Introduction à Python (html)

Cours d'introduction au langage Python. Version html.

Didacticiel Python

Auteur: Olivier Laurent
Organisation: Python Blanc Bleu Belge
Date: 23/01/2005

Table des matières

1 Introduction

Ce didacticiel est destiné à vous plonger rapidement dans le langage Python.

Guido van Rossum, le créateur du langage Python a composé un didacticiel plus complet. Il fait partie de la documentation officielle du langage et est écrit en anglais, mais il a été heureusement traduit par Olivier Berger, Daniel Calvelo Aros et Bruno Liénard. Il est disponible ici: http://olberger.club.fr/python/doc/tut/tut.html

2 Qu'est-ce que Python ?

Python est un langage de plus en plus utilisé à l'heure actuelle. Bien qu'il soit relativement jeune, il comprend déjà un nombre impressionnant de bibliothèques dans des domaines très divers et d'applications très performantes.

Pour la petite histoire, Python fut développé en 1990 par Guido van Rossum à l'université d'Amsterdam. Python était destiné à être un langage avancé de scripting pour le système d'exploitation distribué Amoeba. Le terme *Python* provient de la série télévisée Monty Python's Flying Circus. Beaucoup de plaisanteries ou d'exemples de code font allusion à cette série ou aux films tournés avec la troupe des Monty Python comme 'Monty Python's Quest for the Holy Grail' (un navigateur Web écrit en Python s'appelle d'ailleurs 'Grail') ou 'La vie de Brian' (titre français).

Voyons quelques uns de ses avantages et ses (peu

nombreux) inconvénients :

- Python est, orienté objet : il est même l'un des plus complets à ce sujet. Il supporte le polymorphisme, l'héritage multiple, la surcharge des opérateurs,... Mais si ces termes ne vous disent rien, n'ayez crainte car Python peut aussi être utilisé comme langage procédural classique.
- Python est librement distribuable et ses sources sont ouvertes. Toute personne désirant contribuer à étendre Python peut donc le faire, Python appartient à la communauté toute entière.
- Python est un langage multi plateformes. Il tourne sous les environnements Unix, BeOs, Windows, Amiga, QNX, OpenVMS, VxWorks, Psion Serie 5, OS2, MacOS, DOS, Win3.11, Windows CE, ...
- Python peut facilement s'interfacer avec d'autres langages de programmation :
    - Le C et Python : Python possède une API d'intégration Python/C. Les programmes Python peuvent être étendus en C et les programmes C peuvent aussi être étendus en Python.
    - Jython : JPython est une implémentation de Python écrite en Java permettant de compiler un programme écrit en Python dans du bytecode Java.
    - Pythonwin : Pythonwin permet aux programmes Python de communiquer avec l'API COM de Windows permettant ainsi de se défaire de l'emprise Visual Basic sur ce type de plates-formes.
- Il n'y a pas de phase de compilation ou d'édition de liens comme en C ou en Java. En réalité, les programmes Python sont automatiquement compilés en bytecode au lancement du programme. Ce qui accélère le processus de développement sans subir les faibles performances des langages purement interprétés.
- Malgré sa puissance, Python est un langage très simple à apprendre grâce à sa syntaxe limpide et son modèle orienté objet très bien construit. Python est tout indiqué comme premier langage aussi bien comme langage purement procédural que

comme langage objet. Il permet de se familiariser aiment avec les concepts fondamentaux de la programmation. C'est donc une excellente approche à des langages procéduraux comme le C ou des langages objet comme l'Objective C ou le C++. La limpidité de son code facilite, de plus, la maintenance des programmes.

- Python est très versatile. Il sert aussi bien de langage de script que pour le codage d'applications plus importantes :
  - Scripts d'administration système : grâce à sa puissante interface avec l'OS (services POSIX, variables d'environnement, fichiers, sockets, pipes, processus, threads, expressions rationnelles, ...)
  - Prototypage Rapide d'Applications : écriture du prototype en Python. Si le prototype fonctionne parfaitement, on a gagné. Sinon, on recode uniquement les sections critiques dans un autre langage comme le C.
  - Programmation Internet :
    - génération de documents HTML, manipulation de fichiers XML, gestion des protocoles HTTP, FTP, Gopher, IMAP, nntp, pop, telnet,...
    - CGI (Common Gateway Interface)
    - Applets Java écrites en Python avec Jython.
    - Medusa est un serveur Web et FTP à haute performance.
    - ZOPE est un serveur d'application permettant de publier des objets Python sur le Web.
  - Bases de données : Python peut interroger les Bases de Données SQL les plus courantes (Sybase, Oracle, Informix, PostgreSQL, MySQL, ODBC,...). Gadfly est une base de données écrite en Python. Elles ne peut rivaliser avec les DB suçotée mais est du même niveau que Microsoft Access. Elle n'est utilisée que dans un

but didactique.
    - Interfaces graphiques : Un grand nombre d'interfaces graphiques sont disponibles sous Python. La plus connues est TKInter (basée sur le langage TK) mais des interfaces vers GTK, WxWindows ou même les MFC de Microsoft (au détriment de la portabilité) sont possibles ainsi que bien d'autres.
    - Traitement d'images : La bibliothèque PIL (Python Imaging Library) permet de manipuler un grand nombre de formats d'images (jpg, gif, png, bmp,...). Une bibliothèque permet même d'écrire, en Python, des script-fu pour GIMP (GNU Image Manipulation Program), un Photoshop-like open-source pour Linux (et Windows).
    - Programmation numérique, Intelligence Artificielle, Objets Distribués, PDFGen (génération de documents au format PDF), HTMLGen (Génération de pages HTML), RE (Module d'expression régulières),...
- Un certain nombre de caractéristiques font de Python un langage très puissant :
    - Les modules appelés par Python peuvent être codés en C. L'utilisateur n'affronte donc pas la complexité du C. Python sert donc d'interface entre l'utilisateur et les modules écrit en C.
    - Python se sert de manière intensive de plusieurs types complexes comme les listes ou les dictionnaires, offrant ainsi souplesse et facilité à l'utilisateur.
    - Python gère la mémoire automatiquement (Ramasse-Miettes ou 'Garbage Collector')

3 Leçon préliminaire

Lancer un programme Python.

Python peut être utilisé de 2 manières différentes :

- Intéractivement à l'aide de

l'interpréteur.
- ou en lançant un fichier python.

3.1 Lancer l'interpréteur

Tapez *python* à l'invite d'une console en mode texte [1]. L'invite interactive Python est représentée par un triple signe 'plus grand que' : >>>. L'interpréteur attend maintenant vos instruction. Pour quitter l'interpréteur, tapez CTRL + D (sous Unix) ou CTRL + Z (sous Windows).

Sous l'interpréteur, le résultat des instructions exécutées sont évaluées à chaque ligne. Exemple :

```
python
>>> print "Lui, le raton ivre."
Lui, le raton ivre.
>>> 2 ** 16 # 2 à la puissance 16
65536
```

3.2 Lancer un programme Python

Les programmes Python se terminent par l'extension .py (exemple : interpretor.py). Pour lancer un programme Python, tapez :

- 'python' + le nom du programme (avec l'extension)
- ou le nom du programme (toujours avec l'extension).

Exemple d'un programme affichant la chaîne de caractère *Lui, le raton ivre.*:

```
c:\>python raton.py
Lui, le raton ivre. # exemple sous Windows
c:\>
% ./raton.py
Lui, le raton ivre. # exemple sous Unix
%
```

[1] Sous windows, pour obtenir une console texte, il faut cliquer sur le menu **démarrer** ou **start** et taper **command** dans la case **executer**. Puis, appuyer sur la touche **enter** ou **return**.

4 Les types de données

Outre les types de données simples rencontrées dans beaucoup d'autres langages comme les chaînes de caractères, les entiers ou les flottants, Python dispose également de types de données complexes comme les listes ou les dictionnaires.

4.1 Les types de données numériques

Type	Notes	Exemples
Entiers	Normalement 32 bits	421 ; -12 ; 0
Entiers longs	Taille illimitée	6542218744664218654OL
Virgules flottantes	Taille entre E-308 et E+308. Intègrent soit un point, soit la lettre e ou E pour les puissances de dix.	1.23 ; 2E2 ; 6.0e-204
Nombres complexes	Taille illimitée. Possèdent une partie réelle et une partie imaginaire	5+7j ; 3+0j ; 4.5j
Nombres octaux et hexadécimaux	Les octaux commencent par '0' et les hexadécimaux par '0x'	0177 ; 0x4E2F

4.1.1 Quelques opérations sur les nombres

```
>>> abs(-2.2)           # Valeur absolue
2.2
>>> pow(2, 4)           # 2 à la puissance 4
16
>>> round(1.12345678, 3) # arrondir à 3 décimales
1.123
```

4.2 Les chaînes de caractères

4.2.1 Petite polémique sur les quotes et les guillemets

Faut il utiliser le terme *quotes*, le terme *guillemets* ou le terme *apostrophes* lorsque l'on parle de caractères comme ' ou " ?

En effet,

Note

1. Le guillemet n'est pas une double apostrophe. Essayez donc ceci:

```
>>> maChaine = ''ceci est ma chaîne''
SyntaxError: invalid syntax
```

Et oui, la deuxième apostrophe ferme la première. Ce n'est donc pas identique à :

```
>>> maChaine = "Ceci est ma chaîne"
>>>
```

2. Certains préconisent ceci:
    - guillemets simples 'x'
    - guillemets doubles "x"
    - guillemets triples """x"""

Ce qui est gênant dans l'histoire, c'est que:

- une apostrophe, ce n'est pas un guillemet (cfr exemple précédent)
- les guillemets doubles sont en fait simples
- les guillemets triples ne sont donc pas les 3/2 de guillemets doubles.

En fin de parcours, on ne sait plus si on parle de __vrais__ guillemets ou de guillemets qui veulent dire quotes, etc.

P3B a donc décidé d'utiliser le terme quote, qui, bien qu'anglophone, a le mérite d'être sans ambiguïtés.

4.2.2 Revenons à nos chaînes de caractères

Les chaînes de caractères, en Python, se placent entre simples quotes, doubles quotes ou triples quotes.

On utilise indifféremment les simples et les doubles quotes pour les textes assez courts. On peut placer des simples et doubles quotes entre des triples quotes.

```
une_simple_chaine = "Programmer en %s, c'est 'fun

un_calcul = "'Exemple : " %i * %i = %i' % (12, 12, 14

un_texte = """Ce texte peut comporter des "doubles
        quotes'. Les caractères d'échappement com
        ligne) ou \\t (tabulation) doivent commencer
```

4.2.3 Opérations sur les chaînes de caractères

```
>>> # concaténation de chaînes de caractères :
>>> 'Bobo' + ' casse' + ' la croûte'
Bobo casse la croûte
>>> len("Bobo")      # longueur de 'Bobo'
4
>>> 'Ni! ' * 4
Ni! Ni! Ni! Ni!
>>> alphabet = 'abcdef'
>>> alphabet[:1]
a
>>> alphabet[1:]
bcdef
>>> alphabet[-1:]
f
>>> alphabet[:-1]
abcde
>>> alphabet[2]
c
```

Attention : Les chaînes de caractères sont non modifiables :

```
>>> un_mot = "ortjographe"
>>> # essayons de remplacer ce 'j' en 'h' :
>>> un_mot[3] = 'h'
Traceback (innermost last):
File "<pyshell#11>", line 1, in ?
  un_mot[3] = 'h'
  TypeError: "object doesn't support item assignment
```

Pour modifier une chaîne, il faut en créer une nouvelle ou recréer la chaîne à modifier à partir de sous chaînes. Exemple :

```
>>> un_mot = "ortjographe"
>>> un_mot = un_mot[:3] + 'h' + un_mot[4:]
>>> un_mot
orthographe
```

4.3 Les listes

Les listes sont, en quelques sortes des tableaux séquentiels

à une dimension. Elles peuvent contenir toutes sortent d'objets différents, même d'autres listes (devenant ainsi des tableaux à plusieurs dimensions).

```
>>> maListe = ['un_m', 45 , [1, 2, 3]]
>>> print maListe
['un_mot', 45, [1, 2, 3]]
>>> print maListe[0]
un_mot
>>> print maListe[2]
[1, 2, 3]
>>> print maListe[2][0]   # élément 0 de l'élément 2
1
```

4.3.1 Opérations sur les listes

```
>>> taListe = ['t', 'a', 'r', 'a', 't', 'a', 't']
>>> taListe.append('a')
>>> print taListe
['t', 'a', 'r', 'a', 't', 'a', 't', 'a']
>>> print taListe.count('a')
# compte le nombre d'occurences de 'a' dans la liste
4
>>> print taListe.count('r')
1
>>> titi = [1,2,3]
>>> bubu = titi.reverse() # inverse l'ordre de titi.
>>> print bubu    # La méthode reverse ne retourne p
None      # elle inverse la liste sur place.
>>> print titi
[3, 2, 1]   # C'est donc bien titi qui a changé
```

Note

Le *point* dans les langages objets est un opérateur. Il permet d'appliquer une méthode à un objet. Ex : dans **taListe.append**, on applique la méthode append à un objet liste.

Les opérations vues précédemment sur les chaînes de caractères sont aussi supportées par les listes. Mais les listes sont modifiables.

4.3.2 La création fonctionnelle de listes

Python 2.0 a introduit un nouveau concept très intéressant: les mutations de listes ou création fonctionnelle de liste (list comprehension ou list mapping en anglais). C'est un moyen très pratique d'appliquer une

fonction sur chaque élément d'une liste afin d'en produire une nouvelle. Un exemple simple:

```
>>> maliste = [num*2 for num in range(1, 11)]
>>> print maListe
[2,4,6,8,10,12,14,16,18,20]
```

Qui est l'équivalent de:

```
>>> maliste2 = []
>>> for num in range(1, 11):
>>>...     maliste2.append(num*2)
```

L'idée est de pouvoir créer des listes d'une manière plus simple et sans passer par map, filter ou lambda. Ce qui permet de rendre le code plus clair.

Syntaxe à utiliser d'après la traduction du didacticiel de Guido, traduit par Daniel Calvelo Aros, Bruno Liénard et Olivier Berger:

Note

Chaque *list comprehension* consiste en une expression suivie d'une clause **for**, puis zéro ou plus clauses **for** ou **if**. Le résultat sera une liste résultant de l'évaluation de l'expression dans le contexte des clauses **for** et **if** qui la suivent. Si l'expression s'évalue en un tuple, elle doit être mise entre parenthèses.

Quelques exemples plus complexes:

La table de deux autrement:

```
>>> maliste2 = []
>>> for num in range(1, 11):
>>>...     maliste2.append(num*2)
```

Des fruits et des légumes:

```
>>> from pprint import pprint # 'pretty print' ou la jolie
>>> fruits = ['pommes', 'oranges', 'bananes']
>>> legumes = ['choux rouges', 'navets', 'carottes']
>>> pprint(["des %s et des %s" % (fruit, legume) for
>>> in legumes])
>>> ['des pommes et des choux rouges'
'des pommes et des navets'
'des pommes et des carottes'
'des oranges et des choux rouges'
'des oranges et des navets'
'des oranges et des carottes'
'des bananes et des choux rouges'
'des bananes et des navets'
'des bananes et des carottes']
```

Quelles sont les possibilités pour obtenir 6 avec 3 dés à 6 faces ?

```
>>> print [(x, y, z)\
... for x in range(1, 7)\
... for y in range(1, 7)\
... for z in range(1, 7)\
... if x + y + z == 6]
>>> [(1, 1, 4), (1, 2, 3), (1, 3, 2), (1, 4, 1), (2, 1, 3), (2
>>> 1), (3, 1, 2), (3, 2, 1), (4, 1, 1)]
```

L'algorithme du QuickSort:

```
def quicksort(lst):
    if len(lst) == 0:
        return []
    else:
        return quicksort([x for x in lst[1:] if x < lst[0]]) + [l
              quicksort([x for x in lst[1:] if x >= lst[0]])
```

4.4 Les tuples

Les tuples sont des listes non modifiables. Ils peuvent, comme les listes, inclure tout type d'objet. Opérations sur les tuples.

```
>>> mon_tuple = (12, 31, 5, 2)
>>> len(mon_tuple)    # longueur du tuple
4
>>> min(mon_tuple)    # plus petite valeur du tuple
2
>>> max(mon_tuple)    # plus grande valeur
31
```

Les tuples ne supportent pas les opérations permettant de modifier une séquence, puisque les tuples ne sont pas modifiables :

```
>>> mon_tuple.append(45)
Traceback (innermost last):
  File "", line 1, in ?
    mon_tuple.append(5)
AttributeError: '"tuple' object has no attribute 'append
```

4.5 Les dictionnaires

Aussi appelés 'hash-tables' ou tableaux associatifs dans d'autres langages, les dictionnaires sont des collections non ordonnées d'objets. A la différences des listes, tuples et chaînes de caractères, les dictionnaires ne sont pas accédés séquentiellement mais par clés.

```
>>> wiwi = {"sept":9, "oct":10, "nov":11}
>>> wiwi["oct"]
10
```

4.5.1 Opérations sur les dictionnaires

```
>>> len(wiwi)
3
>>> wiwi.keys()
['nov', 'oct', 'sept']
>>> wiwi.values()
[11, 10, 9]
>>> wiwi.has_key('oct')   # wiwi possède-t-elle la clé
1       # oui
>>> wiwi.has_key('dec')   # wiwi possède-t-elle la clé
0       # non
>>> wiwi['dec']=13    # assignation d'une nouvelle va
>>> wiwi.has_key('dec')
1
>>> del wiwi['oct']   # effacer une clé
>>> wiwi['dec']=12    # modifier une valeur
>>> wiwi
{'nov': 11, 'sept': 9, 'dec':12}
```

4.6 Récapitulatif

Type	Symbole	Modifiable ?	Accès
Chaîne de caractère	" " ou ' ' ou """ """ ou ''' '''	Non	Séquentiel
Listes	[ ]	Oui	Séquentiel
Tuples	( )	Non	Séquentiel
Dictionnaires	{ }	Oui	Par clé

Une question revient souvent lorsque l'on apprend Python : Pourquoi existe t-il des listes et des tuples ?

Les tuples, de par leur nature non modifiable, apportent une certaine sécurité. Si vous devez créer une liste tout en étant sûr qu'elle ne puisse pas être modifiée, créez plutôt un tuple. De plus, certaines opérations (comme *apply*) nécessitent des tuples et ne fonctionnent pas avec des listes.

Il existe d'autres types de données mais la majorité des programmes utilisent ceux-ci. Sachez qu'il existe aussi des tableaux normaux comme en C. Ils sont surtout utilisés dans le domaine mathématique.

5 Les spécificités de Python

5.1 Programmer en Python

La facilité avec laquelle on programme en Python est due à un certain nombre de ses caractéristiques. Outre les types de données évolués comme les dictionnaires et les listes,

citons :

- l'abscence de délimiteur de blocs.
- le typage dynamique.

5.1.1 Délimitation des blocs de code

En Python, les blocs de code ne sont pas délimités par des points-virgules ou des accolades comme dans d'autres langages comme le C, Java ou Perl, mais par l'indentation. Un exemple :

Vous le constatez, le code est beaucoup plus lisible et plus compact. Veillez à garder une indentation homogène : soit des tabulations, soit des espaces (toujours le même nombre).

```
def plusPetitPlusGrand(a, b):
    if a < b:
        print "%d est plus petit que %d" % (a, b)
    else:
        print "%d est plus grand que %d" % (a, b)

if __name__ == "__main__":
    print plusPetitPlusGrand(5, 12)
```

Vous le constatez, le code est beaucoup plus lisible et plus compact. Veillez à garder une indentation homogène : soit des tabulations, soit des espaces (toujours le même nombre).

Notez la petite astuce en avant dernière ligne qui permet d'exécuter le module comme programme principal mais qui permet aussi d'importer les fonctions et méthodes d'un module sans les exécuter (dans ce cas, le __name__ du module n'est pas exécuté en __main__). Notez que ce sont deux doubles underscores.

5.1.2 Typage dynamique

En Python, pas besoin de déclarer les variables avant de les utiliser. Cela élimine une contrainte souvent assez lourde dans d'autres langages mais implique plus de rigueur dans l'écriture du code.

```
>>> a = "vile rat noir élu."
>>> type(a)               # quel est le type de a ?
<type 'string'>          # a est une chaîne de caractère
>>> a = 1
>>> type(a)
<type 'int'>             # a est un entier
```

6 Les structures de contrôle

- les boucles WHILE
- les boucles FOR
- l'instruction de test : IF
- une instruction de choix : ELIF
- sortir d'une boucle : BREAK
- retourner au début de la boucle : l'instruction CONTINUE

6.1 Les boucles 'tant que' ou boucles WHILE

Elle effectue les instructions du corps de la boucle tant que le test de la boucle est évalué à VRAI. Le corps n'est pas effectué si le test est d'emblée FAUX.

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

counter = 10
while counter > 0:
  print counter
    counter = counter -1
```

Résultat:

```
10
9
8
7
6
5
4
3
2
1
```

6.2 Itération sur une séquence : la boucle FOR

La boucle FOR itère sur les membres d'une séquence (liste, chaîne de caractères, tuples,...).

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

for x in range(10, 0, -1):
  print x
```

Résultat :

```
10
9
8
7
6
5
4
3
2
1
```

La fonction range crée une liste d'entiers conformément au format suivant:

range(start, end, step) où start est le premier entier, (end+1) le dernier et step, le pas d'incrémentation. Notez que le dernier élément de la liste est 1 et non 0.

La plupart du temps, on itère sur une séquence déjà existante. Exemple:

```
ma_liste = ['abcd', 456, 'wxyz', 789, 123]
for element in ma_liste:
  print element
```

Résultat:

```
abcd
456
wxyz
789
123
```

6.3 IF : l'instruction de test

Les instructions du corps sont exécutées si le test est évalué à VRAI. Si le test est FAUX, ce sont les instructions du ELSE (s'il est présent) qui sont exécutées.

```
nbr = long(raw_input("Entrez un nombre\n"))
if abs(nbr) >= 2 ** 31:
  print "votre nombre est un LONG INTEGER"
else:
  print "votre nombre est un INTEGER"
```

Note

La fonction **raw_input** permet à l'utilisateur d'entrer une chaîne de caractères. **long()** convertit cette chaîne en LONG INTEGER et **abs()** retourne la valeur absolue du nombre.

6.4 ELIF, l'instruction de choix

Cette instructions signifie en fait '**ELSE-IF**' et correspond au **CASE** et **SWITCH** des autres langages de programmation. Très utiles lorsque l'on doit choisir entre plusieurs options possibles.

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

opt = raw_input("Votre choix ?\n")
if opt == 'a':
  # '==' est un opérateur de comparaison à la différer
  # de '=' qui est un opérateur d'assignation.
  # Confondre ces deux opérateurs est une source
  # d'erreurs très fréquente.
    print "votre choix est a"
elif opt == 'b':
    print "votre choix est b"
elif opt == 'c':
    print "votre choix est c"
elif opt == 'd':
    print "votre choix est d"
else:
    print "à bientôt"
```

6.5 Sortir d'une boucle : l'instruction BREAK

Petit exemple :

```
from time import *

while 1:
    print "-"*32
    print "t : afficher l'heure"
    print "j : afficher le jour"
    print "x : sortie du programme"
    print "-"*32
    opt = raw_input("Votre choix ?\n")
    if opt == 'x':
        print "adieu"
        break
    elif opt == 't':
        time_tuple = localtime(time())
        hour = str(time_tuple[3])
        min = str(time_tuple[4])
        sec = str(time_tuple[5])
        print hour + "H" + min + " et " + sec + "s"
    elif opt == 'j':
        time_tuple = localtime(time())
        day = str(time_tuple[2])
        month = str(time_tuple[1])
        year = str(time_tuple[0])
        print "nous sommes le " + day + " du " + month
```

Remarques concernant le programme :

from time import *: Nous verrons plus tard les importations de modules. En fait, on importe, ici, les méthodes du module time, gérant tous ce qui concerne le temps (dates, heures,...)

while 1: Crée une boucle infinie (puisque 1 est VRAI)

break: sort de la boucle while infinie et donc du programme

time_tuple = localtime(time()): time() renvoie le temps en secondes depuis l'EPOQUE, qui est en fait une date précise dépendant du système d'exploitation (Sur UNIX, l'EPOQUE est le 1er janvier 1970. localtime() convertit une date en seconde en un tuple représentant le temps local.

6.6 Retourner au début de la boucle : l'instruction CONTINUE

Exemple :

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

# créons une liste de -10 à 10
int_list = range(-10, 10 ,1)

for nbr in int_list:
  if nbr < 0:    # passe les éléments négatifs
    continue
  print nbr
```

7 Les fonctions

7.1 Définir une fonction

Le mot-clé def permet de définir une fonction. Comme exemple générique, créons une fonction qui multiplie deux valeurs:

```
def multiply(param_one, param_two):
  returnValue = param_one * param_two
  return returnValue   # valeur de retour de la fonctio
```

```
>>> multiply(3, 8)
    24
```

Une fonction n'est pas obligée de retourner une valeur. Dans d'autres langages, on appelle une telle fonction une procédure. Elle effectue une tâche mais ne retourne rien. Exemple:

```
def disBonjours():
    print "Bonjours"
```

```
>>> disBonjour()
    Bonjour
```

En fait, une fonction qui ne retourne pas explicitement une valeur retourne, en fait, la valeur **None**, qui est une valeur toujours nulle.

7.2 Paramètres par défaut

Les fonctions peuvent aussi avoir des paramètres par défaut. Modifions notre fonction pour qu'elle multiplie une valeur par 10 si un seul argument est passé.

```
def multiply(param_one, param_two=10):
    returnValue = param_one * param_two
    return returnValue
```

```
>>> multiply(3)
    30

>>> multiply(5, 3)
    15
```

7.3 Passage d'arguments par mots-clés

Lorsque le nombre d'arguments est plus important, il peut être utile de nommer les arguments de la fonction. Python a prévu ce cas de figure en laissant la possibilité au programmeur de transmettre des arguments en les nommant.

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

def printargs(x, y, z):
    print x, y, z

printargs(y='r', z=(12, 't'), x=2)
```

Résultat :

```
2 r (12, 't')
```

7.4 Nombres d'arguments variables

En Python, les fonctions peuvent aussi accepter un nombre d'arguments variable. Il existe deux façons de procéder. La première consiste à accepter les arguments supplémentaires dans un tuple:

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

def addition(a, *args):
    ret = a
    for x in args:
        ret = ret + x
    return ret

print addition(5, 5, 5)   # a = 5 et args = (5, 5)
```

Résultat :

```
15
```

La seconde méthode consiste à transmettre les arguments dans un dictionnaire. Mais comme on dit parfois à l'école, vous verrez ça par vous même.

8 Importation de modules

Comme dans les autres langages de programmation, vous pouvez utiliser des méthodes et des classes définies dans d'autres fichiers. Il faut donc importer ces fichiers (appelé modules en Python) dans votre programme avec le mot-clé **import**

Il existe deux façons d'importer un module en Python:

8.1 Importer un module sans son espace de nom

Cela signifie que, pour utiliser une méthode du module importé, vous devrez précéder la méthode du nom du module. En clair, voici comment procédez pour utiliser la méthode **getcwd()** du module **os**.

Voici la syntaxe à utiliser:

import **<module>**

Note

Le module **os** contient les services couvrant du système d'exploitation. La méthode **getcwd()** récupère le répertoire de travail. getcwd = 'get current working directory'

```
>>> import os
>>> print os.getcwd()
   /home/oli/python
```

8.2 Importer un module avec son espace de nom

Si vous voulez que les objets importés se retrouvent directement dans votre espace de noms, voici la syntaxe à utiliser:

from **<module>** import **<method>**

Dans ce cas, il n'y a plus besoin d'utiliser le nom du module pour appeler une méthode.

```
>>> from os import getcwd
>>> print getcwd()
   /home/oli/python
```

Mais veillez à ce que le nom de la méthode d'un module ne soit pas identique au nom de la méthode d'un autre module. exemple : la méthode **open** qui existe en standard et la méthode **open** du module os. Dans ce cas, préférez la première solution.

Si vous devez utiliser plusieurs méthodes d'un même module, vous pouvez importer toutes les méthodes du module comme ceci:

from **<module>** import *

On peut mélanger les deux solutions suivant l'utilité :

```
import sys, os, cgi
from stat import *
from time import localtime
```

Important : pour importer un module, il faut qu'il soit dans le **PYTHONPATH**.

Qu'est-ce donc que cela ? Le PYTHONPATH est une variable d'environnement composées de tous les chemins vers les répertoires contenant les modules Python.

Sous DOS et Windows, Vous pouvez déclarer le PYTHONPATH dans le fichier autoexec.bat en lui rajoutant, par exemple, une ligne ressemblant à celle-ci : set PYTHONPATH=c:pythonLib;c:pythonlib-tk;... où <c:python> est le répertoire où est installé Python. La liste des répertoires est, comme vous pouvez le constater, séparée par des points virgules.

9 Les exceptions

Le exceptions permettent d'intercepter certaines erreurs, évitant ainsi au programme de planter. Par exemple, un mécanisme d'exception peut intercepter une division par zéro qui, sans cela, ferait planter le programme en laissant un message d'erreur:

```
>>>  12/0
Traceback (innermost last):
  File "<pyshell#1>", line 1, in ?
    print 12/0
ZeroDivisionError: integer division or modulo
>>>
```

Pour intercepter cette erreur, voici une manière simplifiée de procéder:

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

def division(a, b):
    ret = a/b
    return ret

try:
    print division(12,0)
except ZeroDivisionError:
    print "attention : division par zéro"
```

La clause **else** permet de recevoir du code qui sera exécuté si l'exception définie par **except** n'est pas déclenchée. Voici un petit programme qui accepte des caractères entrés au clavier par l'utilisateur et les convertit en majuscules. Pour en sortir, il faut faire CTRL + C qui normalement déclencherait une KeyboardInterrupt Exception ou une EOFError suivant les environnements et les conditions d'utilisation..

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

from string import upper

while 1:
    try:
        line = raw_input()
    except (KeyboardInterrupt, EOFError):
        print "Adios"
        break
    else:
        print upper(line)
```

Il existe aussi une clause **finally** qui permet d'exécuter du code, qu'une exception soit déclenchée ou non.

La clause **raise**, quant à elle, permet de déclencher une exception. Sachez enfin que l'on peut créer soit même les exceptions, mais vous verrez bien çà par vous même.

10 Les expressions lambda

L'expression **lambda** permet de définir une fonction anonyme. C'est une expression et non une instruction. Par conséquent, on peut placer une expression lambda là où on ne pourrait pas placer une instruction def.

Attention!

On ne peut pas placer d'instructions dans une expression lambda.

La syntaxe d'une expression lambda est la suivante:

lambda paramètre1, paramètre2, ..., paramètreX: **<expression de retour>**.

Voici un petit exemple :

```
>>> x = lambda a, b, c: a + b + c
>>> x(1, 2, 3)
6
```

L'équivalent de cet exemple en utilisant une instruction def donnerait ceci:

```
def x(a, b, c):
    valeurDeRetour = a + b + c
    return valeurDeRetour
```

On peut affecter des valeurs par défaut aux paramètres.
Exemple:

```
>>> x = lambda a, b, c=0: a + b + c
>>> x(1, 2)
3
```

11 L'instruction Map

L'instruction **map** applique une fonction passée en argument à un objet à accès séquentiel (liste, tuple, ...).

La syntaxe est:

map(<fonction>, <sequence>)

Exemple:

```
def func(a):
    return a * 3

print map(func, (1,2,3,4,5,6))
```

Résultat:

```
[3, 6, 9, 12, 15, 18]
```

On peut coupler l'instruction map avec une expression lambda:

```
>>> map((lambda a: a * 3), (1, 2, 3, 4, 5, 6))
[3, 6, 9, 12, 15, 18]
>>>
```

map est très pratique pour itérer sur les valeurs et clés d'un dictionnaire:

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

month = {'Jan': 'Janvier', 'Feb': 'Février', 'Mar': 'Mars',
        'Apr': 'Avril', 'May': 'Mai', 'jun': 'Juin',
        'Jul': 'Juillet', 'Aug': 'Août', 'Sept': 'Septembre',
        'Oct': 'Octobre', 'Nov': 'Novembre', 'Dec': 'Décer

for m in map(None, month.keys(), month.values()):
    print m
```

Résultat:

```
('Oct', 'Octobre')
('Sept', 'Septembre')
('Dec', 'Décembre')
('Nov', 'Novembre')
('Apr', 'Avril')
('Jan', 'Janvier')
('Jul', 'Juillet')
('jun', 'Juin')
('May', 'Mai')
('Aug', 'Août')
('Feb', 'Février')
('Mar', 'Mars')
```

12 L'instruction Apply

Apply invoque une fonction indirectement avec un tuple ou un dictionnaire comme paramètres. L'avantage étant de pouvoir construire la liste de paramètres sans la connaître au départ.

La syntaxe de apply est la suivante :

apply(<fonction>, [(<tuple>), [<dictionnaire>]])

Note

Le tuple ou le dictionnaire sont optionnels (pour les fonctions sans arguments).

Voici un petit exemple :

```
def func(a, b):
    ret = a + b
    return ret

print apply(func, (5, 6))
```

Résultat:

```
11
```

13 Les décorateurs de fonctions

Les *decorators* sont sont une nouvelle fonctionnalité disponible depuis la version 2.4 de Python. Ils permettent de décorer une fonction. Ce sont en fait des fonctions qui prennent une fonction en paramètre et retourne cette fonction (modifiée par le décorateur) ou, éventuellement, un autre objet.

En fait, il existait déjà des décorateurs disponibles avec la version 2.2 de Python: **staticmethod** et **classmethod**

staticmethod permettait de déclarer une méthode comme étant statique. L'avantage étant de pouvoir appeler cette méthode sans créer d'instance de cette classe.

classmethod permettait de déclarer une méthode comme étant une méthode de classe.

Ils étaient appelé comme ceci:

```
class A:
   def meth(arg):
      return arg.upper()
   meth = staticmethod(meth)
```

Le problème venait du fait que la décoration devait se faire après la définition de la méthode. Dans le cas de longue méthode, cela était problématique car il fallait accéder à la fin de la méthode avant de savoir que cette méthode était décorée.

Python 2.4 introduit une nouvelle façon de déclarer cette décoration.

Voici l'équivalent de la décoration précédente avec la syntaxe Python 2.4:

```
class A:
   @staticmethod
   def meth(arg):
      return arg.upper()
```

Comme vous le voyez, nous savons maintenant que cette

méthode est décorée dès le début. Plus besoin d'aller à la fin de la méthode.

Mais quel peut être l'utilité de décorer des méthodes, mis à part définir une méthode statique ou de classe ?

13.1 Voici quelques exemples

13.1.1 Vérifier le type des arguments de la méthode (méthode simple):

```
def require_int (func):
    def wrapper (arg):
        assert isinstance(arg, int)
        return func(arg)
    return wrapper

@require_int
def meth (arg):
    print arg
```

13.1.2 Vérifier le type des arguments de la méthode (méthode plus complexe):

Voici une méthode plus complexe et plus flexible de vérification du type des arguments:

```
def require(*types):
    """Return a decorator function that requires specif
       types -- tuple each element of which is a type or
                several types or classes.
       Example to require a string then a numeric argu
       @require(str, (int, long, float))
       will do the trick"""
    def deco(func):
        """Decorator function to be returned from requir
           wrapper that validates argument types."""
        def wrapper (*args):
            """Function wrapper that checks argument ty
            assert len(args) == len(types), 'Wrong numbe
            for a, t in zip(args, types):
                if type(t) == type(()):
                    # any of these types are ok
                    msg = """%s is not a valid type. Valid typ
                    assert sum(isinstance(a, tp) for tp in t) >
                assert isinstance(a, t), '%s is not a %s type
            return func(*args)
        return wrapper
    return deco

@require(int)
def inter(int_val):
    print 'int_val is ', int_val

@require(float)
def floater(f_val):
    print 'f_val is ', f_val

@require(str, (int, long, float))
def nameAge1(name, age):
    print '%s is %s years old' % (name, age)

# another way to do the same thing
number = (int, float, long)
@require(str, number)
def nameAge2(name, age):
    print '%s is %s years old' % (name, age)

nameAge1('Emily', 8)       # str, int ok
nameAge1('Elizabeth', 4.5) # str, float ok
nameAge2('Romita', 9L)     # str, long ok
nameAge2('Emily', 'eight') # raises an exception!
```

13.1.3 Afficher les arguments passés à une fonction

Lorsque la fonction est appelée, les arguments passés à la

fonction sont affichés:

```
def dumpArgs(func):
    "This decorator dumps out the arguments passed
    argnames = func.func_code.co_varnames[:func.fu
    fname = func.func_name
    def echoFunc(*args,**kwargs):
        print fname,":", ', '.join('%s=%r' % entry
                                   for entry in zip(argnames,args)
        return func(*args, **kwargs)
    return echoFunc

@dumpArgs
def f1(a,b,c):
    print a + b + c

f1(1, 2, 3)
```