

INTRODUCTION AU LANGAGE *PYTHON*

Pierre Puisseux, Université de Pau et des Pays de l'Adour

pierre.puisseux@univ-pau.fr

www.univ-pau.fr/~puisseux

RÉSUMÉ. Ce document reprend en très grande partie le tutoriel de Guido Van Rossum [VR].
Quelques modifications et simplifications y sont apportées.

1. INTRODUCTION

Python est développé depuis 1989 par [Guido van Rossum](#) et de nombreux contributeurs.

Un classement des langages de programmation les plus [populaires](#), par

<http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>

The ratings are calculated by counting hits of the most popular search engines. The search query that is used is + "<language> programming". The search query is executed for the regular Google, Google Blogs, MSN, Yahoo!, Wikipedia and YouTube web search for the last 12 months.

2009	2010	Langage	Ratings Jan 2010	delta
1	1	Java	17.482%	-1.54%
2	2	C	16.215%	+0.28%
5	3	PHP	10.071%	+1.19%
3	4	C++	9.709%	-0.41%
4	5	(Visual) Basic	7.354%	-1.81%
6	6	C#	5.767%	+0.16%
7	7	<i>Python</i>	4.453%	-0.28%
8	8	Perl	3.562%	-0.74%
9	9	JavaScript	2.707%	-0.65%
11	10	Ruby	2.474%	-0.67%
10	11	Delphi	2.392%	-0.91%
37	12	Objective-C	1.379%	+1.24%
-	13	Go	1.247%	+1.25%
14	14	SAS	0.809%	+0.01%
13	15	PL/SQL	0.718%	-0.29%
18	16	ABAP	0.641%	+0.10%
15	17	Pascal	0.624%	-0.04%
23	18	Lisp/Scheme	0.576%	+0.14%
20	19	ActionScript	0.566%	+0.11%
24	20	MATLAB	0.540%	+0.11%

A noter : le succès improbable du langage [go](#) (Google) réunissant les avantages de C++ et *Python*, évitant les écueils de C++ et de *Python*.

1.1. Caractéristiques du langage *Python*.

- ▷ langage qui évolue → version 3
- ▷ logiciel libre, licence [PSF](#) (Python Software Foundation), GPL-compatible → perrène, ouvert
- ▷ portable (Mac OS, Linux, Unix, Windows...) → universalité
- ▷ une librairie standard très fournie → voir <http://docs.python.org/library/>
- ▷ nombreuses extensions et paquetages (Qt, VTK, SQL...) → puissance
- ▷ inter-opérable, extensible (avec Fortran, Java, C, C++...) → extensions
- ▷ *Jython*, est écrit en Java et génère du bytecode Java
- ▷ refcounting Garbage Collector → gestion mémoire par comptage de références (sans intervention du programmeur)
- ▷ **Pas** de pointeurs explicites en *Python*)

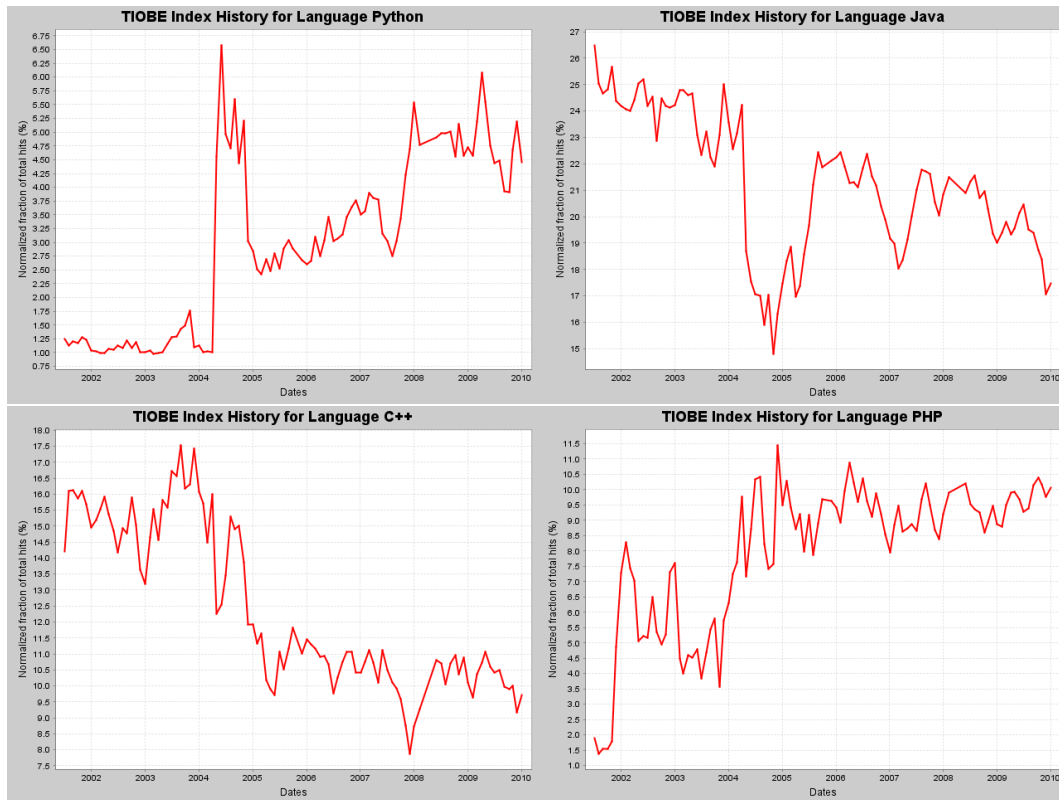


FIG. 1.1. Popularité des différents langages

- ▷ (optionnellement) multi-threadé¹
- ▷ orienté-objet → réutilisable
 - ▶ héritage multiple
 - ▶ surcharge des opérateurs
 - ▶ toutes les méthodes sont virtuelles
- ▷ système d'exceptions → gestion des erreurs simplifiée
- ▷ dynamique → la fonction `eval()` peut évaluer des chaînes de caractères représentant des expressions ou des instructions *Python*
- ▷ typage dynamique → tout objet possède un type bien défini à l'exécution, qui n'a pas besoin d'être déclaré à l'avance.
- ▷ orthogonal → un petit nombre de concepts suffit à engendrer des constructions très riches
- ▷ réflexif → supporte la métaprogrammation, (`setattr()`, `getattr()`, `delattr()`).
- ▷ introspectif → `debugger` ou le `profiler`, sont implantés en *Python* lui-même
- ▷ L'utilisation de la fonction `property()` permet de simuler les variables privées.
- ▷ Facile à utiliser et apprendre, intuitif → pédagogie

¹Wikipedia : un thread ou processus léger, également appelé fil d'exécution (autres appellations connues : unité de traitement, unité d'exécution, fil d'instruction, processus allégé), est similaire à un processus car tous deux représentent l'exécution d'un ensemble d'instructions du langage machine d'un processeur. Du point de vue de l'utilisateur, ces exécutions semblent se dérouler en parallèle. Toutefois, là où chaque processus possède sa propre mémoire virtuelle, les threads d'un même processus se partagent sa mémoire virtuelle. Par contre, tous les threads possèdent leur propre pile d'appel.

- ▷ Langage interprété (et compilé) → développements rapides

1.2. *Python* pourquoi faire ?

- ▷ Calculatrice vectorielle évoluée
- ▷ Traitements de fichiers texte
- ▷ scripts, ou commandes `unix` pour traitements de fichiers par lot par exemple.
- ▷ Langage "glue" pour enchaîner les traitements par différents programmes. Par exemple un mailleur produit un maillage, repris par un code éléments finis pour simulation numérique, dont les résultats sont à leur tour exploités pour visualisation.
- ▷ Ergonomie, interface homme machine : la plupart des bibliothèques de "widgets" (`qt`, `wxwidget`, `tk`, ...) sont interfacées pour *Python*.

1.3. Exécution d'un programme *Python*. Trois mode d'exécution :

- ▷ en ligne de commande
- ▷ par fichier module, ou scripts *Python*
- ▷ par script *Unix*

1.3.1. *Exécution en ligne de commande.* lancer la commande *Python* en mode interactif (ou calculatrice de luxe), depuis une console :

```
$ python
>>>
```

le prompt de *Python* est >>> ou bien ... lorsque *Python* attend une suite.

ipython propose quant à lui

```
In [n] :
```

pour les entrées et

```
Out [n+1] :
```

pour les sorties, n étant le numéro de l'instruction.

pour quitter *Python* : *ctl-d*

```
>>> import sys #importez le module sys :
>>> sys.argv #consultez le contenu de la variable sys.argv:
```

Les lignes de continuation sont nécessaires lorsqu'on saisit une construction sur plusieurs lignes. Comme exemple, voici une instruction *if* :

```
>>> le_monde_est_plat = 1
>>> if le_monde_est_plat:
...     print "Gaffe à pas tomber par dessus bord!"
...
```

1.3.2. *Exécution de fichiers de modules.* On peut sauvegarder un programme exécutable dans un fichier *module* *toto.py*,

```
import sys
print "Les arguments sont :",sys.argv
```

puis exécuter ce programme en invoquant *Python*, suivi du nom du fichier et de paramètres :

```
puiseux@iplmap109:~$ python toto.py -i escalade -o Ossau
['toto.py', '-i', 'escalade', '-o', 'Ossau']
```

1.3.3. *Exécution de scripts Unix.* dans ce cas, le programme exécutable est placé dans un fichier script Unix, c'est à dire un fichier dont la première ligne indique quel interpréteur doit être utilisé. Ici, l'interpréteur est *Python*, bien sur. Par exemple, le script Unix *Ossau.py*

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
import sys
print u"Lez arguments :",sys.argv
```

sera rendu exécutable, puis exécuté directement :

```
puiseux@iplmap109 :~$ chmod +x Ossau.py
puiseux@iplmap109 :~$ ./Ossau.py -i SudEst -o TD
['./Ossau.py', '-i', 'SudEst', '-o', 'TD']
```

1.3.4. Les éditeurs et environnements Python.

▷ Ligne de commande `python`

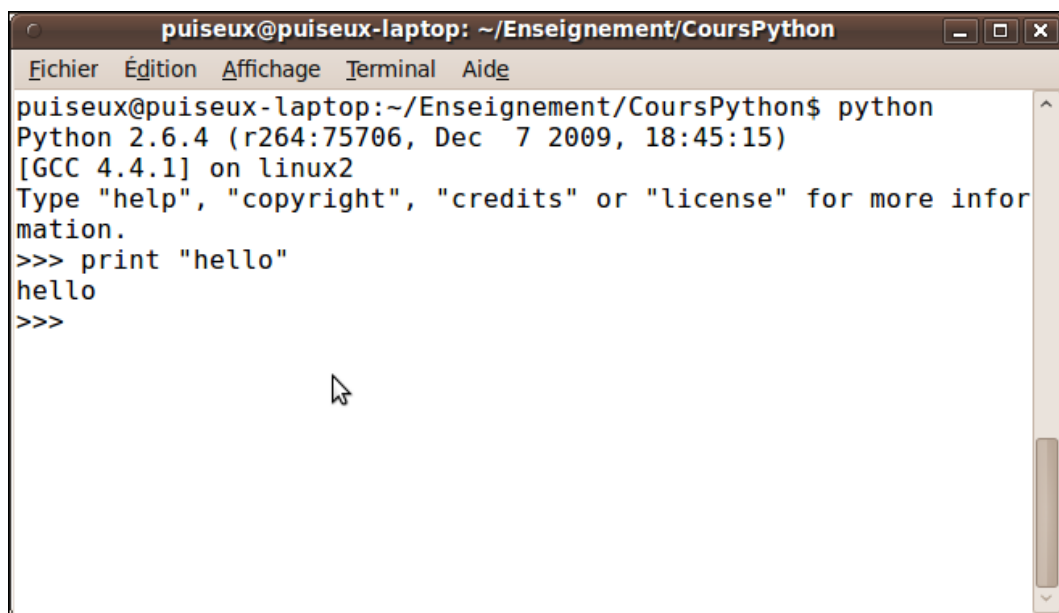
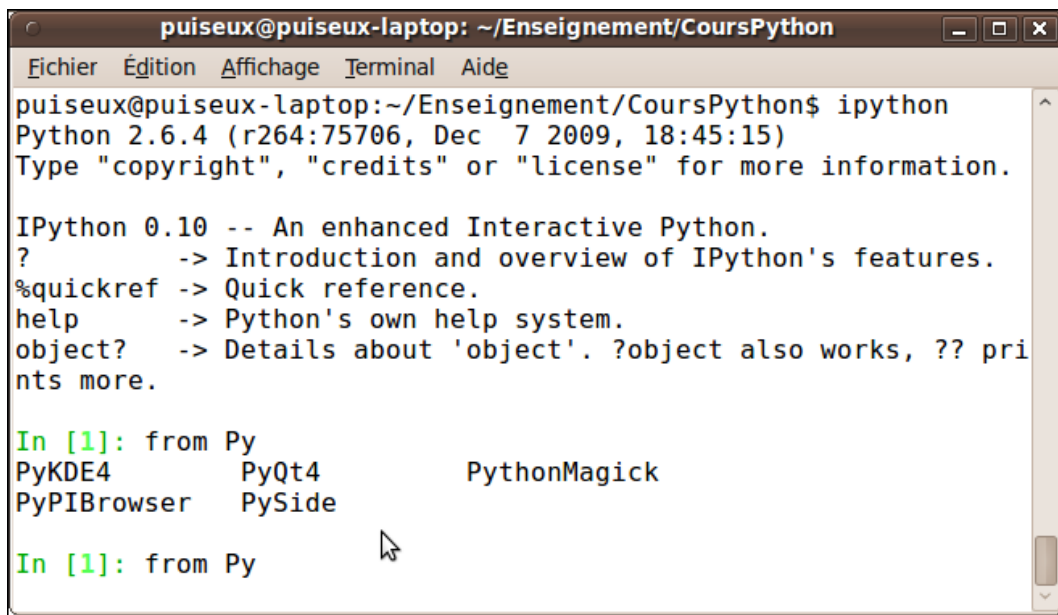


FIG. 1.2. Python en ligne de commande

▷ `ipython` est une interface à l'interpréteur *Python* offrant de nombreuses facilités dont une complétion automatique efficace.



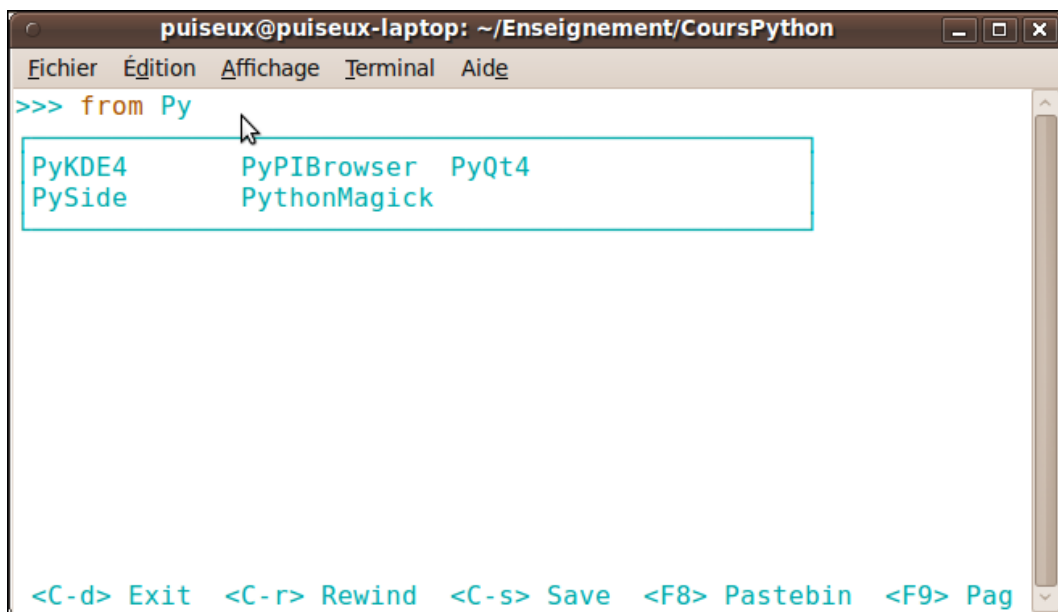
```
puiseux@puiseux-laptop: ~/Enseignement/CoursPython
Fichier  Édition  Affichage  Terminal  Aide
puiseux@puiseux-laptop:~/Enseignement/CoursPython$ ipython
Python 2.6.4 (r264:75706, Dec  7 2009, 18:45:15)
Type "copyright", "credits" or "license" for more information.

IPython 0.10 -- An enhanced Interactive Python.
?      -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help    -> Python's own help system.
object? -> Details about 'object'. ?object also works, ?? prints more.

In [1]: from Py
PyKDE4      PyQt4      PythonMagick
PyPIBrowser PySide
In [1]: from Py
```

FIG. 1.3. ipython

▷ `bpython` également



```
puiseux@puiseux-laptop: ~/Enseignement/CoursPython
Fichier  Édition  Affichage  Terminal  Aide
>>> from Py
PyKDE4      PyPIBrowser  PyQt4
PySide      PythonMagick

<C-d> Exit  <C-r> Rewind  <C-s> Save  <F8> Pastebin  <F9> Pag
```

FIG. 1.4. bpython

▷ `idle` est un l'environnement complet de programmation *Python*, écrit en *Python* `tkinter` par Van Rossum lui-même

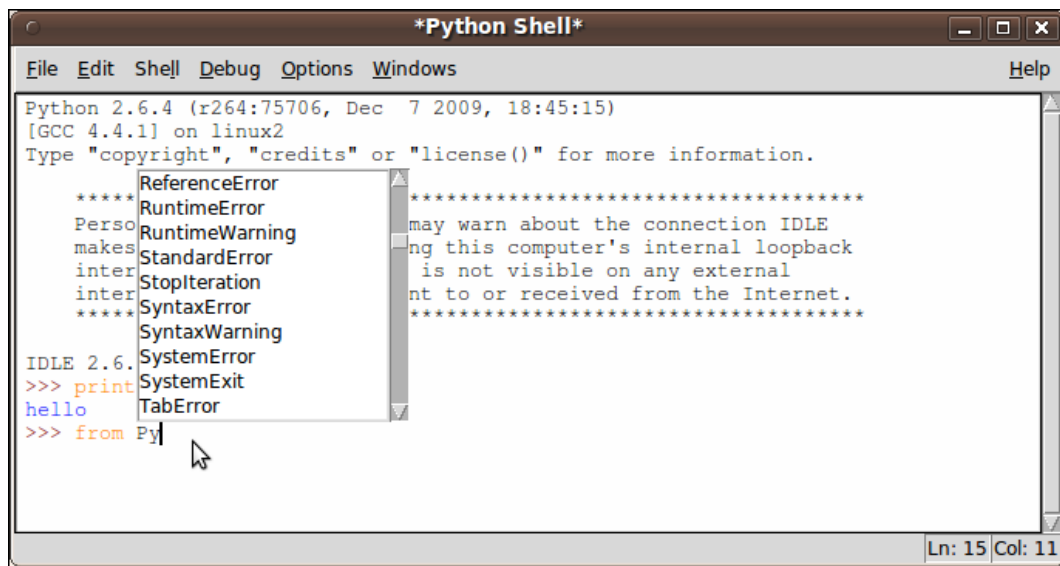


FIG. 1.5. idle

▷ `eclipse+PyDev` le plus abouti

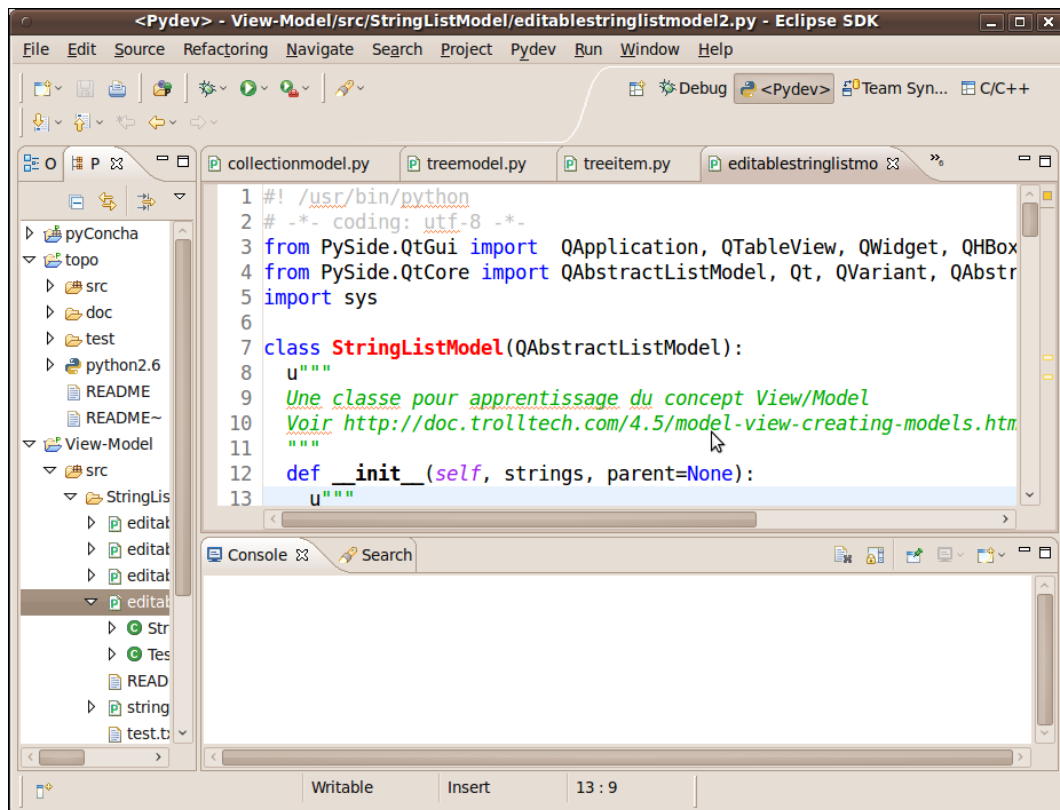


FIG. 1.6. Pydev

- ▷ [Eric](#)
- ▷ etc...

1.4. **Où trouver la documentation ?** La documentation *Python* est très riche et plutôt bien organisée.

- (1) Le document de base est <http://docs.python.org> et surtout
 - (a) <http://docs.python.org/tutorial>
 - (b) <http://docs.python.org/library/index.html>
- (2) Une bonne entrée en matière pour comprendre les concepts : <http://diveintopython.org/>
- (3) Pour un résumé très complet :
 - (a) <http://rgruet.free.fr/#QuickRef>
 - (b) <http://www.python.org/doc/QuickRef.html>

1.5. Exercices.

- (1) Trouver dans la documentation la fonction `input()` et testez cette fonction.
- (2) Que signifie [PEP](#) pour un programmeur *Python* averti ?
- (3) On suppose que vous avez écrit une classe `MonObjet`. Quelle méthode de cette classe devez-vous définir pour pouvoir additionner (c'est à dire utiliser l'opérateur d'addition) deux instances de `MonObjet` ainsi :

```
a = MonObjet(1)+MonObjet(2)
```

2. *Python* COMME CALCULATRICE

Ouvrez une session *Python* puis testez, jouez avec les instructions suivantes :

2.1. Commentaires.

```
# voici le premier commentaire
SPAM = 1 # et voici le deuxième commentaire
        # ... et maintenant un troisième!
STRING = "# Ceci nest pas un commentaire."
```

2.2. Nombres, opérations.

```
>>> 2+2
>>> (50-5*6)/4

>>> # La division des entiers retourne l'entier immédiatement inférieur
>>> 7/3 #Attention
>>> 7/-3
>>> 7%4

>>> # Le signe =
>>> largeur = 20
>>> hauteur = 5*9
>>> largeur * hauteur
>>> x = y = z = 0 # Mettre à zéro x, y et z
>>> x,y,z = 0,0,0

>>> # Nombres à virgule flottante, conversions
>>> 3 * 3.75 / 1.5
>>> 7.0 / 2
```

2.3. Complexes.

```
>>> 1j * 1J
>>> 1j * complex(0,1)
>>> 3+1j*3
>>> (3+1j)*3
>>> (1+2j)/(1+1j)
>>> a = 1.5+0.5j
>>> a.real
>>> a.imag
>>> abs(a)
```

2.4. La variable `_`.

```
>>> tva = 12.5 / 100
>>> prix = 100.50
>>> prix * tva
>>> prix + _
```

2.5. Chaînes de caractères.

- ▷ simples quotes (apostrophes) ou doubles quotes (guillemets) ou triples guillemets :

```
>>> 'spam eggs '
>>> 'n\'est-ce pas '
>>> "n'est-ce pas"
>>> '"Oui," dit-il.'
>>> "\'Oui,\" dit-il."
>>> '"N\'est-ce pas," repondit-elle.'
>>> u'Pour une chaîne contenant des caractères non ascii, \
... préférez le type unicode'
>>> salut = u"""Ceci est une chaîne unicode plutot longue contenant
plusieurs
lignes de texte exactement
    Notez que les blancs au début de la ligne et les sauts de ligne
sont significatifs."""
>>> print salut
```

- ▷ Les chaînes peuvent être concaténées (accollées) avec l'opérateur `+`, et répétées avec `*` :

```
>>> word = 'Help' + 'A'
>>> '<' + word*5 + '>'
```

- ▷ Pas de type caractère (`char`) : un caractère est simplement une chaîne de taille un.

```
>>> type('a')
>>> type("aaa")
```

- ▷ Les chaînes peuvent être décomposées (indexées) le premier caractère d'une chaîne est en position 0.

```
>>> mot = 'HelpA'
>>> mot[4]
>>> mot[0:2]
>>> mot[2:4]
```

- ▷ Valeurs d'index par défaut

```
>>> mot[:2] # Les deux premiers caractères
>>> mot[2:] # Tout sauf les deux premiers caractères
```

Voici un invariant utile des opérations de découpage : `s[:i] + s[i:] == s`

```
>>> mot[:2] + mot[2:]
>>> mot[:3] + mot[3:]
```

▷ Gestion des indices de découpage erronés :

```
>>> mot[1:100]
>>> mot[10:]
>>> mot[2:1]
```

▷ Les indices peuvent être des nombres négatifs, pour compter à partir de la droite.

```
>>> mot[-2] # Lavant dernier caractère
>>> mot[-2:] # Les deux derniers caractères
>>> mot[:-2] # Tout sauf les deux derniers caractères
```

▷ Que retourne la fonction intégrée `len()` ?

```
>>> len(' anticonstitutionnellement ')
```

2.6. Listes.

▷ Une **liste** de valeurs (éléments) entre crochets et séparés par des virgules. Les éléments d'une liste n'ont pas nécessairement le même type.

```
>>> a = [ 'spam', 'eggs', 100, 1234 ]
>>> a
```

▷ Comme les indices des chaînes, les indices des listes **commencent à 0**, et les listes peuvent être découpées, concaténées, et ainsi de suite :

```
>>> a[0]
>>> a[3]
>>> a[-2]
>>> a[1:-1]
>>> a[:2] + [ 'bacon', 2*2 ]
>>> 3*a[:3] + [ 'Boe! ']
```

▷ A la différence des chaînes, qui sont non-modifiables, il est possible de changer les éléments individuels d'une liste

```
>>> a[2] = ''
>>> a
```

▷ l'affectation dans des tranches est aussi possible, et cela peut même changer la taille de la liste :

```
>>> a[0:2] = [1, 12]
>>> a[0:2] = []
>>> a[1:1] = ['bletch', 'xyzzzy']
>>> a[:0] = a # Insère (une copie de) soi-même au début
```

▷ La fonction intégrée `len()` s'applique aussi aux listes :

```
>>> len(a)
```

▷ Il est possible d'emboîter des listes (créer des listes contenant d'autres listes), par exemple :

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
>>> p[1]
>>> p[1][0]
>>> p[1].append('extra')
>>> p
>>> q
```

Notez que dans l'exemple précédent, `p[1]` et `q` se réfèrent réellement au même objet ! Nous reviendrons plus tard sur la sémantique des objets.

2.7. Dictionnaires. fonctionne par couple (clé :valeur)

Voici un petit exemple utilisant un dictionnaire :

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
>>> tel['jack']
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
>>> tel.keys()
>>> tel.values()
>>> tel.items()
>>> tel.has_key('guido')
>>> tel.pop('guido')
```

2.8. Premiers pas vers la programmation. Bien sûr, nous pouvons utiliser *Python* pour des tâches plus compliquées que d'ajouter deux et deux. Par exemple, nous pouvons écrire une sous-séquence de la suite de Fibonacci de la façon suivante :

```
>>> # Suite de Fibonacci
... # La somme de deux éléments définit le suivant
... a, b = 0, 1
>>> while b < 10:
...     print b
```

```
... a, b = b, a+b
```

Cet exemple introduit plusieurs fonctionnalités nouvelles.

- ▷ La première ligne contient une **affectation multiple** : les variables `a` et `b` prennent simultanément les nouvelles valeurs 0 et 1.
- ▷ Les expressions en partie droite sont d'abord toutes évaluées avant toute affectation.
- ▷ La boucle **`while`** s'exécute tant que la condition (ici : `b < 10`) reste vraie. En *Python*,
 - ▶ toute valeur entière différente de zéro est vraie ;
 - ▶ zéro est faux.
 - ▶ N'importe quoi avec une longueur différente de zéro est vrai,
 - ▶ les séquences vides correspondent à faux.
 - ▶ Le test utilisé dans l'exemple est une simple comparaison. Les opérateurs de comparaison standard sont : `<`, `>`, `==`, `<=`, `>=` et `!=`.
- ▷ Le corps de la boucle est **INDENTÉ** : l'indentation est le moyen par lequel *Python* regroupe les instructions.
- ▷ l'instruction **`print`** écrit la valeur de la ou des expressions qui lui sont données. Elle accepte plusieurs expressions et chaînes. Une **virgule finale** empêche le retour chariot après l'affichage :

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print b,
...     a, b = b, a+b
... 
```

2.8.1. *Une première fonction.* On peut faire de la suite d'instructions précédente une fonction qui renvoie le n -ème terme de la suite :

```
>>> def Fibonacci(n):
...     a,b = 0,1
...     for i in range(n) : a,b = b,a+b
...     return b
...
>>> [Fibonacci(i) for i in range(8)]
```

2.9. Exercices.

Exercice 1. Utiliser une boucle **`while`**, la fonction **`type()`** pour déterminer la valeur, M , du plus grand entier, en *Python*. Montrer que $\exists n \in \mathbb{N} : M + 1 = 2^n$. Calculer M^2

Exercice 2. Quel est le plus petit réel strictement positif de la forme 2^{-n} , $n \in \mathbb{N}$?

Exercice 3. Quelle différence y a-t-il entre les fonctions **`input`** et **`raw_input`** ? On pourra tester les instructions suivantes :

```
>>> x = input()
3.14
>>> type(x)
>>> x = raw_input()
3.14
>>> type(x)
>>> x = input()
u'Demain il fait beau, je vais à l'Ossau'
>>> x
```

3. PLUS DE DÉTAILS SUR...

Ce chapitre décrit avec plus de détail quelques éléments que vous avez déjà étudié, et ajoute aussi quelques nouveautés.

3.1. ... les chaînes de caractères.

3.1.1. *Unicode et str*. En *Python3.x*, le traitement des chaînes de caractères évolue vers plus de facilité :

- ▷ Le type `unicode` de *Python2.x* devient le type `str` de *Python3* (standard)
- ▷ Le type `str` de *Python2* devient le type `byte` de *Python3* (obsolète)

- (1) *Python 2.x* : dès que l'on doit manipuler des chaînes de caractères contenant les caractères *non-ASCII*, il est conseillé d'utiliser le type `unicode`, obtenu en faisant précéder la chaîne par la lettre `u`.

```
>>> u=u 'é€ï'
>>> u
>>> print u, type(u)
>>> v= 'ebèç'
>>> v
>>> print v, type(v)
>>> u+v      #ouuups !
>>> v=u 'ebèç'
>>> u+v      #yes !
```

- (2) En *Python3* on écrira plus simplement :

```
>>> u=u 'é€ï'
>>> u, print(u), type(u)
>>> v= 'ebèç'
>>> type(v), print(v)
>>> u+v      #ok !
>>> u=b 'abc'   # le type byte de Python3 est le type str de Python2.x
>>> v=b 'é€ï'   # ouuups non ascii!
```

3.1.2. Méthode propres aux chaînes de caractères.

Exercice 4. Que fait l'instruction

```
F=open('toto.csv')
[[[g.strip() for g in f.strip().split(';')] for f in e.strip().split('\n')] for e in F.readlines()]
F.close()
```

3.2. ... les entrées et sorties.

3.2.1. *Un formatage personnalisé, %, str() et repr()*. Jusqu'ici nous avons rencontré deux manières d'afficher des valeurs : les instructions d'expression et l'instruction `print`.²

Il y a deux manières de formater vos sorties ;

²(Une troisième manière est d'utiliser la méthode `write()` des objets fichier ; le fichier de sortie standard peut être référencé par `sys.stdout`. Voyez le manuel Library Reference pour plus d'informations.)

- (1) Manipulez vous-même les chaînes de caractères. Le module standard `string` contient quelques opérations utiles pour remplir des chaînes de caractères à une largeur de colonne donnée. Pour convertir n'importe quelle valeur en chaîne de caractères : passez-la à la fonction `repr()`, ou `str()`
- (2) La deuxième manière est d'utiliser l'opérateur de formatage `%` :

`format%variables`

comme pour la fonction `sprintf()` du langage C. Cette forme retourne une chaîne de caractères résultant de cette opération de formatage.

La fonction `str()` pour renvoie des représentations faciles à lire par les humains, alors que `repr()` est renvoie des représentations lisibles par l'interpréteur. De nombreuses valeurs, comme les nombres ou les structures comme les listes et les dictionnaires, ont la même représentation dans les deux fonctions. Les chaînes et les nombres à virgule flottante, ont deux représentations distinctes.

Quelques exemples :

```
>>> s = 'Salut, tout le monde.'
>>> str(s), repr(s)
>>> s
>>> str(0.1)

>>> x,y = 3.25, 150
>>> print 'x=%f tandis que y=%f'%(x,y)
>>> print 'x=' + x + ' tandis que y=' + y
```

Convertir une chaîne ajoute des quotes de chaîne et des antislash :

```
>>> saluts = 'salut'
>>> print saluts
```

Exercice 5. écrire une table des carrés et des cubes présentée comme suit :

0	0	0
1	1	1
2	4	8
3	9	27
4	16	64
5	25	125
6	36	216
7	49	343
8	64	512
9	81	729
10	100	1000

(Notez qu'un espace entre chaque colonne a été ajouté à cause de la façon dont `print` fonctionne : elle ajoute toujours des espaces entre ses arguments.)

3.2.2. *Le module `string`*. La fonction `string.ljust()`, justifie à droite une chaîne de caractères dans un champ d'une largeur donnée en la complétant avec des espaces du côté gauche.

```
>>> a = 'eee'
>>> string.ljust(a, 2)
'eee'
>>> string.ljust(a, 20)
'eee'
>>> string.ljust(a, 20, '*')
'eee*****'
```

Il y a les fonctions semblables `string.ljust()` et `string.center()`. Ces fonctions n'écrivent rien, elles renvoient juste une nouvelle chaîne de caractères. Si la chaîne de caractères d'entrée est trop longue, elles ne la tronquent pas, mais la renvoient sans changement ; cela gâchera votre présentation de colonne mais c'est habituellement mieux que l'alternative, qui serait de tricher au sujet d'une valeur. (Si vous voulez vraiment la troncature vous pouvez toujours ajouter une opération de découpage, comme `string.ljust(x, n)[0 :n]`.)

Il y a une autre fonction, `string.zfill()`, qui complète une chaîne de caractères numérique du côté gauche avec des zéros. Elle sait gérer les signes positifs et négatifs :

```
>>> import string
>>> string.zfill(12, 5)
>>> string.zfill(-3.14, 7)
>>> string.zfill(3.14159265359, 5)
```

L'utilisation de l'opérateur `%` ressemble à ceci :

```
>>> import math
>>> print 'La valeur de PI est approximativement %5.3f.' % math.pi
La valeur de PI est approximativement 3.142.
```

S'il y a plus d'un descripteur de format dans la chaîne de caractères, vous devez passer un tuple comme opérande de droite, comme dans cet exemple :

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> for nom, telephone in table.items():
...     print '%-10s ==> %10d' % (nom, telephone)
...
```

La plupart des formats fonctionnent exactement comme en C et exigent que vous passiez le type approprié.

3.2.3. *Lire et écrire des fichiers*. `open()` renvoie un objet de type fichier, et est utilisée plus généralement avec deux arguments : `open(nomfichier, mode)`.

```
>>> f=open('/tmp/fichiertravail', w)
>>> print f
```

Le premier argument est une chaîne de caractères contenant le nom du fichier. Le deuxième argument est une autre chaîne de caractères contenant quelques caractères décrivant la manière d'utiliser le fichier. `mode` vaut

- ▷ `r` quand le fichier doit être seulement lu,
- ▷ `w` pour seulement écrit (un fichier déjà existant avec le même nom sera effacé),
- ▷ et `a` ouvre le fichier en ajout ; les données écrites dans le fichier seront automatiquement ajoutées à la fin.
- ▷ `r+` ouvre le fichier pour la lecture et l'écriture.

L'argument `mode` est facultatif ; `r` sera pris par défaut s'il est omis.

3.2.4. *Méthodes des objets fichiers.* On supposera qu'un objet fichier appelé `f` a déjà été créé.

- (1) Pour lire le contenu d'un fichier, appeler `f.read(taille)`, qui lit une certaine quantité de données et les retourne en tant que chaîne de caractères. Quand `taille` est omis ou négatif, le contenu entier du fichier est lu et retourné

```
>>> f.read()
```

- (2) `f.readline()` lit une seule ligne à partir du fichier ; un caractère de fin de ligne (`\n`) est laissé à l'extrémité de la chaîne de caractères lue

```
>>> f.readline()
Ceci est la première ligne du fichier.\n
>>> f.readline()
Deuxième ligne du fichier\n
>>> f.readline()
```

- (3) `f.readlines()` renvoie une liste contenant *toutes* les lignes de données dans le fichier. La liste retournée est entièrement faite de lignes complètes.

```
>>> f.readlines()
[Ceci est la première ligne du fichier.\n, Deuxième ligne du fichier\n]
```

- (4) Une approche alternative est de boucler sur l'objet fichier.

```
>>> for line in f:
...     print line,
Ceci est la première ligne du fichier.
Deuxième ligne du fichier
```

- (5) `f.write(chaine)` écrit le contenu de chaîne dans le fichier, en retournant `None`.

```
>>> f.write(Voici un testn)
```

Pour écrire quelque chose d'autre qu'une chaîne il est nécessaire de commencer par le convertir en chaîne :

```
>>> value = (the answer, 42)
>>> s = str(value)
>>> f.write(s)
```

- (6) `f.tell()` renvoie un nombre entier donnant la position actuelle, mesurée en octets depuis le début du fichier. Pour changer la position, employez
- (7) `f.seek(decalage, point_depart)`. La position est calculée en ajoutant `decalage` à un point de référence; le point de référence est choisi par l'argument `point_depart`. Une valeur de 0 pour `point_depart` fait démarrer au début du fichier, 1 utilise la position courante du fichier, et 2 utilise la fin de fichier comme point de référence. `point_depart` peut être omis et prend alors 0 pour valeur par défaut comme point de référence.

```
>>> f=open('/tmp/fichiertravail', r+)
>>> f.write(0123456789abcdef)
>>> f.seek(5) # Saute jusqu'au 6ème octet dans le fichier
>>> f.read(1)
5
>>> f.seek(-3, 2) # Saute jusqu'au 3ème octet avant la fin
>>> f.read(1)
d
```

- (8) Appeler `f.close()` pour le fermer.

```
>>> f.close()
>>> f.read()
```

Les objets fichier ont quelques méthodes supplémentaires, telles que `isatty()` et `truncate()` qui sont moins fréquemment utilisées; consultez la Library Reference pour un guide complet des objets fichier.

3.2.5. *Le module `pickle`*. Pour lire et écrire les nombres : la méthode `read()` renvoie une chaîne de caractères, qui devra être convertie avec `int()`, `float()`...

Pour sauvegarder des types de données complexes (listes, dictionnaires, ...) : le module standard appelé `pickle` convertit en chaîne de caractères presque n'importe quel objet *Python* : ce processus s'appelle *pickling*.

Reconstruire l'objet à partir de sa représentation en chaîne de caractères s'appelle *unpickling*.

```
>>> f=open('pik.txt', 'w')
>>> x=[1,2,3, 'hello', 1j]
>>> pickle.dump(x, f)
```

Pour “unpickler” l'objet, si `f` est un objet fichier ouvert en lecture :

```
>>> x = pickle.load(f)
```

3.3. ... les listes. Le type de données liste possède d'autres méthodes. Voici toutes les méthodes des objets listes :

`append(x)` : équivalent à `a.insert(len(a), x)`
`extend(L)` : rallonge la liste en ajoutant à la fin tous les éléments de la liste donnée ; équivalent à `a[len(a):] = L`.
`insert(i, x)` : insère un élément à une position donnée. Le premier argument est l'indice de l'élément avant lequel il faut insérer, donc `a.insert(0, x)` insère au début de la liste, et `a.insert(len(a), x)` est équivalent à `a.append(x)`.
`remove(x)` : enlève le premier élément de la liste dont la valeur est `x`. Il y a erreur si cet élément n'existe pas.
`pop([i])` : enlève l'élément présent à la position donnée dans la liste, et le renvoie. Si aucun indice n'est spécifié, `a.pop()` renvoie le dernier élément de la liste. L'élément est aussi supprimé de la liste.
`index(x)` : retourne l'indice dans la liste du premier élément dont la valeur est `x`. Il y a erreur si cet élément n'existe pas.
`count(x)` : renvoie le nombre de fois que `x` apparaît dans la liste.
`sort()` : trie les éléments à l'intérieur de la liste.
`reverse()` : renverse l'ordre des éléments à l'intérieur de la liste.

Exemple. Un exemple qui utilise toutes les méthodes des listes :

```
>>> a = [66.6, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.6), a.count(x)
>>> a.insert(2, -1)
>>> a.append(333)
>>> a.index(333)
>>> a.remove(333)
>>> a.pop(1)
>>> a.reverse()
>>> a.sort()
```

3.3.1. *Utiliser les listes comme des piles.* Les méthodes des listes rendent très facile l'utilisation d'une liste comme une pile, où le dernier élément ajouté est le premier élément récupéré (LIFO, "last-in, first-out"). Pour ajouter un élément au sommet de la pile, utilisez la méthode `append()`. Pour récupérer un élément du sommet de la pile, utilisez `pop()` sans indice explicite.

Example.

```
>>> pile = [3, 4, 5]
>>> pile.append(6)
>>> pile.append(7)
>>> pile.pop()
>>> pile.pop()
>>> pile.pop()
>>> pile
```

3.3.2. *Utiliser les listes comme des files.* Vous pouvez aussi utiliser facilement une liste comme une file, où le premier élément ajouté est le premier élément retiré (FIFO, "first-in, first-out"). Pour ajouter un élément à la fin de la file, utiliser `append()`. Pour récupérer un élément du devant de la file, utilisez `pop(0)` avec 0 pour indice.

Example.

```
>>> file = ["Eric", "John", "Michael"]
>>> file.append("Terry") # Terry arrive
>>> file.append("Graham") # Graham arrive
>>> file.pop(0)
>>> file.pop(0)
>>> file
```

3.3.3. *List Comprehensions.* Les list comprehensions fournissent une façon concise de créer des listes.

```
>>> fruitfrais = ['banane', 'myrtille', 'fruit de la passion']
>>> [projectile.strip() for projectile in fruitfrais]
```

```
>>> vec, vec1 = [2, 4, 6], [8, 9, 12]
>>> [3*x for x in vec]
>>> [3*x for x in vec if x > 3]
>>> [v[0]*v[1] for v in zip(vec, vec1)]
```

3.3.4. *l'instruction del.* Il y a un moyen d'enlever un élément d'une liste en ayant son indice au lieu de sa valeur : l'instruction `del`. Cela peut aussi être utilisé pour enlever des tranches dans une liste (ce que l'on a fait précédemment par remplacement de la tranche par une liste vide). Par exemple :

```
>>> a=[-1, 1, 66.6, 333, 333, 1234.5]
>>> del a[0]
>>> del a[2:4]
```

`del` peut aussi être utilisé pour supprimer des variables complètes :

```
>>> del a
```

3.4. ... les tuples et séquences. Un tuple consiste en un ensemble de valeurs séparées par des virgules, par exemple :

```
>>> t = 12345, 54321, 'salut !'
>>> t[0]
>>> t
```

les tuples peuvent être imbriqués :

```
>>> u = t, (1, 2, 3, 4, 5)
>>> u
```

Les tuples sont toujours entre parenthèses.

l'instruction `t = 12345, 54321, 'salut !'` est un exemple d'**emballage** en tuple (tuple packing) : les valeurs 12345, 54321 et 'salut !' sont emballées ensemble dans un tuple. l'opération inverse (**déballage** de tuple -tuple unpacking-) est aussi possible :

```
>>> x, y, z = t
```

3.5. ... les ensembles. Python comporte également un type de données pour représenter des ensembles. Un set est une collection (non rangée) sans éléments dupliqués. Les emplois basiques sont le test d'appartenance et l'élimination des entrées dupliquées. Les objets ensembles supportent les opérations mathématiques comme l'union, l'intersection, la différence et la différence symétrique. Voici une démonstration succincte :

```
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a
>>> a - b
>>> a | b
>>> a & b
>>> a ^ b
```

3.6. ... les dictionnaires. Un autre type de données intégré à *Python* est le dictionnaire. Les dictionnaires sont indexés par des clés, qui peuvent être de n'importe quel type non-modifiable.

Les chaînes et les nombres peuvent toujours être des clés.

Il est préférable de considérer les dictionnaires comme des ensembles non ordonnés de couples (clé : valeur), avec la contrainte que les clés soient uniques (à l'intérieur d'un même dictionnaire).

Un couple d'accolades crée un dictionnaire vide : `{}`.

Placer une liste de couples clé : valeur séparés par des virgules à l'intérieur des accolades ajoute les couples initiaux (clé : valeur) au dictionnaire ; c'est aussi de cette façon que les dictionnaires sont affichés. Les opérations principales sur un dictionnaire sont :

- ▷ le stockage d'une valeur à l'aide d'une certaine clé et
- ▷ l'extraction de la valeur en donnant la clé.

▷ Il est aussi possible de détruire des couples (clé : valeur) avec `del`.

La méthode `keys()` d'un objet de type dictionnaire retourne une liste de toutes les clés utilisées dans le dictionnaire, dans un ordre quelconque (si vous voulez qu'elle soit triée, appliquez juste la méthode `sort()` à la liste des clés). Pour savoir si une clé particulière est dans le dictionnaire, utilisez la méthode `has_key()` du dictionnaire.

Voici un petit exemple utilisant un dictionnaire :

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
>>> tel['jack']
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
>>> tel.keys()
>>> tel.values()
>>> tel.items()
>>> tel.has_key('guido')
>>> tel.pop['guido']
>>> for nom, num in tel.iteritems() : print nom, num
```

Le constructeur `dict()` construit des dictionnaires directement à partir de listes de paires clé-valeur rangées comme des n-uplets. Lorsque les paires forment un motif, les list list comprehensions peuvent spécifier de manière compacte la liste de clés-valeurs.

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
>>> dict([(x, x**2) for x in (2, 4, 6)]) # use a list comprehension
```

Plus loin dans ce tutoriel nous étudierons les “expressions générateurs” qui sont l'outil idéal pour fournir des paires clé-valeur au constructeur `dict()`. Lorsque les clés sont de simples chaînes il est parfois plus simple de spécifier les paires en utilisant des arguments à mot-clé :

```
>>> dict(sape=4139, guido=4127, jack=4098)
```

3.7. ... les techniques de boucles.

▷ Lorsqu'on boucle sur un dictionnaire, les clés et les valeurs correspondantes peuvent être obtenues en même temps en utilisant la méthode `iteritems()`

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.iteritems() : print k, v
gallahad the pure
robin the brave
```

▷ Lorsqu'on boucle sur une séquence, l'indice donnant la position et la valeur correspondante peuvent être obtenus en même temps en utilisant la fonction `enumerate()`.


```
>>> for i, v in enumerate(['tic', 'tac', 'toe']): print i, v
```

est préférable à :

```
>>> a = ['tic', 'tac', 'toe']
>>> for i in range(len(a)) : print i, a[i]
```

- ▷ Pour boucler sur deux séquences, ou plus, en même temps, les éléments peuvent être appariés avec la fonction `zip()`.

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print What is your %s? It is %s. % (q, a)
...
```

- ▷ Pour boucler à l'envers sur une séquence, spécifiez d'abord la séquence à l'endroit, ensuite appelez la fonction `reversed()`.

```
>>> for i in reversed(xrange(1,10,2)): print i
```

- ▷ Pour boucler sur une séquence comme si elle était triée, utilisez la fonction `sorted()` qui retourne une liste nouvelle triée tout en laissant la source inchangée.

```
>>> basket = [apple, orange, apple, pear, orange, banana]
>>> for f in sorted(set(basket)): print f
```

3.8. ... les conditions. Les conditions utilisées dans les instructions `while` et `if` peuvent contenir d'autres opérateurs en dehors des comparaisons.

Les opérateurs de comparaison `in` et `not in` vérifient si une valeur apparaît (ou non) dans une séquence.

Les opérateurs `is` et `is not` vérifient si deux objets sont réellement le même objet

```
>>> a=[1,2,3]
>>> b=a
>>> a is b
>>> 1 in a
>>> b=a[:]
>>> b is a
```

Tous les opérateurs de comparaison ont la même priorité, qui est plus faible que celle de tous les opérateurs numériques. Les comparaisons peuvent être enchaînées. Par exemple, `a < b == c` teste si `a` est strictement inférieur à `b` et de plus si `b` est égal à `c`.

Les opérateurs `and` et `or` ont une priorité inférieure à celle des opérateurs de comparaison ; et entre eux, `not` a la plus haute priorité, et `or` la plus faible, de sorte que

$A \text{ and not } B \text{ or } C \iff (A \text{ and } (\text{not } B)) \text{ or } C.$

Les opérateurs `and` et `or` sont dits *court-circuit* : leurs arguments sont évalués de gauche à droite, et l'évaluation s'arrête dès que le résultat est trouvé.

Exemple, si **A** et **C** sont vrais mais que **B** est faux, **A and B and C** n'évalue pas l'expression **C**. Il est possible d'affecter le résultat d'une comparaison ou une autre expression Booléenne à une variable. Par exemple,

```
>>> chaine1, chaine2, chaine3 = , 'Trondheim', 'Hammer Dance'
>>> non_null = chaine1 or chaine2 or chaine3
>>> non_null
```

Notez qu'en *Python*, au contraire du C, les affectations ne peuvent pas être effectuées à l'intérieur des expressions. Les programmeurs C ronchonneront peut-être, mais cela évite une classe de problèmes qu'on rencontre dans les programmes C : écrire = dans une expression alors qu'il fallait ==.

Une instruction commode :

```
>>> a = 1 if condition else 2
```

affecte 1 à **a** si la condition est vraie, 2 sinon

3.9. ... les modules. Un module peut contenir des instructions exécutables aussi bien que des définitions de fonction.

Chaque module a sa propre table de symboles privée.

Vous pouvez accéder aux variables globales d'un module avec la même notation que celle employée pour se référer à ses fonctions, **nommodule.nomelem**. Les noms du module importé sont placés dans la table globale de symboles du module importateur.

Exercice 6. Écrire la fonction fibonacci dans un fichier **fibonacci.py** et testez le instructions

```
>>> from fibo import fib, fib2
>>> fib(500)
>>> import fibo
>>> fibo.fib(20)
```

Il y a une variante pour importer tous les noms qu'un module définit :

```
>>> from fibo import *
>>> fib(500)
```

Cela importe tous les noms excepté ceux qui commencent par un tiret-bas (**_**).

3.9.1. Le chemin de recherche du module. Quand un module nommé **spam** est importé, l'interpréteur recherche un fichier nommé **spam.py**

- ▷ dans le répertoire courant, et puis
- ▷ dans la liste de répertoires indiquée par la variable d'environnement **PYTHONPATH**.
- ▷ dans un chemin d'accès par défaut, dépendant de l'installation ; sur **UNIX**, c'est habituellement **/usr/local/lib/python**.

En fait, les modules sont recherchés dans la liste de répertoires donnée par la variable **sys.path**.

Exercice 7. Consultez la liste des répertoires dans lesquels l'interpréteur recherche les modules. Combien y a-t-il de répertoires *distincts* ?

3.9.2. *Fichiers “compilés” de Python.* Pour accélérer le temps de lancement, si un fichier appelé `spam.pyc` existe dans le répertoire où `spam.py` se trouve, il est supposé contenir une version du module `spam` déjà compilée “en `byte-code`”. L’heure de modification de la version de `spam.py` employée pour créer `spam.pyc` est enregistrée dans `spam.pyc`, et le fichier est ignoré si ceux-ci ne s’accordent pas.

Toutes les fois que `spam.py` est compilé avec succès, une tentative est faite pour écrire la version compilée sur `spam.pyc`.

Le contenu du fichier `spam.pyc` est indépendant de la plate-forme, ainsi un répertoire de module de Python peut être partagé par des machines d’architectures différentes.

3.9.3. *Modules standard.* Python est livré avec une bibliothèque de modules standard, décrite dans un document séparé, *Python Library Reference* (“Library Reference” ci-après).

3.10. ... les fonctions builtins.

▷ la fonction `dir()` :

► renvoie une liste des noms qu’un module définit.

```
>>> import fibo
>>> dir(fibo)
```

► Sans arguments, `dir()` énumère les noms que vous avez définis :

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo, sys
>>> fib = fibo.fib
>>> dir()
```

► Notez qu’elle énumère tous les types de noms : les variables, les modules, les fonctions, etc. `dir()` n’énumère pas les noms des fonctions et des variables intégrées. Si vous en voulez une liste, elles sont définies dans le module standard `__builtin__` :

```
>>> import __builtin__
>>> dir(__builtin__)
```

▷ la fonction `range(min,max)` : renvoie une liste de d’entier compris en min et max

▷ La fonction `type(objet)` : renvoie le type d’une variable

▷ `len(obj)` : renvoie la taille d’un objet

▷ `abs(x)` : valeur absolue ou module pour un complexe

▷ `quit()` :

▷ `locals()` : retourne un dictionnaire représentant la liste des symboles locaux.

▷ `exec(string)`

Permet d’exécuter une instruction Python sous forme de chaîne de caractères. Par exemple :

```
>>> pi = 3.14
>>> todo = 'L=[1,3,pi] '
>>> exec (todo)
>>> L
[1, 3, 3.1400000000000001]
```

▷ `assert(condition)`

Pour déclencher une exception si la condition est fausse

```
>>> import os
>>> filename = 'toto.txt'
>>> assert(os.path.isfile(filename))
```

donne le même résultat que

```
*>>> if not os.path.isfile(filename) : raise AssertionError
```

Exercice 8. Depuis une session *Python*,

- ▷ déterminer le type de l'objet `__builtins__`
- ▷ Affecter à la variable `L` une liste des clés de `__builtins__`
- ▷ que vaut `L[42]` ?
- ▷ Trouver l'aide sur la fonction `sum`

3.11. ... **les paquetages**. Les paquetages sont un moyen de structurer l'espace des noms de modules *Python* en utilisant "les noms de modules pointés".

Par exemple, le nom de module `A.B` désigne un sous-module nommé `B` dans un module nommé `A`.

3.12. ... **Les exceptions**. Lorsqu'un erreur d'exécution survient, une exception est levée. Le programme stoppe, et *Python* affiche la pile d'appels, et l'exception.

Il est possible d'écrire des programmes qui prennent en charge des exceptions spécifiques. Regardez l'exemple suivant, qui interroge l'utilisateur jusqu'à ce qu'un entier valide ait été saisi, mais lui permet d'interrompre le programme en utilisant **Control-C** ou une autre combinaison de touches reconnue par le système d'exploitation (il faut savoir qu'une interruption produite par l'utilisateur est signalée en levant l'exception `KeyboardInterrupt`).

```
while 1:
    try:
        x = int(raw_input(u"Veuillez entrer un nombre : "))
        break
    except ValueError:
        print u"Aïe! Ce n'était pas un nombre valide. Essayez encore..."
```

Fonctionnement :

La clause `try` : les instructions entre les mots-clés `try` et `except` est exécutée.

- ▷ S'il ne se produit pas d'exception, la clause `except` est ignorée, et l'exécution du `try` est terminée.
- ▷ Si une exception se produit, le reste de la clause `try` est ignoré. Puis si son type correspond à l'exception donnée après le mot-clé `except`, la clause `except` est exécutée, puis l'exécution reprend après l'instruction `try`.
- ▷ Si une exception se produit qui ne correspond pas à l'exception donnée dans la clause `except`, elle est renvoyée aux instructions `try` extérieures.

L'instruction `raise` vous permet de lever vous même une exception.

Par exemple :

```
def get_age():
    age = input('Please enter your age: ')
    if age < 0:
        raise ValueError, '%s is not a valid age' % age
    return age
```

L'instruction `raise` prend deux arguments : le type de l'exception et une chaîne de caractère qui décrit l'exception.

`ValueError` est une exception standard.

La liste complète des exceptions :

```
>>> help('exceptions')
```

Les plus courantes :

- ▷ Accéder à une clé non-existante d'un dictionnaire déclenche une exception `KeyError`
- ▷ Chercher une valeur non-existante dans une liste déclenche une exception `ValueError`.
- ▷ Appeler une méthode non-existante déclenche une exception `AttributeError`.
- ▷ Référencer une variable non-existante déclenche une exception `NameError`.
- ▷ Mélanger les types de données sans conversion déclenche une exception `TypeError`.

Exercice 9. Testez les instructions suivantes, quel est le nom de l'exception ?
une division par zéro :

```
>>> 2/0
```

un indice hors tableau :

```
>>> a=[]
>>> a[2]
```

assigner une valeur à un item dans un tuple :

```
>>> tuple=(1,2,3,4)
>>> tuple[3]=45
```

3.13. Exercices.

Exercice 10. Écrire une expression qui vaut `True` si l'entier `n` est pair et `False` dans le cas contraire.

Exercice 11. Transformer la chaîne de caractère 3.14 en un flottant.

Exercice 12. Soit la chaîne de caractères `a='1.0 3.14 7 8.4 0.0'`. Écrire une instruction unique utilisant les fonctions et méthode `float()` et `split()`, qui construit la liste de réels `[1.0, 3.14, 7.0, 8.4, 0.0]` à partir de `a`.

Exercice 13. Soit la chaîne de caractères `c="(1.0, 2.0) (3, 4)n"`. Trouver une suite d'instructions permettant d'en extraire les deux nombres complexes `x=1+2j` et `y=3+4j`. On pourra par exemple :

- (1) Nettoyer `c` de son `'n'` (méthode `strip()`) \rightarrow `c='(1.0, 2.0) (3, 4)'`
- (2) Remplacer `','` par `','` \rightarrow `c='(1.0,2.0) (3,4)'`
- (3) Scinder (`=split()`) la chaîne en deux \rightarrow `c=['(1.0,2.0)', '(3,4)']`
- (4) Supprimer `'('` et `)'` de chaque terme \rightarrow `c=['1.0,2.0', '3,4']`
- (5) Instancier les complexes `1+2j` et `3+4j` à partir des deux sous chaînes `c[0]` et `c[1]`

Exercice 14. Lecture-écriture de fichiers

- ▷ lire le contenu du fichier `humour.txt`, écrire son contenu à l'écran en majuscule.
- ▷ demander à l'utilisateur un nom de fichier, le lire et recopier son contenu, avec fer à droite, dans le fichier `HUMOUR.txt`.
- ▷ Créer une liste de votre choix, picklez-la dans `liste.pic`, unpicklez-la dans une variable `zoe`

Exercice 15. Que fait la séquence d'instructions suivante :

```
>>> def echo(msg): print msg
>>> x=echo
>>> x("ca marche")
```

Exercice 16. Ecrire un script `rilex.py` qui calcule la richesse lexicale d'un texte contenu dans un fichier dont le nom est passé en argument. La richesse lexicale d'un texte est définie comme le quotient entre le nombre de mots différents et le nombre total de mots du texte. Dans cet exercice, on définit la notion de "mot" comme toute séquence de taille supérieure ou égale à quatre, formée exclusivement de caractères alphabétiques (on ne distinguera pas les majuscules des minuscules). Pour l'implémentation, on utilisera la structure native de dictionnaire fournie par le langage *Python*.

Exercice 17.

- (1) Lors de la lecture d'un fichier, le programme lit la chaîne de caractères `s='1;2; 3; 4; 5'`. Quelle instruction unique permet d'en faire la liste d'entiers `l=[1,2,3,4,5]`
- (2) idem, avec `s='1;2; 3; 4; 5;;'`
- (3) Idem, avec `s='1;2; 3; -4; 5;;'` et on ne garde dans `l` que les entiers positifs.
- (4) Ecrire un script qui crée un dictionnaire, `d`, dont les clés et les éléments sont lus sur un fichier `octave`. Les clés sont les noms des variables et les éléments sont les valeurs de ces variables. On testera les instructions programmées sur le fichier `vars.oct`.

Exercise 18. On donne une liste de d'articles et la liste des prix correspondants. Écrire une instruction qui imprime la liste des (prix : article). Par exemple

```
>>> articles = [ 'mousqueton ', 'reverso ', 'crochet ', 'baudrier ', 'chauss  
>>> prix = [5,25,5,80,120,2]
```

produira l'affichage

```
mousqueton : 5€  
reverso : 25€  
crochet : 5€  
baudrier : 80€  
chausson : 120€  
magnesie : 2€
```

Exercise 19.

Quel est le contenu du module `matplotlib`? Dans quel répertoire se trouve ce module?

Exercise 20. En utilisant le mécanisme des exceptions, écrire une fonction `isFile(fichier)` qui renvoie *True* si le fichier existe, *False* sinon

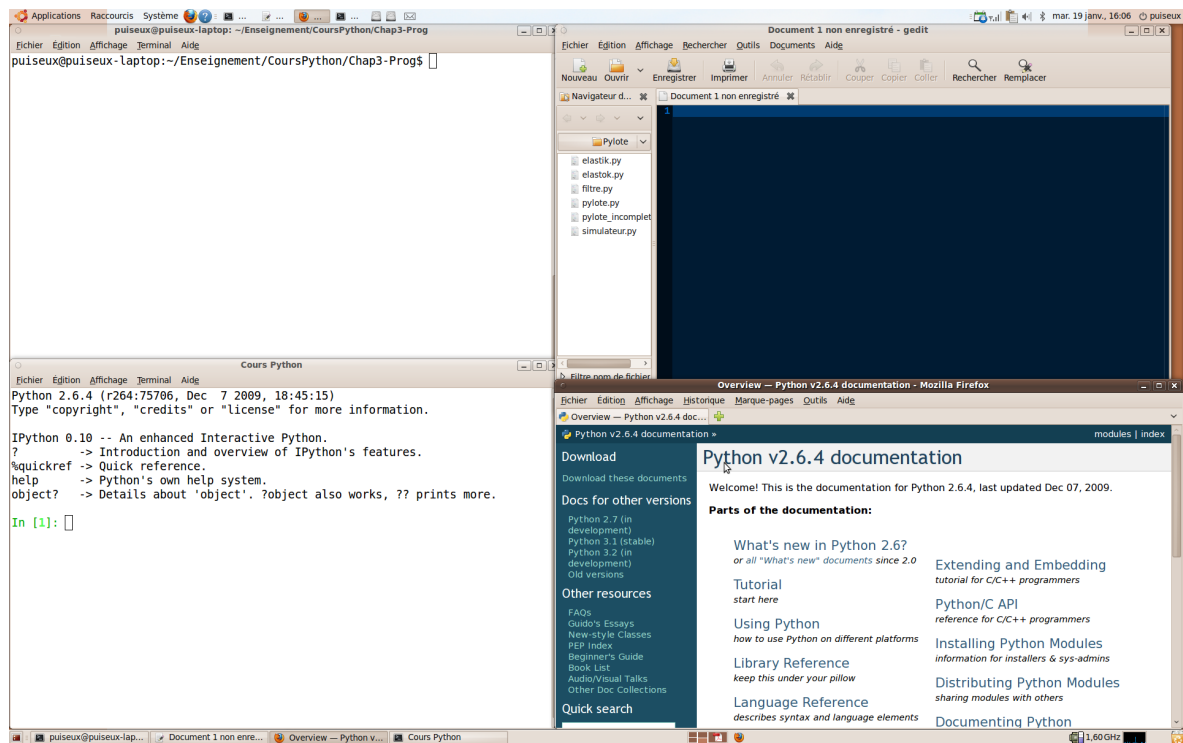


FIG. 4.1. Le bureau d'un développeur Python

4. PROGRAMMATION

Pour programmer en *Python*, nous pouvons utiliser l'environnement de programmation le plus rudimentaire qui soit : la ligne de commande et un éditeur de texte.

Il existe des environnements plus élaborés, comme *idle*, *eric*, *eclipse-PyDev*, *pype*, *pyragua*, *spe*.

Au besoin nous utiliserons *idle*, le plus rustique mais le plus standard de ces environnements.

4.1.1. Mise en route. Dans un terminal :

- ▷ créer un répertoire et s'y déplacer :

```
$ mkdir python-chapitre3 & cd Python-chapitre3
```

- ▷ ouvrir un éditeur de texte et créer un fichier *test.py*

```
$ gedit tests.py &
```

- ▷ ouvrir un nouvel onglet et lancer *Python*

```
$ python
```

- ▷ ouvrir *firefox* et la doc *Python*

4.1.2. Exécuter un script ou une fonction Python. Voir (1.3)

Exercice 21. Ecrire une fonction récursive `hanoi(n, depuis, vers, par)` qui implémente l'algorithme célèbre des tours de Hanoï. voir <http://www.mah-jongg.ch/towerofhanoi/> ou <http://fr.wikipedia.org/>

4.1.3. *Déboguer un programme.* Pour déboguer un programme, on le lance depuis le débogueur Python : ce débogueur est un module Python nommé `pdb` qu'il suffit d'importer. Comme de coutume, taper `?` ou `help` pour connaître les commandes disponibles

```
>>> import pdb
>>> pdb.run("tests.py")
```

Les commandes de débogage les plus courantes :

- ▷ `b(reak)` 32 : mettre un point d'arrêt ligne 32
- ▷ `n(ext)` : instruction suivante, exécuter les fonctions sans y entrer
- ▷ `s(tep)` : avancer d'un pas, entrer dans les fonctions
- ▷ `p(rint)` a : écrire la valeur de la variable a
- ▷ `c(ontinue)` : continuer jusqu'au prochain point d'arrêt
- ▷ `l(ist)` : lister le source autour de la ligne courante

Il est également possible d'utiliser `winpdb`, ou encore d'invoquer `pdb`, directement depuis le shell

```
$ pdb toto.py ses arguments
$ winpdb toto.py ses arguments
```

Exercice 22. Utiliser un debogueur pour trouver les deux (au moins!) bugs dans le programme `oct2dic_bug.py`

4.2. Fonctions Python, détails.

4.2.1. *premier exemple.* Dans un fichier `fibonacci.py`, nous pouvons créer une fonction qui écrit la série de Fibonacci jusqu'à une limite quelconque :

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
def fib(n):
    """Affiche une suite de Fibonacci jusqu'à n."""
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

if __name__ == "__main__":
    fib(500)
```

Le mot-clé `def` débute la définition d'une fonction. Il doit être suivi par le nom de la fonction et une liste entre parenthèses de paramètres formels.

Les instructions qui forment le corps de la fonction commencent sur la ligne suivante, *indentée par une tabulation*.

La première instruction devrait être la chaîne de documentation de la fonction, ou *docstring*. Il y a des outils qui utilisent les docstrings pour générer automatiquement de la documentation papier, ou pour permettre à l'utilisateur de naviguer interactivement dans le code.

L'exécution d'une fonction génère une nouvelle *table de symboles*, utilisée pour les variables locales de la fonction.

On ne peut pas affecter directement une valeur aux variables globales à l'intérieur d'une fonction (à moins de les déclarer avec une instruction *global*), bien qu'on puisse y faire référence.

Les vrais paramètres (arguments) d'un appel de fonction sont introduits dans la table de symboles locale de la fonction appelée quand elle est appelée ; ainsi, les arguments sont passés en utilisant un *passage par valeur*.

Une fonction est un objet *Python* comme un autre. Il est possible de renommer une fonction :

```
>>> from fibo import fib
>>> fib
<function object at 10042ed0>
>>> f = fib
>>> f(100)
1 1 2 3 5 8 13 21 34 55 89
```

La procédure *fib* malgré les apparences, retourne une valeur, bien qu'elle soit plutôt décevante. Cette valeur est appelée *None* (c'est un nom intégré).

```
>>> print fib(0)
None
```

Ecrire une fonction qui retourne une liste des nombres de la suite de Fibonacci, au lieu de les imprimer, est très simple :

```
def fib2(n):
    """Retourne une list contenant
    la série de Fibonacci jusqu'à n"""
    resultat = []
    a, b = 0, 1
    while b < n:
        resultat.append(b) # voir ci-dessous
        a, b = b, a+b
    return resultat

if __name__ == '__main__':
    fib2(100)
```

Cet exemple démontre quelques nouvelles caractéristiques de *Python* :

- ▷ l'instruction *return a* termine une fonction en renvoyant la valeur *a*. Sans argument, *return* renvoie *None*.
- ▷ l'instruction *resultat.append(b)* appelle une *méthode* de l'objet *resultat*. En programmation objet, une *méthode* est une fonction qui appartient à un objet et est nommée *obj.nommethode*

- ▷ Vous pouvez définir vos propres types d'objets et méthodes, en utilisant des classes, cf plus bas. La méthode `append()` montrée précédemment, est définie pour les objets de type `list`; elle ajoute un nouvel élément à la fin de la liste.
- ▷ L'instruction `def` crée un objet de type fonction, et l'affecte à son nom.

Exercice 23. Créer une fonction qui prend en argument deux listes et qui renvoie l'intersection des deux listes. Testez-la.

4.2.2. *Chaînes de documentation* (Docstrings). Il existe des conventions émergentes à propos des *docstrings*.

- ▷ La première ligne devrait toujours être un résumé concis des objectifs de l'objet, devrait toujours commencer par une lettre majuscule et finir par une virgule.
- ▷ S'il y a d'autres lignes dans la chaîne de documentation, la deuxième ligne devrait être vide, séparant visuellement le résumé du reste de la description.
- ▷ Les lignes suivantes devraient constituer un ou plusieurs paragraphes décrivant les conventions d'appel des objets, ses effets de bord, etc.

Voici un exemple de docstring multi-ligne :

```
def ma_fonction():  
    """Ne fait rien, mais le documente.  
  
    Non, vraiment, elle ne fait rien.  
    """  
    pass  
if __name__ == '__main__':  
    print ma_fonction.__doc__
```

Les docstrings sont également utilisées par la fonction `help` comme le montre l'exemple suivant.

```
>>> def carre(x):
...     """Retourne le carre de son argument."""
...     return x*x
...
>>> help(carre)
Help on function carre:

carre(x)
    Retourne le carre de son argument.
```

4.2.3. *Règles de portée.* Chaque fonction définit son propre *espace de noms*. Une variable déclarée dans une fonction est *inaccessible* en dehors de la fonction. Elle appartient à l'espace de noms de la fonction.

Exemple.

▷ Variable `x` non globale

```
>>> def f():
...     x=12
...
>>> x=1
>>> f()
>>> x
1
```

▷ Variable `x` globale

```
>>> def f():
...     global x
...     x=12
...
>>> x=0
>>> f()
>>> x
12
```

4.2.4. *Passage d'arguments .*

- ▷ Les arguments sont passés aux fonctions *par affectation*, ce qui signifie que les arguments sont simplement affectés à des noms locaux (à la fonction).
- ▷ Pratiquement, ce mode de passage par affectation a les conséquences suivantes :
 - ▶ Les arguments *non modifiables* miment le comportement du passage par valeur de C, mais *ne sont pas copiés* (gain de temps)
 - ▶ Les arguments *modifiables* se comportent comme des arguments passés par adresse en C, et ne sont bien sûr pas copiés non plus.

- ▷ Attention aux effets de bord : l'affectation (comme le passage d'argument) crée une référence sur la valeur affectée (sur l'argument) et non pas une copie profonde.

Exemple 24. Tester les instructions :

```
>>> def modif(x):
...     x[0] += 10
...
>>> a = [1, 2, 3]
>>> modif(a);
>>> print a
>>> def fidom(x):
...     x = 0
>>> b = 1
>>> fidom(b);
>>> print b
```

- ▷ Il est possible de définir des fonctions à nombre d'arguments variable. Il y a plusieurs façons de faire, qui peuvent être combinées.

4.2.5. *Valeurs d'argument par défaut.* La technique la plus utile consiste à spécifier une *valeur par défaut* pour un ou plusieurs arguments. Cela crée une fonction qui peut être appelée avec moins d'arguments qu'il n'en a été défini.

```
def interrogatoire(nom, prenom = '', presume = 'coupable'):
    print u"Bonjour, monsieur ", nom, prenom
    print u"Vous êtes présumé ", presume
```

Cette fonction peut être appelée soit comme ceci :

```
>>> interrogatoire('Puisseux')
```

ou comme ceci :

```
>>> interrogatoire('Puissant', '', 'innocent')
```

4.2.6. *Arguments à mot-clé.* Les fonctions peuvent aussi être appelées en utilisant des arguments mots-clés de la forme `motcle = valeur`. Par exemple, pour la fonction précédente :

```
>>> interrogatoire(prenom='Pierre', presume='innocent', nom='Puisseux')
```

4.3. Les modules `os`, `os.path`.

4.3.1. `os`.

- ▷ `os.walk` parcourt d'une arborescence. Par exemple :

```
>>> import os
>>> for root, dirs, files in os.walk('.'):
...     print "Liste des fichiers du repertoire : %s => %s" \
              %(root, files)
```

- ▷ `os.system(cmd)` exécution de la commande shell `cmd` (string)

```
>>> os.system('mkdir toto')
```

`os.path`

- ▷ `abspath(chemin)` : `os.path.abspath('devel/toto.cxx')` → le chemin absolu de `'devel/toto.cxx'`
- ▷ `basename(chemin)` : `os.path.basename('devel/toto.cxx')` → `toto.cxx`
- ▷ `curdir` le répertoire courant
- ▷ `dirname(chemin)` `os.path.dirname('devel/toto.cxx')` → `devel`
- ▷ `exists(chemin)` renvoie vrai ou faux suivant que le chemin existe ou non
- ▷ `expanduser(chemin)` expansion du `~` en `/home/mon_repertoire`
- ▷ `sep` = `'/'` en unix, `'\'` en window\$
- ▷ Caractéristiques (date, taille, .. d'un fichier ou répertoire) :
 - ▶ `getatime(fichier)`
 - ▶ `getsize(fichier)`
- ▷ `isabs(chemin)` le chemin est-il absolu ?
- ▷ `isdir(chemin)` est-ce un répertoire existant ?
- ▷ `isfile(chemin)` un fichier ?
- ▷ `islink(chemin)` un lien ?
- ▷ `ismount(chemin)` le chemin est-il un point de montage ?
- ▷ `join os.path.join('devel', 'toto.cxx')` → `devel/toto.cxx`
- ▷ `split os.path.split('devel/toto.cxx')` → `('devel', 'toto.cxx')`
- ▷ `splitext os.path.splitext('devel/toto.cxx')` → `('devel/toto', '.cxx')`

Exercice 25. utilisation de `os.path`

- (1) Trouver, à l'aide du module `os.path`, l'adresse absolue de votre répertoire utilisateur.
- (2) Lister récursivement l'adresse absolue de tous les fichiers `.py` et `.txt` qui s'y trouvent.

```
(3) #!/usr/bin/python
#-*- coding : utf-8 -*-
if __name__ == "__main__" :
    import os.path, os
    root = os.path.expanduser ('~/devel/Version2/Concha/')
    print root#, os.path.isdir(root)
    for r,d,F in os.walk(root) :
        for f in F :
            if os.path.splitext(f)[1] in ['.py', '.txt'] :
                print os.path.basename(f)
```

4.4. **Le module `optparse`.** Pour écrire un script *Python* utilisant le mécanisme des options *Unix*, le module `optparse` est bien adapté. On trouvera la documentation du module dans la section 14.3 de la documentation des librairies *Python*.

Par exemple si dans un script `tests.py`, on trouve les instructions :

```
parser = OptionParser("Voici comment utiliser la commande xxx")
parser.add_option("-o", "--output",
                  dest="fileoutname",
                  help=u"nom du fichier sortie")
(opts, args) = parser.parse_args()
```

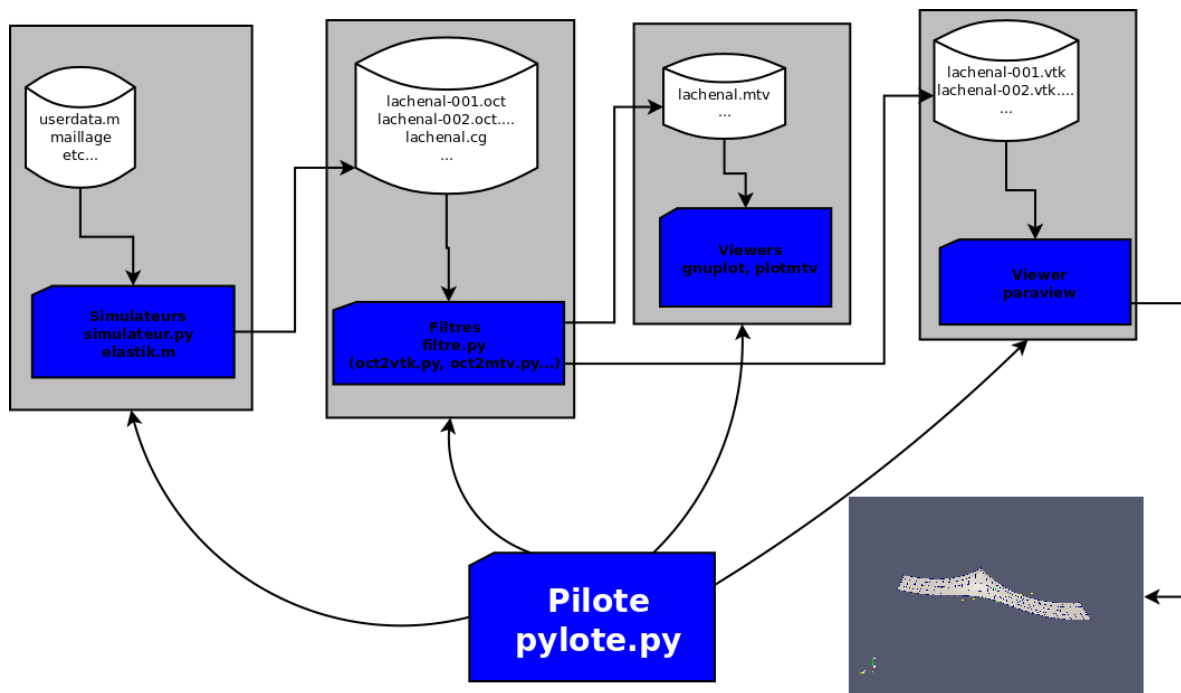


FIG. 4.2. Pilotage d'une application

alors la variable `opts` est le dictionnaire `opts={'fileoutname': None}` parmi les options disponibles pour l'exécution de `tests.py`, l'option `-o` (option longue `--output`) sera utilisée pour préciser le nom du fichier d'entrée, qui sera affecté dans le script à l'attribut `parser.fileoutname`, la chaîne de caractères `help= "... "` est utilisée pour documenter le script lors d'un appel à

```
$ toto.py -h
```

Exercice 26. Exécutez le script `options.py` avec différentes options

```
$ options.py -h
$ options.py --out=filout -c -v filin
$ options.py -o filout -c filin
$ options.py -w
$ options.py --out=filout
```

déduisez-en les règles de formation des options dans un script de ce type.

Notez la différence entre options avec valeurs (`-o`) et options sans valeur (`-h`, `-v`)

4.5. Python pour piloter une application.

Exercice 27. Python pour piloter une application (voir (4.2))

On dispose des composants suivants :

- (1) le simulateur (`simulateur.py`) programme *Python*
- (2) les filtres (`filtre.py`), module *Python*, qui transforment les résultats du simulateur au format idoine pour la visualisation (vtk par exemple)
- (3) `paraview`, pour la visualisation, un programme non *Python*

Ouvrir dans un éditeur le script `pylote_incomplet.py`. À partir de celui-ci, écrire un script *Python* nommé `pilote.py`, utilisant le mécanisme des options, que l'on appellera ainsi :

```
$ python pilote.py -v session
```

- ▷ l'argument **session** est obligatoire, il indique le répertoire de la session, qui contient :
 - ▶ le fichier de données **userdata.m** nécessaire au fonctionnement du **simulateur.py**,
 - ▶ les données et
 - ▶ les résultats après simulation
- ▷ L'option **-v** est facultative, si elle est présente, le viewer **paraview** est lancé.

4.6. Les portées et les espaces de noms en *Python*. D'abord quelques définitions.

- (1) Un *espace de noms* (name space) est une relation entre des noms et des objets. Par exemple
- (a) l'ensemble des noms intégrés (les fonctions telles que `abs()`, et les noms d'exception intégrés) ;
 - (b) les noms globaux dans un module ;
 - (c) les noms locaux au cours d'un appel de fonction.

il n'y a absolument aucune relation entre les noms contenus dans les différents espaces de noms

par exemple, deux modules différents peuvent définir tous les deux une fonction “maximise” sans confusion possible — les utilisateurs des modules doivent préfixer par le nom du module à l'utilisation.

- ▷ L'espace de noms appelé `__builtin__` qui contient les noms intégrés est créé au lancement de l'interpréteur *Python*, et n'est jamais effacé.
 - ▷ L'espace de noms global pour un module est créé quand la définition du module est chargée.
 - ▷ Les instructions exécutées à l'invocation de l'interpréteur font partie d'un module appelé `__main__`, elles ont donc leur propre espace de noms global.
 - ▷ L'espace de noms local à une fonction est créé quand celle-ci est appelée et il est effacé quand la fonction se termine
- (2) Une *portée* (*scope*) est une région textuelle d'un programme *Python* dans laquelle un espace de noms est directement accessible.

A n'importe quel moment de l'exécution, exactement trois portées imbriquées sont utilisées (exactement trois espaces de noms sont accessibles directement) :

- (a) la *portée immédiate*, qui est explorée en premier, contient les noms locaux,
- (b) la *portée intermédiaire*, explorée ensuite, contient les noms globaux du module courant, et
- (c) la *portée extérieure* (explorée en dernier) correspond à l'espace de noms contenant les noms intégrés.

Si un nom est déclaré *global* alors les références et affectations le concernant sont directement adressées à la portée qui contient les noms globaux du module.

5. LES CLASSES EN PYTHON

Le mécanisme de classe en **Python** permet d'introduire les classes avec un minimum de syntaxe et sémantique nouvelles.

Les caractéristiques les plus importantes des classes sont pleinement présentes :

- ▷ le mécanisme d'*héritage* permet la multiplicité des classes de base,
- ▷ une classe dérivée peut *surcharger* n'importe quelle méthode de sa ou ses classes de base,
- ▷ une méthode peut appeler une méthode de sa classe de base avec le même nom.
- ▷ Les objets peuvent contenir un nombre arbitraire de *données privées*.
- ▷ tous les membres d'une classe (dont les données membres) sont *publics*,
- ▷ toutes les fonctions membres sont *virtuelles*. Il n'y a pas de constructeurs ou de destructeurs particuliers.
- ▷ il n'y a pas de raccourcis pour faire référence aux membres d'un objet à partir de ses méthodes : une méthode est déclarée avec un premier argument explicite (**self** le plus souvent) qui représente l'objet, qui est fourni implicitement à l'appel.
- ▷ les classes sont elles-mêmes des *objets*, mais dans un sens plus large : en *Python*, tous les types de données sont des objets. Cela fournit la sémantique pour l'importation et le renommage.
- ▷ les types intégrés *ne peuvent pas* être utilisés comme classes de base pour des extensions par l'utilisateur.
- ▷ la plupart des *opérateurs* intégrés qui ont une syntaxe particulière (opérateurs arithmétiques, indigage, etc.) peuvent être *redéfinis* pour des instances de classe.

5.1. **Une première approche des classes.** Les *classes* introduisent un peu de syntaxe nouvelle et quelques points de sémantique supplémentaires.

Syntaxe de la définition de classe. La forme la plus simple de définition de classe ressemble à ceci :

```
class NomClasse:
    <instruction-1>
    .
    .
    .
    <instruction-N>
```

5.1.1. Une classe *Complexe* : déclaration.

```
5 class Complexe:
6     """Une classe complexe sommaire"""
7     def __init__(self, re=0.0, im=0.0):
8         self._x, self._y = re, im
9
10    def arg(self):
11        return atan2(self._x, self._y)
12
13    if __name__ == "__main__":
14        z0 = Complexe(1, 2)
15        print z0._x, z0._y
16        print z0.arg()
```

- (1) `class` est un mot clé, `Complexe` le nom de la classe. Cette ligne alerte l'interpréteur : je veux créer un objet *Python* de type `class`, une nouvelle classe (et non pas un objet de type `Complexe` qui, lui, sera créé plus loin). La classe `Complexe` hérite de la classe `object`, c'est à dire que tous les attributs de la classe `object` peuvent être utilisés par notre nouvelle classe.
- (2) un *docstring* pour expliquer le fonctionnement
- (3) la première *méthode* est un constructeur (méthode spéciale) : `__init__` ses trois arguments sont
 - (a) `self` : désigne l'objet non encore instancié, de type `Complexe`, qui appellera cette méthode. Autrement dit, `self` est un `Complexe`, qui n'existe pas encore.³
 - (b) `re` et `im` sont les parties réelle et imaginaire, (par défaut 0.0 et 0.0)

Il y a création de deux attributs `_x` et `_y`, dans `self`. Ces attributs prendront corps en même temps que `self`, lorsque un `Complexe` sera *instancié*.

- (4) `arg()` est une *méthode* qui calcule l'argument. L'argument `self` prendra la valeur `Complexe z` lorsque sera exécutée l'instruction `z.arg()`

³ De manière analogue, au moment de la définition d'une fonction (par exemple `def f(x) : return x*x`) `x` ne désigne *aucune* variable, mais prendra corps seulement lorsque la fonction sera appelée par exemple `a=3;f(a)`.

5.1.2. *Complexe* : instantiation. l'instruction :

```
z0=Complexe(1,2)
```

déclenche les actions suivantes :

▷ appel à `Complexe.__init__(1,2)` , \implies `z0` est *instancié*. Les attributs `z0._x`, `z0._y` deviennent effectif, à l'intérieur de `z0`.

5.1.3. *Complexe* : utilisation.

```
print z0._x,z0._y
print z0.arg()
```

▷ `z0._x` et `z0._y` valent 1 et 2. Il serait commode d'avoir un instruction `print z0`

▷ `z0.arg()` \iff `Complexe.arg(z0)` dans la méthode `Complexe.arg(self)`, `self` vaut `z0`

Plus généralement si `uneMethode(a)` est une méthode de la classe `MaClasse`, et si `X` est un objet de type `MaClasse`, alors

```
X.uneMethode(a)  $\iff$  MaClasse.uneMethode(X,a)
```

Exercice 28. Ecrire la méthode `abs()` de la classe *Complexe*

5.2. **Méthodes spéciales.** Les méthodes *spéciales* commencent et finissent par `__` (deux underscores '`__`'). Elles sont héritées de la classe `object`.

On trouve dans la documentation *Python* de la classe `object` quelle sont ces méthodes.

Par exemple, si l'on veut pouvoir écrire `z=z1+z2` pour `z1` et `z2` de type *Complexe*, il suffit de surcharger la méthode `Complexe.__add__(self, z)` dans la classe *Complexe* en sachant que

```
z=z1+z2  $\iff$  z=z1.__add__(z2)
```

Les principales méthodes que l'on peut surcharger pour une classe donnée sont :

Méthode de la classe à surcharger	utilisation
<code>object.__add__(self, other)</code>	<code>self+other</code>
<code>object.__sub__(self, other)</code>	<code>self-other</code>
<code>object.__mul__(self, other)</code>	<code>self*other</code>
<code>object.__and__(self, other)</code>	<code>self and other</code>
<code>object.__or__(self, other)</code>	<code>self or other</code>
<code>object.__len__(self)</code>	<code>len(self)</code>
<code>object.__getitem__(i)</code>	<code>x=self[i]</code>
<code>object.__setitem__(i)</code>	<code>self[i]=y</code>
<code>object.__call__(self[, args...])</code>	<code>self(args)</code>
<code>object.__str__()</code>	<code>print self</code>
<code>object.__repr__()</code>	<code>self</code>

Exercice 29. implémentez une méthode `__str__()` dans la classe *Complexe* , afin qu'elle affiche de manière agréable le Complexe `self`. Testez ensuite les instructions

```
>>> z0=Complexe(1,2)
>>> print z0
```

Exercice 30. Méthodes spéciales

Implémenter une méthode `__add__()` et une méthode `__mul__()`. Testez l'instruction

```
print z0+z1, z1*z0
```

5.3. Héritage simple, multiple. L'instruction :

```
class Complexe(object):
```

signifie que la classe `Complexe` *hérite* de la classe `object` (qui est une classe *Python* de base).

Donc `Complexe` *EST* un `object` de *Python*.

Grâce à cet héritage, tous les attributs et les méthodes de la classe `object` peuvent être utilisés et/ou surchargés par les instances de la classe `Complexe`.

5.3.1. Héritage ou pas ? On utilise un héritage lorsque l'héritier *EST* un parent, comme dans une filiation de famille ordinaire. Le fils Dupond *est* un Dupont.

La bonne question à se poser pour décider si B doit hériter de A est :

B *is* A ou bien B *has* A ?

▷ Si la réponse est *B is A*, alors B doit *hériter* de A,

▷ Si la réponse est *B has A*, alors B doit avoir un *attribut* A.

Par exemple : un cercle n'*EST* pas un `Complexe`, ni même un centre. Un cercle *POSSÈDE* un centre (donc un `Complexe`), une classe `Cercle` n'héritera pas de `Complexe`, mais possèdera un attribut `Complexe`, qu'on appellera probablement `_centre`.

Par contre un carré *EST* un polygone. Un triangle aussi

5.3.2. Sans héritage : la classe `Cercle` s'écrira :

```
class Cercle(object) :
    def __init__(self, centre, rayon) :
        self._centre = centre #attribut
        self._rayon = rayon
    def perimetre(self) : pass
    def aire(self) : pass
```

5.3.3. Avec héritage : en supposant que l'on dispose d'une classe `Polygone`, comme celle-ci,

```
class Polygone(object):
    def __init__(self, liste_de_points) :
        self._points=liste_de_points
    def isClosed(self) :
        return self._points[0] == self._points[-1]
    def close(self) :
        if not self.isClosed() :
            self._points.append(self._points[0])
    def longueur(self):
        p=self._points#alias
        return sum([abs(z1-z0) for (z0,z1) in zip(p[:-1],p[1:])])
    def __str__(self) : return "%s"%self._points
    def __getitem__(self,i) : return self._points[i]
```

on pourra définir une classe `Triangle` comme ceci :

```
class Triangle(Polygone):
    def __init__(self, a, b, c) :
        Polygone.__init__(self, [a,b,c])
    def __str__(self):
        return "Triangle : "+Polygone.__str__(self)
    def hauteur(self):#blabla, spécifique à Triangle
    def isRectangle(self) : #etc...
```

et l'utiliser ainsi :

```
t=Triangle(complex(1,0), complex(0,1), complex(1,1))
t.longueur() # méthode de Polygone
t.isClosed() # méthode de Polygone
t[1]
t.hauteur() # méthode spécifique à Triangle
```

etc...

5.3.4. *Surcharge*. Dans cet exemple, la méthode spéciale `Triangle.__str__` surcharge celle de `Polygone.__str__`.

Exercice 31. Ecrire la classe `Triangle` qui hérite de `Polygone` (dans le même fichier `polygone.py`) et tester ses différents composants. Ecrire en particulier une méthode `aire()` qui calcule l'aire du triangle.

5.3.5. *Héritage multiple*. Python supporte aussi une forme limitée d'héritage multiple. Une définition de classe avec plusieurs classes de base ressemble à :

```
class NomClasseDerivee(Base1, Base2, Base3):
    <instruction-1>
    .
    .
    .
    <instruction-N>
```

La seule règle permettant d'expliquer la sémantique de l'héritage multiple est la règle de résolution utilisée pour les références aux attributs. *La résolution se fait en profondeur d'abord*, de gauche à droite. Donc, si un attribut n'est pas trouvé dans `NomClasseDerivee`, il est cherché dans `Base1`, puis (récursivement) dans les classes de base de `Base1`, et seulement s'il n'y est pas trouvé, il est recherché dans `Base2`, et ainsi de suite.

L'utilisation banalisée de l'héritage multiple est un cauchemar de maintenance.

5.4. **Quelques remarques.** Les données attributs écrasent les méthodes de même nom ;
 ⇒ utiliser une convention qui minimise les chances de conflit.

Autrement dit, les classes ne sont pas utilisables pour implémenter des types abstraits purs.

Par *convention*, le premier argument d'une méthode est souvent appelé `self`.

Les méthodes peuvent appeler d'autres méthodes en utilisant les attributs méthodes de l'argument `self` :

```
class Sac:
    def vider(self):
        self.donnees = []
    def ajouter(self, x):
        self.donnees.append(x)
    def ajouterdoublon(self, x):
        self.ajouter(x)
        self.ajouter(x)
```

Les méthodes peuvent faire référence à des noms globaux de la même façon que les fonctions ordinaires. La portée globale associée à une méthode est celle du module qui contient la définition de la classe. (La classe elle-même ne sert jamais de portée globale!)

5.5. Objets en rapport avec les classes.

5.5.1. *Objets classes.* Les objets classe admettent deux sortes d'opérations : la référencement des attributs et l'instanciation.

- ▷ Les *références* aux attributs (*attribute references*) utilisent la syntaxe standard utilisée pour toutes les références d'attribut en *Python* : `obj.nom`.

Par exemple, si la définition de classe ressemble à :

```
class MaClasse:
    u"""Une classe simple pour exemple"""
    i = 12345
    def f(self):
        return u"Bonjour tout l'monde"
```

alors `MaClasse.i` et `MaClasse.f` sont des références d'attribut *valides*.

Et on peut leur affecter une valeur : `MaClasse.i = 12` et `MaClasse.f = g` sont des affectations valides.

`__doc__` est un attribut valide, en lecture exclusive, qui renvoie la *docstring* correspondant à la classe.

- ▷ L'instanciation de classe utilise la notation d'appel de fonction. Faites comme si l'objet classe était une fonction sans paramètres qui renvoie une instance nouvelle de la classe. Par exemple, (avec la classe précédente) :

```
x = MaClasse()
```

crée une nouvelle instance de la classe et affecte cet objet à la variable locale `x`.

Une classe peut définir une méthode spéciale nommée `__init__()`, comme ceci :

```
def __init__(self):
    self.donnee = []
```

Dans ce cas, l'instanciation de la classe appelle automatiquement `__init__()`.

Bien-sûr, la méthode `__init__()` peut avoir des arguments pour offrir plus de souplesse. Par exemple,


```

class Complexe:
    def __init__(self, partiereelle, partieimaginaire):
        self.r = partiereelle
        self.i = partieimaginaire

z = Complexe(3.0, -4.5)
print z.r, z.i

```

5.5.2. *Objets instances.* Que peut-on faire avec les objets instances? Les seules opérations acceptées par des objets instance sont des références à leurs attributs. Il y a deux sortes de noms d'attributs valides.

- (1) J'appellerai la première “*données attributs*” (*data attributes*). Ils correspondent aux “*données membres*” (*data members*) en C++. Les données attributs n'ont pas besoin d'être déclarées; comme les variables locales, elles apparaissent lorsqu'on leur affecte une valeur pour la première fois. Par exemple, si `x` est l'instance de `MaClasse` créée précédemment, le morceau de code suivant affichera la valeur 16, sans laisser de trace :

```

x.compteur = 1
while x.compteur < 10:
    x.compteur = x.compteur * 2
print x.compteur
del x.compteur

```

- (2) La seconde sorte de référence d'attribut acceptée par les objets instance sont les *méthodes* (*methods*). Une méthode est une fonction qui “appartient” à un objet.

5.5.3. *Objets méthodes.* D'habitude, une méthode est appelée de façon directe :

```
x.f()
```

Dans notre exemple, cela renverrait la chaîne “*Bonjour tout l'monde*”. L'objet `x.f` est un objet méthode, il peut être rangé quelque part et être appelé plus tard, par exemple :

```

xf = x.f
while 1:
    print xf()

```

continuera à afficher “*Bonjour tout l'monde*” jusqu'à la fin des temps.

La particularité des méthodes est que l'objet est passé comme premier argument à la fonction.

Dans notre exemple, l'appel `x.f()` est l'équivalent exact de `MaClasse.f(x)` .

5.6. **Variables privées.** Il y a un *support limité pour des identificateurs privés* dans une classe. Tout identificateur de la forme `__spam` (au moins deux tirets-bas au début, au plus un tiret-bas à la fin) est maintenant textuellement remplacé par `_nomclasse__spam` , où `nomclasse` est le nom de classe courant, duquel les tirets-bas de début ont été enlevés.

5.6.1. *Itérateurs*. Vous avez probablement remarqué que la plupart des conteneurs peuvent être parcourus en utilisant une instruction *for* :

```
for element in [1, 2, 3]:
    print element
for element in (1, 2, 3):
    print element
for key in {'one':1, 'two':2}:
    print key
for char in "123":
    print char
for line in open("myfile.txt"):
    print line
```

Ce style d'accès est clair, concis et pratique. L'utilisation d'itérateurs imprègne *Python* et l'unifie.

En coulisse,

- ▷ l'instruction *for* appelle *iter()* sur l'objet conteneur.
- ▷ Cette fonction renvoie un objet *itérateur* définissant une méthode *next()* qui accède les éléments dans le conteneur, un à la fois.
- ▷ Lorsqu'il n'y a plus d'éléments, *iter()* lève une exception *StopIteration* qui dit à la boucle *for* de se terminer.

Exemple 32. itérateur :

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> it.next()
'a'
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()
Traceback (most recent call last):
File "<pyshell#6>", line 1, in -toplevel
it.next()
StopIteration
```

Pour ajouter un comportement d'itérateur à vos classes, définissez une méthode *__iter__()* qui renvoie un objet ayant une méthode *next()*. Si la classe définit *next()*, alors *__iter__()* peut se limiter à renvoyer *self* :

Exemple 33. Itérateur

```

class Reverse:
    """Iterator for looping over a sequence backwards"""
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def next(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]

>>> for char in Reverse('spam'):
...     print char
...
m
a
p
s

```

5.6.2. Générateurs.

5.6.3. Expressions générateurs.

5.7. *property*. Une *property* est un attribut de la classe, gérée par un *getter* et un *setter*.

Exemple 34. basique

```

class Complexe(object):
    u"""Démonstration property"""
    def __init__(self, x, y):
        self._x = x
        self._y = y
    def getPrettyPrint(self):
        return "(%f, %fi)" % (self._x, self._y)
    pp = property(getPrettyPrint)

>>> z = Complexe(1,2)
>>> print z.pp
(1.000000, 2.000000i)
>>> print z.getPrettyPrint()
(1.000000, 2.000000i)

```

- ▷ Pour l'utilisateur, la notion de *property* est transparente. Elle permet de définir, comme en C++ des *getters* et des *setters*, avec une vérification de type si nécessaire, mais avec une syntaxe légère pour l'utilisateur.

```

Complexe z() ; z.SetRe(14.5); cout<<z.GetRe(); //En C++
z=Complexe() ; z.x=14.5 ; print z.x #En Python, x est une property

```

▷ Au niveau de la programmation, cela demande au programmeur un effort initial.

La signature de *property* est :

```
property(fget=None, fset=None, fdel=None, doc=None)
```

▷ *fget* est la méthode *getter* de l'attribut

▷ *fset* le *setter*

▷ *fdel* est le *destructeur* de l'attribut

▷ *doc* est une chaîne de caractères contenant la *documentation* de l'attribut

Exemple 35. Pour une classe *Complexe* de base : définissons un getter, un setter, une doc pour accéder à la partie réelle :

```

class Complexe(object):
    u"""Démonstration property"""
    def __init__(self, x, y):
        self._x = x
        self._y = y
    def _getx(self) : return self._x
    def _setx(self, value) : self._x = float(value)
    x = property(fget = _getx,
                 fset = _setx,
                 doc = u"Partie réelle entière")

```

Exécution dans une session *Python* :

```

>>> z = Complexe(1,2)
>>> z.x = 5
>>> print z.x, z._y
5.0 2
>>> z.x = "7.2"
>>> print z.x, z._y
7.2 2
>>> z.x = "ah"
>>> print z.x, z._y
ah 2

```

5.8. Décorateurs. Un décorateur python est une fonction qui prend en paramètre une autre fonction, pour la modifier, lui ajouter des fonctionnalités, la substituer ou simplement exécuter un travail avant ou après l'avoir appelé.

Un décorateur est caractérisé par le mot clé `@` placé au dessus de la fonction ciblée.

Exemple 36. Décorateur

```
>>> def mon_decorateur(f):
...     def _mon_decorateur():
...         print "decorator stuff"
...         f()
...         print "other stuff"
...         return
...     return _mon_decorateur
...
>>> @mon_decorateur
... def decorate():
...     print "fonction stuff"
...
>>> decorate()
decorator stuff
fonction stuff
other stuff
```

Comme vous pouvez le remarquer sur l'exemple, dès qu'on applique un décorateur sur une fonction c'est le décorateur qui prend le contrôle, autrement dit, il peut carrément ignorer la fonction `decorate()`.

5.9. Manipulation dynamique des attributs.

```
>>> class A(object):
...     pass
...
>>> a=A()
>>> setattr(a, 'un_attribut', 3.14)
>>> hasattr(a, 'un_attribut')
True
>>> a.un_attribut
3.1400000000000001
>>> delattr(a, 'un_attribut')
>>> hasattr(a, 'un_attribut')
False
```

- ▷ `setattr(objet, nom, valeur)` pour ajouter l'attribut `nom` à `objet`, avec la valeur `valeur`
- ▷ `delattr(objet, nom)` pour supprimer l'attribut `nom` à `objet`
- ▷ `hasattr(objet, nom)` renvoie `True` ou `False` suivant que `objet` possède ou non un attribut `nom`.

5.10. Exercices.

Exercice 37. Ecrire une classe `Tableau` qui hérite de la classe `list`,

- ▷ dont la fonction `__init__(self, l)` permet de vérifier que l'argument `l` est une liste dont tous les éléments sont d'un type numérique : `int`, `long`, `float` ou `complex`
- ▷ qui permet d'écrire les instructions :

```
>>> a=Tableau([1,2,3])
>>> a.norm()
>>> a.norm(1)
>>> len(a)
>>> b=Tableau([2,3,4])
>>> a+b
>>> a*b #produit scalaire
>>> a(3) = 12
>>> a[3] = 12
```

Exercice 38. Implémenter une classe `Complexe`, qui hérite de la classe `object`, et qui permette d'écrire les instructions :

```
>>> z0=Complexe(1,2) #__init__
>>> z0                # définir __repr__
1+2I
>>> print z0          # définir __str__
1+2I
>>> print z0._x
1
>>> print z0.arg()
1.40564764938
>>> z1=Complexe(2,3)
>>> print z1.module()
3.60555127546
>>> print z1+z0       #__add__
14+5I
```

Exercice 39. Utiliser la notion de `property` pour effectuer une vérification sur le type et modifier le comportement de la classe `Complexe` de la manière suivante :

```
>>> z0=Complexe(1,2)
>>> z0.x = "12.0"
Traceback (most recent call last):
  File "./prop.py", line 40, in <module>
    z0.x = "12"
```

```
File "./prop.py", line 29, in _setx
    raise TypeError, "Complexe.x : %s non supporte"%(type(value))
TypeError: Complexe.x : <type 'str'> non supporte
```

Exercice 40. Décorateur pour vérification de type. Prévoir le comportement des instructions suivantes :

```
def ArgEntier(f) :
    def _ArgEntier(arg):
        if not isinstance(arg,int) :
            raise TypeError( "%s : %s n'est pas de type entier" \
                             % (arg,type(arg)))
        else : return f(arg)
    return _ArgEntier

@Entier
def h(x) :
    return x*x

print h(1)
print h(1.2)
```

Exercice 41. Décorateurs : soit la fonction

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

- (1) Ajouter un décorateur à la fonction `f`, qui vérifie que l'argument `x` est de type `float` et lève une exception `TypeError` si ce n'est pas le cas.
- (2) Ajouter un décorateur à la fonction `f`, qui lève une exception adaptée si le nombre d'arguments est différent de 1

Exercice 42. Soit la fonction `def g(x) : print x`. Ecrire un décorateur `Majuscule`, appliquer le à `g` de sorte que :

```
>>> g('hello')
```

produise le résultat

```
HELLO
```

RÉFÉRENCES

- [LA] Mark Lutz, David Ascher, Introduction à Python, ed O'Reilly, Janvier 2000, ISBN 2-84177-089-3
- [HPL] Hans Petter Langtangen, Python Scripting for Computational Science. Simula Research Laboratory and Department of Informatics University of Oslo.
[/home/puiseux/enseignement/CoursOutilsInformatiques/Python_scripting_for_computational_science.04.pdf](#)
- [MP] Mark Pilgrim, dive into Python (Plongez au coeur de Python) <http://diveintopython.org/>, traduction française : Xavier Defrang, Jean-Pierre Gay, Alexandre Drahon.
[/home/puiseux/doc/python/frdiveintopython.pdf](#)
- [VR] Guido van Rossum, Tutoriel Python, Release 2.0.1. [/home/puiseux/doc/python/tut-fr.pdf](#)
- [GS] Gérard Swinnen, Apprendre à programmer avec Python, <http://www.librecours.org/documents/5/577.pdf>
- [1] Jeffrey Elkner, Allen B. Downey and Chris Meyers, <http://openbookproject.net//thinkCspy>

INDEX

court-circuit, 25

docstring, 33

None, 34

packing, 23

unpacking, 23