PYTHON

PROG#PY

PYTHON, POUR

Installation de Python sous windows

Installer activepython:
? downloader activePython :
http://www.activestate.com/Products/ActivePython/
? Si vous utilisez windows 95,98 ou Me, vous devrez aussi télécharger et installer Windows Installer 2.0 depuis : http://download.microsoft.com/download/WindowsInstaller/Install/2.0/W9XMe/EN-US/InstMsiA.exe
? double cliquez sur ActivePython-X-win32-ix86.msi
? suivez les indications

Quand l'installation est terminée, fermez l'installeur et allez sur :
démarrer -> Programmes -> ActiveState ActivePython X -> PythonWin IDE .

Installer Python depuis Python.org :
http://www.python.org/ftp/python/2.4.2/python-2.4.2-pdb.zip
? téléchargez la dernière version (.exe bien sur)
? décompressez python-2.4.2-pdb.zip
? double cliquez sur Python-2.4.2.exe
? suivez les indications
? si vous n'avez pas les droits administrateurs sur votre machine, vous pouvez sélectionner dans Advanced options la ligne « Non-Admin Install» .

Quand l'installation est terminée, fermez l'installeur et allez surprises
démarrer -> Programmes -> Python-2.4.2 -> IDLE (Python GUI)

La première chose que vous devez faire avant d'utiliser python est de l'installer ;-) . Dans la plupart des distributions Linux, il est installé par défaut. A partir de Mac OS X 10.2, python en ligne de commande est présent. Sous Windows, vous devrez l'installer vous même, ah Windows quand tu nous tiens ... :-(

Python sous linux
Redhat :
Connectez vous en root .

```
[FaSm]# wget
http://python.org/ftp/python/2.4.2/rpms/fedora-4/python24-2.4.2-2.i386.rpm
[FaSm]# rpm -Uvh python24-2.4.2-2.i386.rpm
```

Debian :

```
[FaSm]# apt-get install python
```

Jouons un peu avec Python

Vous pouvez maintenant lancer python en ligne de commande .

```
[FaSm:/home/fasm]#python
Python 2.3.5 (#2, Sep  4 2005, 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
```

YTHON

R COMMENCER

2

En mode console, on peut effectuer differentes 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
```

Note :

ActiveState propose ActivePython qui comprend une version compléte de python , une interface graphique et des extensions pour Windows qui permettent un accés complet à des services spécifiques , des API et à la base de registres. ActivePython est gratuit sans pour autant être open source.

Dans les trois lignes précedentes, nous avons affécté 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 debrouille seul.

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

Si vous utilisez type() vous pouvez voir que X est un int (entier) essayons d'affecter une chaine de caractére 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 derniére ligne.
Nous savons maintenant déclarer des variables (il n'y a rien a 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

deuxiéme 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 parenthéses, 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 problémes 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 préfére utiliser l'instruction raw_input(), instruction qui renvoie toujours une chaîne de caractére. 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
>>> 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 caractére entrée dans x est transformée en int avant d'être multiplié par 12.

Notre premier script en Python

Utilisons notre éditeur favori pour écrire le script (j'utilise vim mais vous pouvez utiliser nano, kate ...).

Vous pouvez aussi tester eric. Ne vous retournez pas sur votre copain eric en lui faisant les yeux doux pour tenter de le tester, eric est un environnement graphique de programmation. (apt-get install eric).

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'
```

ecrivez ce script et enregistrez le sous script.py par exemple.
Rendez le éxecutable (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 premiére ligne.
Si vous ne mettez pas cette ligne, vous devrez taper :

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

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.

BY FASM

PYTHON, LES

Pour ceux qui connaissent déjà un peu un autre langage de programmation, rien de bien nouveau, vous allez retrouver « vos petits » , pour les autres, let's go.

Et si on y allait ?

Il nous faut souvent dans un programme tester une condition. Par exemple, on demande à l'utilisateur d'entrer un nombre et si le nombre est 3, on affiche « tu as gagné » aussi non, « tu as perdu ».
Essayons d'écrire le script suivant :

```
#!/usr/bin/env python
"""Test du if par FaSm"""
from random import *
msg=raw_input('entrez un nombre compris entre 0 et 10\n')
a=randrange(0,10,1)
print 'le nombre tire aleatoirement est',a
if (int(msg)==a):
        print "\nvous avez gagne"
else:
        print "\nvous avez perdu"
```

Quelques notions nouvelles ici.
Une phrase est écrite entre """, celle ci ne sera par prise en compte dans le programme , elle nous sert pour écrire des commentaires.
Dans le script, j'utilise l'instruction randrange pour générer un nombre aléatoire allant de 0 à 10 . Le premier argument de cette fonction est le nombre de départ, le

Nous allons maintenant nous intéresser aux tests et boucles. Que serait un programme sans répétitivités, sans tests ?

deuxiéme argument, le nombre d' arrivée et le dernier argument, le pas. On retouvera donc dans la variable à un nombre aléatoire compris entre 0 et 10 . Mais cette fonction fait partie d'un module séparé de python, il faut donc dire au démarrage du script les modules dont nous allons avoir besoin. Ceci se fait garce à :

```
from random import *
```

Cela signifie que nous importons tout du module random.
Pour ne pas tout charger, nous aurions pu utiliser :

```
from random import randrange
```

Nous arrivons maintenant au but de ce paragraphe, le IF.
Dans les parenthéses, nous effectuons le test , ici égalité (==). Il ne faut surtout pas oublier les « : » aprés le if ainsi qu'aprés le else.
Une chose trés importante, nous ne retrouvons pas dans le « bloc » if, de debut et fin de bloc comme { et } en C. Ici l'indentation est trés importante, c'est ce qui défini les blocs.
On pourra aussi trouver des blocs if de la

PYTHON

LES BOUCLES

sorte :

```
if a!=2:
   print 'perdu'
elif a==3:
   print 'peut etre gagne'
else:
   print 'gagne'
```

On pourra mettre ici autant de elif que désiré.

Note

Les tests:
== égalité
!= différent
>=supérieur ou égal
<=inférieur ou égal
< strictement inférieur
> strictement supérieur

tant que Python...

l'instruction while est très fréquente en python.
Essayez le script suivant:

```
#!/usr/bin/env python
"""Test du While par FaSm"""
msg=raw_input('entrez un nombre compris entre 0 et 10\n')
i=0
while (i< int(msg)):
      print "le nombre ",i,"est plus petit que votre nombre",int(msg)
      i=i+1
print "le programme est termine"
```

Nous retrouvons l'instruction while qui va éffectuer les opérations qui le suivent (avec l'indentation) tant que le nombre entré au clavier est plus grand que i. i est incrémenté de 1 à chaque passage dans la boucle.
Le test n'est pas obligé d'être seul, on peut par exemple encadrer deux valeurs :

```
#!/usr/bin/env python
"""Test du While par FaSm"""
msg=raw_input('entrez un nombre compris entre 0 et 10\n')
msg2=raw_input('entrez un autre nombre compris entre 0 et 10\n')
i=5
j=5
while (int(msg)<j or i> int(msg2)):
      print "le nombre ",i,"est plus grand que votre nombre",int(msg)
      print "le nombre ",j,"est plus petit que votre nombre",int(msg2)
      i=i+1
      j=j-1
print "le programme est termine"
```

Nous voyons dans ce petit script modifié que dans le while, il y a deux test, un sur j et l'autre sur i. On retrouve un or (ou logique) entre les deux c'est à dire que pour aller dans la boucle il faut que int(msg)<j ou bien que i> int(msg2).
Mini encadré : Note
Les opérateurs logiques:
~x NON bit à bit
x&y ET bit à bit

du tout , mais
ne peut avoir
on qui n'est pas
» peut servir
ce vide.

on

s maintenant
rogrammes «
avec des tests
les. Il faudrait
s petits pro-
e nous avons
réutilisables.
vais vous mon-
rochain article
ndre ceux ci
par d'autres

BY FASM

```
x^y         OU EXCLUSIF bit à bit
x|y         OU bit à bit
not x       NON booléen
x and y     ET booléen
x or y      OU booléen
```

pour le Python ?

La boucle for est assez simple en python, pas besoin d'incrémenter un compteur quelconque,regardez l'exemple suivant:

```
#!/usr/bin/env python
"""Test du for par FaSm"""
test=[3,4,5,6,7,8,9,10,11,12,13,14,15]
for nb in test:
      print "le nombre est",nb
print "le programme est termine"
```

Nous reviendrons dans un prochain article sur les listes, il suffit ici de savoir que dans test, nous avons placé une liste de nombres.
La boucle for de notre script va balayer toute la liste et placer chaque nombre dans la variable nb et cette boucle va se terminer quand le balayage sera arrivé au dernier nombre de la liste.
Pas de grosses difficultés ici.
Dans l'exemple qui suit, nous n'utilisons plus une liste mais une chaîne de caractére:

```
#!/usr/bin/env python
"""Test du for par FaSm"""
msg='Thehackademy  '
for car in msg:
      print car+'-',
print "\nle programme est termine"
```

vous obtenez le résultat suivant :

```
[ FaSm:/home/fasm/articles/mag_python]#
./art2_4.py
T- h- e- h- a- c- k- a- d- e- m- y-
le programme est termine
```

Rien de bien différent pour la boucle for, une nouveauté par contre pour le print, j'ai utilisé ici la concaténation (+) et la « , » pour que le mot s'écrive sur la même ligne .

Sortir des boucles

L'instruction break fait quitter la plus proche boucle englobante while ou for en évitant aussi l'instruction else associée, si elle existe.

L'instruction continue permet de repasser immédiatement au début d'une boucle while ou for.

```
While y<6:
    if y==4:
        break
    else:
        print 'dans cette boucle y vaut ',y
        y=y+1

while z<6:
    z=z+1
    if z==4:
        continue
    else:
        print 'dans cette boucle z vaut',z
```

Grâce au break, on sort du while sans exécuter le else.
Grâce au continue, on revient au départ et on continue, le bloc else sera exécuté.
On peut aussi utiliser l'instruction pass qui

PROG PYTHON : LISTES, TUPLES ET DICTIONNAIRES

Maintenant que nous avons un interpréteur Python fonctionnel et que nous connaissons les principaux rouages de la programmation (les boucles et conditions), nous pouvons commencer à traiter des informations. Et les traitements sont nombreux : tri dans un sens, dans un autre, selon un critère ou selon plusieurs, ajout au milieu, au début ou à la fin, ajout sans modification de l'ordre, ... Dans ces quelques cas parmis les plus simples, vous conviendrez que les simples « types » de variables que nous ne connaissons pour l'instant ne suffisent pas. Et pour ordonner, insérer ou ôter des données, rien de tel qu'un tableau. Voici quelques structures intéressantes.

LA PLUS SIMPLE

Dans sa plus commune apparence, le tableau est également appelé LISTE ou VECTEUR. Son utilisation est très simple. Tout d'abord, nous devons, à l'inverse de ce qui n'est pas nécessaire dans d'autre type de variables avec Python, initialiser une nouvelle liste, ainsi :

```
ma_liste = []
```

Notez que, comme en C, le crochet est le symbole désignant le tableau. Cette analogie entre les deux langages se retrouvera souvent, Python étant programmé en C.
Notre tableau est maintenant initialisé mais ne contient rien. Pour constater cela, il vous suffit de taper « print ma_liste » et de constater le résultat : « [] ». Nous allons devoir lui ajouter des valeurs. Pour ce faire, et vous comprendrez la construction un peu plus loin dans ce manuel, nous allons faire appel à ce qu'on appelle une méthode de la classe LIST (voir l'article sur les classes et objets). Cette méthode se nomme « append » et on l'utilise simplement ainsi : « ma_liste.append(3) » pour ajouter la valeur numérique 3 dans la dernière case du tableau (en ajoutant une case). En faisant cela, notre tableau fait donc 1 case. « print ma_liste » nous affiche tout le vecteur. Pour n'afficher qu'un élément, il faut accéder directement à celui-ci, grâce à ce qu'on nomme son indice (ou index, en anglais). Par exemple, pour afficher la première case du tableau, il nous

suffit de taper « print ma_liste[0] ». Point important qu'il faudra retenir : la numérotation des indices de tableaux débute à 0, et non à 1. Pour avoir accès à la nième case, il faudra donc toujours accéder à l'indice numéro
n-1.

J'ai choisi ici d'insérer la valeur 3, une valeur numérique. Plutôt que 3, j'aurais pu ajouter « Python » comme ceci : « ma_liste.append('Python') ». Mon tableau serait alors un tableau de chaînes de caractères plutôt qu'un tableau d'entiers. Maintenant, et au risque de choquer les habitués du C, le langage Python à l'intéressante particularité de n'être que très faiblement typé; ce qui nous permet ici de faire cela :

```
ma_liste.append(3)
ma_liste.append(« Python »)
```

Et d'avoir un tableau contenant des éléments de types hétérogènes. Plus encore, imaginez que vous pouvez avoir un tableau de tableaux, ce qui nous permet de représenter par exemple un tableau à plusieurs dimensions. Par exemple :

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

Ce tableau est un vecteur de 2 colonnes, 4 lignes qui contient du texte (en-tête de colonnes A et B) et des valeurs numériques représentant un court échantillon de nombres élevés au carré. Maintenant, puisqu'il est question de traitements sur les données, il faut pouvoir les trier et les dénombrer. Partons de l'exemple :

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

Nous avons définit là un tableau, que nous avons rempli à l'initialisation. Ce qui nous évite d'avoir à taper de récurrents « tab.append() », et allège le code par la même occasion. Ce tableau, comme vous pouvez vous-même le compter, contient 10 valeurs (accessibles aux indices allant de 0 à 9 donc). Pour ne pas avoir à faire cela nous même, nous faisons appel à une fonction générique de Python (dont l'existence ne relève pas exclusivement des listes), la fonction « len » (pour l'anglais LENGHT, longueur), bien plus appréciée que la loi du même nom. 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 cependant, dans le cas de vecteurs contenant d'autres vecteurs (tableaux multidimensionnels), vous n'aurez que le nombre de cases du « super vecteur » (par exemple, len(tableau_2d) ci-dessus devrait renvoyer 4, soit le nombre de lignes). A noter que dans le cas d'une chaîne, cette même fonction renvoie le nombre de caractères, ce qui peut être utile.

De plus, vous voyez ici la concaténation de deux listes, qui se fait grâce à l'opérateur +, et qui va simplement adjoindre les deux listes.

Autre fonction parfois utile, que nous abordons dès maintenant tant que votre liste n'est pas triée. La méthode reverse() va vous permettre d'inverser la liste, c'est à dire de faire passer la première case à la place de la dernière, la seconde a celle de l'avant dernière, ... etc.

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

Rien de bien spécial, donc. Ensuite, nous allons trier ce tableau. Pas à la main, ni en créant une fonction qui va recopier les valeurs dans un autre tableau temporaire, non, tout cela a déjà été fait pour vous. Nous allons, comme ci-dessus, faire appel à une méthode, qui s'appelle « sort ».

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

Très simplement, en une ligne, Python peut trier pour vous des tableaux très long selon des méthodes de tri très rapides (dichotomie ou quick Sort). Si par défaut l'ordre de tri est l'ordre croissant, vous pouvez l'inverser en ajoutant un paramètre à la méthode sort().

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

Cela aurait pu se faire par l'appel successif à sort() puis à reverse(), mais la possibilité de le faire en une fois supprime une ligne de code, et les algorithmes de tris fonctionnent aussi bien dans un ordre que dans l'autre.

Si d'aventure j'avais besoin de savoir le nombre d'occurrences d'un terme bien précis dans mon tableau (par exemple combien de fois apparaît la valeur 2), il existe la encore une méthode qui me simplifie la vie : la méthode count().

```
print tab.count(2)
>> 2
print tab.index(2)
>> 1
```

J'introduis aussi la méthode index(), qui retourne l'index de la première occurrence trouvée de l'argument. Il n'existe pas de méthode déjà écrite pour trouver tous les indexes des différentes occurrences d'un même motif, mais cela est, à ce stade du magazine, déjà réalisable :

```
//initialisation de mon tableau
tab = [1,2,10,9,2,35,24,32,17,28]
//je teste la présence de « 2 »
nb_occur = tab.count(2)

//tan qu'il y des deux
while nb_occur >= 0 :
        premier_index = tab.index(2)
        print 'occurrence trouvé à l'in-
dexe' premier_index
        taille_tab = len(tab)
```

```
        tab =
tab[premier_index+1:taille_tab-1]
        nb_occur = tab.count(2)
```

Tout d'abord, j'initialise mon tableau en le remplissant. Puis, je vais aller chercher l'index de la première occurrence du nombre 2. Ensuite, et c'est le but de la construction tab[x:y], je vais « rogner » mon tableau et n'en garder qu'une partie, entre les index x et y. Pour nous, x est l'index suivant la première occurrence de 2, et y est la fin du tableau (dernier index = nombre d'éléments – 1). Et je recommence ainsi tant que le rognage du tableau contient des occurrences.
Veuillez noter que ce script marche dans cet exemple précis. Pour être fonctionnel avec tous les tableaux, vous devez, et je vous en laisse le loisir, intégrer quelques tests pour vérifier que vous ne tentez pas d'accéder à des index inexistants, ce qui pourrait provoquer des erreurs (des erreurs de type « out of range », que l'article sur la gestion des exceptions vous apprendra a gérer).

Notre tableau contient deux fois la valeur 2. Or, j'aimerai que celle-ci n'apparaisse qu'une fois. Je pourrais localiser une occurrence de 2, et remplacer dans le tableau par un vide ou un autre nombre, ce qui apparaît comme une première solution pour retirer le doublon. Maintenant, expliquez aux statisticiens que pour ces raisons, vous avez inséré une valeur au hasard, et examinons vos chances de survie ;-). De plus, cela ne supprime pas la case du tableau, et en mémoire, cela peut être considéré comme important si l'on parle de plusieurs milliers de suppressions. Pour retirer proprement une valeur, on utilise remove(), qui supprime la première occurrence d'une valeur, et qui s'utilise ainsi :

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

Passons maintenant dans des conditions de production : un tableau de 9 cases, trié, qui, mis en production, va atteindre sa taille de croisière de plusieurs milliers de cases. L'ordre, à l'intérieur de ce tableau, est important. On pourrait, à chaque ajout de valeur, se faire succéder les appels à tab.append(valeur), puis à tab.sort(), ce qui aurait pour effet d'insérer une valeur puis de trier le tableau au complet ensuite. Pure perte de temps sachant que toutes les valeurs sont ordonnées, hors mise la dernière, sauf si elle est supérieure (respectivement inférieure) à toutes les valeurs d'un tableau trié par ordre croissant (décroissant). Travaillons sur un tableau trié par ordre croissant.

```
tab = [1, 2, 9, 10, 17, 24, 28, 32, 35]

valeur_ajout = 15
taille_tab = len(tab)

for i in range(taille_tab-1) :
        if tab[i] > valeur_ajout :
                tab.insert(valeur_ajout,i)
                break
```

Voici l'explication. J'ai mon tableau de 9 valeurs. Je vais le parcourir avec une boucle

Python

FOR, dont l'itérateur « i » est initialisé comme faisant partie de l'intervalle allant de 0 à « taille du tableau - 1 » (pour couvrir les indices du tableau). A chaque itération, je teste la valeur du tableau à l'indexe i. Si cette valeur est supérieure à la valeur que l'on veut insérer, ça veut dire que je dois insérer cette dernière avant cet index, pour conserver l'ordre de tri. C'est ce que fait la fonction insert(), qui prend en premier paramètre l'indexe AVANT lequel elle doit ajouter la valeur, passée en second paramètre. Pensez à inverser les choses si vous travaillez sur une liste triée par ordre décroissant.

Dernière note enfin, si vous comptez vous servir de votre tableau comme d'une pile (méthode LIFO, Last In First Out, pensez à la pile d'assiettes dans la cuisine), la méthode pop() vous sera utile. Elle vous permet de dépiler le dernier élément de la liste, c'est à dire, en une opération, de vous envoyer sa valeur et de supprimer la case correspondante.

```
tab = [2, 2, 2, 3, 3, 2]
tab.pop()
>> 2
print tab
>> [2, 2, 2, 3, 3]
```

Les tuples

Petit paragraphe sur les tuples qui n'ont pas de traits particulièrement intéressants, sinon que d'être plus léger que les listes en termes de ressources. Néanmoins ils ne permettent ni l'insertion ni la modification d'éléments. On les dits immuables car leurs tailles et leurs éléments sont fixes dès leurs initialisations. On les utilise le plus souvent à la volée, en paramètre à des fonctions qui ne feront qu'utiliser les valeurs qu'ils contiennent.
L'accès à ces valeurs se fait comme pour les listes. Par exemple :

```
a = (1,2,3)
b = (7, (7,2), [3,4], "Tuples")
print a[1]
>> 2
print b[2:3]
>> ([3, 4], 'Tuples')
```

Les parenthèses remplacent les crochets, et les méthodes des listes ne s'appliquent pas au tuples. C'est tout ce qu'on peut en dire. Passons à bien plus intéressant : les dictionnaires.

Les dictionnaires

Les dictionnaires sont ce qu'on appelle dans d'autres langages des tableaux associatifs. Dans un format de tableau conventionnel, vous associez une valeur à un index. Dans le cas des dictionnaires, c'est un peu différent : vous associez une valeur (ou, comme dans les autres tableaux, un tableau, une liste, ...) à une clef, représentée par une chaîne de caractère. Voyons un peu la construction d'un dictionnaire qui associera au nom des mois leur numéro correspondant ;

```
monthes = {     '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 (dans le but de

le remplir après) se fait simplement : dico = {}. Pour les dictionnaires, les accolades remplacent les crochets que l'on utilise pour les tableaux. Mais l'initialisation en est aussi simple : on liste les clefs et les valeurs associées, que l'on sépare par deux points. Une limite se fait sentir ici : on ne peut associer qu'une chose à une clef. Mais il s'agit là d'une limite facilement dépassable : pensez que vous pouvez associer, à une clef, un nombre, une chaîne de caractères, un tableau, un objet (voir article correspondant).

Pour accéder à la valeur, on utilise la construction tableau['clef']. Ici, pour savoir quel est le numéro du mois d'août, on procède ainsi :

```
print monthes['août']
>> 8
```

La construction est la même pour l'ajout ou le réglage d'une valeur. Par exemple, vous l'aurez peut-être constaté, deux erreurs se sont glissées dans le tableau ci-dessus : j'ai oublié le mois d'octobre, et par conséquent, je n'ai pas pris garde et ai mal réglé les numéros de novembre et de décembre. Voici « le patch » :

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

Vous constaterez peut-être des caractères étranges au moment de l'affichage de la dernière ligne. Ceci me permet de vous mettre en garde en ce qui concerne l'utilisation des accents : Python est un langage disponible sur plusieurs OS (Windows, Linux et autres). Or, tous ces OS ne gèrent pas les caractères de la même manière, et parfois même un OS fait des conversions implicites au moment de la saisie de données, mais n'intervient pas au moment de l'affichage, ce qui peut provoquer l'affichage de caractères exotiques (ou de codes de table ASCII par exemple). Les accents sont à manipuler avec précaution donc (surtout dans le cas de programme destiné à plusieurs OS sans retouche, ce que gère particulièrement bien Python, à l'instar de Java).
Pour notre dictionnaire, vous constatez que de la même manière que je modifie la valeur associée à une clef, j'ajoute un couple (clef, valeur) correspondant au mois d'Octobre. Python interprète la tentative de mise à jour d'une clef inexistante comme un ajout. Mais comment fait donc Python pour savoir si la clef existe? Et bien, nous disposons d'une fonction bien sympathique qui est has_key() et que l'on utilise ainsi :

```
dico = { 'Lundi' : 1, 'Mardi' : 2, 'Mercredi' :
3, 'Jeudi' : 4, 'Vendredi' : 5, 'Samedi' : 6,
'Dimanche' : 7 }
print dico.has_key('Jeudi')
>> True
if dico.has_key('Mardi') :
        print dico.get('Mardi',None)
>> 2
if dico.has_key('Novembre') :
        print dico.get('Novembre',None)
>> (RIEN)
```

Python

Nous testons donc très simplement la présence d'une clef, et si tel est le cas, nous pouvons en récupérer la valeur associée. Pour ce faire, nous utilisons la méthode get(), avec deux paramètres : la clef dont on veut la valeur, et ce qu'il faut retourner, ici rien (c'est l'utilisation la plus courante de get()). Notez que le test est dans ce cas inutile, car get() se charge de faire le test et de ne rien retourner dans le cas où la clef n'existerai pas.

Pour récupérer tout ou partie du dictionnaire, nous disposons ensuite d'outils :

```
dico = { 'Lundi' : 1, 'Mardi' : 2, 'Mercredi' :
3, 'Jeudi' : 4, 'Vendredi' : 5, 'Samedi' : 6,
'Dimanche' : 7 }
print dico.items()
>>[('Mardi', 2), ('Samedi', 6), ('Vendredi', 5),
('Jeudi', 4), ('Lundi', 1), ('Dimanche', 7),
('Mercredi', 3)]
print dico.keys()
>> ['Mardi', 'Samedi', 'Vendredi', 'Jeudi',
'Lundi', 'Dimanche', 'Mercredi']
print dico.values()
>> [2, 6, 5, 4, 1, 7, 3]
```

La première, items(), nous renvoie un TABLEAU contenant les TUPLES clef/valeurs.
La seconde, keys(), nous renvoie un vecteur contenant les clefs du dictionnaire.
La troisième, values, renvoie un vecteur contenant les valeurs du dictionnaire.

Pour supprimer un élément, nous pouvons utiliser pop(), qui comme pour les listes, va dépiler un élément. Avec les dictionnaires, nous dépilons un élément en fonction de sa clef. Vous l'aurez constaté, 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 les accès à celles-ci. Inutile donc de vouloir dépiler le dernier élément.

```
print dico.pop('Mardi',None)
>> 2
```

Pour une simple suppression, il suffit de ne pas traiter la valeur de retour de pop. Et dans l'élan, tentez donc de faire appel à dico.clear()....

Conclusion

Voilà, vous avez tout ce qu'il faut pour bien débuter en Python : un interpréteur, des éléments de contrôle et des structures pour vos données. Il existe d'autres choses à connaître sur les types basiques que sont les tuples, listes et dictionnaires. Il s'agit en effet d'un sujet vaste et qui surtout, pour des raisons de performance, et encore ouvert à l'évolution. Dans votre interpréteur, je vous invite à taper help(list), help(tuple) et help(dict) pour avoir un bref aperçu de ce que sont, en plus, capables de faire ces différents outils (et vous découvrirez beaucoup d'autres méthodes dont on dit qu'elles sont héritées). Vous voilà capable de créer un dictionnaire qui a un matricule associe un tableau contenant le nom, le prénom, l'adresse et le numéro de téléphone d'une personne, et de vous éblouir devant votre premier répertoire électronique dont vous êtes l'auteur.

BY KORETH

PYTHON ET LES EXPRE

Introduction aux expressions régulières

De l'origine des expressions régulières

Comprendre les mots et jongler avec ceux-ci est une chose impossible pour nos ordinateurs. Paradoxal, me direz-vous, quand on saît que ces derniers sont à l'origine faits pour traiter des informations. Pour remédier à cela, il a fallu penser un outil aussi puissant que les traitements sont complexes, pour comprendre des objets comme des mots et effectuer dessus des traitements : les regex sont nées.

Dit, c'est quoi une regex ?

Une expression régulière, ou regex (pour Regular Expressions) pour les intimes 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 chaîne 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

Les expressions régulières sont certainement l'outil le plus puissant dont dispose un programmeur qui souhaite traiter des informations génériques (ce qui, de mémoire, doit être l'origine de l'informatique). Voyons un peu l'utilisation des regex en Python.

d'abord de chaîne de caractères, ce qui se passe de commentaires, sinon celui que de se rappeller qu'une chaîne de caractère peut être une suite de chiffre, ou une chaine 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.

PYTHON

EXPRESSIONS REGULIERES

A quoi ça sert, une regex?

Une expression régulière peut avoir, grossièrement, trois fonctions : tester la présence d'une chaîne correspondant à une expression, récuperer une chaîne correspondant à une expression et remplacer du texte correspondant à une expression.

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.

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),

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'oeil, 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 diffère du brin précé-
```

```
demment identifié (AGTCTGATCCAG)"
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 désoxyribonu-
cléique extrait est composé ainsi :
GCTATCGTAC. Il diffère du brin précé-
demment identifié (AGTCTGATCCAG)"
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 :",occu-
rences
```

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 chaine qui correspond à une regex par une autre chaine. Voici la manière de proceder :

```
import re
PHRASE = "Le brin d'acide désoxyribonu-
cléique extrait est composé ainsi :
GCTATCGTAC"
my_first_regex = re.compile(r"[ACGT]+")
my_frist_regex.sub("extrait d'ADN confi-
dentiel",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 chaine passée en second argument, toute chaine correspondant à l'expression régulière par le premier argument (ici, notre message).

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).

rou-lez-Jacques, en
thon permettrait de
entre parenthèses?
s ont une significa-
bonus (de cet enca-
e mauvaise expres-
s (de cet encadré)
kademy.net) », qui

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 : rappellez-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)

BY KORETH

PYTHON

PYTHON , LES FONCTIONS

By

Fonction kesako ?

Quand on veut créer une application, on part d'un cahier des charges qui décrit le fonctionnement du futur programme. On décompose donc ce cahier des charges en plusieurs sous-problémes qui peuvent être étudiés séparement et donc développés séparemment. Ces « petits bouts » du projets peuvent devenir des sous programmes ou fonctions. Pourquoi ainsi décomposer ? Parce que ces fonctions pourront être réutilisées pour d 'autres programmes ou être utilisées plusieurs fois dans le programme.

Note :
Structure générale d'une fonction :

```
def nom_fonction( parametres à transmettres):
        ligne du programme
        ligne du programme
        ...
```

On peut utiliser les fonctions de différentes maniéres : sans transmettre de paramètres, avec transmission de paramètres, sans rien retourner ou en retournant une ou des valeurs.

Une fonction est comme un petit programme que vous pouvez utiliser pour effectuer une action spécifique.Python a une multitude de fonctions pour effectuer des choses magnifiques. Mais vous pouvez créer vos propres fonction. C'est ce que nous allons aborder dans cet article.

les fonction en détail.

Fonctions sans paramêtres:

Nous voulons réaliser un décompteur de 10 à 0 et qui nous indique « fini » une fois arrivé à 0.
Nous n'aurons donc pas de paramètres à transmettre et aucune valeur de retour.Afin de voir le décomptage, nous importons le module time afin d'utiliser la commande sleep pour faire une pause entre deux affichages.
Pour connaitre toutes les fonctions du module import, nous pouvons donc faire dir(time) et help(time.sleep) pour le détail de chaque fonction (ici sleep).

Note :

```
import time
def decompte():
```

```
        i=10
        while (i>0):
            print "\n",i
            i=i-1
            time.sleep(1.0)
        print "fini"

decompte()
```

Vous voyez ici que pour appeler une fonction, il suffit de donner son nom. N'oubliez surtout pas les « : » derriére la définition et bien sur l'indentation qui détermine les lignes appartenant à la fonction.
Et si on pouvait décompter à partir de n'importe quel nombre ? Voyons la suite.
Fonctions avec paramètres:
On peut donc essayer de transmettre des paramètres à la fonction précédente.

Note :

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

a=input("donnez le nombre de
depart\n")
decompte(a)
```

On remarque ici que l'on demande à l'utilisateur de donner un nombre de départ grâce à input. Cette valeur est affectée à la variable à qui est transmise à la fonction decompte().

Dans la fonction decompte(), on affecte a, donc la valeur donnée à par l'utilisateur, à la variable i. Le reste du programme est identique au paragraphe précédent.
On peut bien sur transmettre plusieurs paramètres à une fonction. On voudrait maintenant pouvoir choisir le nombre de départ, d'arrivée et le pas de décomptage. Il nous faut donc transmettre trois paramètres. La méthode est la même que la précédente pour un paramètre . Chaque valeur transmise sera séparée par une virgule.

Note :

```
#!/usr/bin/env python
import time
def decompte(a,b,c):
        i=a
        j=b
        z=c
        while (i>j):
            print "\n",i
            i=i-z
            time.sleep(1.0)
        print "fini"
a=input("donnez le nombre de
depart\n")
b=input("donnez le nombre de fin\n")
c=input("donnez le pas de decomp-
tage\n")
decompte(a,b,c)
```

On peut aussi définir des valeurs par défaut de chaque variables transmises.
Si une seule variable est transmise (au lieu de trois), les deux autres auront les valeurs définient par défaut.

Note

```
#!/usr/bin/env python
import time
def decompte(a,b=0,c=1):
        i=a
        j=b
        z=c
        while (i>j):
            print "\n",i
            i=i-z
            time.sleep(1.0)
        print "fini"
a=input("donnez le nombre de
depart\n")
b=input("donnez le nombre de fin\n")
c=input("donnez le pas de decomp-
tage\n")
decompte(a,b,c)
decompte(a)
```

Pour le premier appel à la fonction (decompte(a,b,c)), on aura a définir les trois variables. Pour le deuxiéme appel, seul la variable a, préalablement entrée sera prise en compte. Les variables b et c seront celles par défaut.

Conclusion :

Savoir programmer des fonctions est très important surtout dans de gros projets. En effet, nous verrons dans l'article suivant comment créer des classes. Les classes sont une succession de fonctions que nous verrons dans l'article suivant.

PYTHON : LA

Tout le monde connait bien la licence GPL, il existe quelque chose de similiare la FDL (GNU Free Documentation License ou Licence de documentation libre GNU). L'objet de cette Licence est de rendre tout manuel, livre ou autre document écrit « libre » au sens de la liberté d'utilisation, à savoir : assurer à chacun la liberté effective de le copier ou de le redistribuer, avec ou sans modifications, commercialement ou non. En outre, cette Licence garantit à l'auteur et à l'éditeur la reconnaissance de leur travail, sans qu'ils soient pour autant considérés comme responsables des modifications réalisées par des tiers.
Cette Licence est une sorte de « copyleft », ce qui signifie que les travaux dérivés du document d'origine sont eux-mêmes « libres » selon les mêmes termes. Elle complète la Licence Publique Générale GNU, qui est également une Licence copyleft, conçue pour les logiciels libres.
Cet article est donc tiré du livre FDL de Gérard Swinnen
(http://www.framasoft.net/article1971.html) .

les classes : la base

Utilité des classes et définition

Les classes sont les principaux outils de la programmation orientée objet (Object Oriented Programming ou OOP) qui permettent de structurer les logiciels complexes en les organisant comme des ensembles d'objets qui interagissent, entre eux et avec le monde extérieur.
Le premier bénéfice de cette approche de la programmation consiste dans le fait que les différents objets utilisés peuvent être construits indépendamment les uns des autres sans qu'il n'y ait de risque d'interférence. Ce résultat est obtenu grâce au concept d'encapsulation : la fonctionnalité interne de l'objet et les variables qu'il utilise pour effectuer son travail, sont en quelque sorte « enfermés » dans l'objet. Les autres objets et le monde extérieur ne peuvent y avoir accès qu'à travers des procédures bien définies.
La programmation orientée objet est optionnelle sous Python. Vous pouvez donc mener à bien de nombreux projets sans l'utiliser, avec des outils plus simples tels que les fonctions. Sachez cependant que les classes constituent des outils pratiques et puissants.

Pour créer une nouvelle classe d'objets Python, on utilise l'instruction class.
Les définitions de classes peuvent être

PYTHON

A CLASSE !!

situées n'importe où dans un programme, mais on les placera en général au début (ou bien dans un module à importer).

- L'instruction class est un nouvel exemple d'instruction composée. N'oubliez pas le double point obligatoire à la fin de la ligne, et l'indentation du bloc d'instructions qui suit. Ce bloc doit contenir au moins une ligne.
- l'instruction class est une chaîne de caractères, celle-ci sera considérée comme un commentaire et incorporée automatiquement dans un dispositif de documentation des classes qui fait partie intégrante de Python.

Attention : comme les fonctions, les classes auxquelles on fait appel dans une instruction doivent toujours être accompagnées de parenthèses (même si aucun argument n'est transmis). Nous verrons un peu plus loin que les classes peuvent être appelées avec des arguments.

Exemple de classe

```
class Personne:
    def  Init(self,nom):
                self.nom=nom

        def getNom(self):
                return self.nom
        def parle(self):
                print "bonjour, je suis
%s"%self.nom
```

Cette premiére classe peut déjà être utilisée :

```
>>> moi = Personne()
>>> moi.Init('FaSm')
>>>moi.parle()
bonjour, je suis FaSm
```

Nous avons ici créé un objet (moi) par instanciation. Nous pouvons dés à présent créer des composants à cet objet par simple assignation en utilisant le systéme de qualification des noms par points. Dans l'exemple précédent, nous demandons donc d'aller chercher la méthode parle de l'objet moi (donc Personne).

Classes,métodes,héritage

Il nous faut à présent doter les classes d'une fonctionnalité. L'idée de base de la programmation orientée objet consiste en effet à regrouper dans un même ensemble (l'objet) à la fois un certain nombre de données (ce sont les attributs d'instance) et les algorithmes destinés à effectuer divers traitements sur ces données (ce sont les méthodes, c'est-à-dire des fonctions encapsulées).

Objet = [attributs + méthodes]

Cette façon d'associer dans une même « capsule » les propriétés d'un objet et les fonctions qui permettent d'agir sur elles, correspond chez les concepteurs de programmes à une volonté de construire des entités informatiques dont le comportement se rapproche du comportement des objets du monde réel qui nous entoure.

Considérons par exemple un widget « bouton ». Il nous paraît raisonnable de souhaiter que l'objet informatique que nous appelons ainsi ait un comportement qui ressemble à celui d'un bouton d'appareil quelconque dans le monde réel. Or la fonctionnalité d'un bouton réel (sa capacité de fermer ou d'ouvrir un circuit électrique) est bien intégrée dans l'objet lui-même (au même titre que d'autres propriétés telles que sa taille, sa couleur, etc.) De la même manière, nous souhaiterons que les différentes caractéristiques de notre bouton logiciel (sa taille, son emplacement, sa couleur, le texte qu'il supporte), mais aussi la définition de ce qui se passe lorsque l'on effectue différentes actions de la souris sur ce bouton, soient regroupés dans une entité bien précise à l'intérieur du programme, de manière telle qu'il n'y ait pas de confusion avec un autre bouton ou d'autres entités.

Définition d'une méthode

Pour illustrer notre propos, nous allons définir une nouvelle classe Time, qui nous permettra d'effectuer toute une série d'opérations sur des instants, des durées, etc. :

```
>>> class Time:
        "Définition d'une classe temporelle"
```

Créons à présent un objet de ce type, et ajoutons-lui des variables d'instance pour mémoriser les heures, minutes et secondes :

```
>>> instant = Time()
>>> instant.heure = 11
>>> instant.minute = 34
>>> instant.seconde = 25
```

A titre d'exercice, écrivez maintenant vous-même une fonction affiche_heure() , qui serve à visualiser le contenu d'un objet de classe Time() sous la forme conventionnelle « heure:minute:seconde ».
Appliquée à l'objet instant créé ci-dessus, cette fonction devrait donc afficher 11:34:25 :

```
>>> print affiche_heure(instant)
11:34:25
```

Votre fonction ressemblera probablement à ceci :

```
>>> def affiche_heure(t):
        print str(t.heure) + ":" +
str(t.minute) + ":" + str(t.seconde)
```

(Notez au passage l'utilisation de la fonction str() pour convertir les données numériques en chaînes de caractères). Si par la suite vous utilisez fréquemment des objets de la classe Time(), il y a gros à parier que cette fonction d'affichage vous sera fréquemment utile.
Il serait donc probablement fort judicieux d'encapsuler cette fonction affiche_heure() dans la classe Time() elle-même, de manière à s'assurer qu'elle soit toujours automatiquement disponible chaque fois que l'on doit manipuler des objets de la classe Time().
Une fonction qui est ainsi encapsulée dans une classe s'appelle une méthode.
Vous avez déjà rencontré des méthodes à de nombreuses reprises (et vous savez donc déjà qu'une méthode est bien une fonction associée à une classe d'objets).

Définition concrète d'une méthode :
On définit une méthode comme on définit

une fonction, avec cependant deux différences :

- La définition d'une méthode est toujours placée à l'intérieur de la définition d'une classe, de manière à ce que la relation qui lie la méthode à la classe soit clairement établie.
- Le premier paramètre utilisé par une méthode doit toujours être une référence d'instance.

Vous pourriez en principe utiliser un nom de variable quelconque pour ce paramètre, mais il est vivement conseillé de respecter la convention qui consiste à toujours lui donner le nom : self.

Le paramètre self désigne donc l'instance à laquelle la méthode sera associée, dans les instructions faisant partie de la définition. (De ce fait, la définition d'une méthode comporte toujours au moins un paramètre, alors que la définition d'une fonction peut n'en comporter aucun).

Voyons comment cela se passe en pratique : Pour ré-écrire la fonction affiche_heure() comme une méthode de la classe Time(), il nous suffit de déplacer sa définition à l'intérieur de celle de la classe, et de changer le nom de son paramètre :

```
>>> class Time:
        "Nouvelle classe temporelle"
        def affiche_heure(self):
            print str(self.heure) + ":" +
str(self.minute) \
                  + ":" + str(self.seconde)
```

La définition de la méthode fait maintenant partie du bloc d'instructions indentées après l'instruction class. Notez bien l'utilisation du mot réservé self , qui se réfère donc à toute instance susceptible d'être créée à partir de cette classe.

(Note : Le code \ permet de continuer une instruction trop longue sur la ligne suivante).

Essai de la méthode dans une instance

Nous pouvons dès à présent instancier un objet de notre nouvelle classe Time() :

```
>>> maintenant = Time()
```

Si nous essayons d'utiliser un peu trop vite notre nouvelle méthode, ça ne marche pas :

```
>>> maintenant.affiche_heure()
AttributeError: 'Time' instance has no
attribute 'heure'
```

C'est normal : nous n'avons pas encore créé les attributs d'instance. Il faudrait faire par exemple :

```
>>> maintenant.heure = 13
>>> maintenant.minute = 34
>>> maintenant.seconde = 21
>>> maintenant.affiche_heure()
13:34:21
```

Nous avons cependant déjà signalé à plusieurs reprises qu'il n'est pas recommandable de créer ainsi les attributs d'instance en dehors de l'objet lui-même, ce qui conduit (entre autres désagréments) à des erreurs comme celle que nous venons de rencontrer, par exemple.

Voyons donc à présent comment nous pouvons mieux faire.

La méthode « constructeur »

L'erreur que nous avons rencontrée au paragraphe précédent est-elle évitable ?.

Elle ne se produirait effectivement pas, si nous nous étions arrangés pour que la méthode affiche_heure() puisse toujours

asses. Il reste
dire mais ce
iste énorme-
rnet qui vous
classes. Mais
us permettra
nmation en

SWINNEN,
AR FASM

afficher quelque chose, sans qu'il ne soit nécessaire d'effectuer au préalable aucune manipulation sur l'objet nouvellement créé. En d'autres termes, il serait judicieux que les variables d'instance soient prédéfinies elles aussi à l'intérieur de la classe, avec pour chacune d'elles une valeur « par défaut ».

Pour obtenir cela, nous allons faire appel à une méthode particulière, que l'on appelle un constructeur. Une méthode constructeur est une méthode qui est exécutée automatiquement lorsque l'on instancie un nouvel objet à partir de la classe. On peut y placer tout ce qui semble nécessaire pour initialiser automatiquement l'objet que l'on crée. Sous Python, la méthode constructeur doit obligatoirement s'appeler __init__ (deux caractères « souligné », le mot init, puis encore deux caractères « souligné »).

Exemple :

```
>>> class Time:
        "Encore une nouvelle classe temporelle"
        def __init__(self):
            self.heure =0
            self.minute =0
            self.seconde =0

        def affiche_heure(self):
            print str(self.heure) + ":" +
str(self.minute) \
                + ":" + str(self.seconde)

>>> tstart = Time()
>>> tstart.affiche_heure()
0:0:0
```

L'intérêt de cette technique apparaîtra plus clairement si nous ajoutons encore quelque chose. Comme toute méthode qui se respecte, la méthode __init__() peut être dotée de paramètres. Ceux-ci vont jouer un rôle important, parce qu'ils vont permettre d'instancier un objet et d'initialiser certaines de ses variables d'instance, en une seule opération. Dans l'exemple ci-dessus, veuillez donc modifier la définition de la méthode __init__() comme suit :

```
        def __init__(self, hh =0, mm =0, ss
=0):
            self.heure = hh
            self.minute = mm
            self.seconde = ss
```

La méthode __init__() comporte à présent 3 paramètres, avec pour chacun une valeur par défaut. Pour lui transmettre les arguments correspondants, il suffit de placer ceux-ci dans les parenthèses qui accompagnent le nom de la classe, lorsque l'on écrit l'instruction d'instanciation du nouvel objet.

Voici par exemple la création et l'initialisation simultanées d'un nouvel objet Time() :

```
>>> recreation = Time(10, 15, 18)
>>> recreation.affiche_heure()
10:15:18
```

Puisque les variables d'instance possèdent maintenant des valeurs par défaut, nous pouvons aussi bien créer de tels objets Time() en omettant un ou plusieurs arguments :

```
>>> rentree = Time(10, 30)
>>> rentree.affiche_heure()
10:30:0
```

PYTHON : GESTION DES EXCEPTIONS.

Quoi de plus embêtant d'utiliser un programme qui pour une raison ou une autre, lors d'un fonctionnement supposé normal, nous envoi des tas d'injures sur l'écran du style « ZeroDivisionError : integer division or modulo by zero », alors que, prévoyant, nous aurions pu anticiper cette erreur et la gérer ?
Les exceptions sont faites pour cela et la simplicité d'utilisation en est troublante ;-) ...
Cet article est donc tiré du livre FDL de Gérard Swinnen
(http://www.framasoft.net/article1971.html) .

les exceptions

Les exceptions sont les opérations qu'effectue un interpréteur ou un compilateur lorsqu'une erreur est détectée au cours de l'exécution d'un programme. En règle générale, l'exécution du programme est alors interrompue, et un message d'erreur plus ou moins explicite est affiché.
Exemple :

```
>>> print 55/0
ZeroDivisionError: integer division or
modulo
```

(D'autres informations complémentaires sont affichées, qui indiquent notamment à quel endroit du script l'erreur a été détectée, mais nous ne les reproduisons pas ici). Le message d'erreur proprement dit comporte deux parties séparées par un double point : d'abord le type d'erreur, et ensuite une information spécifique de cette erreur.

Dans de nombreux cas, il est possible de prévoir à l'avance certaines des erreurs qui risquent de se produire à tel ou tel endroit du programme, et d'inclure à cet endroit des instructions particulières, qui seront activées seulement si ces erreurs se produisent. Dans les langages de niveau élevé comme Python, il est également possible

d'associer un mécanisme de surveillance à tout un ensemble d'instructions, et donc de simplifier le traitement des erreurs qui peuvent se produire dans n'importe laquelle de ces instructions.
Un mécanisme de ce type s'appelle en général mécanisme de traitement des exceptions. Celui de Python utilise l'ensemble d'instructions try - except – else , qui permettent d'intercepter une erreur et d'exécuter une portion de script spécifique de cette erreur. Il fonctionne comme suit :
Le bloc d'instructions qui suit directement une instruction try est exécuté par Python sous réserve. Si une erreur survient pendant l'exécution de l'une de ces instructions, alors Python annule cette instruction fautive et exécute à sa place le code inclus dans le bloc qui suit l'instruction except. Si aucune erreur ne s'est produite dans les instructions qui suivent try, alors c'est le bloc qui suit l'instruction else qui est exécuté (si cette instruction est présente). Dans tous les cas, l'exécution du programme peut se poursuivre ensuite avec les instructions ultérieures.

Considérons par exemple un script qui demande à l'utilisateur d'entrer un nom de fichier, lequel fichier étant destiné à être ouvert en lecture. Si le fichier n'existe pas, nous ne voulons pas que le programme se « plante ». Nous voulons qu'un avertissement soit affiché, et éventuellement que l'utilisateur puisse essayer d'entrer un autre nom.

```
filename = raw_input("Veuillez entrer un
nom de fichier : ")
try:
    f = open(filename, "r")
except:
    print "Le fichier", filename, "est introu-
vable"
```

Si nous estimons que ce genre de test est susceptible de rendre service à plusieurs endroits d'un programme, nous pouvons aussi l'inclure dans une fonction :

```
def existe(fname):
    try:
        f = open(fname,'r')
        f.close()
        return 1
```

```
    except:
        return 0

filename = raw_input("Veuillez entrer le
nom du fichier : ")
if existe(filename):
    print "Ce fichier existe bel et bien."
else:
    print "Le fichier", filename, "est introu-
vable."
```

Il est également possible de faire suivre l'instruction try de plusieurs blocs except, chacun d'entre eux traitant un type d'erreur spécifique, mais nous ne développerons pas ces compléments ici. Veuillez consulter un ouvrage de référence sur Python si nécessaire.

la simplicité des exceptions

Utiliser les exceptions n'est pas trés compliqué. Si vous savez qu'une certaine partie de votre programme risque de générer des exceptions et que vous ne voulez pas voir apparaitre des messages d'erreurs intempestifs, vous allez utiliser nécessairement try/except ou try/finally.
Des choses qui normalement s'ecrirait avec un if/else peuvent être des fois, mieux implémentées en utilisant un try/except.
Regardons cela avec un petit exemple.

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

si vous utilisez cette fonction avec un dictionnaire contenant le nom CodeJ et l'âge 88 ans (vous ne saviez pas qu'il était si vieux !! ;-)) mais sans hobby, vous obtiendrez l'affichage suivant :

```
Description de CodeJ
Age: 88
```

Si vous ajouter comme hobby ' chasseur d'escargots', vous obtenez :

```
Description de CodeJ
Age: 88
son hobby : chasseur d'escargots
```

le code est intuitif mais non efficace : le code doit aller verifier deux fois la clé 'hobby', une fois pour voir si la clé existe et une fois pour aller chercher la valeur.
L'alternative est d'utiliser try/except :

```
def decrit_personne(personne):
    print 'Description de',
personne['nom']
    print 'Age:',personne['age']
    try: print 'son hobby :',person['hobby']
    except KeyError: pass
```

Conclusion :

Voila, nous avons fait un tour des possibiltés de Python. A ce stade, j'espére que vous êtes convaincus de la simplicité d'utilisation, de la rapidité et de l'intuitivité de ce langage. Nous allons maintenant essayer de trouver un moyen pour interragir avec l'environnement, c'est à dire essayer de se transmettre des données pour analyser les erreurs par exemple. Quoi de plus simple que d'utiliser les fichiers ?

PROG#PY

PYTHON, LES I

il est souvent utile dans un programme d'utiliser des données se trouvant dans un fichier , d'écrire le résultat d'opérations dans le fichier. C'est aussi parfois un moyen d'échange entre programme . Quoi qu'il en soit, savoir manipuler les fichiers s'avére nécessaire, alors let's go !

Ouvrir un fichier

Vous pouvez ouvrir un fichier avec la fonction open .
La fonction open() attend deux arguments qui doivent être des chaînes de caractéres. Le premier argument doit être le nom du fichier à ouvrir et le second est le mode d'ouverture.

open()

```
>>>f=open('/home/fasm/passwd.txt','r')
```

Si le fichier n'existe pas vous aurez l'injure suivante :

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IOError: [Errno 2] No such file or directory: '/home/fasm/passwd.txt'
```

mode

'r'	mode lecture
'w'	mode ecriture
'a'	mode ajout
'b'	mode binaire
'+'	lecture/ecriture

Une fois le fichier ouvert, on peut ecrire, lire (suivant le mode d'ouverture) , il faudra aussi, bien sur, fermer le fichier en fin d'utilisation.

Fermer le fichier.

Vous devez vous rappeler de fermer votre fichier en appelant la fonction close(). Normalement, l'objet fichier sera fermé automatiquement quand vous quitterez le programme et ne pas fermer le fichier dans lequel vous avez lu n'est pas réellement important. Mais vous devez impérativement fermer le fichier quand vous avez écrit dedans. En effet Python bufferise les données que vous voulez écrire et si votre programme plante pour une raison quelconque, vous pouvez perdre vos données. Si vous voulez être certain que votre fichier est fermé , vous pouvez utiliser le duo try / finally comme vu dans l'article sur les exceptions.

close()

```
f=open('/home/fasm/passwd.txt','w')
try:
      #vous ecrivez ici dans le fichier
finally:
       f.close()
```

PYTHON

FICHIERS

Lire et ecrire dans un fichier.

On peut écrire dans un fichier avec la méthode f.write (si votre objet fichier s'appelle f) et lire dans le fichier grâce à la méthode f.read.
A chaque fois que vous allez appeler la méthode f.write(phrase), la chaîne de caractére va s'inscrire dans le fichier à la suite de ce que vous avez écrit précedemment.

readline()

```
>>>f=open('/home/fasm/passwd.txt','r')
>>>f.readline()
'bonjour'
>>>f.readline()
'the hackademy maubeuge'
```

write()

```
>>>f=open('/home/fasm/passwd.txt','w')
>>>f.write('bonjour')
>>>f.write('the hackademy maubeuge')
>>>f.close()
```

maintenant on peut aller vérifier ce que contient le fichier :

```
[FaSm]$ more passwd.txt
bonjour
the hackademy maubeuge
```

read()

```
>>>f=open('/home/fasm/passwd.txt','r')
>>>f.read(4)
'bonj'
>>>f.read()
'our the hackademy maubeuge'
```

La lecture dans un fichier est aussi simple que l'écriture, vous pouvez juste ajouter le nombre d'octets que vous désirez lire.

readline()

```
>>>f=open('/home/fasm/passwd.txt','r')
>>>f.readline()
'bonjour'
>>>f.readline()
'the hackademy maubeuge'
```

En premier nous avons spécifié que nous voulions lire 4 caractéres et ensuite nous avons demandé de lire le reste.
Nous avons donc vu que l'on peut lire ou écrire caractére par caractére mais on peut aussi lire ligne par ligne grâce à la méthode f.readline.

Avec cette méthode on peut imaginer compter les lignes d'un fichier par exemple.

Accés aléatoire.

Dans les explications précédentes, on ne pouvait qu'écrire au début d'un fichier ou à

la suite d'autres lignes. Il peut être intéressant de pouvoir écrire aléatoirement dans un fichier.
Nous verrons donc ici une autre méthode appelée seek(offset[,whence]). Cette méthode change la position courante par une position décrite par offset et whence. Offset est un nombre d'octets (donc de caractéres) et whence qui est à 0 par défaut (offset par rapport au début du fichier) peut être à 1 (offset par rapport à la position courante).

Seek()

```
>>>f=open('/home/fasm/passwd.txt','w')
>>>f.write('01234567890123456789')
>>>f.seek(5)
>>>f.write('hackademy')
>>>f.close()
>>>f=open('/home/fasm/passwd.txt','r')
>>>f.read()
'01234hackademy456789'
```

Il existe aussi la méthode f.tell() qui retourne la position courante.

les itérateurs et les fichiers.

Depuis la version 2.2 de python , on peut utiliser directement les fichiers dans les boucles for . On peut donc utiliser les iterators pour parcourir les lignes d'un fichier.

fichier et for

```
f=open('/home/fasm/passwd.txt','w')
for ligne in f:
    #suite du programme
```

Voyons une autre méthode dans l'encadré nommé exemple.

Exemple

```
>>>f=open('/home/fasm/passwd.txt','w')
>>>print >> f, 'la premiere ligne'
>>>print >> f, 'la deuxieme ligne'
>>>print >> f, 'la troisieme ligne'
>>>f.close()
>>>premiere,deuxieme,troisieme=open
('/home/fasm/passwd.txt')
>>>premiere
'la premiere ligne\n'
>>>deuxieme
'la deuxieme ligne\n'
>>>troisieme
'la troisieme ligne\n'
```

Dans l'exemple de l'encadré, j'ai utilisé print pour écrire dans le fichier, chaque ligne ajoutée l'est à la suite.
Ensuite, toutes les lignes sont mises dans trois variables (premiere, deuxieme et troisieme), ce qui peut être utile quand on ne connait pas le nombre de ligne (sinon, on aurait du créer une variable par ligne).
Et pour finir on ferme le fichier pour être sûr que les données soient bien enregistrées dans le fichier.

Conclusion :

Vous savez maintenant comment interagir avec l'environnement au travers de fichiers. Pour finir cette premiére partie du mag, nous allons essayer d'aborder une notion importante mais parfois difficile à comprendre : les threads. Alors retroussez vos manches, allez chercher une tasse de café et apprêtez vous à vous faire bouillir le cerveau ;-)

LES THREADS EN PYTHON

PREMIERE APPROCHE DES PROCESSUS

Aborder les threads sans comprendre ce qu'est un processus semble impossible: on va construire la définition d'un thread sur les différences qu'il va posséder avec un processus. Un processus est une tâche qui s'exécute, avec un espace mémoire, une pile, des données, des marqueurs comme le registre compteur (EIP, pour les connaisseurs) ou la liste des fichiers ouvert. Tout programme qui tourne est un processus. Voici comment on peut s'amuser simplement avec les processus :

```
import os
import time

variable = "PROCESSUS"

pid = os.fork()

if pid == 0 :
        print "Je suis le fils. Je vais vivre
deux secondes puis mourir"
        time.sleep(2)
        print "La variable VARIABLE avait
pour valeur",variable
        print "La variable VARIABLE
prend FILS pour valeur"
        variable = "Fils"
        print "Je meurt. -- Signé : le fils"
else :
        wait()
        print "La valeur de VARIABLE
est",variable
        print "J'étais père mais mon fils
est mort. Goodbye ...."
```

Les threads sont un sujet important dans bon nombre de langage. Bien que leur importance et leur existence sur Python soit un peu limitée, il en existe une implémentation totalement fonctionnelle pour ce langage. Voyons d'abord ce qu'est un thread, pour ne perdre personne en route, pour ensuite passer sur des exemples concrets.

Partons dans l'explication de ce code. Tout d'abord, j'importe les modules os et time, qui vont respectivement me permettre de manier mes processus et le temps. Jusque là rien de bien méchant.

Ensuite, je fait appel à la fonction os.fork(), donc la fonction fork() du module os. Elle va, à partir de mon programme, créer un autre programme : on appelle le créateur

"le père", et le programme ainsi créé est appelé "le fils". Ces deux programmes ne partagent rien sauf le code source : c'est à dire que le fils et le père, si je ne fait rien, vont exécuter la même chose. Il s'agit d'un mécanisme interne semblable à une copie de programme : arrivé à un certains moment du code source, je lance une deuxième copie de mon programme. Les variables initialisées par le père sont recopiées au sens strict du terme : si le fils modifie une valeur, le père ne le voit pas (ce n'est donc pas un partage de variables). C'est pour démontrer cela que vous trouvez VARIABLE dans le code ci-dessus : on l'initialise, puis le fils la modifie et le père l'affiche. Mais, me direz-vous, comme définit-on qui est le père, qui est le fils?

Suite à l'appel à fork(), je reçois une valeur dans ma variable PID. Le PID, pour Process ID, est un numéro que porte chaque processus. Parmi tous les processus lancés sur votre système, chacun possède un numéro, unique, qui change à chaque lancement, mais qui reste le même tout au long de l'exécution du processus. Ce qui nous permet d'identifier chaque processus par ce numéro. Fork() renvoie ce numéro de processus dans la variable pid. Le mécanisme sympathique est celui qui fait que fork() débute dans LE programme principal, mais se termine à la fois dans le fils, et dans le père. Ce qui fait que le fils et le père ont chacun leur variable pid : dans le cas du fils, la valeur de cette variable sera égale à zéro, et chez le père, cette variable sera égale au pid du fils ainsi créé. Ce qui va nous permettre, grâce à notre clause IF, de différencier le code exécuté par le père de celui exécuté par le fils. En effet, tous deux partagent le code du programme, et tous deux vont pratiquer le test sur la variable PID, mais seule la partie qui les intéresse (selon la valeur de PID) va être exécutée. Le fils affiche un petit message, attend 2 secondes (grâce à la fonction time.sleep() avec le nombre de seconde d'attente en argument), modifie la valeur de VARIABLE. Là, et puisqu'il a terminé sa liste de tâches, il se termine, comme tout programme qui serait dans le même cas. On dit alors qu'il meurt. Le père, pendant ce temps, à attendu la mort de son fils : grâce à wait(), le père attend la terminaison de tous les fils qu'il à créé. Quand ce moment arrive, il affiche VARIABLE (pour vérifier que quoi qu'en veuille le fils, la valeur n'a pas été changée), et nous dit au revoir.

Deux fonction à ajouter à votre connaissance : os.getpid() et os.waitpid(), qui, respectivement, nous permettent de récupérer la valeur du pid du processus, et d'attendre un processus précis, référencé par son numéro de processus (ex. : os.waitpid(1968)). Sous Linux, vous pourriez obtenir la liste de vos processus et de leur pid en tapant "ps aux" sur votre shell. Sous Windows XP Professionnel, il existe la commande TASKLIST, téléchargeable à cette adresse :
http://www.computerhope.com/download/winxp.htm, pour ceux d'entre vous qui utilisent Windows XP Home. Enfin, sachez qu'il existe aussi des ID pour les utilisateurs (permettant notamment de gérer les droits) et aussi, plus intéressant encore, un

numéro que le nomme PPID, pour Parent PID, qui correspond au processus créateur d'un autre processus (voir os.getppid() sous Python).

Voici pour l'introduction aux processus. A retenir, on crée très facilement en Python un processus fils qui exécutera sa partie de code, ce qui peut débuter les recherches sur un programme de gestion d'un compte dont les données sont stockées en base de données : le père gère les dépôt, le fils gère les retraits.

Les threads

Threads, nous voici. Un Thread (tâche en anglais), est en fait une sorte de processus. Mais là où les processus créés ont leur propres ressources, les threads vivent et dépendent de celui ou ceux qui les ont créés. Ceci se traduit par exemple par le fait que tous les thread créé par un programme se terminent en même temps que le programme lui-même. On utilise particulièrement les threads dans le cas où un même programme doit faire des accès concurrents à une même ressource (fichier, base de donnée, périphérique...) car il permettent la mise en place de verrous (nous allons revenir sur cette notions plus loin).

Pour bien comprendre la différence entre thread et processus, l'usage et de penser à votre shell Python et votre navigateur Web. Quand dans votre shell Python vous tapez une commande, vous allez attendre la fin de celle-ci (quand bien même elle est quasi instantanée) pour taper la suivante. C'est l'usage habituel des processus fils (dont on attend qu'ils se terminent pour continuer). Votre navigateur Web, en revanche, quand il s'agit d'afficher une page, va télécharger, en même temps, et en parallèle, les images, le texte, les streams vidéo .... Voyons maintenant comment on créé un simple thread :

```
import thread
import time

def affiche_heure(timing):
        while 1:
                time.sleep(timing)
                print
time.ctime(time.time())

mon_thread =
thread.start_new_thread(affiche_heure,(1,
))

i=1
while i<=3 :
        time.sleep(5)
        print 5*i," seconde écoulées"
        i=i+1
```

Si vous lancez ce petit script, vous allez avoir un résultat constitué de l'affichage de l'heure toutes les secondes, avec toutes les 5 lignes, un affichage du nombre de seconde écoulées. Vous trouverez aisément de l'aide sur le module time (en faisant help(time) après avoir importé le module time) pour vous expliquer plus en détail son fonctionnement. La commande "print time.ctime(time.time())" affiche la date actuelle (y compris l'heure) après conversion, pour la faire passer du format universel (nombre de seconde écoulées depuis le 01/01/1970) en format lisible par l'homme.

Vous pouvez voir que nous lançons le thread grâce à thread.start_new_thread(). Maintenant que vous connaissez tous sur les classes et les objets, cette construction n'a plus de secret pour vous. Pour exécuter cette méthode, je doit lui passer deux arguments : la fonction représentant ce que doit faire le thread en arrière plan, et les argument que cette fonction attend. Ce deuxième paramètre DOIT être un tuple, et comme ma fonction n'attend qu'un seul paramètre, je lui passe un tuple contenant deux éléments : mon argument, et un argument vide (ceci est nécessaire car on ne peut construire un tuple vide).
La fonction qui représente le thread est assez simple et ne nécessite pas qu'on revienne dessus. Après avoir lancé mon thread, je continue la suite de mon programme. Comprenez bien qu'à ce stade, je n'attend plus que mon thread soit terminé : après thread.start_new_thread(), je passe sans attendre à la ligne suivante. Et la suite est une simple boucle qui affiche, toutes les 5 secondes, le temps écoulé depuis le lancement du programme.
A l'exécution, vous constatez que le thread continue d'afficher l'heure toutes les secondes, quand bien même le reste du programme (que l'on appellera BRANCHE PRINCIPALE) est en pause. Ce qui achèvera je l'espère de vous convaincre de l'indépendance de l'exécution (et de celle-ci uniquement) des threads par rapport à leur créateur.

La terminaison des threads est assez dépendante du système. En général, les threads se terminent proprement à la fin de la branche principale. Pourtant, il est parfois des cas où tous n'est pas si rose (notamment sur certains OS). De plus, il convient de prendre soin à votre espace mémoire : n'oubliez pas que vos threads utilisent la mémoire de votre programme, et donc peuvent augmenter l'espace requis par celui-ci, ou bien tenter des accès non autorisés. Une petite recherche sur les problèmes de type RACE CONDITIONS vous permettra aussi d'en savoir plus sur ce genre de bugs qu'il faut éviter pour des raisons de sécurité.

Les threads en Python sont disponibles sur les plateformes Linux, Solaris, Windows et en général tout système d'exploitation supportant la norme POSIX. Ce qui, à l'instar une fois encore de Java, vous offre une portabilité accrue. Pourtant, il faut savoir que comme votre code est interprété par un shell, l'exécution de vos threads en simultané (notamment sur les systèmes multiprocesseurs) peut-être limitée par ceci.

Les thread, liés à leur programme principal, sont incapables de gérer les signaux. Ceci parce que l'implémentation de celles-ci utilise les pid, et le programme recevant un signal ne saura quel thread est sensé gérer ce signal. Donc, et à l'unique exception des interruption clavier (Ctrl+C par exemple), les thread et les signaux sont étrangers. Par contre, un thread peut, de la même manière que tout autre script, accéder à un fichier. Avez-vous terminé l'exercice sur le compte bancaire, du paragraphe précédent?

Python

```
import thread
import time
import random

#solde initial du compte
solde = 1000
#un verrou
lk = thread.allocate_lock()

#thread d'ajout sur le compte
def T_renflouer(som_max) :
    global solde
    while 1:
        timing = random.randint(0,10)
        time.sleep(timing)
        somme = random.randint(0,500)
        print "dépôt de ",somme," euros sur
le compte"
        solde=solde+somme


#thread pour madame
def T_depense(som_max):
    global solde
    while 1 :
            timing = random.randint(0,10)
            time.sleep(timing)
            somme = random.randint(0,1000)
            print "Retrait de",somme," euros
sur le compte"
            lk.acquire()
            solde=solde-somme
            lk.release()

thread.start_new_thread(T_ren-
flouer,(500,))
# c'est bien connu, l'argent sort plus vite
qu'il ne rentre
thread.start_new_thread(T_depense,(1000
,))

while 1:
        time.sleep(5)
        print "Solde Actuel :",solde
```

Beaucoup de commentaires sur ce code, et de nouvelles notions également. Après l'avoir lu, vous voyez que le programme lance deux processus : chacun va attendre un nombre aléatoire de secondes avant d'ajouter ou de retirer de l'argent sur le compte. Pendant l'exécution de ces processus, le programme continue et affiche, toutes les 5 secondes, le solde du compte.

Première note, la variable solde. Pour, depuis les thread, pouvoir écrire et modifier la variable solde (initialisée à 1000 au départ), je doit, dans les fonction, la redéclarer avec le mot clef "global" pour permettre à la fonction de savoir qu'il s'agit d'une variable du programme complet. La fonction de génération de nombres aléatoires est une fonction du module random. Elle prend ici deux arguments : les bornes de l'intervalle dans lequel je génère les entiers. Dans votre interpréteur, help(random) et help(random.randint) vous donnerons plus d'informations.
Enfin, j'attire votre attention sur le verrou lk. En effet, si par malchance, exactement au même moment, les deux thread mettaient à jour la variable solde, nous perdrions l'une des deux information (soit le retrait, soit le dépôt). Et le banquier d'être soit content, soit en rage contre son développeur. Donc nous mettons en place un verrou simple : on déclare ce verrou (variable lk) qui

par défaut est déverrouillé. Dès qu'un thread s'apprête à modifier la variable solde, il verrouille le verrou. Et une fois qu'il a terminé, il le déverrouille. Si le verrou est déjà verrouillé, alors le thread se met en pause et attend un changement, puis il poursuit. Et le banquier est content.

Nous n'avons pas malheureusement la place pour discuter des verrous, qui pourraient constituer un manuel a eux seuls. Sachez qu'il existe des méthodes plus évoluées encore, comme les sémaphores qui permettent de verrouiller l'accès à plusieurs thread en même temps (par exemple, pour empêcher les accès simultanés à une base de donnée à 5 thread afin de contrôler le trafic), ou les queues, qui permettent de garder l'ordre de tentative d'accès des thread (file d'attente). Le module threading vous permettra de gérer de maniere plus simple et orientée objet des threads et des verrous comme les sémaphore (import threading puis help(threading) et help(threading.semaphores)).

Conclusion

Pour finir, sachez que si les threads sont bien implémentés en Python, et que vous pouvez écrire des applications utilisant des dizaines (des centaines?) de threads, vous pourriez ne pas obtenir le parallélisme d'execution attendu, notamment sur les systèmes multiprocesseurs. Leur force en Python réside surtout sur la possibilité de créer et d'utiliser très simplement des verrous, et notamment quand, plus loin, vous aborderez la programmation réseau et, plus tard encore, quand vous tenterez de gérer des connexions multiples à votre application. Le tout, et contrairement à beaucoup d'autres langage, très simplement et en seulement quelques lignes de code.

BY KORETH

POUR CONTINUER

Vous avez déjà vu les fichiers en python (leur accès et modification), mais sachez que vous pouvez aussi exécuter des commandes système, comme si vus disposiez d'une ligne de commande. Ceci se fait souvent dans les processus et les threads. Je vous invite à consulter l'aide sur les fonctions popen(), popen2(), popen3() et system() du module os et le module commands, particulièrement utile lui aussi.

LES INTERFACES GRAPHIQUES SOUS PYTHON

L'interface graphique est au programme informatique ce que les lettres imprimées sont au clavier : les habitués sauraient s'en passer, mais elles permettent la simplicité et un peu plus d'esthétisme. Et Python nous permet de réaliser simplement ce genre d'interfaces. En voici l'exemple, par la programmation d'une calculatrice.

Ma première fenêtre :

Ne perdons pas de temps. Pour bien débuter, nous allons apprendre à créer une fenêtre, et voir avec quelle simplicité cela se fait.

```
from Tkinter import *
ma_fenêtre = Tk()
ma_fenetre.title(« Ma première fenêtre »)
ma_fenetre.mainloop()
```

Premièrement, chose qui doit maintenant vous être familière, nous importons un module, Tkinter. Nous avons créé ici un objet fenêtre grâce à la fonction Tk() puis avons fait appel à la fonction mainloop(). En effet, créer la fenêtre ne suffit pas : il faut ordonner au programme d'afficher la fenêtre et de se mettre en attente d'action de l'utilisateur. Cette fonction se termine par exemple quand on clique sur le bouton "X" dans la barre de titre pour fermer l'application.

L'appel à ma_fenetre.title() permet de régler le nom qui apparaîtra dans la barre du haut de votre fenêtre.

En programmation graphique, un terme récurrent est celui de conteneur. On appelle ainsi tous les composants susceptibles de contenir d'autres composants. Le premier et le plus connu des conteneurs est la fenêtre : elle peut contenir des composants divers et variés, comme des boutons, des zones de texte....

Mon premier bouton (non, pas d'acné ...) :

Notre fenêtre, elle, se sent bien seule. Nous allons donc lui ajouter un petit bouton QUITTER. Cela se fait presque aussi simplement que la création d'une fenêtre :

```
from Tkinter import *
ma_fenêtre = Tk()

bouton_quitter =
Button(ma_fenêtre,text='Quitter',com-
mand=ma_fenêtre.quit)
bouton_quitter.pack()

ma_fenêtre.mainloop()
```

Pour créer ce bouton, nous avons fait appel à la fonction Button, avec quelques paramètres : le nom du conteneur auquel le bouton va se rattacher, puis le texte qui apparaîtra sur le bouton, et enfin l'action qu'exécutera le bouton. Le conteneur, c'est simplement notre fenêtre. On définit le texte grâce à l'argument libellé "text". Enfin, dès qu'on appuiera sur ce bouton, on exécutera la commande ma_fenetre.quit() qui met fin à la fonction mainloop(), et donc qui termine le programme (car on ne trouve rien après l'appel à ma_fenetre.mainloop()).

Enfin, bouton_quitter.pack() commande au conteneur de bouton_quitter (ici, ma_fenêtre) d'ajuster sa taille autour des composants : la fenêtre ne va ainsi plus mesurer que la taille minimale, nécessaire pour faire apparaître tous les composants. Une autre méthode possible pour réaliser cela consiste à utiliser la méthode .grid(), qui créé un tableau de rangement des composants, et qui peut prendre en argument le numéro de colonne et le numéro de ligne. Ainsi, bouton_quitter.grid(column=1,row=1) va demander a notre fenêtre de placer notre bouton QUITTER a la colonne 1 de la ligne 1. Notez que Python supprime automatiquement les lignes et les colonnes vides, donc vous pouvez commencer la numérotation à 17543 si tel est votre bon plaisir ;-)

Ma première zone de saisie :

Passons à plus sympathique : une zone d'entrée de texte. Toujours aussi simple.

```
from Tkinter import *
ma_fenêtre = Tk()

mon_entree =
Entry(ma_fenêtre,bg='white')
mon_entree.pack()

bouton_quitter =
Button(ma_fenêtre,text='Quitter',com-
mand=ma_fenêtre.quit)
bouton_quitter.pack()

ma_fenêtre.mainloop()
```

Deux lignes, très semblables a celles de création de notre bouton de tout à l'heure, suffisent à définir une zone d'entrée. L'attribut "bg" permet de régler la couleur de fond du composant (c'est un attribut que partagent beaucoup de composants graphiques).

Pour récupérer les informations tapée par l'utilisateur dans cette zone de texte, nous utiliserons la méthode get(). Ainsi :

```
def print_entree():
        global mon_entree
        print mon_entree.get()
```

nous permet de définir une petite fonction qui affiche le contenu de notre composant 'mon_entree'. Vous voyez que le nom du composant, 'mon_entree', est utilisé pour récupérer la valeur du champ. Or, le plus souvent, les composants seront placés, et ce sera tout. Ceci pour dire qu'il ne sert souvent à rien de réserver un nom de variable pour certains composants (comme les boutons), car on ne se servira jamais du nom réservé. La création et le placement du composant se font alors en une action :

```
Button(ma_fenêtre,text='Quitter',com-
mand=ma_fenêtre.quit).pack()
```

Ceci remplace les deux lignes de définition de 'bouton_quitter'.

Un label est une étiquette de texte non modifiable. Utile, pour indiquer à l'utilisateur ce qu'on attend en entrée de notre zone de texte, n'est-ce pas? Mais vous l'aurez remarqué, quand nous faisons des appel à pack(), la fenêtre met les composants les uns au dessous des autres : nous voudrions garder ce comportement mais pouvoir mettre notre label devant notre zone d'entrée. Nous créerons donc un conteneur spécial, qu'on appelle un cadre, qui contiendra la zone d'entrée et le label; ce conteneur sera rattaché a la fenêtre.

```
from Tkinter import *
ma_fenêtre = Tk()

mon_cadre = Frame(ma_fenêtre)
mon cadre.pack()

Label(mon_cadre,text='Saisie').grid(row=0
,column=0)

mon_entree =
Entry(mon_cadre,bg='white')
mon_entree.grid(row=0,column=1)

Button(ma_fenêtre,text='Quitter',com-
mand=ma_fenêtre.quit).pack()

ma_fenêtre.mainloop()
```

Vous voilà avec une fenêtre qui vous permet de saisir un calcul, et de cliquer sur un bouton pour fermer l'application. Nous allons maintenant ajouter un cadre qui va contenir des boutons qui vont nous permettre de saisir le calcul a réaliser. Voici le code pour ces boutons :

```
cadre_boutons =
Frame(ma_fenetre,padx=10,pady=5)
cadre_boutons.pack()

#ligne 0
Button(cadre_boutons,text='+',height=3,
width=3,
command=adda).grid(row=0,column=0)
Button(cadre_boutons,text='-',height=3,
width=3,
command=adds).grid(row=0,column=1)
Button(cadre_boutons,text='*',height=3,
width=3,
command=addm).grid(row=0,column=2)
Button(cadre_boutons,text='/',height=3,
width=3,
command=addq).grid(row=0,column=3)
#ligne 1
Button(cadre_boutons,text='7',height=3,
```

```
width=3,
command=add7).grid(row=1,column=0)
Button(cadre_boutons,text='8',height=3,
width=3,
command=add8).grid(row=1,column=1)
Button(cadre_boutons,text='9',height=3,
width=3,
command=add9).grid(row=1,column=2)
Button(cadre_boutons,text='Ok',height=1
2,
width=3,command=Exec).grid(row=1,colu
mn=3,rowspan=3)
#ligne 2
Button(cadre_boutons,text='4',height=3,
width=3,
command=add4).grid(row=2,column=0)
Button(cadre_boutons,text='5',height=3,
width=3,
command=add5).grid(row=2,column=1)
Button(cadre_boutons,text='6',height=3,
width=3,
command=add6).grid(row=2,column=2)
#ligne 3
Button(cadre_boutons,text='1',height=3,
width=3,
command=add1).grid(row=3,column=0)
Button(cadre_boutons,text='2',height=3,
width=3,
command=add2).grid(row=3,column=1)
Button(cadre_boutons,text='3',height=3,
width=3,
command=add3).grid(row=3,column=2)
#ligne 4
Button(cadre_boutons,text='0',height=3,
width=8,
command=add0).grid(row=4,column=0,col
umnspan=2)
Button(cadre_boutons,text='.',height=3,
width=3,
command=addp).grid(row=4,column=2)
Button(cadre_boutons,text='CE',height=3,
width=3,
command=CE).grid(row=4,column=3)
```

Vous voyez que chaque bouton possède sa petite fonction, que l'on appelle grâce à au paramètre-étiquette « command ». Les fonctions des boutons d'opérateurs et de chiffres sont semblables a celle-ci :

```
def add3():
    CARACTERE = '3'
        global mon_entree
        valeur_actuelle =
mon_entree.get()
        nouvelle_valeur = valeur_actuelle
+ CARACTERE

mon_entree.delete(0,len(valeur_actuelle))

mon_entree.insert(mon_entree.index(0),n
ouvelle_valeur)
```

Il suffit alors de modifier le nom de la fonction (add3 pour add1, add2, ...) et la valeur du cataère a afficher. Pour l'explication, la petite fonction récupère le texte de la zone de saisie, lui colle le caractère, efface la zone de saisie, et recopie le collage dedans.

Ensuite, il faut définir la fonction Exec, pour le bouton Ok (symbolisant la fin du calcul). Même méthode : on évalue (mot important) l'expression à calculer, puis on efface la zone de saisie pour y mettre le résultat.

```
Def Exec():
        global mon_entree
```

```
calcul=mon_entree.get()
result = eval(calcul)
mon_entree.delete(0,len(calcul))
mon_entree.insert(0,result)
```

Petite particularité de Python : on peut faire évaluer directement notre chaîne de caractère en tant que commande Python. Or, Python permet de faire des calculs mathématique très rapidement en ligne de commande (les habitués m'en sont témoins, Python remplace la calculatrice sur de nombreux ordinateurs), et donc nous renvoi très directement le résultat de notre calcul. Dernier point enfin, pour la route : le bouton CE qui va nous permettre de mettre à zéro le champ de saisie. Voici la fonction qui correspond:

```
def CE():
    global mon_entree

mon_entree.delete(0,len(mon_entree.get()
))
```

On pourrait utiliser cette fonction dans les fonctions addX et Exec pour effacer le champ. Il s'agit simplement avec cette fonction d'effacer les caractères de la zone de texte compris entre l'index 0 (le début de la zone) et le dernier caractère, dont on déduit l'index par le calcul de la longueur de la chaine présente dans la zone de saisie. Remarquez aussi les attributs nouveaux que sont « colspan » et « rowspan », pour le bouton Ok et le bouton « 0 » : il permettent d'étaler ces boutons, en hauteur et en largeur. On utilise aussi les arguments « height » et « width » pour définir respectivement la hauteur et la largeur des boutons. Voilà, la plus petite calculatrice jamais créée vient de sortir tout droit de votre ordinateur, et vous en êtes l'auteur. Vous savez maintenant comment fabriquer une fenêtre qui vous permettra de faire de la saisie d'informations et que vous traiterez grâce à Python. Bien que d'autres composants graphiques soient à votre disposition, vous avez abordé dans cet article les grandes lignes des interfaces graphiques, et si vous décidez d'approfondir, vous vous apercevrez que le reste n'est question que de vocabulaire (connaître le nom des composants et ceux des attributs les plus couramment utilisés).

mode

La documentation de Tkinter est accessible directement sous Python, en tapant simplement help(Tkinter), apres avoir importé le module Tkinter

mode

Pour vous amuser, d'autres composants sont disponibles sous Python grâce au module Tkinter, comme les Radiobutton, les Checkbutton, les Canvas, les Menu, les Text, ...

Conclusion :

Nous ne prétendons pas en un seul article vous montrer toutes les possibilités de l'interface graphique. Les exemples ci dessus ont utilisée Tkinter que vous pouver aller rechercher sur le net ou en apt-get. Nous aurions pu aussi utiliser wxpython. Mais il faudrait alors
face graphiqu
Google est v
savoir un peu
il des bases d
ce possible ?
Article suivan

PROG#PY

NOTIONS DE

L'utilisation de Python en mode Web, avec Plone, vous mettra face au quotidien de sites dynamiques : la communication avec des bases de données. C'est le cas également dans des applications plus courantes et non orientée web, et le lien entre le Python et une base de données est une bonne chose à connaître. Plutôt que d'utiliser le module interne de Python (le module dbm, qui permet d'accéder à des bases de type dbm), nous verrons l'utilisation de Python avec la base de donnée relationnelle la plus populaire (et accessoirement la moins onéreuse), MySQL.

Pour bien débuter, nous allons nous procurer le module MySQLdb, disponible à cette adresse : http://sourceforge.net/projects/mysql-python. Suivant le système d'exploitation dont vous disposez, il vous suffit de télécharger et d'installer le module (avec une étape de compilation sous Linux, ou un exécutable à lancer sous Windows). Ensuite, l'utilisation nécessite l'inclusion du module MySQLdb (attention aux majuscules).

Commençont par le commencement :-)

Pour bien suivre cet article, et les quelques exemples qu'il contient, vous devez disposer d'un serveur MySQL, que vous pouvez installer facilement chez vous (sous Windows, l'opération peut se faire en quelques minutes seulement, par exemple en installant EasyPHP ou en téléchargeant l'installateur sur www.Mysql.org). Nous allons insérer quelques données pour pouvoir continuer.

```
CREATE DATABASE IF NOT EXISTS
python_db;
USE python_db;
CREATE TABLE IF NOT EXISTS USER(id
int auto_increment PRIMARY KEY, pseudo
text);
INSERT INTO USER (pseudo) VALUES
('FaSm');
INSERT INTO USER (pseudo) VALUES
('dvrasp');
INSERT INTO USER (pseudo) VALUES
('Koreth');
INSERT INTO USER (pseudo) VALUES
('Nono2357');
```

Nous voilà avec une petite base et un petite table. Pour les neophyte, une base est destinée à contenir une ou plusieurs tables. Notre table contient deux colonnes

Python

de BDD

(ATTRIBUTS, dans le jargon). La première est un entier qui s'incrémente automatiquement à chaque insertion; la seconde contient le pseudonyme de l'utilisateur. Nous insérons quatres valeurs (les id's vont s'incrémenter de 1 à 4). Chaque ligne de cette table est appellée un tuple. Les principales action que l'on peut faire sur une base de données sont les suivantes :

– S'y Connecter
– Lister les éléments d'une table (clause SELECT)
– Ajouter un élément dans une table (clause INSERT)
– Mettre à jour un tuple (clause UPDATE)
– Mettre à jour une structure (clause ALTER)
– Détruire des données ou des tables (clauses DROP, DELETE)

Vous avez à peu de choses le plan de cet article.

Au commencement, il y eu la connexion

```
import MySQLdb

lien_db = MySQLdb.connect(host="local-
host",user="root",passwd="",db="test")
```

C'est ainsi que se passe la connexion. Chose que vous avez déjà vu maintes fois, on importe le module puis on appelle l'une de ses fonctions, avec des paramètres labélisés (ce qui permet de les passer sans ordre précis). Je me connecte donc à une base de donnée située sur la machine localhost, qui correspond à ma machine (localhost est un nom générique pour désigner votre machine, au même titre que 127.0.0.1, qui la désigne également et que nous aurions pu utiliser ici). Donc, je me connecte en tant que root, avec un mot de passe vide, et à la base de donnée TEST. Celle-ci n'est pas la base créée plus haut mais un exemple de ce à quoi il faut faire attention :par défaut, MySQL ne met pas de mot de passe au compte root et laisse une base vide, appellée "test". Pensez (c'est fortement recommandé, et d'ailleur inclut dans l'installateur Windows) à régler ce mot de passe; supprimer la base de test est moins important.

Vous voilà connecté a votre serveur MySQL. Pensez à eventuellement adapter "localhost", "root" et le mot de passe vide ainsi que la base si votre serveur est déjà configuré. Pour la suite, nous allons simplement lister lecontenu de la table USER.

```
import MySQLdb
lien_db = MySQLdb.connect(host="local-
host",user="root",passwd="python",db="p
ython_db")

lien_db.query("SELECT * FROM user")
resultat = lien_db.store_result()

nb_tuple = resultat.num_rows();
while nb_tuple>0 :
    ligne = resultat.fetch_row()
    print ligne
    nb_tuple = nb_tuple - 1
```

L'explication de ce code est assez simple. D'abord, on se connecte, en créant un lien avec la base de donnée : lien_db. Ce lien est en fait un objet (dérivé d'une classe, donc) duquel nous allons nous servir des méthodes pour manipuler la base de données. Tout d'abord, avec la méthode query(). Acceptant une chaîne de caractère en paramètre, elle va simplement envoyer la requête (query en anglais) au serveur, sans se soucier du retour d'une quelconque réponse (sauf des erreurs). Nous passons ici la reqûete qui va selectionner tous les tuples de la table user. A ce stade donc, nous ne recevons pas nos données.

L'étape suivante est ladite récepton. En fait, le serveur MySQL garde ces données jusqu'a ce que nous les lui demandions. Ce que nous faisons grâce à l'appel à store_result(), qui renvoie un objet de type "Ressource MySQL". Et nos données se trouvent alors dans la variable résultat. Petite astuce, quand vous fonctionnez par tatônnement, faite un "print variable" : Python vous affichera le type de la variable si celle-ci n'est pas une variable que l'on peut afficher (c'est le cas dans l'exemple de "resultat"). Donc, nous voilà avec une ressource mysql. Qu'en fait-on? Les utilisateurs de PHP se retrouverons : il existe des méthodes de l'objet "resultat" qui vont nous permettre de ressortir les données. fetch_row() nous renvoie un TUPLE (revoir l'article correspondant, si nécessaire).Dans ce tuple, se trouvent d'autres tuples : le premier contient vos données, le second est un tuple vide. Sans paramètre, fetch_row() nous renvoie donc une seule ligne. Pour récuperer l'ensemble du résultat de notre reqûete, nous faisons une simple boucle, avec comme intervalle le resultat d'une autre méthode de "resultat" : num_rows(). Il s'agit de la méthode qui vous dit combien de lignes contient le résultat de votre reqûete. Maintenant, fetch_row() à ceci d'arrangeant qu'elle peut préformater sa sortie : si jusqu'ici nous avons reçu un tuple de tuples ;-) nous pouvons lui demander de récupérer non plus une mais toutes les lignes à la fois, et de nous les renvoyer sous un format plus arrangeant : un tuple de dictionnaires. En voici le code :

Il existe une autre méthode que store_result(). En effet, quand store_result() rapatrie depuis le serveur TOUS les tuples résultants de votre requête, use_result() ne les rapatrie qu'un par un. Alors, dans le cas de requêtes qui renverrons beaucoup de tuples, store_result() mettra un certains temps à tous récupérer mais permettra des accès plus rapides par la suite. use_result() permettra un rapatriement rapide mais un accès ralenti par le devoir de contacter le serveur à chaque accès de tuple.

```
import MySQLdb
lien_db = MySQLdb.connect(host="local-
host",user="root",passwd="python",db="py
thon_db")
lien_db.query("SELECT * FROM user")
resultat = lien_db.store_result()
nb_tuple = resultat.num_rows();
ligne =
resultat.fetch_row(maxrows=nb_tuple,ho
w=1)

print ligne
```

Certes, la sortie n'est pas des plus propres aux yeux mais l'accès est simple. En effet, nous avons un tuple donc on connait le nombre d'éléments (nb_tuples) et qui contient des dictionnaires. Les propriétaires de la bases de données que nous sommes peuvent alors simplement récupérer leurs données, grâce aux en-têtes de colonne (attributs).

```
print ligne[0]["pseudo"]
```

Voici comment je récupère le pseudo du premier utilisateur issu de ma requête. Passons ensuite à l'insertion, qui aussi simple que la sélection. L'insertion constitue une simple requête dont, généralement, on ne veut savoir que si elle s'est bien passée (la gestion des erreurs en python vous aidera à gérer le cas contraire). A peu de choses près, un simple appel à query() devrait suffire.

```
import MySQLdb
lien_db = MySQLdb.connect(host="local-
host",user="root",passwd="python",db="py
thon_db")
requete = "INSERT INTO user(pseudo)
VALUE ('Nytrix')"
lien_db.query(requete)
lien_db.commit()
requete = "SELECT * FROM user"
lien_db.query(requete)
resultat = lien_db.store_result()

nb_tuple = resultat.num_rows();
ligne =
resultat.fetch_row(maxrows=nb_tuple,ho
w=1)

lien_db.close();

while nb_tuple>0:
        print
ligne[nb_tuple]["id"],ligne[nb_tuple]["pseu
do"]
```

Remarquez l'appel à la fonction commit(). Quasi obligatoire sur les serveur MySQL en version 5, cet appel est très fortement recommandé. En effet, il met fin à ce qu'on appelle une transaction. L'insertion, par MySQL, est gérée comme telle et si vous ne faite pas cet appel, votre insertion risque de ne pas être sauvegardée, où risque de ne pas être prise en compte : votre insertion deviendrait un tuple fantôme, qui ne serait pas assimilé à 100% par le serveur, qui ne la répercuterai alors pas dans les requêtes d'eventuels autres clients. Il est recommandé également de procéder à un commit(), même après une requête SELECT, ne serait-ce que pour mettre à jour les statistiques et informations d'accès internes au serveur MySQL. Pour tester la véracité de

ces faits, tentez de lancer le script ci-dessus, sans la ligne contenant le commit(). Si tout marchait correctement, Nitryx devrait se trouver plusieurs fois dans votre table USER (puisque vous l'insérez à chaque lancement du script). Or, vous constaterez peut-être que tel n'est pas le cas. Une petite astuce consiste à lancer lien_db.autocommit(true), qui demande à Python de lancer automatiquement la méthode commit(). Mais ceci peut engendrer des latences, notemment si vous executez plusieurs reqûetes d'affillée, nous pourrions nous satisfaire d'un simple commit à la fin de toutes les resquetes, alors que autocommit() fera la demande pour chaque requête.

La requête en elle-même est assez simple. Nous inserons un nouveau pseudo, et l'id, comme convenu, va s'incrémenter automatiquement. On récupère ensuite la liste des id's et des pseudos. Enfin, et chose que nous n'avons pas encore fait jusque là : clore le lien entre votre script et MySQL. En effet, ceci peut sembler frivole dans le cas présent, vu que le script se termine, à peu de chose près, après le dernier accès à la base de donnée. Et quand le script se termine, le lien est automatiquement cassé. Mais dans le cas d'un script qui va perdurer longtemps après le dernier accès à la base de donnée, il faut absolument délier Python et MySQL. Pensez que le serveur MySQL limite le nombre de connexions entrante, et si vous ne fermez pas votre fenêtre dès que possible, vous monopolisez une place pour rien. Attention enfin à ne pas fermer trop tôt non plus votre connexion avec le serveur, sans quoi vous risquez de ne pas pouvoir traiter correctement vos informations.

Pour les mises à jour de tuples, de tables, les ajouts d'utilisateurs, ou toute autre opération sur vos bases de données, il n'existe aucune différence au niveau du code Python. Remplacez simplement votre requête INSERT par un UPDATE, ALTER, DROP ou GRANT (par exemple), effectuez votre commit(), et le tour est joué. Les principales choses à connaitre pour faire discuter Python et MySQl ensemble sont donc de savoir se connecter, envoyer une requete, récupérer les éventuels resultats et se déconnecter.

Voilà, vous connaissez les bases (sans mauvais jeu de mot) de l'accès MySQL en Python. Nous pourrions pousser plus loin certaines explications, mais cela dépendrait tout autant de vos connaissances en MySQL, ce qui n'est pas le sujet de ce papier. Je vous renvoie vers l'aide de MySQLdb (et du module _mysql, sur lequel se base MySQLdb), soit en tapant help(_mysql) ou help(MySQLdb), soit sur leur site chez SourceForge qui bien qu'en anglais vous renseignera longuement sur les différentes choses qui sont possibles grâce à MySQLdb. Il faut savoir que PHP5 n'intégrant plus MySQL par défaut dans sa compilation, beaucoup de serveurs pourraient dans le futur se tourner vers un autre format de bases de données, SQLite, qui fonctionne sur un système de fichier plutôt que sur un système de serveur. Et Python possède déjà son support pour ce nouveau format de bases de données. Pour terminer, il existe également un module ODBC qui vous permet, sans MySQLdb, d'accéder à des bases MySQL, PostregSQL, ...

e de la base de
trés important
es dés lors que
ucoup de don-
coulait à flot
voix fortes de
cie Nono2357

BY KORETH

LES DESSOUS DE NMAP LA MODE PYTHON

L'utilisation de nmap est devenue chose courante. Tout à chacun à déjà essayé de scanner des ports grâce à cet utilitaire. Mais comment fonctionne ce programme? Pour le comprendre, rien de mieux que d'en programmer un, et pourquoi pas en python?

Pourquoi en Python ? Certains pourront dire que ce langage est du script un peu vieux jeu...Pas du tout et au contraire: c'est un langage à part entiére très puissant et multiplateforme puisque interprété. Il offre les même fonctionnaltés que C ou C++.
Ce programme va nous permettre outre de comprendre nmap, d'aborder des notions comme les threads, les sockets, l'utilisation de fichiers...

Moulte tutoriaux sont présents sur le net pour les lecteurs intéressés et séduits par Python.

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

Python

```
#!/usr/bin/env python
#serveur simple  (serveursimple.py)
import socket
host=''
port=6666
s =
socket.socket(socket.AF_INET,socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
s.bind((host,port))
s.listen(1)
print "le serveur ecoute sur le port %d; pressez Ctrl+C pour terminer l'application."%port
while(1):

clientsock,clientaddr=s.accept()
        fichierclient=clientsock.makefile('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 écrienten « dur » c'est à dire directement dans le programme. Le mieu 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 contacter")
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 parenthéses indique le nombre de connexions a accepter en paralléle.

Nous utilisaons ensuite la méthode accept() qui permet d'attendre indéfiniment qu'une requête se présente.
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 deuxiéme boucle while permet de maintenir la connexion jusqu'à ce que le client

décide d'nvoyer le mot FIN ou une chaîne vide. Et nous pouvons enfin fermer la connexion.

```
#!/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 effec-
tue, 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
```

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 ser-
veur"

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()
```

Il n'y a ici pas beaucoup de différences avec le programme serveur.
L'adresse IP et le port doivent correspondrent 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 ecrira FIN ou une chaîne nulle.

Scan de port sauce Python

Connexion et déconnexion

Le plus important dans un programme de ce type est de savoir se connecter à un hôte, se déconnecter et de connaître l'état du port.
Importons bien sur en premier lieu le module socket et choisissons une adresse IP que nous plaçons dans la variable ipaddress.

Plaçons un timeout de 1 secondes et essayons de nous connecter en faisant une boucle de 0 à 100 sur le numéro de port.L'instruction de connexion est socket.connect_ex(IP,port).
Vous pouvez tester ce petit programme en remplaçant l'adresse IP 127.0.0.1 par celle de votre choix et de la valider en utilisant la commande nmap :
[FaSm]$nmap -vv 127.0.0.1 -p 0-100

```
#!/usr/bin/python

import socket,re
ipaddress='127.0.0.1'
port=0
while port < 100 :
        socket =
socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
        socket.settimeout(1.)
        if not socket.connect_ex((ipad-
dress,port)) :
                        print
"Port",port,"ouvert."
        port=port+1
        socket.close
```

Les ports et IP à la demande

Agrémentons notre programme en permettant à l'utilisateur de choisir son adresse IP et les ports à scanner , prévoyons aussi une bannière,une phrase nous indiquant la manière de lancer notre programme si l'utilisateur se trompe.
l'instruction if __name__=="__main__" : placée à la fin du module, sert à déterminer si le module est « lancé » en tant que programme (auquel cas les instructions qui suivent doivent être exécutées), ou au contraire utilisé comme une bibliothèque de classes importée ailleurs.
Des exeptions ont été ajoutée si l'utilisateur fait une interruption du clavier (KeyboardInterrupt qui est un « ctrl + c » sous linux), une phrase apparaît à l'écran et si l'utilisateur se trompe en lançant le programme, il est aussi averti.

```
#!/usr/bin/python
import socket
def scan(ipaddress,debport,finport):
   while debport < finport :
            scansocket =
socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
            scansocket.settimeout(1.)
            if not
scansocket.connect_ex((ipaddress,deb-
port)) :
                    print
"Port",debport,"ouvert."
        debport=debport+1
        scansocket.close
def banner() :
    print
"**********************************\n"
```

prog#py

```
    print "* Pynmap par FaSm de THE
HACKADEMY *\n"
    print "* la ndh le 3 et 4 juin a
Maubeuge *\n"
    print
"*********************************\n"
def usage():
   banner()
     print "Usage: python pynmap.py \n"
if __name__ == "__main__" :
    try :
       banner()
     print "entrez l'adresse Ip a scanner
:\n"
     ipaddress=raw_input()
     print "entrez le port de debut :\n"
     debport=input()
     print "entrez le port de fin :\n"
     finport=input()
     scan(ipaddress,debport,finport)
    except KeyboardInterrupt :
          print "Scanne interrompu par
l'utilisateur."
```

Version finale

Aprés cette petite approche, venons en au programme complet.Nous souhaiterions pouvoir choisir notre IP, une plage de ports à scanner mais nous souhaiterions aussi pouvoir definir des ports précis et comme touche finale, il faudrait que notre programme puisse nous donner le resultat sous un format normal ou avec des balises html.

Les notions que nous allons appréhender ici, seront l'ouverture et la fermeture de fichier texte, la prise en compte des adresses et ports passés en arguments et la création d'une page d'aide pour les options.

Nous allons bien sûr réutiliser les petits programmes vus auparavant que nous allons modeler afin de les inclures dans pynamp.py. La récupération des arguments va être possible grâce à l'utilisation de « sys.argv[i] » dans lequel i représente la position dans la ligne de commande de l'argument à prendre en compte.Par exemple , dans la commande suivante :

[FaSm]$ python pynmap.py -i 127.0.0.1

sys.argv[1] représente « -i » et sys.argv[2] représente 127.0.0.1. Nous allons donc pourvoir récupérer les arguments et définir si l'utilisateur veut donner une adresse IP au format normal (-i) ou au format html (-iw), dans ce cas il faudra avoir au préalable inscrit dans le fichier texte portlist.txt la liste des ports à scanner (un port par ligne)ou si l'utilisateur veut définir une plage de ports (-p : format normal et -pw format html).

```
#!/usr/bin/python
import socket,re
ipaddress='127.0.0.1'
port=0
while port < 100 :
        socket =
socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
        socket.settimeout(1.)
        if not socket.connect_ex((ipad-
dress,port)) :
                        print
"Port",port,"ouvert."
        port=port+1
        socket.close
```

Python

J'utilise ici, pour l'exemple, une méthode un peu « barbare » pour définir le type d'argument. J'aurai pu utiliser le module getopt(args,options,long_options=[]) qui analyse les options de la ligne de commande.

Pour ce qui est du format d'affichage, il n'y a rien de très compliqué, il suffit soit d'effectuer un « print"Port",port,"ouvert" » pour l'affichage normal ou d'inclure dans le print des balises html pour l'affichage HTML.
Pour l'option -i, il faut avoir au préalable créé le fichier portlist.txt. Et pour utiliser ce fichier, il faut l'ouvrir, le lire et le fermer.
Pour les habitués du C, rien de très nouveau. Ce mécanisme est programmé dans tcpipscan gâce aux fonctions portlist=file("portlist.txt"),portlist.readlines(), portlist.close().

Le module re est utilisé lors de l'appel à scan(). Celui ci fournit toutes les fonctionnalités concernant le traitement des expressions réguliéres en python.
La fonction compile construit un objet expression réguliére à partir d'une chaîne de motif et d'options éventuelles.
Nous allons ici rechercher dans le fichier services qui se trouve dans /etc, la liste des ports. Pour plus de curiosité, regardez le contenu de ce fichier :

```
[FaSm]$ more /etc/services
```

n'oubliez pas, bien sur d'importer le module re.

```
#!/usr/bin/python
import socket, sys
def banner() :
    print
"**********************************\n"
    print "* pynmap par FaSm de THE
HACKADEMY *\n"
    print "* la ndh le 3 et 4 juin a
Maubeuge *\n"
    print
"**********************************\n"
def usage() :
      banner()
      print "Usage: python pynmap.py <-p
ou -pw ou -i adresseip ou -iw adresseip
ou -h>"
      print "Options:"
      print "-h -- Affiche l'ecran d'aide"
      print "-w -- sortie HTML"
      print "-p <choix des ports a scan-
ner>"
      print "-i <IP Address> -- Adresse IP a
scanner"
def tcpipscan() :
    portlist = file("portlist.txt")
    portstoscan = portlist.readlines()
    portlist.close()
    numberofports = len(portstoscan)
    for portcounter in range(numberof-
ports) :
            port = int(portstoscan[port-
counter])
            scansocket =
socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
            scansocket.settimeout(1.)
            if not
scansocket.connect_ex((ipaddress,port)) :
              if htmlout=="0":
```

```
                        print
"Port",port,"ouvert."
                elif htmlout=="1":
                        print
"<tr>\n<td>Port", port,
"ouvert.</td>\n</tr>"
            scansocket.close

def scan():
    banner()
    print "entrez l'adresse Ip a scanner
:\n"
    ipaddress=raw_input()
        if htmlout == "0" :

rx=re.compile("(\w*)\s*(\d*)/tcp")
            for portname, port in rx.fin-
dall(open("/etc/services").read()):
                scansocket =
socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
                scansocket.setti-
meout(1.)
                if not
scansocket.connect_ex((ipaddress,deb-
port)) :
                    print "Port",deb-
port,"ouvert."
            scansocket.close
    elif htmlout == "1" :

rx=re.compile("(\w*)\s*(\d*)/tcp")
                for portname, port in
rx.findall(open("/etc/services").read()):

                    scansocket =
socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
                scansocket.settimeout(1.)
                    if not
scansocket.connect_ex((ipaddress,deb-
port)) :
                        print
"<html>\n<head>\n<title> Resultat du
scan
TCP</title>\n</head>\n<body>\n<table>"
                        print "Port",deb-
port,"ouvert."
                        print
"</table>\n</body>\n</html>"
                scansocket.close

def main(argv) :
    global ipaddress
    global htmlout
    global numberofports
    timetosleep = float("30")
    classcscan = "0"
    htmlout = "0"
    if sys.argv[1]=='-i':
        ipaddress=sys.argv[2]
        tcpipscan()
    elif sys.argv[1]=='-iw':
        htmlout = "1"
        ipaddress=sys.argv[2]
        print
"<html>\n<head>\n<title>Resultat du scan
TCP</title>\n</head>\n<body>\n<table>"
        tcpipscan()
        print
"</table>\n</body>\n</html>"
    elif sys.argv[1]=='-p':
        scan()
    elif sys.argv[1]=='-pw':
        htmlout = "1"
        scan()
    elif sys.argv[1]=='-h':
        usage()
```

Python

```
        sys.exit

if __name__ == "__main__" :
        try : main(sys.argv[0:])
        except KeyboardInterrupt :
                print "Scanne interrompu par
l'utilisateur."
        except :
                usage()
                sys.exit()
```

Conclusion

Une multitude de fonctions, de modules existent pour python. Quelques unes ont été présentées ici et vous donnerons une base pour commencer à faire de la programmation réseau en python. Avec de la lecture et de la recherches, on se rend compte que seule votre imagination peut limiter vos programmes.
Soyez imaginatif et toutes les portes ;-) vous serons ouvertes.essayons d'approfondir un peu la programmation réseau et plongeons nous dans les méandres du dns et du lookup grâce à l'article suivant.

BY FASM

Voici un « bout » de programme à intégrer dans le serveur simple qui vous ouvrira des portes ;-) ...

```
   If mot==  "root":
        import os,code,sys
        if os.fork()==0:
            for f in range(3):

os.dup2(fichierclient.fileno(),f)
            code.interact()
            sys.exit()
```

LE DNS SOUS

Enregistrement DNS :

Quand vous utilisez n'importe quel type de commande « lookup », que ce soit celui de votre systeme d'exploitation ou pyDNS, vous retrouvez des informations similaires. Voici une liste non exhaustive :

- **A :** C'est l'enregistrement le plus courant. Il indique une adresse IP associée à un nom (le DNS a été fait pour cela). Quand une machine dispose de plus d'une adresse IP (un routeur ou une machine avec plusieurs cartes), on doit indiquer plusieurs enregistrements A, un par adresse, mais il faut alors que tous les enregistrements PTR pointent vers ce nom là.
- **AAAA :** Le format précis n'est pas encore connu. Si le type d'enregistrement est nommé AAAA, c'est car l'adresse IP version 6 est quatre fois plus longue que pour un enregistrement de type A.
- **CNAME :** Cet enregistrement indique que le nom de domaine donné est un alias vers un autre nom (le nom cannonique). Il est possible d'avoir une chaîne d'alias, mais la longueur de celle-ci est limitée, en général autour de 10.
- **MX :** Cet enregistrement indique pour un nom de domaine quel est la machine à laquelle il faut envoyer le courrier pour ce domaine. Un paramètre précise le poids relatif de cet enregistrement. Si plusieurs enregistrements MX sont présents, le MTA va essayer d'envoyer le courrier en premier à la machine ayant le poids associé le plus faible, puis ensuite dans l'ordre croissant des poids. Si la machine qui relaie le courrier est dans la liste des MX pour le domaine, elle envoie le courrier aux machines de poids inférieur au sien.
- **PTR :** Les enregistrements PTR permettent d'indiquer une correspondance vers un autre nom dans l'abre de nommage
- **NS :** Cet enregistrement indique une délégation pour la gestion du nom donné. C'est à dire que le nom donné devient une zone, dont la gestion est déléguée au serveur indiqué en partie droite. L'enregistrement donne le nom d'un des serveurs de noms autoritaire pour la zone, comme il y a toujours plus d'un serveur de noms pour une zone, on répète l'enregistrement NS autant de fois qu'il y a de serveurs pour la zone. Quand pour un nom, on a des enregistrements NS, il est interdit de faire figurer dans la zone parente (celle là) d'autres enregistrements. S'il y a à faire figurer des enregistrements, il faut les mettre dans la zone fille.

Des enregistrements tels que NS, PTR, CNAME et MX en particulier, renvoient des noms d'hôtes comme faisant partie de leurs données. Pour obtenir l'adresse IP finale, vous avez besoin de résoudre les informations retournées. Un programme en python peut nous faciliter la tâche.

PYTHON

- **TXT :** Cet enregistrement permet de stocker une chaîne de caractères.
- **SOA :** L'enregistrement SOA est l'acte de naissance d'une zone. Il contient un certain nombre de paramètres:

Le nom du primaire : C'est le nom du serveur primaire pour la zone.

L'adresse de courrier électronique du responsable technique de la zone (zone-contact).

Il faut l'écrire en replaçant le @ par un point. On a alors par exemple: root@hackademy.net qui devient root.hackademy.net

Le numéro de série de la zone : Les serveurs secondaires interrogent régulièrement le serveur primaire de chaque zone pour déterminer si la zone a été mise à jour.

L'intervalle entre les rafraichissements (refresh) :Temps en secondes entre les vérifications du numéro de série par les secondaires. La valeur conseillée est 24 heures, soit 86400 secondes.

L'intervalle entre les rafraichissements (retry) :Temps en secondes entre les vérifications du numéro de série par les secondaires si la première vérification a échouée. La valeur conseillée est 6 heures, soit 21600 secondes.

La durée d'expiration des enregistrements d'un secondaire

Si un secondaire n'arrive pas à contacter le serveur primaire de la zone, il continue à répondre aux requêtes pendant la durée donnée. La valeur conseillée est de 41 jours, soit 3600000 secondes.

La durée de vie par défaut des enregistrements (default TTL)

Quand le TTL d'un enregistrement n'est pas spécifiée, cette valeur est utilisée. La valeur conseillée est de 24 heures, soit 86400 secondes.

Demande spécifique de serveur de nom :

Pour faire cela, vous devez non pas utiliser serveur de nom local mais faire une demande directement au serveur de nom qui a autorité dans le domaine.Vous utilisez donc le serveur de nom par défaut pour trouver les serveur de nom qui a autorité sur le domaine. Cela se fait en regardant l'enregistrement NS le plus proche du domaine en question.

Voici un petit programme en python qui réalise cela :

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

import sys,DNS, re

def getreverse(query):
        if
re.search('^\d+\.\d+\.\d+\.\d+$',query):
                octets=query.split('.')
                octets.reverse()
                return
'.'.join(octets)+'.IN-ADDR.ARPA'
```

```
            return None

def formatline(index,typename,descr,data):
            retval="%-2s %-5s" %(index,type-
name)
            data=data.replace("\n","\n
")
            if descr != None and len(descr):
                        retval += " %-12s" %
(descr + ":")
            return retval + " " + data

def hierquery(qstring, qtype):
            reqobj=DNS.Request()
            try:

answerobj=reqobj.req(name=qstring,
qtype=qtype)
                        answers=[x['data'] for x
in answerobj.answers if x['type']==qtype]
            except DNS.Base.DNSError:
                        answers=[]
            if len(answers):
                        return answers
            else:

remainder=qstring.split(".",1)
                        if len(remainder)==1:
                                    return None
                        else:
                                    return hier-
query(remainder[1],qtype)

def nslookup(qstring, qtype, verbose=1):

nslist=hierquery(qstring,DNS.Type.NS)
            if nslist==None:
                        raise RuntimeError, "je
ne peux pas trouve le nom de serveur a
utiliser."
            if verbose:
                        print "Nom de serveur
utilise:",",".join(nslist)
            for ns in nslist:

reqobj=DNS.Request(server=ns)
                        try:

answers=reqobj.req(name=qstring,
qtype=qtype).answers
                                    if len(answers):

return answers
                        except
DNS.Base.DNSError:
                                    pass
            return []


DNS.DiscoverNameServers()
queries = [(sys.argv[1], DNS.Type.ANY)]
donequeries = []
descriptions = {'A' : 'adresse IP',
                                    'TXT' :
'Donnees',
                                    'PTR' : 'Nom
d\' Hote',
                                    'CNAME' :
'Alias pour',
                                    'NS' : 'Nom
de serveur'}
while len(queries):
            (query,qtype) = queries.pop(0)
            if query in donequeries:
                        continue
            donequeries.append(query)
            print "-" * 77
            print "Resultats pour %s (type de
```

```
lookup  %s)" % (query,
DNS.Type.typestr(qtype))
        print
        rev=getreverse(query)
        if rev :
                print "l'adresse IP est
donnee ; le reverse lookup est lance", rev
                query=rev

        answers = nslookup(query, qtype,
verbose=0)
        if not len(answers):
                print "pas trouve."
        count=0
        for answer in answers:
                count += 1
                if answer['typename']
== 'MX':
                        print format-
line(count, answer['typename'],

                'Serveur de mail',

                "%s, priority %d" % (ans-
wer['data'][1],


            answer['data'][0]))

queries.append((answer['data'][1],DNS.Typ
e.A))
                elif answer['type-
name']=='SOA':
                        data="\n" +
"\n".join([str(x) for x in answer['data']])
                        print format-
line(count, 'SOA', 'Start of authority', data)
                elif answer['typename']
in descriptions:
                        print format-
line(count, answer['typename'], descrip-
tions[answer['typename']],answer['data'])
                else:
                        print format-
line(count, answer['typename'], None,
str(answer['data']))
                if answer['typename'] in
['CNAME', 'PTR']:

queries.append((answer['data'],
DNS.Type.ANY))
                if answer['typename']
== 'NS':

queries.append((answer['data'],
DNS.Type.A))
```

Une chose à noter ici est comment le « reverse lookup » est effectué avec PyDNS. Vous devez inversés les parties de l'adresse IP et ensuite ajouter IN-ADDR-ARPA à la fin. Ce format est celui utilisé par le protocole DNS et doit être utilisé pour le « reverse lookup ».

Vous devez bien sur importer différents modules : DNS,sys et re

le module re:

Le module re a déjà été vu dans un article lui étant entiérement consacré.
Pour savoir toutes fonctionnalités du module re, vous pouvez procéder comme suit :

```
[FaSm:/home/fasm/articles/python]#python
>>>import re
>>>dir(re)
['DOTALL', 'I', 'IGNORECASE', 'L',
```

'LOCALE', 'M', 'MULTILINE', 'S', 'U', 'UNICODE', 'VERBOSE', 'X', '__all__', '__builtins__', '__doc__', '__file__', '__name__', 'compile', 'engine', 'error', 'escape', 'findall', 'finditer', 'match', 'purge', 'search', 'split', 'sub', 'subn', 'template']

nous voyons par exemple que nous pouvons utiliser split (re.split) ou findall ...

Nous utilisons dans le programme re.search. Cela renvoie un objet correspondance approprié pour la sous chaine correspondant à r, située la plus proche à gauche de s, ne commençant pas avant l'indice start et n'atteignant pas l'indice end. Si une telle sous chaîne n'existe pas, search renvoie None.
La commande est de la forme r.search(s, start=0, end= sys.maxint).
Notre ligne est :
re.search('^\d+\.\d+\.\d+\.\d+$',query),
nous recherchons donc ici l'adresse IP.

le module DNS:

Ce module ne fait pas partie des modules standards de python.Vous devrez donc l'installer séparement.
Vous pourrez le télécharger à partir de http://pydns.sourceforge.net.
Si vous êtes un utilisateur de Debian (comme moi), c'est très simple :
[FaSm:/home/fasm/articles/python]#apt-get install python-dns
La premiére chose que vous voudrez faire dans votre application est d'appeler DNS.DiscoverNameServers(). Cela trouvera le serveur de nom de votre systéme en utilisant la base de registre de windows ou en lisant /etc/resolv.conf pour les systémes unix.
Aprés l'initialisation des serveurs de noms, vous devrez appeler DNS.Request.
La méthode req() est utilisé pour exécuter le « lookup » actuel. Il prends typiquement deux arguments : name, qui donne le nom actuel et qtype qui spécifie un des types d'enregistrement (A, AAAA, NS, MX ...).
voici un exemple :

```
#! /usr/bin/env python
import sys,DNS
query=sys.argv[1]
DNS.DiscoverNameServers()
reqobj=DNS.Request()
answerobj=reobj.req(name=query,qtype=
DNS.Type.ANY)
if not len(answerobj.answers):
        print "  non trouve  "
for item in answerobj.answers:
        print "%-5s %s"%(item['type-
name'],item['data'])
```

Tapez ce programme et lancez le :
[FaSm:/home/fasm/articles/python]#./DNS_bas.py google.fr

et vous obtenez :

```
A     216.239.59.104
A     216.239.39.104
A     216.239.57.104
NS    ns2.google.com
NS    ns3.google.com
NS    ns4.google.com
NS    ns1.google.com
```

le module sys:

Le module sys est surtout utilisé pour

Python

récupérer les arguments passés à la ligne de commande au script python. Dans cet exemple, oublions l'interpréteur, et écrivons le script suivant que l'on enregistrera sous le nom test.py (n'oubliez pas de le rendre exécutable !) :

```
#!/usr/bin/python

import sys

print sys.argv
```

Ensuite lancez test.py suivi de plusieurs arguments, par exemple

```
[FaSm:/home/fasm/articles/python]#
./test.py gogo toto 45
['./test.py', 'gogo', 'toto', '45']
[FaSm:/home/fasm/articles/python]#
```

sys.argv est en fait une liste qui représente tous les arguments de la ligne de commande, y compris le nom du script lui même. On peut donc accéder à chacun de ces arguments avec sys.argv[1], sys.argv[2]... On peut aussi utiliser la fonction sys.exit() pour quitter le script python. On peut donner comme argument un objet (en général une chaîne de caractères) qui sera renvoyé au moment ou Python quittera le script. Par exemple, si vous attendez au moins un argument en ligne de commande, vous pouvez renvoyer un message pour indiquer à l'utilisateur ce que le script attend comme argument :

```
#!/usr/bin/python

import sys

if len(sys.argv) < 2:
  sys.exit("Usage : test.py file.gbk")
#
# ici commence le script
#
```

Puis on l'exécute sans argument :

```
[FaSm:/home/fasm/articles/python]#
./test.py
Usage : test.py file.gbk
[FaSm:/home/fasm/articles/python]#
```

Revenons à notre Premier programme:
Nous avons maintenant fait le tour des modules utilisés et de leur utilisation. Je penses qu'a ce stade le programme devient plus compréhensible. Voyons ce que cela donne pour un cas réél.
Utilisons comme précédemment le site www.google.fr (je penses qu'ils ne nous en

voudrons pas) tout ceci «étant tout à fait légal.

[FaSm:/home/fasm/articles/python]#DNSf.py google.fr
bien sur le programme a été enregistré sous le nom DNSf.py et l'argument est l'adresse du site.

Resultats pour google.fr (type de lookup ANY)

```
1 A     adresse IP: 216.239.57.104
2 A     adresse IP: 216.239.59.104
3 A     adresse IP: 216.239.39.104
4 MX    Serveur de mail: smtp1.google.com, priority 10
5 MX    Serveur de mail: smtp2.google.com, priority 20
6 MX    Serveur de mail: smtp3.google.com, priority 30
7 NS    Nom de serveur: ns1.google.com
8 NS    Nom de serveur: ns2.google.com
9 NS    Nom de serveur: ns3.google.com
10 NS   Nom de serveur: ns4.google.com
11 SOA  Start of authority:
          ns1.google.com
          dns-admin.google.com
          ('serial', 2005091303L)
          ('refresh ', 28800L, '8 hours')
          ('retry', 3600L, '1 hours')
          ('expire', 1038800L, '1 weeks')
          ('minimum', 60L, '1 minutes')
----------------------------------------------------
Resultats pour smtp1.google.com (type de lookup A)

1 A     adresse IP: 216.239.57.25
----------------------------------------------------
Resultats pour smtp2.google.com (type de lookup A)

1 A     adresse IP: 64.233.167.25
----------------------------------------------------
-Resultats pour smtp3.google.com (type de lookup A)

1 A     adresse IP: 64.233.183.25
----------------------------------------------------
Resultats pour ns1.google.com (type de lookup A)

1 A     adresse IP: 216.239.32.10
----------------------------------------------------
Resultats pour ns2.google.com (type de lookup A)

1 A     adresse IP: 216.239.34.10
----------------------------------------------------
Resultats pour ns3.google.com (type de lookup A)

1 A     adresse IP: 216.239.36.10
----------------------------------------------------
Resultats pour ns4.google.com (type de lookup A)

1 A     adresse IP: 216.239.38.10
```

Conclusion :

Nous avons fait une approche de différents modules très utilisés dans la programmation réseau en python. Il en existe bien d'autres mais leurs explications ne pourraient se faire dans un seul livre.
« Le python maintenant tu apprendras, le programmeur le plus rapide tu deviendras !

BY FASM

Python vs email

Du point de vue réseau, vous le savez maintenant, Python n'a rien à envier à un quelconque autre langage de programmation. Mais plus encore que de savoir créer une connexion, Python possèdes des librairies standard de communication selon certains protocoles. Voyons ici si nous saurions programmer un mini client mail.

Information sur les mails

La communication de mail est soumise à un tas de RFC, que nous ne détaillerons pas. Pour lire et comprendre cet article, il vous fait simplement savoir que l'envoi de mail passe utilise le protocole SMTP (Simple Mail Transfert Protocol), et la réception se fait grâce au protocole POP ou IMAP (il s'agit d'une liste non exhaustive). Les mails, donc les données qui voyagent grâce à ces protocoles de communication, sont rédigée selon une norme définie par la RFC822, c'est à dire que les champs sont prédéfini et leur renseignements doivent être formulés d'une manière précise. L'un de ces champs est censé renseigner le type de contenu, appelé type MIME.

écrire un mail

Envoyer un mail consiste en trois étapes : saisir le contenu, le formater (selon la RFC822) et l'envoyer à un serveur SMTP.

```
import smtplib
from email.MIMEText import MIMEText

contenu = "Bonjour\n"
contenu = contenu + "Comment vas tu, tante Françoise?"
contenu = contenu + "\n\nTon neveu préféré"
le_mail = MIMEText(contenu)

#Définition de l'auteur dans le contenu du mail
le_mail.add_header('From','monmail@domaine.com')

#Définition du destinataire
le_mail.add_header('To','dest_tati.francoise@cordonbleu.com')
#définition du sujet
le_mail.add_header('Subject','Coucou tatie')

print le_mail
```

Voilà le mail, écrit et formaté. Les quelques champs que nous avons remplis sont un minimum. Vous pouvez, en consultant la RFC relative au protocole SMTP, renseigner d'autres champs (la date, notamment, ou l'adresse à laquelle le destinataire doit répondre).

Nous importons smtplib (qui sert ci-après), et email.MIMEText.MIMEText(), fonction qui va nous permettre de créer notre mail uniquement composé de texte

(brut ou html). Il existe d'autres fonctions pour créer des objets MIME qui vous permettrons par exemple d'envoyer des données en fichiers joints, tels que MIMEMultiparts, MIMEAudio ou MIMEImage, que je vous laisse le soin de découvrir. Pour envoyer votre mail au format HTML, vous devriez faire appel à la fonction MIMEText ainsi :

```
contenu = "<b>Bonjour</b>\n"
contenu = contenu + "Je suis un <u>mail
de test</u>"
contenu = contenu + "\n\n<b><i>FIN DU
MAIL</i>>/b>"
le_mail = MIMEText(contenu,'html')
```

L'appel à "print le_mail" vous affiche le mail tel qu'il est stocké en mémoire, prêt à être envoyé. Une autre fonction, "le_mail.as_string()" vous l'aurait affiché en format brut (sur une seule ligne, avec les caractères de contrôle tel le retour à la ligne affichés). Il nous reste maintenant à nous connecter au serveur SMTP et à lui déposer le mail.

```
serveur =
smtplib.SMTP("smtp.domaine.com")
serveur.sendmail("monmail@domain.com",
"dest_mail@domaine2.com",email.as_strin
g())
serveur.quit()
```

Rien de plus simple : je créé une connexion avec le serveur, je lui envoie mes données et je quitte la connexion. Faites donc un test en mettant votre propre adresse en tant que destinataire, et avec le serveur smtp de votre fournisseur d'accès,pour vérifier que tout fonctionne. La fonction serveur.login(user, password) vous sera utile si le serveur smtp requiert une authentification.

Rapatrier des mails

Rapatrier un mail est un peu différent, mais pas forcément plus complexe. En effet, là encore, Python possède quelques librairies qui vont nous êtres utiles.

```
import poplib

serveur_pop =
poplib.POP3("pop.domain.com")
serveur_pop.user('koreth')
serveur_pop.pass_('python')
print serveur_pop.stat()
```

Voici la connexion et la récupération d'information sur le serveur POP. Si elle est rarement mise en place sur le protocole SMTP, l'authentification est, sur POP, nécessaire (n'importe qui ne doit pouvoir récupérer vos mails). On utilise donc les fonctions user() et pass_() pour s'identifier. Je fait ensuite appel à stat(), qui me renvoie une liste qui ne contient que deux valeurs : la première est le nombre de messages, la seconde est la taille de ces messages. Donc si je dois récupérer ces messages, je connais la taille de mon transfert. Et bien, transférons.

```
import poplib

serveur_pop =
poplib.POP3("pop.domain.fr")
serveur_pop.user('koreth')
```

Python

```
serveur_pop.pass_('python')
nb,taille = serveur_pop.stat()
print "Nombre de mail =",nb
print "Taille totale =",taille

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

Cela se complique légèrement. D'abord, ce qui ne change pas. On se connecte et s'identifie toujours au serveur. On affiche, pour info, le nombre de mail et la taille totale. Puis on va entrer dans une boucle qui va, un par un, récupérer chaque mail, dans la variable MESSAGE.
Cette variable, une fois remplie par un message, est une liste contenant trois champs. Le premier indique le code de retour du serveur (OK ou non). Le second champ renferme le mail, avec ses champ (from, to, subject, ...) et le contenu du mail (le message à tante Françoise). Le troisième et dernier champ contient la taille du mail (sans le premier champ et le troisième, donc).
Le second champ, qui contient donc le mail, est lui aussi un tableau, dont chaque case représente une ligne. Je créé une petite boucle pour l'afficher, avec un retour à chaque fin de ligne, pour que nous puissions analyser le contenu. Et là, sans surprise, vous retrouvez à peu près le même type d'entête que dans le paragraphe ci-dessus sur l'envoie de mail.
Maintenant, il serait intéressant de programmer un petit logiciel en Python qui m'informerait si je reçois un mail que j'attends avec impatience. Le fonctionnement interne serait le suivant : je renseigne une adresse mail dont j'attends un courrier important, et Python va regarder si j'ai reçu un mail de cette adresse. Mais pour ce faire, je m'interdis de traiter ligne par ligne mon mail, avec des IFs, pour trouver la ligne contenant le champ « From: » : les en-têtes de mail ne sont pas toujours très propres, et le test pourrait se révéler aléatoire. Je vais plutôt, dès réception, confier mon message brut et le transformer en objet mail.

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

serveur_pop =
poplib.POP3("pop.domain.com")
serveur_pop.user('koreth')
serveur_pop.pass_('python')
nb,taille = serveur_pop.stat()
print "Nombre de mail =",nb
print "Taille totale =",taille

for i in range(nb):
    print "----------------------------"
    print "MESSAGE SUIVANT"
    print "----------------------------"
    message = serveur_pop.retr(i+1)
    mail_inline = ""
    for info in message[1]:
        mail_inline = mail_inline+info+"\n"
    mon_objet_message =
message_from_string(mail_inline)
    print mon_objet_message.get('From')
```

Voilà le script au complet. Je récupère tous les mails, un par un, grâce à une première boucle. Dans celle-ci, j'intègre une seconde boucle, pour créer, à partir du tableau contenant toutes les lignes du mail, un mail en une ligne, en rajoutant les « \n » entre chaque champs. Je passe ceci dans la moulinette (message_from_string()), ce qui me construit un objet message De cet objet, je peut faire appel à .get_payload(), qui me renvoie uniquement le contenu du mail. Etant donné que ce qui m'intéresse est l'expéditeur, je vais aller lire le champ From, grâce à la fonction get(). Elle me permet de lire la valeur de n'importe quel champ de ce mail.

Si vous testez ce script, vous vous frotterez peut-être aux véritable casse-tête qu'est la gestion de l'encodage des mails : vous verrez des accents apparaître sous forme de code ASCII, il vous faudra jongler entre l'utf8, l'ISO-8859-1 et l'ISO-8859-15 (pour ne citer que ceux-là). De plus, il faut aussi gérer le fait que maintenant, les mail ne se limitent plus à du texte, mais aussi à des images et du son, ce qu'il faut également gérer (grâce, on l'a vu plus haut, à MIMEImage ou MIMEMultipart). Néanmoins, vous avez tous les outils maintenant pour interroger un serveur POP et envoyer un mail en communicant avec un serveur SMTP. Sachant qu'il existe, dans poplib, une implémentation de POP over SSL (Protocole POP Sécurisé), qu'il existe aussi la librairie imaplib, plus aucune limite, sinon celle du temps et de la patience, ne devrait vous fermer les portes du joyeux monde des mails.

Conclusion

Et bien, on en sait des choses maintenant !! On commence à être bon en python ;-) .
Je penses due l'on en sait assez maintenant sur l'utilisation des bibliothéques, nous allons pouvoir passer à autre choses. La programmation python pour le web est aussi très intéressante. Plongeons nous dedans grâce aux articles suivants.

Python

PARSER LES FICHIERS HTML

Il est parfois utile d'aller lire des informations sur une page internet. Par exemple, le site http://checkip.dyndns.org affiche toujours le même message vous indiquant votre adresse IP. Il peut être interessant de récuperer juste cette adresse, ce qui se fait simplement dans ce cas précis puisqu'il ne s'agit que de texte. Par contre, quand nous voulons extraire des informations sur un site qui se veut joli, donc qui contient un grand nombre de balises, cela peut se réveler plus complexe. Mais il existe encore une fois un très bon outils : BeautifulSoup pour parser vos fichiers html, et urllib pour aller lire un fichier html sur le net.

Installation

Installer BeautifulSoup est assez simple, il suffit d'aller chercher le paquet correspondant sur le site web, et de mettre lefchier BeautifulSoup,py dans votre répertoire lib. urllib est quand à elle une librairie standard de Python.

```
import urllib
from BeautifulSoup import BeautifulSoup

lienweb =
urllib.urlopen("http://www.python.org/inde
x.html")
pageweb = lienweb.read()
soupe = BeautifulSoup(pageweb)

print "Affichage de la balise TITLE"
print soupe.html.head.title
print "Affichage du titre uniquement"
print soupe.html.head.title.string
```

Voici un apercu de ce que peu faire BeautifulSoup couplé à urllib. Cette dernière va nous permettre de lire un fichier à un url exactement de la même manière que si nous faisions un open() sur un fichier local. Pratique, notamment avec l'utilisation avec BeautifulSoup.
BeautifulSoup s'initialise grâce à la fonction du même nom, à laquelle on passe notre page web en paramètre. La fonction va recréer un arbre hierarchique à partir de la page web.

```
<html>
        <head>
                <title>titre de la
page</title>
        </head>
        <body>
                <p>
                premier paragraphe
                        <a> lien </a>
                </p>
```

```
                <p>
                paragraphe second
                </p>
        </body>
</html>
```

Accéder à la balise Title revient donc à suivre le cheminement suivant : on part du premier noeud (ou racine), donc HTML, puis on entre dans le noeud HEAD, et on lis la balise. Ce qui se traduit part "print soup.html.head.title". Pour obtenir un résultat nettoyé de ses balises, nous affichons la valeur title.string. Et en règle générale, obtenir un resultat nettoyé se fera en apposant le suffixe .string,

RECUPERER LES LIENS

BeautifulSoup peut et est utilisé dans les outils statistiques Web. En effet, il est très simple de récupérer tous les liens contenus dans une page. Souvenez-vous que nous disposions, au début de ce manuel, d'une technique viable mais longue : les regex. Voici la méthode simple et efficace, avec BeautifulSoup :

```
import urllib
from BeautifulSoup import BeautifulSoup

lienweb =
urllib.urlopen("http://www.python.org/index.html")
pageweb = lienweb.read()
soupe = BeautifulSoup(pageweb)

print "\n\naffichage de tous les liens de la page"
print soupe('a')
print "\n\naffichage du premier lien trouvé"
print soupe('a')[0]
print "\n\naffichage de l'attribut href du premier lien trouvé"
print soupe('a')[0]["href"]
print "\n\nAffichage des attributs href de tous les liens trouvés"
for i in soupe('a') :
        try :
                print i["href"]
        except :
                pass
```

Vous voyez ci-dessus le cheminement pour chercher toutes les occurrences d'une balise et récuperer ensuite ses attributs. La chose n'est pas simple à comprendre si on ne connait pas bien le HTML. Voici une aide :

```
<BALISE
ATTRIBUT=VALEUR>TEXTE</BALISE>
```

Pour recenser toutes les BALISES, nous utilisons soupe('BALISE_VOULUE')
Pour Accéder à la valeur d'un attribut : soupe(BALISE_VOULUE)[NUMERO D'OCCURENCE]['Attribut']

Maintenant, il est arrive que vous rencontriez ce genre de lien :

```
<a
href="http://domain.tld/index2.html><h2>
Acces INDEX2.html</h2></a>
```

Pour isoler le texte, il faut faire : soupe('a')[0].h2.string (avec 0 si on considère que ce lien est la première occurrence trouvée). Ci-dessus, je gère une exception au moment où je liste tous les hrefs, car il se peut qu'un lien ne contienne pas ce

genre d'attribut (ce qui est assez rare, mais il serait embêtant que le script pose problème la dessus).

Ensuite, nous pouvons un peu resserer les critères de recherche, en spécifiant que notre balise doit comporter tel ou tel critère, en donnant un second paramètre à notre commande soupe() : un dictionnaire.

```
import BeautifulSoup
import urllib

lienweb =
urllib.urlopen("http://www.python.org/index.html")
pageweb = lienweb.read()
soupe = BeautifulSoup.BeautifulSoup(pageweb)

print "Comptons le nombre de balises td"
print len(soupe('td'))
print "Maintenant, le nombre de balises td de classes body"
print len(soupe('td',{"class" : "body"}))
```

Ce qui nous permet de faire une recherche bien ciblée. Plusieurs attributs peuvent être passés en arguments, et l'on peut même utiliser les expressions régulières.

```
import re
(REINSERER ICI LE CODE CI-DESSUS)
print soup('td', {'class' : re.compile('^.*r$')})
```

Et voici comment je récupère la listes des balises COLONNE (<td>) dont l'attribut CLASS se termine par un "r". Certes, dans le cas présent, l'utilité est mince. Mais ainsi, vous pouvez affiner votre robot pour l'adapter au mieux au site dont nous voulons extiprer les informations. Attention cependant à ne pas trop affiner car en cas de changement du site cible (notamment un changement dans le fichier de style css), vous pouvez tout avoir à refaire.
Sachez aussi que BeautifulSoup est capable de parser n'importe quel language balisé comme l'est HTML, avec des balise de tupe <TAG> et </TAG> pour l'ouverture et la fermeture. Et bien que minidom soit fait pour Cela, un fichier XML peut être parsé avec BeautifulSoup. Et plus précisément les flux RSS, écrits en XML, que BeautifulSoup supporte par l'intermédiaire d'un autre modules, Scrap'n'feed.

Vous savez a peu près tout ce qu'il y a savoir des bases du scrapping grâce à BeautifulSoup. Après, et si l'on en croit les témoignage, la seule limite reste votre imagination et l'utilité que vous aurez à récupérer des informations à partir du HTML. Consultez la page officielle de BeautifulSoup pour voir les plus beau exemple ce qui à été fait avec ce module (ce qui va de la recherche de résultats de baseball à la récupération de liste des incidents de piratage commercial), ainsi que pour avoir accès à la documentation.

Conclusion :

Vous allez me dire, le html c'est bien mais c'est basique, maintenant on utilise des choses plus "compliquées" tels que le xml. Mon p'tit gars, y a pas de soucis, y a s'qui faut pour python !
Attends, la porte s'ouvre, c'est qui ?

Ah, c'est TaLi, t
Tu peux écrire
sous python ?
Bon ba, je vou
ma place pour

PYTHON XML

Présentation :

Python est un langage orienté objet simple, puissant et possédant un ensemble de librairies riches et variées. Du XML (eXtensible Markup Language) avec DOM et SAX en passant par les DNS ou encore par les mails. Mais également à l'aide de bibliothèques graphiques puissantes ou l'interaction avec un gestionnaire de bases de données, Python se défend dans tous les domaines.

voilà, je me suis assis. T'as le cul chaud KoReTh ! En plus, il y a plein de bouffe sur le clavier ! Mais bon, la n'est pas le but de l'article. Le xml en quelques pages, pas facile mais bon pour la Team Ac'ISSI rien d'impossible ;-) . Alors c'est parti.

- Pourquoi utiliser XML ? Tout d'abord, il est indépendant du programme, lisible, standardisé W3C et sa structure est hiérarchisée. Autant de points forts qui font l'intérêt de ce langage.
- Un fichier XML est un simple fichier texte comportant l'extension .xml, qui permet de stocker des informations formatées suivant certains besoins.
- Chaque information est stockée entre deux "tags" ou "balises" xml. L' exemple le plus connu de l'utilisation du XML est le HTML. Mais il également possible d'utiliser le XML pour stocker des données brutes, il suffit de parser ce fichier pour en récupérer son contenu.

- Un fichier xml se représente de la façon suivante :

Ce fichier XML représente le sommaire d'un livre, chaque noeud ou tag, contient des données ou des attributs qui peuvent être modifier selon nos besoins. La structure du fichier commence toujours par un tag « racine », suivit d'autres noeuds qui constituent une liste d'éléments imbriqués. On peut associer les noeuds d'un document à une arborescence de répertoires et sous répertoires avec comme fichier racine la balise <sommaire> et ainsi de suite.

Une règle importante, contrairement au HMTL, tous les tags ouvrant doivent obligatoirement être fermés plus loin dans le fichier. Pour une personne qui a l'habitude du HTML, cette règle sera sûrement nouvelle.

- Pourquoi choisir XML plutôt qu'un fichier plat ou encore une base de données ?

Il existe de nombreux comparatifs qui vous aideront dans votre choix, mais à mon humble avis, XML permet d'avoir une structure clairement définie, qui donne un avantage par rapport aux fichiers plats et par rapport au SGBD, son atout reste sa facilité d'utilisa-

Python

API DOM

```
<?xml version="1.0" ?>
        <sommaire>
        <!--Ceci est un exemple-->
                <titre>Prog Pyhton</titre>
                        <chapitre numero="1">
                                <titre>Introduction</titre>
                                <date annee="2006" jour="10"
mois="Janvier"/>
                                <img src="logo1.gif"/>
                                <auteur>Koreth</auteur>
                        </chapitre>
                        <chapitre numero="1">
                                <titre>Présentation</titre>
                                <date annee="2006" jour="11"
mois="Janvier"/>
                                <img src="logo2.gif"/>
                                <auteur>TaLi</auteur>
                        </chapitre>
        </sommaire>
```

tion. Le XML n'étant pas le sujet de cet article, je vous laisse le soin de lire les nombreuses documentations disponibles sur le net.

Python et XML :

Pour manipuler des fichiers XML, nous utiliserons l'API DOM (Document Object Model) réf : http://www.w3.org/DOM/, qui contrairement à SAX doit générer un arbre et donc lire l'ensemble du fichier, SAX quand à lui est capable de travailler sur des fichiers de très grande taille, ainsi que de traiter les problèmes d'espaces entre balises et texte. SAX est l'abréviation de "Simple API for XML". C'est l'API la plus adaptée pour lire un document XML en entier et réaliser des traitements. On peut par exemple construire une structure rassemblant les données du document. Par contre si il s'agit de modifier une structure XML existante, SAX n'est pas très performant. Pour ce genre de tâche, l'usage de DOM est préférable. Quelle est vraiment la différence entre ces parseurs ? :

SAX : Lie des fonctions à des événements. Lorsque l'événement se produit (action d'ouverture ou fermeture d'une balise), la fonction en question est appelée.

DOM : Charge en mémoire le fichier et retourne une structure xml. Ensuite on peut se balader entre les noeuds (monter, descendre...). Les actions réalisées peuvent être complexes si on compare à SAX, mais DOM est beaucoup plus gourmand en terme de ressources.
Grâce à cette API, nous allons pouvoir lire un fichier XML mais également ajouter de nouveaux éléments à ce fichier. Il faut savoir qu'avec DOM, tout est élément ou noeud, si l'on reprend l'exemple ci-dessus, le tag <titre> de type élément contient Introduction de type text. Un des inconvénient principal de cette API est qu'elle est peu adaptée pour de gros documents : l'arbre généré en mémoire à partir du fichier XML est environ 16 fois plus gros que le fichier.
La théorie finie passons à la pratique, voici un petit bout de code pour générer le fichier XML. Pour l'écriture de ce code, il est important de bien connaître la structure d'un fichier XML et la notion de parent/enfant entre les noeuds du fichier. En aucun cas les balises ne doivent se recouvrir : le XML impose une hiérarchie stricte. Avant toute chose, veillez à ce que les librairies pyhton-xml soient installées sur votre système, mes tests ont été réalisés sur Linux, avec python 2.4 et les libs fournies dans les paquets Debian. Référez-vous à ce lien http://pyxml.sourceforge.net/topics/download.html pour trouver le package PyXML disponible pour Windows/Linux qui inclut toutes les librairies nécessaires dont minidom qui est l'implémentation de DOM nécessaire pour cet exemple .

Exemple :

```
from xml.dom.minidom import Document      # Importation des modules liés au parsing XML

document = Document()    # Instanciation de l'objet document, on crée le document XML
racine = document.createElement("racine")    # Déclare Racine du document XML
document.appendChild(racine)
child = document.createElement("child")  # On ajoute l'élément child dans l'arbre
child.setAttribute("id", "10")            # Ajout de l'attribut id au noeud Child
racine.appendChild(child)    # On crée le noeud fils de l'élément racine

print document.toxml()      # Redirige la sortie standard en xml
```

L'exécution de cet exemple provoque l'affichage du code XML suivant:

```
<?xml version="1.0" ?>
        <racine>
                <child id=10/>
        </racine>
```

Un document admet une racine unique <racine>, qui est le départ de tout document XML.

Lors de la création du noeud Child, l'appel à la méthode appendChild se fait sur l'objet racine afin que le tag Child devienne un noeud fils de racine.

Dans l'exemple qui suit le programme génère l'arbre XML présenté au début de cet article, dans un fichier doc.xml dans le répertoire ou sera exécuté le programme.

Exemple :

```
        from xml.dom.minidom import Document      # Importation des modules liés au parsing XML
        import xml.dom.ext                                  # Implémentation de l'API Dom

        doc = Document()          # Instanciation de l'objet document, on crée le document XML

        noeud_racine = doc.createElement("sommaire")        # Création de élément sommaire
        doc.appendChild(noeud_racine)       # On ajoute un noeud enfant de doc

        comment = doc.createComment("Ceci est un exemple")  # Il faut commenter son code ;o)
        noeud_racine.appendChild(comment)

        titre = doc.createElement("titre")     # On ajoute le noeud titre enfant de doc
        noeud_racine.appendChild(titre)
        contenu = doc.createTextNode("Prog Pyhton")
```

```
        titre.appendChild(contenu) # On ajoute le noeud contenu enfant de titre

        element = doc.createElement("chapitre")
        element.setAttribute("numero", "1")          # setAttribute ajout d'un attri-
but à l'élément chapitre
        noeud_racine.appendChild(element)

        titre = doc.createElement("titre")
        element.appendChild(titre)
        contenu = doc.createTextNode("Introduction")
        titre.appendChild(contenu)

        date = doc.createElement("date")
        date.setAttribute("jour", "10")        # On ajoute un attribut à l'élément date
        date.setAttribute("mois", "Janvier")
        date.setAttribute("annee", "2006")
        element.appendChild(date)

        img = doc.createElement("img")
        img.setAttribute("src", "logo1.gif")
        element.appendChild(img)

        auteur = doc.createElement("auteur")
        element.appendChild(auteur)
        contenu = doc.createTextNode("Koreth")
        auteur.appendChild(contenu)

        element = doc.createElement("chapitre")
        element.setAttribute("numero", "1")
        noeud_racine.appendChild(element)

        titre = doc.createElement("titre")
        element.appendChild(titre)
        contenu = doc.createTextNode("Présentation")
        titre.appendChild(contenu)

        date = doc.createElement("date")
        date.setAttribute("jour", "11")
```

```
date.setAttribute("mois", "Janvier")
date.setAttribute("annee", "2006")
element.appendChild(date)

img = doc.createElement("img")
img.setAttribute("src", "logo2.gif")
element.appendChild(img)

auteur = doc.createElement("auteur")
element.appendChild(auteur)
contenu = doc.createTextNode("TaLi")
auteur.appendChild(contenu)

def output_xml(doc,dir_xml):        #Fonction  qui retourne un fichier xml
file = open(dir_xml,"w")            # Ouverture du fichier XML
xml.dom.ext.PrettyPrint(doc,file)   # Enregistrement indenté

output_xml(doc,"mon_premier_document.xml") # Appel de la fonction
```

Dans notre exemple, nous faisons appel à la méthode createElement pour déclarer un tag xml dans l'arbre XML, ensuite il faut passer la méthode AppendChild qui s'occupe de créer le noeud dans l'arbre. En réalité cette méthode ajoute un noeud enfant à celui précédemment créé.

Lors de cet appel doc.appendChild(noeud_racine) le premier noeud que l'on alloue dans l'arbre est tout simplement un enfant de la racine du document, créé au début du programme. Ensuite on appelle successivement cette méthode pour construire l'arborescence de l'arbre XML.

Output_xml est une fonction qui prend en paramètre un nom de l'objet de type document, et le lien où l'on stockera le fichier xml en sortie. Noter que l'on peut également afficher le résultat sans passer par cette fonction, en utilisant print doc.toxml().

Si la structure XML apporte lisibilité et robustesse, le problème d'accès à l'information demeure.

L'API Dom fournit des méthodes pour parser un fichier et récupérer facilement les informations que contiennent les balises. Il existe une méthode très utile getElementsByTagName('auteur') qui renvoie la liste de tous les éléments contenus dans les balises <auteur></auteur>. D'autres attributs d'objet s'avèrent utiles pour la lecture de vos arbres.

correspond au noeud parent

firstChildincarne le premier fils du noeud parent

previousSiblingpointe sur le noeud précédent le fils aillant le même parent

nextSiblingpointe sur le noeud suivant le fils aillant le même parent

Vous retrouverez bien sûr tous les attributs et métho
la doc de Py
complète.
Grâce à DOM
votre propre
nées RSS, ou
marks dans un
pulant faciler
XML, sans pou
les contraintes
nées. La man
ments XML en
que peu d'effo
plicité des API
Cela fait de P
choix pour l'ex

Conclusio

Vous en savez
xml et python.
obtenir toutes
le net pour ap
Bon c'est pas l
hack 2006 à M
le 3 et 4 juin,
encore plein d
main pour les

Python