dans l’exemple initial, appelant le constructeur *DummyClass()* avec une chaîne en argument l’exécution se terminera avec une exception *TypeError*. On peut ensuite, en utilisant la fonction `pdb.pm()` ou `pdb.post_mortem()` retrouver le contexte d’exécution au moment où l’erreur s’est produite:

```
>>> import pdb_sample
>>> import pdb
>>> pdb_sample.DummyClass("abc").do_something()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "pdb_sample.py", line 8, in do_something
  last = first + len(alphabet)
TypeError: cannot concatenate 'str' and 'int' objects
>>> pdb.pm()
> /home/poli/mypython-course/source/includes/pdb_sample.py(8)do_something()
-> last = first + len(alphabet)
(Pdb)
```

NB: Plusieurs outils de débogage basés sur **pdb** existent. Ils ont le mérite de proposer une ergonomie améliorée. Parmi eux on peut citer:

- **pudb** : https://pypi.python.org/pypi/pudb proposant une interface un mode caractère.
- **spyder** : https://pypi.python.org/pypi/spyder Environnement intégré (IDE) proposant une interface graphique.

11.4 Le module __future__

Les fonctionnalités Python 3 incompatibles avec la version 2 peuvent être testées et utilisées “par anticipation” dans l’ancienne version. C’est une stratégie possible pour préparer le passage à la nouvelle version mais sa pertinence est à évaluer au cas par cas.

Les propriétés importantes par rapport à la version 3 sont:

- division
- absolute_import
- print_function
- unicode_literals

Illustration:

```
>>> 5/6
0
>>> from __future__ import division
>>> 5/6
0.8333333333333334
>>> print sys.version
2.7.3 (default, Feb 27 2014, 19:58:35)
[GCC 4.6.3]
>>> from __future__ import print_function
>>> print sys.version
  File "<stdin>", line 1
    print sys.version
              ^
SyntaxError: invalid syntax
>>> print(sys.version)
2.7.3 (default, Feb 27 2014, 19:58:35)
[GCC 4.6.3]
>>>
```