Full Circle

LE MAGAZINE INDÉPENDANT DE LA COMMUNAUTÉ UBUNTU LINUX

ÉDITION SPÉCIALE SÉRIES PROGRAMMATION

ÉDITION SPÉCIALE
SÉRIES PROGRAMMATION

PROGRAMMER EN PYTHON

Volume trois

About Full Circle

Full Circle is a free, independent, magazine dedicated to the Ubuntu family of Linux operating systems. Each month, it contains helpful how-to articles and reader- submitted stories. Full Circle also features a companion podcast, the Full Circle Podcast which covers the magazine, along with other news of interest.

Please note: this Special Edition is provided with absolutely no warranty whatsoever; neither the contributors nor Full Circle Magazine accept any responsibility or liability for loss or damage resulting from readers choosing to apply this content to theirs or others computers and equipment.

Spécial Full Circle Magazine

Full Circle

LE MAGAZINE INDÉPENDANT DE LA COMMUNAUTÉ UBUNTU LINUX

Welcome to another 'single-topic special'

In response to reader requests, we are assembling the content of some of our serialised articles into dedicated editions.

For now, this is a straight reprint of the series 'Programming in Python', Parts 22-26 from issues #48 through #52; nothing fancy, just the facts.

Please bear in mind the original publication date; current versions of hardware and software may differ from those illustrated, so check your hardware and software versions before attempting to emulate the tutorials in these special editions. You may have later versions of software installed or available in your distributions' repositories.

Enjoy!

Rejoignez-nous

Site web :
http://www.fullcirclemagazine.org/

Forums :
http://ubuntuforums.org/forumdisplay.php?f=270

IRC : #fullcirclemagazine on chat.freenode.net

Équipe éditoriale
Rédacteur en chef : Ronnie Tucker (aka: RonnieTucker)
ronnie@fullcirclemagazine.org
Webmaster : Rob Kerfia (aka: admin / linuxgeekery-
admin@fullcirclemagazine.org
Podcaster : Robin Catling (aka RobinCatling)
podcast@fullcirclemagazine.org
Dir. comm. : Robert Clipsham (aka: mrmonday) -
mrmonday@fullcirclemagazine.org

TUTORIEL

Programmer en Python - Partie 17

Greg Walters

En terminant l'article précédent de notre série, j'ai reçu un courriel au sujet d'une compétition de programmation. Nous n'avons pas le temps de traiter celle-ci, mais plusieurs sites proposent des compétitions de programmation au cours de l'année. L'information concernant cette compétition, si vous êtes intéressé, se trouve ici : http://www.freiesmagazin.de/third programming contest. Ceci m'a fait penser que nous n'avons pas encore parlé de programmation client/serveur. Nous allons donc nous y lancer avec cette idée derrière la tête et nous verrons bien où cela nous mène.

Alors, qu'est-ce qu'une application client/serveur ? Pour faire simple, à chaque fois que vous utilisez un programme (ou même une interface web) qui accède à des données d'une autre application ou d'un autre ordinateur, vous utilisez un système client/serveur. Examinons un exemple que nous avo,s déjà utilisé. Vous souvenez-vous de notre programme de livre de recettes ? C'était un exemple TRÈS simple (et pas très bon) d'une application client/serveur. La base SQLite était le serveur, l'application que nous avons décrite était le client. L'exemple suivant serait meilleur : imaginez qu'une base de données soit sur un ordinateur à un autre endroit de votre entreprise, à un autre étage ; elle contient des informations sur l'inventaire du magasin dans lequel vous travaillez. Vous utilisez une caisse à un des dix points de vente dans le magasin : chacune de ces caisses est un client et la base de données située ailleurs est le serveur.

Même si nous ne cherchons pas à créer un tel système ici, nous pouvons apprendre quelques-unes des bases.

La première chose à laquelle il faut réfléchir est l'emplacement de notre serveur. De nombreuses personnes n'ont qu'un seul ordinateur à la maison. Certains en ont parfois 7 ou 8.

Pour utiliser un système client/serveur, nous devons nous connecter à la machine serveur depuis la machine cliente. On fait cela avec ce qu'on appelle un « pipe » ou un « socket » [Ndt : un connecteur]. Si vous avez déjà fabriqué un téléphone avec deux boîtes de conserves quand vous étiez enfant, vous avez une idée de ce dont je vais vous parler. Sinon, laissez-moi vous décrire une scène d'autrefois. D'abord il fallait demander à votre mère qu'elle vous garde deux boîtes de conserves de haricots ou autre. Puis il fallait les nettoyer soigneusement et les apporter au garage. Ensuite, en utilisant un petit clou et un marteau, vous perciez un petit trou sur le fond de chaque boîte. Puis avec 40 cm de ficelle (que vous avait donné votre maman chérie), il fallait passer la ficelle dans les deux boîtes et faire un gros nœud à chaque bout à l'intérieur des boîtes. Enfin, vous alliez chercher votre meilleur copain et, en tendant bien fort la corde, vous pouviez crier dans une des boîtes pendant que le copain plaçait l'autre près de son oreille. Les vibrations du fond de la boîte traversaient la corde tendue et faisaient vibrer le fond de l'autre boîte. Bien sûr, vous pouviez entendre sans la boîte, mais c'était hors de propos. C'était chouette. Le « socket » est à peu près la même chose. Le client a une connexion directe (pensez à la ficelle) au serveur. Si plusieurs clients sont connectés au serveur, chaque client aurait sa propre boîte de conserves et le pauvre serveur devrait avoir le même nombre de boîtes avec des ficelles bien tendues jusqu'à chaque client. L'essentiel ici est que chaque client ait sa propre ligne jusqu'au serveur.

Fabriquons un serveur et un client simples. Commençons par le serveur. En pseudo-code, voici ce qui se passe.

```
Créer un connecteur.
Récupérer le nom du serveur.
Choisir un port.
Relier le connecteur à
l'adresse et au
port.
Écouter s'il y a une
connexion.
Si c'est le cas :
accepter la connexion ;
afficher qu'on est connecté ;
fermer la connexion.
```

Vous pouvez voir le code pour le serveur en bas à gauche de la page suivante.

Ainsi, on crée le connecteur, on récupère le nom de la machine sur

laquelle tourne le serveur, on relie le connecteur au port et on commence à écouter. Quand on reçoit une demande de connexion, on l'accepte, on affiche que l'on vient d'établir une connexion, on envoie « Bonjour et au revoir » et on ferme le connecteur.

Maintenant, il nous faut un client pour que l'ensemble fonctionne (voir en bas à droite).

Le code est presque le même que pour le serveur, mais cette fois-ci on se connecte, on affiche ce qu'on a reçu et on ferme la connexion.

Les sorties du programme dont très prévisibles. Du côté du serveur, on obtient :

```
Mon nom est terre.
Je suis connecte a
('127.0.0.1', 45879).
```

et du côté du client, on obtient :

```
Bonjour et au revoir.
```

C'est donc plutôt simple. Maintenant faisons quelque chose de plus réaliste. Nous allons créer un serveur qui va vraimet faire quelque chose. Le code pour la version 2 du serveur est disponible ici http://fullcirclemagazine.pastebin.com/jZJvqPym

Décortiquons-le. Après les « import », on règle quelques variables. BUFSIZ contient la taille du tampon qui sera utilisé pour contenir l'information que l'on recevra du client. On règle aussi le port sur lequel on va écouter et une liste contenant le nom de l'hôte et le numéro du port. Ensuite on crée une classe nommée « ServCmd ». Dans la routine init, on crée un connecteur et on relie l'inter-

```
#!/usr/bin/env python
#server1.py
import socket
soc = socket.socket()
nom_hote = socket.gethostname()
print "Mon nom est ", nom_hote
port = 21000
soc.bind((nom_hote,port))
soc.listen(5)
while True:
    con,adresse = soc.accept()
    print "Je suis connecte a ",adresse
    con.send("Bonjour et au revoir")
    con.close(
```

```
#!/usr/bin/env python
# client2.py
from socket import *
from time import time
from time import sleep
import sys
BUFSIZE = 4096
class CmdLine:
    def __init__(self,host):
        self.HOST = host
        self.PORT = 29876
        self.ADDR = (self.HOST,self.PORT)
        self.sock = None
    def seConnecte(self):
        self.sock = socket( AF_INET,SOCK_STREAM)
        self.sock.connect(self.ADDR)
    def envoieCommande(self, cmd):
        self.sock.send(cmd)
    def recupResultats(self):
        data = self.sock.recv(BUFSIZE)
        print data
if __name__ == '__main__':
    conn = CmdLine('localhost')
    conn.seConnecte()
    conn.envoieCommande('ls -al')
    conn.recupResultats()
    conn.envoieCommande('AU REVOIR')
```

```
#!/usr/bin/python
# client1.py
#====================
import socket
soc = socket.socket()
nom_hote = socket.gethostname()
port = 21000
soc.connect((nom_hote, port))
print soc.recv(1024)
soc.close
```

face à ce connecteur. Dans la routine « run », on commence à écouter et on attend une commande du client.

Lorsqu'on reçoit une commande du client, on utilise la routine os.popen() qui, en résumé, crée une invite de commande et exécute la commande.

Passons au client (en haut à droite, page précédente), qui est sensiblement plus simple. On ne va expliquer ici que la commande « send », car vous avez maintenant assez de connaissances pour comprendre le reste par vous-mêmes. La ligne coo.envoieCommande() (ligne 31) envoie une simple requête « ls -al ». Voici à quoi ressemblent mes réponses, les vôtres seront quelque peu différentes.

Serveur :

```
python server2.py
Ecoute le client...
Connecte a ('127.0.0.1',
42198)
Recu la commande : ls -al
Recu la commande : AU REVOIR
Ecoute le client...
```

Client :

```
python client2a.py total 72
drwxr-xr-x 2 greg greg 4096
2010-11-08 05:49 .
drwxr-xr-x 5 greg greg 4096
2010-11-04 06:29 ..
-rw-r--r-- 1 greg greg 751
2010-11-08 05:31 client2a.py
-rw-r--r-- 1 greg greg 760
2010-11-08 05:28
client2a.py~
-rw-r--r-- 1 greg greg 737
2010-11-08 05:25 client2.py
-rw-r--r-- 1 greg greg 733
2010-11-08 04:37 client2.py~
-rw-r--r-- 1 greg greg 1595
2010-11-08 05:30 client2.pyc
-rw-r--r-- 1 greg greg 449
2010-11-07 07:38 ping2.py
-rw-r--r-- 1 greg greg 466
2010-11-07 10:01
python_client1.py
-rw-r--r-- 1 greg greg 466
2010-11-07 10:01
python_client1.py~
-rw-r--r-- 1 greg greg 691
2010-11-07 09:51
python_server1.py
-rw-r--r-- 1 greg greg 666
2010-11-06 06:57
python_server1.py~
-rw-r--r-- 1 greg greg 445
2010-11-04 06:29 re-test1.py
-rw-r--r-- 1 greg greg 1318
2010-11-08 05:49 server2a.py
-rw-r--r-- 1 greg greg 1302
2010-11-08 05:30
server2a.py~
-rw-r--r-- 1 greg greg 1268
2010-11-06 08:02 server2.py
-rw-r--r-- 1 greg greg 1445
2010-11-06 07:50 server2.py~
-rw-r--r-- 1 greg greg 2279
2010-11-08 05:30 server2.pyc
```

On peut aussi se connecter depuis une autre machine sans aucun changement, à la seule exception de la ligne 29 : conn = CmdLine('localhost') dans le programme client. Dans ce cas, remplacez « localhost » par l'adresse IP de la machine sur laquelle tourne le serveur. Chez moi, j'utilise la ligne suivante :

```
conn =
CmdLine('192.168.2.12')
```

Et voilà, maintenant on peut envoyer de l'information d'une machine (ou d'un terminal) à une autre.

La prochaine fois, nous rendrons notre application client/serveur plus robuste.

Idées et auteurs souhaités

Nous avons créé les pages du projet Full Circle et de son équipe sur Launchpad. L'idée étant que des personnes qui ne sont pas auteurs puissent aller sur la page du projet, cliquer sur « Answers » [Ndt : Réponses] en haut de la page et laisser leurs idées d'article, mais merci d'être précis dans vos idées ! Ne laissez pas seulement « article sur les serveurs », spécifiez s'il vous plaît ce que le serveur devrait faire !

Les lecteurs qui aimeraient écrire un article, mais qui ne savent pas à propos de quoi écrire, peuvent s'inscrire sur la page de l'équipe du Full Circle, puis s'attribuer une ou plusieurs idées d'articles et commencer à écrire ! Nous vous demandons expressément, si vous ne pouvez terminer l'article en quelques semaines (au plus un mois), de rouvrir la question pour laisser quelqu'un d'autre récupérer l'idée.

La page du projet, pour les idées :
https://launchpad.net/fullcircle
La page de l'équipe pour les auteurs :
https://launchpad.net/~fullcircle

TUTORIEL

Programmer en Python - Partie 18

Greg Walters

La dernière fois, nous avons créé un système client/serveur très simple. Cette fois-ci, nous allons l'améliorer un peu. Le serveur est un plateau de jeu Tic-Tac-Toe (ou jeu du morpion). La partie client sert à faire les saisies et l'affichage.

Nous allons repartir du code du serveur de la dernière fois et le modifier au fur et à mesure. Si vous n'avez pas gardé ce code, rendez-vous sur http://fullcirclemagazine.pastebin.com/jZJvqPym. Pour récupérer le code source de la dernière fois avant de continuer. Le premier changement se situe dans la routine __init__ où nous initialisons deux variables supplémentaires : self.joueur et self.plateau. Le plateau de jeu est simplement un tableau, autrement dit une liste de listes. On peut y accéder comme ci-dessous (c'est plus visuel qu'une liste plate). La liste contiendra nos données. Il y a trois entrées possibles par cellule : « - » signifie que la cellule est vide, « X » signifie qu'elle est occupée par le joueur 1 et « O » qu'elle est occupée par le joueur 2. La grille ressemble à ceci si on la représente en deux dimensions :

```
[0][0] | [0][1] | [0][2]
[1][0] | [1][1] | [1][2]
[2][0] | [2][1] | [2][2]
```

Donc, en repartant du code du serveur de la dernière fois, dans la routine __init__, ajoutez les lignes suivantes :

```
# Les trois prochaines lignes
sont nouvelles

self.joueur = 1

self.plateau = [['-','-','-'],
['-','-','-'],['-','-','-']]

self.run()
```

Les routines run, ecoute et servCmd sont inchangées, donc nous allons nous concentrer sur les changements de la routine exeCommande.

Dans l'article précédent, le serveur attendait une commande du client, puis l'envoyait à la routine os.popen. Cette fois-ci, nous allons analyser la commande envoyée. Dans notre cas, nous avons trois commandes différentes qui peuvent arriver : « Debut », « Deplace » et « AU REVOIR ». Lorsqu'on reçoit la commande « Debut », le serveur doit initialiser le plateau de jeu en plaçant des « - » partout, puis envoyer un « Afficher le plateau » au client.

La commande « Deplace » est une commande composée, dans le sens où elle contient la commande, mais aussi la position où le joueur souhaite se déplacer. Par exemple, « Deplace A3 ». On analyse la commande pour obtenir trois morceaux : la commande « Deplace » elle-même, la ligne et la colonne. Finalement, la commande « AU REVOIR » remet simplement le plateau à zéro pour un nouveau jeu.

Ainsi, on reçoit la commande du client dans la routine exeCommande. On examine ensuite la commande pour vérifier ce qu'on doit faire. Dans la routine exeCommande, cherchez la cinquième ligne et, après la ligne qui contient « if self.boucleGestion: », enlevez le reste du code. Maintenant, nous allons régler les commandes comme nous l'avons prévu. Voici le code pour la commande « Debut » :

```
if self.boucleGestion:
if cmd == 'Debut':
self.InitialisePlateau()
self.AffichePlateau(1)
```

Ensuite, nous allons regarder la partie « Deplace » de la routine (voir ci-dessous). Nous vérifions d'abord les sept premiers caractères de la commande passée pour voir s'il s'agit de « Deplace ». Si oui, on prend alors de reste de la chaîne en commençant à la position 8 (puisqu'on compte à partir de 0), et on l'assigne à la variable position. On vérifie ensuite si le premier caractère est « A » ou « B » ou « C ». Il représente la ligne que le client a envoyée. On récupère ensuite l'entier dans le caractère suivant et c'est le numéro de colonne :

```
if cmd[:7] == 'Deplace':
print "COMMANDE DEPLACE"
position = cmd[8:]
if position[0] == 'A':
    ligne = 0
elif position[0] == 'B':
    ligne = 1
elif position[0] == 'C':
    ligne = 2
else:
    self.cli.send('Position
invalide')
    return
col = int(position[1])-1
```

Puis on vérifie rapidement que le numéro de la ligne est dans la plage

de valeurs autorisées :

```
if ligne < 0 or ligne > 2:

self.cli.send('Position invalide')

return
```

Enfin, on vérifie que la position est libre (« - ») et, si le joueur actuel est le joueur 1, on y place un « X », sinon un « O ». On appelle ensuite la routine AffichePlateau avec le paramètre 0 :

```
if self.plateau[ligne][col] == '-':

  if self.joueur == 1:

    self.plateau[ligne][col] = "X"

else:

  self.plateau[ligne][col] = "O"

self.AffichePlateau(0)
```

Ceci termine des changements apportés à la routine exeCommande.

Il faut maintenant écrire la routine « Initialiser le plateau de jeu ». Elle se contente de régler chaque position à « - », puisque la logique de déplacement utilise cela pour vérifier qu'un emplacement est libre :

```
def InitialisePlateau(self):
self.plateau = [['-','-','-'],['-','-','-'],['-','-','-']]
```

La routine AffichePlateau (ci-dessous) affiche le plateau de jeu, appelle la routine verifGagnant et règle le numéro du joueur. On construit une grande chaîne de caractères à envoyer au client pour qu'il n'ait à appeler la routine une fois par tour. Le paramètre premiereFois est là pour envoyer un joli plateau vide quand le client se connecte pour la première fois ou qu'il réinitialise le jeu (en bas) :

Ensuite, on vérifie si le paramètre premiereFois vaut 0 ou 1 (ci-dessous). S'il vaut 0, on vérifie si le joueur courant a gagné et, si oui, on ajoute le texte « Joueur X GAGNE ! » à la chaîne de sortie. Si le joueur courant n'a pas gagné, on ajoute le texte « Saisir le déplacement... » à la chaîne de sortie. Finalement, on envoie la chaîne au client avec la routine cli.send :

Pour finir, sur la page suivante, vous verrez la routine verifGagnant.

```
if premiereFois == 0:
    if self.joueur == 1:
        ret = self.verifGagnant("X")
    else:
        ret = self.verifGagnant("O")
    if ret == True:
        if self.joueur == 1:
            outp += "Joueur 1 GAGNE !"
        else:
            outp += "Joueur 2 GAGNE !"
    else:
        if self.joueur == 1:
            self.joueur = 2
        else:
            self.joueur = 1
        outp += ('Saisir le deplacement du joueur %s' %
self.joueur)
    self.cli.send(outp)
```

On a déjà réglé le joueur à « X » ou « O », donc on commence à utiliser une boucle « for » simple. Si on trouvez un gagnant, la routine renvoie True. La variable « C » représente chaque colonne dans notre liste de listes.

Le Client

À nouveau, on commence par la routine simple de la fois précédente.

```
def AffichePlateau(self,premiereFois):
# affiche la premiere ligne
outp = ('   1   2   3') + chr(13) + chr(10)
outp += (" A {0} | {1} | {2}".format(self.plateau[0][0],self.plateau[0][1],self.plateau[0][2])) + chr(13)+chr(10)
outp += (' -----------')+ chr(13)+chr(10)
outp += (" B {0} | {1} | {2}".format(self.plateau[1][0],self.plateau[1][1],self.plateau[1][2]))+ chr(13)+chr(10)
outp += (' -----------')+ chr(13)+chr(10)
outp += (" C {0} | {1} | {2}".format(self.plateau[2][0],self.plateau[2][1],self.plateau[2][2]))+ chr(13)+chr(10)
outp += (' -----------')+ chr(13)+chr(10)
```

On commence par vérifier si une ligne horizontale est gagnante :

```
def verifGagnant(self,joueur):
    # boucle sur les lignes et colonnes
    for c in range(0,3):
    # verifie si on a une ligne horizontale
        if self.plateau[c][0] == joueur and
self.plateau[c][1] == joueur and self.plateau[c][2] == joueur:
          print "*********\n\n%s gagne\n\n*********" % joueur
          joueurGagne = True
          return joueurGagne
```

Puis on vérifie si une colonne est gagnante :

```
        # verifie si on a une ligne verticale
        elif self.plateau[0][c] == joueur and
self.plateau[1][c] == joueur and self.plateau[2][c] == joueur:
          print "*********\n\n%s gagne\n\n*********" % joueur
          joueurGagne = True
          return joueurGagne
```

Maintenant on vérifie si une diagonale de gauche à droite gagne...

```
        # verifie si on a une diagonale (gauche a droite)
        elif self.plateau[0][0] == joueur and
self.plateau[1][1] == joueur and self.plateau[2][2] == joueur:
         print "*********\n\n%s gagne\n\n*********" % joueur
          joueurGagne = True
          return joueurGagne
```

Puis de droite à gauche...

```
        #verifie si on a une diagonale (droite a gauche)
        elif self.plateau[0][2] == joueur and
self.plateau[1][1] == joueur and self.plateau[2][0] == joueur:
          print "*********\n\n%s gagne\n\n*********" % joueur
          joueurGagne = True
          return joueurGagne
```

Finalement, si rien ne gagne, on renvoie faux

```
    else:
        joueurGagne = False
        return joueurGagne
```

Les changements commencent juste après l'appel à conn.seConnecte. On envoie un « Debut », plusieurs « Deplace » et, enfin, une commande « AU REVOIR ». La chose la plus importante ici est qu'on doit envoyer une commande, puis obtenir une réponse avant d'envoyer une autre commande. Pensez à une conversation polie. Effectuez vôtre demande, attendez la réponse, puis effectuez une autre demande, attendez la réponse et ainsi de suite. Dans cet exemple, on utilise raw_input simplement pour qu'on puisse voir ce qui se passe :

```
if __name__ == '__main__':
conn = CmdLine('local-
host')
conn.seConnecte()
conn.envoieCommande('De-
but')
conn.recupResultats()
conn.envoieCommande('De-
place A3')
conn.recupResultats()
r = raw_input("Presser
Entree")
conn.envoieCommande('De-
place B2')
conn.recupResultats()
r = raw_input("Presser
Entree")
```

Continuez la suite de routines envoieCommande, recupResultats, raw_input avec les commandes suivantes (le code pour A3 et B2 est déjà écrit) : C1, A1, C3, B3, C2 puis terminez par une commande « AU REVOIR ».

Aller plus loin

Et maintenant, voici un « devoir » à faire à la maison. Dans la partie client, enlevez les mouvements codés en dur et utilisez raw_input() pour demander au joueur de saisir ses déplacements sous la forme « A3 » ou « B2 », puis ajoutez la commande « Deplace » devant et envoyez le tout au serveur.

La prochaine fois, nous modifierons le serveur pour qu'il tienne le rôle de l'autre joueur.

Les codes sources complets du client et du serveur sont ici : http://fullcirclemagazine.pastebin.com/5iNkC5Fr ou là, en anglais : http://the-designatedgeek.com

TUTORIEL Programmer en Python - Partie 19

Greg Walters

Cette fois-ci, nous allons travailler sur la fin de notre programme de Tic-Tac-Toe. Cependant, contrairement à mes autres articles, je ne fournirai pas le code : vous le ferez ! Je vous donnerai quand même les règles du jeu. Après 18 mois, vous avez les outils et les connaissances pour terminer ce projet. J'en suis persuadé.

Tout d'abord, examinons la logique du jeu de Tic-Tac-Toe. Nous verrons cela sous forme de pseudo-code. Regardons d'abord le plateau de jeu, qui se présente ainsi…

```
Coin   |   Côté    |  Coin
-------+--------+-------
 Côté  | Centre   | Côté
-------+--------+-------
Coin   |   Côté    |  Coin
```

Celui qui est « X » commence à jouer. Le meilleur premier mouvement est de prendre un des coins. N'importe lequel, ça n'a pas d'importance. Nous traiterons en premier les permutations lorsque « X » joue ; on les voit à droite.

Le point de vue du joueur « O » est indiqué en bas à droite.

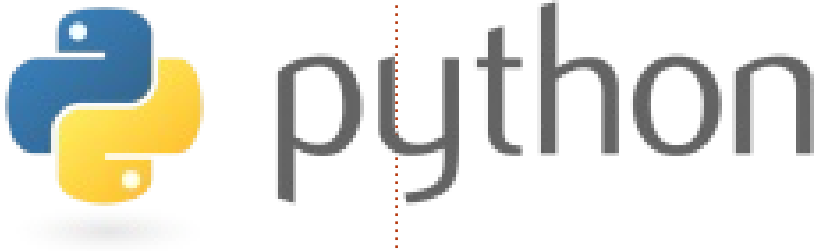
```
SI « O » prend un COIN ALORS
  # Scenario 1
     « X » devrait prendre un des coins restants. N'importe lequel.
     SI « O » bloque la victoire ALORS
     « X » prend le coin restant.
     Terminer en gagnant.
SINON
     Terminer en gagnant.
SINON SI « O » prend un CÔTÉ ALORS
   # Scénario 2
     « X » prend le CENTRE
SI « O » bloque la victoire ALORS
     « X » prend le coin qui n'est pas voisin d'un « O »
     Terminer en gagnant.
SINON
     Terminer en gagnant.
SINON
     # « O » a joué au CENTRE — Scénario 3
     « X » prend le coin opposé en diagonale au premier coup
     SI « O » joue dans un coin
     « X » joue dans le coin restant
     Terminer en gagnant.
SINON
     # le jeu sera nul — Scénario 4
     Bloquer la victoire de « O ».
     Bloquer tout victoire possible
```

Certaines possibilités de jeu sont indiquées à la page suivante.

Comme vous pouvez le voir, la logique est un peu compliquée, mais peut se ramener facilement à une suite d'instructions SI (notez que j'utilise ALORS, mais en Python on utilise plutôt «_:_»). Vous devriez arriver à modifier le code du mois dernier pour gérer tout cela, ou au moins écrire à partir de rien un programme simple de jeu de Tic-Tac-Toe de bureau.

```
SI « X » ne joue pas au centre
ALORS
     « O » prend le centre
     SI « X » a un coin ET
     un côté ALORS
  # Scénario 5
     « O » prend le coin
     Bloquer les victoires possibles pour
un nul.
     SINON
     # « X » a deux côtés
  # Scénario 6
     « O » prend le coin
     entouré par deux « X »
     SI « X » bloque la victoire ALORS
     « O » prend n'importe quelle case
     Bloquer et forcer un nul
          SINON
      Terminer en gagnant.
```

```
Scenario 1
X | - | -    X | - | -    X | - | -    X | - | -    X | - | X    X | - | X    X | X | X
- | - | -    - | - | -    - | - | -    O | - | -    O | - | -    O | O | -    O | O | -
- | - | -    - | - | O    X | - | O    X | - | O    X | - | O    X | - | O    X | - | O

Scenario 2
X | - | -    X | - | -    X | - | -    X | - | -    X | - | X    X | - | X    X | X | X
- | - | -    O | - | -    O | X | -    O | X | -    O | X | -    O | X | -    O | X | -
- | - | -    - | - | -    - | - | -    - | - | O    - | - | O    O | - | O    X | - | O

Scenario 3
X | - | -    X | - | -    X | - | -    X | - | X    X | O | X    X | O | X    X | O | X
- | - | -    - | O | -    - | O | -    - | O | -    - | O | -    - | O | -    - | O | X
- | - | -    - | - | -    - | - | X    O | - | X    O | - | X    O | - | X    O | - | X

Scenario 4
X | - | -    X | - | -    X | - | -    X | - | -    X | - | -    X | - | X    X | O | X
- | - | -    - | O | -    - | O | O    X | O | O    X | O | O    X | O | O    X | O | O
- | - | -    - | - | -    - | - | X    - | - | X    O | - | X    O | - | X    O | - | X

Scenario 5
X | - | -    X | - | -    X | - | -    X | - | -    X | - | -    X | - | -    X | - | X
- | - | -    - | O | -    - | O | X    - | O | X    X | O | X    X | O | X    X | O | X
- | - | -    - | - | -    - | - | -    - | - | O    - | - | O    O | - | O    O | - | O

Scenario 6
- | - | -    - | - | -    - | - | -    - | - | -    - | - | X    O | - | X    O | - | X
X | - | -    X | O | -    X | O | -    X | O | -    X | O | -    X | O | -    X | O | -
- | - | -    - | - | -    - | X | -    O | X | -    O | X | -    O | X | -    O | X | O
```



Idées et auteurs souhaités

Nous avons créé les pages du projet Full Circle et de son équipe sur Launchpad. L'idée étant que des personnes qui ne sont pas auteurs puissent aller sur la page du projet, cliquer sur « Answers » [Ndt : Réponses] en haut de la page et laisser leurs idées d'article, mais merci d'être précis dans vos idées ! Ne laissez pas seulement « article sur les serveurs », spécifiez s'il vous plaît ce que le serveur devrait faire !

Les lecteurs qui aimeraient écrire un article, mais qui ne savent pas à propos de quoi écrire, peuvent s'inscrire sur la page de l'équipe du Full Circle, puis s'attribuer une ou plusieurs idées d'articles et commencer à écrire ! Nous vous demandons expressément, si vous ne pouvez terminer l'article en quelques semaines (au plus un mois), de rouvrir la question pour laisser quelqu'un d'autre récupérer l'idée.

La page du projet, pour les idées :
https://launchpad.net/fullcircle
La page de l'équipe pour les auteurs :
https://launchpad.net/~fullcircle

TUTORIEL

Greg Walters

Programmer en Python - Partie 20

Nous voici de retour. Cette fois-ci nous allons refaire de la programmation graphique, mais en utilisant la bibliothèque pyGTK. Nous ne travaillerons pas avec un éditeur d'interfaces pour le moment, mais seulement avec la bibliothèque.

Utilisez Synaptic pour installer python-gtk2, python-gtk2-tutorial et python-gtk2-doc.

Et maintenant commençons tout de suite à écrire notre premier programme avec pyGTK ; voyez en haut à droite.

Nous allons travailler pendant un certain temps sur ce simple bout de code. Vous voyez une nouvelle instruction à la ligne 3 : « pygtk.require ('2.0') » signifie que l'application ne fonctionnera pas si le module pygtk n'est pas au moins en version 2.0. Dans la routine __init__, nous assignons une fenêtre à la variable self.fenetre (ligne 8), puis nous l'affichons (ligne 9). Souvenez-vous que la routine __init__ est appelée dès qu'on instancie la classe (ligne 13). Sauvegardez le code sous le nom « simple1.py ».

Exécutez-le dans un terminal : vous verrez une simple fenêtre s'afficher quelque part sur votre bureau ; sur le mien, elle s'affiche dans le coin supérieur gauche du bureau. Pour terminer le programme, vous devrez utiliser Ctrl-C dans le terminal. Pourquoi ? Nous n'avons pas ajouté le code pour détruire et donc terminer l'application. C'est ce que nous allons faire maintenant. Ajoutez la ligne suivante avant self.fenetre.show() :

```
self.fenetre.connect("delete_event", self.evenement_supprimer)
```

Puis ajoutez la routine suivante après l'appel à gtk.main() :

```
def evenement_supprimer(self, widget, event, data=None):
    gtk.main_quit()
    return False
```

Sauvegardez votre appli sous le nom « simple2.py » et exécutez-la à nouveau dans un terminal. Désormais, lorsque vous cliquez sur le « X » de la barre de titre, l'application se terminera. Que se passe-t-il vraiment ici ? La première ligne que nous avons ajoutée (self.fenetre.connect...) relie

```
# simple.py
import pygtk
pygtk.require('2.0')
import gtk

class Simple:
    def __init__(self):
        self.fenetre = gtk.Window(gtk.WINDOW_TOPLEVEL)
        self.fenetre.show()
    def main(self):
        gtk.main()
if __name__ == "__main__":
    simple = Simple()
    simple.main()
```

l'événement de fermeture de fenêtre à une routine, en l'occurrence self.evenement_supprimer. En retournant « False » au système, cela détruit du même coup la fenêtre actuelle dans la mémoire système.

Je ne sais pas pour vous, mais moi je préfère que mes applications s'ouvrent au centre de l'écran, plutôt que n'importe où aléatoirement, ou dans un coin (où elles risquent d'être masquées par autre chose). Modifions donc le code : il suffit d'ajouter la ligne suivante avant la ligne self.fenetre.connect dans la fonction __init__ :

```
self.fenetre.set_position(gtk.WIN_POS_CENTER)
```

Comme vous pouvez le deviner, ceci règle la position de la fenêtre au centre de l'écran. Sauvegardez l'appli sous le nom « simple3.py » et exécutez-la.

C'est plus joli, mais il n'y a pas grand-chose à voir. Allez, essayons d'ajouter un composant. Si vous vous souvenez du temps où nous avons travaillé avec Boa Constructor, les composants sont simplement des contrôles prédéfinis que l'on peut ajouter à la fenêtre pour faire des choses. L'un des contrôles les plus simples à ajouter est un bouton. Ajoutons le code suivant juste après la ligne self.fenetre.connect dans la routine __init__ du programme précédent :

```
self.button = gtk.Button
("Ferme-moi")

self.button.connect("cli-
cked",self.clicBouton1,None)

self.fenetreadd(self.button)

self.bouton.show()
```

La première ligne définit le bouton et le texte sur sa surface. La ligne suivante permet de le connecter à un événement de clic. La troisième ligne ajoute le bouton à la fenêtre et la quatrième affiche le bouton sur la surface de la fenêtre. En regardant la ligne self.bouton.connect, vous pouvez voir qu'il y a trois arguments. Le premier est l'événement auquel on veut se connecter, le deuxième est la routine qui sera appelée lorsque l'événement surviendra, dans ce cas « self.clicBouton1 » et la troisième est l'argument (s'il existe) qui sera passé à la routine précédemment indiquée.

Ensuite, nous devons créer la routine self.clicBouton1. Ajoutez ceci après la routine self.evenement_supprimer :

```
def clicBouton1(self,widget,da-
ta=None):
print "Clic bouton 1"
gtk.main_quit()
```

Comme vous pouvez le voir, la routine ne fait pas grand-chose. Elle affiche « Clic bouton 1 » dans le terminal, puis appelle la routine gtk.main_quit(), ce qui fermera la fenêtre et quittera l'application (comme si vous aviez cliqué sur le « X » dans la barre de titre). Sauvegardez à nouveau tout cela sous le nom « simple4.py » et exécutez-le dans un terminal. Vous verrez notre fenêtre centrée avec un bouton affichant « Ferme-moi ». Cliquez dessus et l'application se fermera, comme prévu. Notez cependant que la fenêtre est un peu plus petite que dans l'application « simple3 ». Vous pouvez redimensionner l'application, mais le bouton s'agrandit avec. Pourquoi ? Eh bien, nous avons simplement placé un bouton dans la fenêtre et la fenêtre s'est dimensionnée pour s'adapter au contrôle.

Nous avons en quelque sorte violé les règles de la programmation graphique en plaçant le bouton directement dans la fenêtre, sans utiliser de conteneur. Souvenez-vous lorsque nous avons utilisé Boa Constructor dans le premier article sur la programmation graphique, nous avions utilisé des boîtes de dimensionnement (des conteneurs) pour contenir nos contrôles. Nous devrions faire pareil, même si nous n'avons qu'un seul contrôle. Pour notre exemple suivant, nous ajouterons une Hbox (une boîte horizontale) pour contenir le bouton, ainsi qu'un deuxième bouton. Pour un conteneur vertical, nous utiliserions Vbox.

Pour commencer, utilisez « simple4.py » comme code de base. Effacez tout ce qui se trouve entre les lignes self.fenetre.connect(...) et self.fenetre.show(). C'est là que nous allons ajouter nos nouvelles lignes. Voici le code pour la Hbox et le premier bouton :

```
self.box1 = gtk.HBox(False,0)

self.fenetre.add(self.box1)

self.bouton = gtk.Button("Bou-
ton 1")

self.bouton.connect("cli-
cked",self.clicBouton1,None)

self.box1.pack_start(self.bou-
ton,True,True,0)

self.bouton.show()
```

Tout d'abord, nous ajoutons une Hbox appelée self.box1 ; les paramètres passés à Hbox sont homogènes (vrai ou faux) et une valeur d'espacement :

```
Box = gtk.HBox(homoge-
neous=False, spacing=0)
```

Le paramètre « homogeneous » indique si chaque widget contenu dans la boîte a la même taille (largeur dans le cas d'une Hbox et hauteur pour une Vbox). Dans ce cas, nous choisissons « false » (faux) et une valeur d'espacement de 0. Ensuite, nous ajoutons la boîte dans la fenêtre. Puis nous créons le bouton comme précédemment et connectons l'événement de clic à notre routine.

Maintenant nous arrivons à une nouvelle commande. La commande self.box1.pack_start sert à ajouter le bouton au conteneur (Hbox). Nous utilisons cette commande au lieu de la commande self.fenetre.add pour les widgets que nous voulons placer dans le conteneur. La commande (comme ci-dessus) est :

```
box.pack_start(widget,expand=
True, fill=True, padding=0)
```

La commande pack_start prend les paramètres suivants : en premier le widget, puis expand (True ou False) et une valeur de « padding ». L'espacement dans un conteneur représente l'espace entre les widgets ; le padding concerne les côtés droit et gauche de chaque widget. L'argument expand permet de choisir si les widgets dans la boîte rempliront tout l'espace de la boîte (True), ou bien si

la boîte se rétrécira pour s'adapter aux widgets (False). L'argument fill (remplir) n'a d'effet que si l'argument expand est à True. Finalement on affiche le bouton. Voici maintenant le code pour le deuxième bouton :

```
self.bouton2 = gtk.Button("Bou-
ton 2")

self.bouton2.connect("cli-
cked",self.clicBouton2,None)

self.box1.pack_start(self.bou-
ton2,True,True,0)

self.bouton2.show()

self.box1.show()
```

Remarquez que le code est à peu près le même que celui pour le premier bouton. La dernière ligne de code affiche la boîte.

À présent il faut ajouter la routine self.clicBouton2. Après la routine self.clicBouton1, ajoutez le code suivant :

```
def btn2Clicked(self,widget,da-
ta=None):

    print "Clic bouton 2"

et commentez la ligne
suivante dans la routine
self.clicBouton1 :

gtk.main_quit()
```

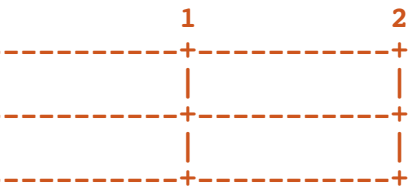
Nous voulons que les deux boutons affichent leur message « Clic bouton X » sans fermer la fenêtre.

Sauvegardez cela dans « simple4a.py » et exécutez-le dans un terrminal. Vous verrez une fenêtre centré avec deux boutons (collés aux bords de la fenêtre) affichant « Bouton 1 » et « Bouton 2 ». Cliquez dessus et remarquez qu'ils répondent correctement aux clics comme nous l'avons prévu.

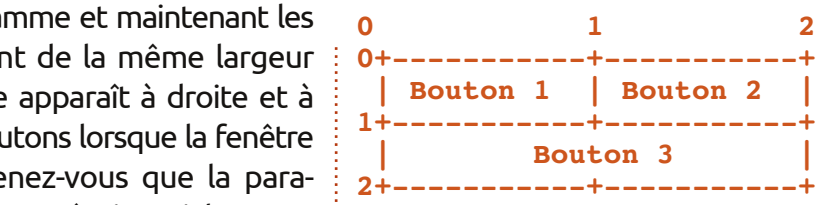
Maintenant, avant de refermer la fenêtre, redimensionnez-la (tirez sur le coin inférieur droit) et remarquez que les boutons grandissent et rétrécissent de façon égale lors du redimensionnement de la fenêtre. Pour comprendre le paramètre expand, modifiez le code de la fonction self.box1.pack_start en remplaçant True par False dans les deux lignes.

Exécutez à nouveau le programme et regardez ce qui se passe : cette fois-ci, la fenêtre semble identique au départ, mais lorsque vous la redimensionnez, les boutons restent de la même largeur, et il y a de l'espace vide à droite lorsque la fenêtre grandit. Remettez ensuite le paramètre expand à True et réglez le paramètre fill à False. Exécutez à nouveau le programme et maintenant les boutons restent de la même largeur et de d'espace apparaît à droite et à gauche des boutons lorsque la fenêtre grandit. Souvenez-vous que la paramètre fill ne sert à rien si le paramètre expand est à False.

Une autre façon de placer les widgets est d'utiliser un tableau. Souvent, si tout ce que vous placez peut tenir facilement dans une structure quadrillée, alors utiliser un tableau est le meilleur choix (et le plus facile). Un tableau est comme une grille de tableur avec des lignes et des colonnes contenant des widgets. Chaque widget peut occuper une ou plusieurs cellules suivant ce que vous souhaitez en faire. Peut-être que le dessin suivant vous aidera à visualiser les possibilités. Voici une grille 2 sur 2 :



Nous placerons deux boutons sur la première ligne, un dans chaque colonne. Sur la deuxième ligne, nous placerons un bouton qui s'étendra sur les deux colonnes. Comme ceci :



Pour dessiner un tableau, on crée un objet de type table et on l'ajoute à la fenêtre. L'instruction pour créer un tableau est :

```
Table = gtk.Table(lignes=1,co-
lonnes=1,homogeneous=True)
```

Si le drapeau « homogeneous » vaut True, la taille des cases du tableau sera calculée en fonction du plus haut widget dans la même ligne et du plus large dans la même colonne. On crée ensuite un widget (comme le bouton ci-dessus) puis on le place dans la bonne case du tableau. Pour le placer, on fait comme ceci :

```
table.attach(widget,gauche,
droite,haut,bas,xoptions=EX-
PAND|FILL,yoptions=EXPAND|
FILL, xpadding=0,ypadding=0)
```

Les seuls paramètres obligatoires sont les cinq premiers. Ainsi, pour placer un bouton dans le tableau en ligne 0 et colonne 0, on peut utiliser l'instruction suivante :

```
table.attach(buttonx,0,1,0,1)
```

Si on voulait le placer en ligne 0 et colonne 1 (souvenez-vous que la numérotation part de 0) comme le bouton ci-dessus, on écrirait :

```
table.attach(buttonx,1,2,0,1)
```

J'espère que c'est aussi clair que de la boue maintenant pour vous. Commençons avec notre code et vous comprendrez mieux. Tout d'abord les parties communes :

```
# table1.py
import pygtk
pygtk.require('2.0')
import gtk
class Table:
    def __init__(self):

        self.fenetre = gtk.
Window(gtk.WINDOW_TOPLEVEL)

        self.fenetre.set_posi-
tion(gtk.WIN_POS_CENTER)

        self.fenetre.set_-
title("Test Table 1")

        self.fenetre.set_bor-
der_width(20)

        self.fenetre.set_size_-
request(250, 100)

        self.fenetre.
connect("delete_event",

self.evenement_supprimer)
```

Il y a plusieurs choses nouvelles ici que nous allons étudier avant d'aller plus loin. La ligne 9 règle le titre de la fenêtre à « Test Table 1 ». On utilise l'instruction set_border_width pour dessiner une bordure de 20 pixels autour de la fenêtre avant d'y placer les widgets. Enfin, on force la taille de la fenêtre à 250x100 pixels en utilisant la fonction set_size_request. Cela vous paraît clair jusqu'ici ? Maintenant, créons le tableau et ajoutons-le à la fenêtre.

```
table = gtk.Table(2, 2, True)
# cree une grille 2x2

self.fenetre.add(table)
```

Ensuite, on crée un premier bouton, on règle l'événement pour le clic, on le place dans le tableau et on l'affiche :

```
bouton1 = gtk.Button("Bouton
1")

bouton1.connect("clicked",
self.gererClic,"bouton 1")

table.attach(bouton1,0,1,0,1)

bouton1.show()
```

Maintenant, le deuxième bouton :

```
bouton2 = gtk.Button("Bouton
2")
bouton2.connect("clicked",
self.gererClic,"bouton 2")

table.attach(bouton2,1,2,0,1)

bouton2.show()
```

C'est presque exactement pareil que pour le bouton 1, mais remarquez le changement dans l'appel à table.attach. Remarquez aussi que la routine que nous utiliserons lors des clics s'appelle self.callback, et est la même pour les deux boutons. Assez pour le moment, vous comprendrez mieux ce que nous faisons un peu plus tard.

Maintenant, le troisième bouton. Ce sera le bouton pour quitter :

```
bouton3 = gtk.Button("Quit-
ter")

bouton3.connect("clicked",
self.Quitter,"bouton 3")

table.attach(bouton3,0,2,1,2)

bouton3.show()
```

Pour finir, on affiche le tableau et la fenêtre. Voici aussi la routine principale et la routine de suppression que nous avons utilisées précédemment :

```
    table.show()

    self.fenetre.show()

def main(self):
    gtk.main()

def evenement_supprimer(self,
widget, event, data=None):

    gtk.main_quit()

    return False
```

Maintenant, voici la partie rigolote. Pour les boutons 1 et 2, on règle la routine de gestion d'événement à self.gereClic. Voici le code pour faire cela :

```
def gererClic(self,widget,da-
ta=None):

    print "clic sur %s" % data
```

Lorsque l'utilisateur clique sur le bouton, l'événement de clic est déclenché et les données fournies lorsqu'on a réglé la connexion à l'événement sont envoyées. Pour le bouton 1, les données envoyées sont « bouton 1 » et « bouton 2 » pour le bouton 2. Tout ce que nous avons à faire ici est d'afficher « clic sur X » dans le terminal. Je suis certain que vous voyez l'intérêt de cet outil lorsqu'on le combine avec une routine bien structurée du type SI | SINON SI| SINON.

Pour terminer, on doit définir la routine Quitter, utilisée lorsqu'on

clique sur le bouton Quitter :

```
def Quitter(self, widget,
event, data=None):

    print "Le bouton Quitter
a ete utilise"

    gtk.main_quit()
```

Et maintenant, le code principal final :

```
if __name__ == "__main__":

table = Table()

table.main()
```

Combinez tout ce code dans une application appelée « table1.py » et exécutez-la dans un terminal.

Pour récapituler, lorsqu'on veut utiliser pyGTK pour créer une application graphique, les étapes sont :

- Créer la fenêtre.
- Créer des Hbox, Vbox ou tableaux pour contenir vos widgets.
- Utiliser pack ou attach pour placer les widgets (selon que vous utilisez des « box » ou des tableaux).
- Afficher les widgets.
- Afficher les « box » ou le tableau.
- Afficher la fenêtre.

Maintenant, nous possédons de nombreux outils et connaissances pour aller plus loin. Tout le code est sur Pastebin : http://fullcirclemagazine.pastebin.com/XxaS0mvJ.

À la prochaine.

EXTRA ! EXTRA !
LISEZ CECI

LE SERVEUR PARFAIT ÉDITION SPECIALE

Il s'agit d'une édition spéciale du Full Circle qui est une réédition directe des articles Le Serveur parfait qui ont déjà été publiés dans le FCM n° 31 à 34.

http://fullcirclemagazine.org/special-edition-1-the-perfect-server/

Des éditions spéciales du magazine Full Circle sont sorties dans un monde sans méfiance*

PYTHON ÉDITION SPECIALE n° 1

Il s'agit d'une reprise de Programmer en Python, parties 1 à 8 par Greg Walters.

http://fullcirclemagazine.org/python-special-edition-1/

* Ni Full Circle magazine, ni ses concepteurs ne s'excusent pour l'hystérie éventuellement causée par la sortie de ces publications.

TUTORIEL

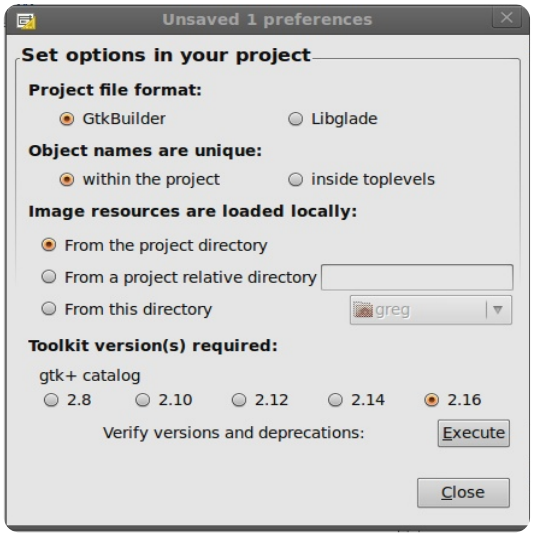
Programmer en Python - Partie 21

Greg Walters

Si vous avez suivi mes articles depuis un certain temps, vous vous souvenez peut-être des parties 5 et 6. Nous avions parlé de l'utilisation de Boa Constructor pour créer l'apparence de notre application graphique. Eh bien, cette fois-ci nous allons utiliser Glade Designer. C'est différent, mais similaire. Vous pouvez l'installer depuis la Logithèque Ubuntu : cherchez glade et installez « GTK+ 2 User Interface Builder ».

Juste pour vous annoncer le programme, nous allons créer une application pour laquelle nous allons avoir besoin de plusieurs articles pour arriver au bout. Le but final est de construire un programme pour créer des listes de lecture pour nos fichiers MP3 et autres fichiers de médias. Ce chapitre du tutoriel sera orienté vers la partie interface graphique. La prochaine fois, nous parlerons du code qui permet de rassembler les dif-férentes parties de l'interface.

Commençons à concevoir notre application. Lorsque vous démarrez Glade Designer, vous aurez une fenêtre de préférences (ci-dessus).

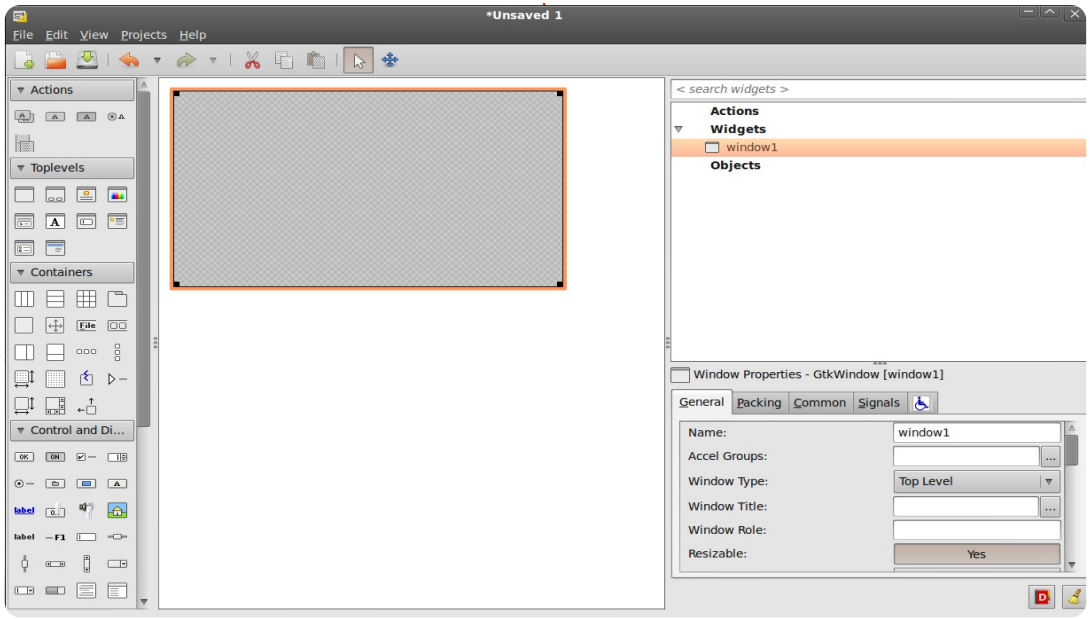


Choisissez Libglade et « à l'intérieur des niveaux supérieurs » puis cliquez sur Fermer. Cela nous amène à la fenêtre principale.

Jetons un coup d'oeil à la fenêtre principale (à droite). Sur la gauche se trouve la boîte à outils, au milieu la zone de conception et, à droite, les attributs et les zones de hiérarchie.

Dans la boîte à outils, cherchez le groupe appelé « Niveaux supérieurs » et cliquez sur le premier outil (si vous le survolez avec la souris, il devrait afficher « Fenêtre »). Cela nous donnera notre « Canvas », une fenêtre vide avec laquelle nous allons travailler.

Remarquez que dans la zone de hiérarchie vous voyez window1 dans la section Widgets. Maintenant, descendez dans la section des attributs, modifiez le nom window1 en FenetrePrincipale et réglez le titre de fenêtre à « Créateur de liste de lecture v1.0 ». Sauvegardez votre travail sous le nom CreateurListeDeLecture.glade. Avant de continuer, cherchez dans la section des attributs de l'onglet Général la liste déroulante « Position de la fenêtre » et réglez-la sur « Centre ». Cochez la case « Largeur par défaut » et indiquez 650. Faites la même chose pour la case « Hauteur par défaut » et indiquez 350. Puis cliquez sur l'onglet « Commun » et descendez jusqu'à l'attribut « Visible ». ASSUREZ-VOUS DE LE RÉGLER SUR « OUI », sinon votre fenêtre ne s'affichera pas. Enfin, allez sur l'onglet Signaux, descendez jusqu'à la section GtkObject et ouvrez-la ; dans « destroy », cliquez sur la liste déroulante de la colonne Gestionnaire et choisissez on_FenetrePrincipale_destroy. Ceci crée un événement qui sera appelé lorsque l'utilisateur fermera notre fenêtre en cliquant sur le « X » dans la barre de titre. Un mot d'avertissement : après avoir réglé l'événement destroy, cliquez quelque part au-dessus ou en



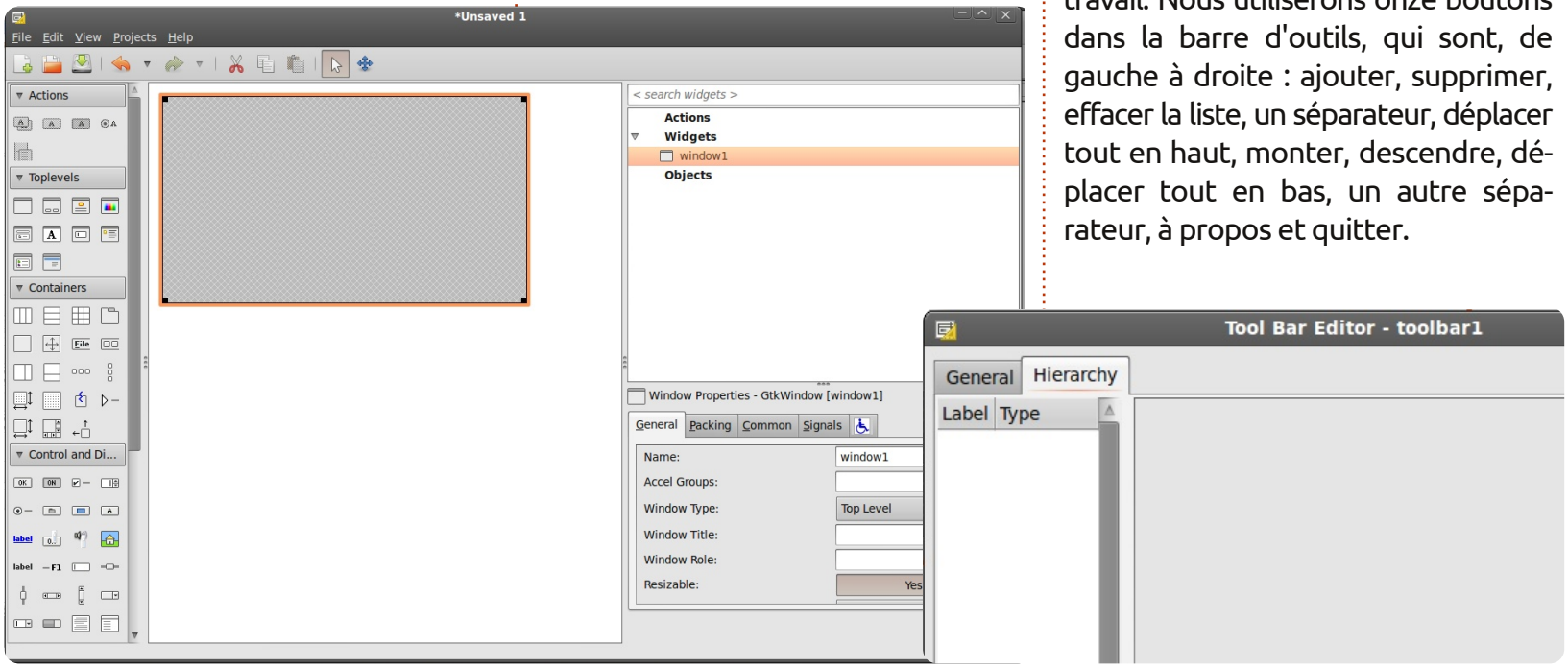
dessous pour valider le changement. Ceci semble être un bogue de Glade Designer. À nouveau, sauvegardez votre projet.

Comme la dernière fois que nous avons conçu une interface graphique, nous devons placer nos widgets dans des « vbox » et des « hbox ». C'est la chose la plus difficile à retenir lorsqu'on fait de la programmation graphique. Nous allons ajouter dans la fenêtre une boîte verticale pour contenir nos widgets ; choisissez « boîte verticale » dans la section Conteneurs de la boîte à outils (deuxième icône de la première ligne), et cliquez sur notre fenêtre vide dans la partie conception. Vous verrez apparaître une boîte de dialogue vous demandant combien d'éléments vous souhaitez. Par défaut, c'est 3, mais nous en voulons 5 : en partant du haut, nous placerons une barre d'outils, une zone pour une vue arborescente, deux zones horizontales pour des étiquettes, des boutons et des zones de saisie de texte, et une barre d'état.

Maintenant nous pouvons commencer à ajouter nos widgets. Tout d'abord, ajoutez une « Barre d'outils » depuis la boîte à outils. Chez moi, c'est la quatrième icône de la deuxième ligne des Conteneurs. Cliquez sur la rangée la plus haute de la vbox ; cette rangée va se rétrécir et presque disparaître, mais ne vous inquiétez pas, nous la récupérerons dans quelques minutes.

Ensuite nous allons ajouter une fenêtre avec ascenseur dans la rangée suivante pour y placer une vue arborescente ; ceci nous permettra de nous déplacer dans la vue. Cherchez l'icône de la fenêtre avec défilement dans la section des Conteneurs de la boîte à outils (chez moi c'est la deuxième icône de la cinquième rangée), et cliquez sur la deuxième rangée de la vbox. Ensuite, nous allons ajouter deux boîtes horizontales dans les deux rangées suivantes ; chacune devra avoir trois éléments. Enfin, ajoutez une barre d'état dans la rangée du bas : vous la trouverez dans la section « Contrôle et affichage » de la boîte à outils, près du bas. Maintenant, votre Glade Designer devrait ressembler à l'image ci-dessous.

Pour terminer, ajoutez à la fenêtre avec défilement une « Vue arborescente » que vous trouverez dans la section « Contrôle et affichage » de la boîte à outils. Vous obtiendrez une boîte de dialogue demandant quel modèle vous voulez utiliser. Cliquez simplement sur le bouton OK pour l'instant, nous nous occuperons de cela plus tard.



Maintenant nous allons nous intéresser quelques instants à la fenêtre avec défilement. Cliquez dessus dans la zone de hiérarchie. Défilez dans l'onglet « Général » jusqu'à trouver « Politique d'affichage des barres de défilement horizontales » ; modifiez cela pour afficher « Toujours », et faites la même chose pour la Politique d'affichage des barres de défilement verticales. Sauvegardez à nouveau.

Bon, maintenant intéressons-nous à la barre d'outils. Cette zone sera située tout en haut de notre application, juste en dessous de la barre de titre. Elle contiendra divers boutons, qui feront la plus grande partie du travail. Nous utiliserons onze boutons dans la barre d'outils, qui sont, de gauche à droite : ajouter, supprimer, effacer la liste, un séparateur, déplacer tout en haut, monter, descendre, déplacer tout en bas, un autre séparateur, à propos et quitter.

Dans la zone de hiérarchie, cliquez sur « barre d'outils 1 ». Elle devrait passer en surbrillance. Tout en haut de Glade Designer, vous verrez quelque chose qui ressemble à un crayon : cliquez dessus. Cela appelle l'éditeur de barres d'outils. Cliquez sur l'onglet « Hiérarchie » et vous verrez quelque chose comme l'image de la page précédente, en bas à droite.

Nous ajouterons tous les boutons de notre barre d'outils à partir de là. Les étapes seront :

- cliquer le bouton ajouter ;
- modifier le nom du bouton ;
- modifier l'étiquette du bouton ;
- choisir l'image.

Nous répéterons cela pour nos onze widgets. Allez, cliquez sur « Ajouter » puis dans la boîte « Nom » et saisissez « boBtnAjouter ». Défilez jusqu'à la partie « Modifier l'étiquette » et saisissez « Ajouter » dans la zone « étiquette » ; puis un peu plus bas vous trouverez « Modifier l'image » et dans la partie « ID prédéfini », utilisez la liste déroulante pour choisir « Ajouter ». Voilà pour notre bouton Ajouter. Nous l'avons nommé « boBtnAjouter » pour pouvoir y faire référence dans le code plus tard. « boBtn » est un raccourci pour « Bouton de barre d'outils ». Ainsi dans notre code il sera facile à repérer et est auto-documenté.

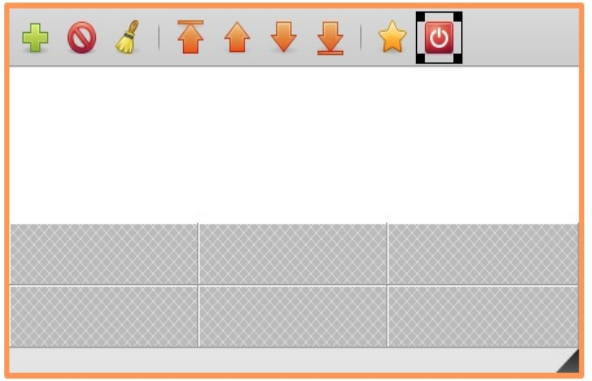
Maintenant il faut ajouter les autres widgets à notre barre d'outils. Ajoutez un autre bouton pour « Supprimer ». Celui-ci s'appellera (comme vous l'aurez deviné) « boBtnSupprimer ». À nouveau, réglez l'étiquette et l'icône. Puis ajoutez un autre bouton en le nommant « boBtnEffacer » et utilisez l'icône « Effacer ». Maintenant nous voulons placer un séparateur ; cliquez donc sur « Ajouter », nommez-le « Sep1 » et choisissez « Séparateur » dans la liste déroulante des types.

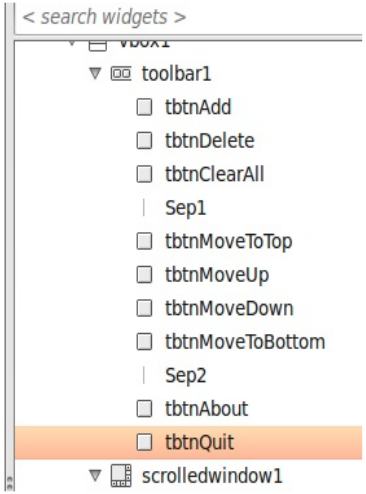
Ajoutez le reste des widgets en les nommant « boBtnHaut », « boBtnMonter », « boBtnDescendre », « boBtnBas », « Sep2 », « boBtnAPropos » et « boBtnQuitter ». Je suis sûr que vous trouverez les icônes correctes. Une fois cela terminé, vous pouvez quitter la fenêtre de hiérarchie et sauvegarder votre travail. Vous devriez avoir quelque chose qui ressemble à l'image ci-dessous, à droite.

Maintenant il nous faut paramétrer les gestionnaires d'événements pour tous les boutons que nous avons créés. Dans la zone de hiérarchie, sélectionnez le widget boBtnAjouter. Cela devrait surligner à la fois la ligne dans la hiérarchie et le bouton lui-même. Retournez à la section des attributs, sélectionnez l'onglet « Signaux » et déployez la ligne GtkToolButton pour afficher l'événement « clicked ». Choisissez comme précédemment « on_boBtnAjouter_clicked » comme gestionnaire de l'événement « clicked », puis cliquez au-dessus ou en dessous pour valider le changement. Faites cela pour tous les autres boutons que nous avons créés (en sélectionnant l'événe-ment « on_boBtnSupprimer_clicked », etc.). Souvenez-vous de cliquer en dehors de la liste déroulante pour valider le changement, puis sauvegardez votre projet. Nos séparateurs n'ont pas besoin d'événements, ne les modifiez pas.

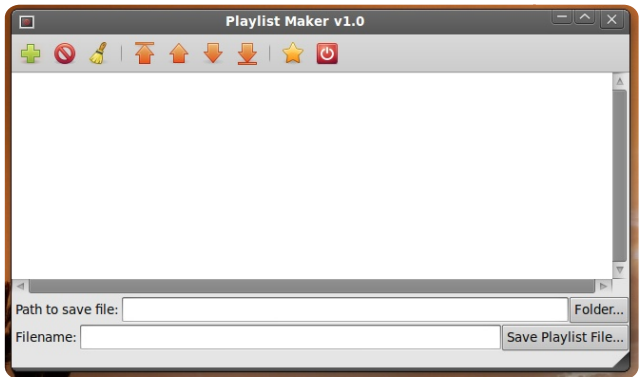
Ensuite, nous devons remplir nos « hbox ». Celle du haut contiendra (de gauche à droite) une étiquette, un widget texte et un bouton. Dans la boîte à outils, sélectionnez le widget « label » (pas le bleu) et placez-le dans l'emplacement de gauche ; puis placez un widget « saisie de texte » au milieu ; et enfin placez un bouton dans l'emplacement de droite. Faites la même chose pour la deuxième hbox.

Il faut maintenant régler les attributs pour les widgets que nous venons d'ajouter. Dans la zone de hiérarchie, sélectionnez label1 sous hbox1. Dans la section des attributs, sélectionnez l'onglet Général, descendez jusqu'à « Modifier l'apparence de l'étiquette » et saisissez « Chemin du fichier à sauvegarder : » comme texte d'étiquette. Puis allez sur l'onglet





« Regroupement » et réglez « Développer » sur « Non ». Vous vous souvenez peut-être de la discussion du mois dernier au sujet du regroupement. Réglez le bourrage à 4, ce qui laissera un peu d'espace à gauche et à droite de l'étiquette. Maintenant, sélectionnez le bouton 1 et réglez également « Développer » sur « Non » sous l'onglet « Regroupement ». Retournez sur l'onglet Général et réglez le nom du bouton à « btnNomRepertoire ». Remarquez que le nom ne commence pas par « bo » car ce n'est pas un bouton de la barre d'outils. Descendez jusqu'à « Étiquette » et saisissez « Répertoire... ». Puis cliquez sur l'onglet « Signaux » et réglez l'événement du bouton à « on_btnNomRepertoire_clicked » dans GtkButton/clicked. Avant de régler les attributs du prochain ensemble de widgets dans l'autre hbox, il nous reste une chose à faire. Sélectionnez hbox1 dans la zone de hiérarchie et réglez « Développer » à « Non » sous l'onglet « Regroupement ». Cela permet à la hbox d'occuper moins d'espace. Pour terminer, réglez le nom du champ de saisie de texte à « txtChemin ».



Maintenant faites la même chose pour hbox2, en réglant « Développer » à « Non », puis le texte de l'étiquette à « Nom de fichier : », « Développer » à « Non » et « Bourrage » à 4. Réglez le nom du bouton à « btnSauvegarderListe », son texte à « Sauvegarder la liste de lecture... », son attribut « Développer » à « Non », réglez son événement « clicked » et réglez le nom du champ de saisie à « txtNomFichier ». Une fois de plus, sauvegardez tout.

Maintenant, notre fenêtre devrait ressembler à l'image ci-dessous à gauche.

Tout cela est bien joli, mais qu'avons-nous fait exactement ? Nous ne pouvons pas exécuter cela comme un programme, puisque nous n'avons pas de code. Nous avons simplement créé un fichier XML nommé CreateurListeDeLecture.glade. Ne vous laissez pas tromper par l'extension : c'est réellement un fichier XML. Si vous faites attention, vous pouvez l'ouvrir dans votre éditeur de texte favori (gedit dans mon cas) et le consulter.

Vous verrez du texte qui décrit notre fenêtre et chacun des widgets avec ses attributs. Par exemple, regardons le code (ci-dessus) pour le widget principal, la fenêtre elle-même.

Vous pouvez voir que le nom du widget est « FenetrePrincipale », son titre « Créateur de liste de lecture v1.0 », le gestionnaire d'événement, etc.

Regardons le code (page précédente, en bas à droite) pour l'un de nos boutons de barre d'outils.

J'espère que cela commence à avoir du sens pour vous. Maintenant nous devons écrire du code pour nous permettre de voir notre beau

```
<widget class="GtkWindow" id="FenetrePrincipale">
    <property name="visible">True</property>
    <property name="title" translatable="yes">Créateur de liste de lecture
v1.0</property>
    <property name="window_position">center</property>
    <property name="default_width">650</property>
    <property name="default_height">350</property>
    <signal name="destroy" handler="on_FenetrePrincipale_destroy"/>
```

```
<child>
  <widget class="GtkToolButton" id="boBtnAjouter">
    <property name="visible">True</property>
    <property name="label" translatable="yes">Ajouter</property>
    <property name="use_underline">True</property>
    <property name="stock_id">gtk-add</property>
    <signal name="clicked" handler="on_boBtnAjouter_clicked"/>
  </widget>
  <packing>
    <property name="expand">False</property>
    <property name="homogeneous">True</property>
  </packing>
</child>
```

travail faire vraiment quelque chose. Ouvrez votre éditeur de code et commençons avec ceci (encadré jaune, miliieu de la première colonne).

Nous avons donc créé nos « import » à peu près comme le mois dernier. Remarquez que nous importons « sys » et « MP3 » depuis mutagen.mp3. Nous avons installé mutagen dans l'article numéro 9 ; si vous ne l'avez pas sur votre système reportez-vous à cet article. Nous aurons besoin de mutagen pour la prochaine fois, et de sys pour que le programme puisse se terminer proprement sur la dernière exception.

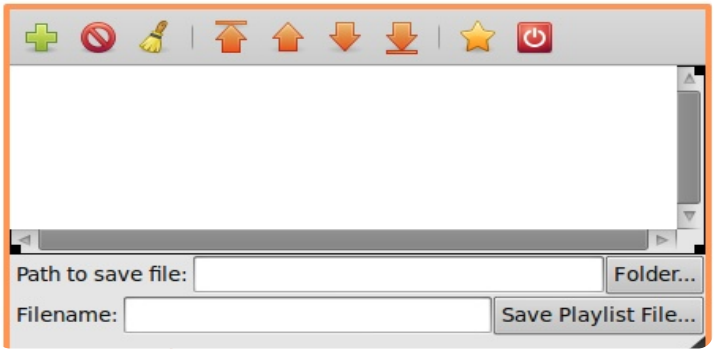
```
#!/usr/bin/env python
import sys
from mutagen.mp3 import MP3
try:
    import pygtk
    pygtk.require("2.0")
except:
    pass
try:
    import gtk
    import gtk.glade
except:
    sys.exit(1)
```

Ensuite, nous devons créer notre classe qui définira notre fenêtre : vous pouvez voir cela page précédente, en haut à droite..

Nous avons déjà fait des choses très ressemblantes. Remarquez les deux dernières lignes : nous définissons le fichier glade (self.gladefile) en indiquant le nom du fichier que nous avons créé avec Glade Designer. Remarquez également que nous n'avons pas indiqué de chemin, juste un nom de fichier. Si votre fichier glade ne se situe pas au même endroit que votre code, vous devez indiquer ici un chemin ; cependant, il est toujours plus malin de les garder ensemble. Ensuite nous définissons notre fenêtre comme « self.wTree » : nous ferons appel à cela à chaque fois qu'on aura besoin de faire référence à la fenêtre. Nous précisons également que le fichier est au format XML, et que la fenêtre que nous utiliserons est celle qui s'appelle « FenetrePrincipale ». Vous pouvez avoir déclaré plusieurs fenêtres dans un seul fichier glade ; nous en reparlerons une autre fois.

```
class CreateurListeDeLecture:
    def __init__(self):
        #================================================
        #                    Creation de la fenetre
        #================================================
        self.gladefile = "CreateurListeDeLecture.glade"
        self.wTree =
gtk.glade.XML(self.gladefile,"FenetrePrincipale")
```

Maintenant il faut gérer les événements. Le mois dernier, nous avons utilisé les appels bouton.connect ou fenetre.connect pour faire référence à nos routines de gestion d'événements. Cette fois-ci, nous allons faire un peu différemment : nous allons utiliser un dictionnaire. Un dictionnaire est comme un tableau, sauf qu'au lieu d'utiliser un index entier, on utilise une clé pour accéder aux données. Clé et donnée : voici un morceau de code qui rendra sans doute cela plus compréhensible. Je ne vais vous montrer que deux événements pour l'instant (bas de la première colonne).



Nous avons donc deux événe-ments : « on_FenetrePrincipale_des-troy » et « on_boBtnQuitter_clicked » sont les clés de notre dictionnaire. Les données correspondantes sont « gtk.main_quit » pour les deux entrées. Lorsqu'un événement est levé par notre interface graphique, le système utilise cet événement pour trouver la clé dans notre dictionnaire, puis sait quelle routine appeler grâce aux données correspondantes. Maintenant nous devons connecter le dictionnaire au gestionnaire de signaux de notre fenêtre ; on fait cela avec la ligne de code suivante : self.wTree.signal_autoconnect(dict)

Nous sommes quasiment prêts. Il ne reste que la routine principale :

```
    #================================================
    #        Creation des gestionnaires d'evenements
    #================================================
    dict = {"on_FenetrePrincipale_destroy": gtk.main_quit,
            "on_boBtnQuitter_clicked": gtk.main_quit}
```

```
if __name__ == "__main__":
        cldl                                    =
CreateurListeDeLecture()
        gtk.main()
```

Sauvegardez ce fichier sous le nom « CreateurLis-teDeLecture.py ». Maintenant vous pouvez l'exécuter (voir page précédente, en haut à droite).

Il ne fait pas grand chose pour l'instant, à part s'ouvrir et se fermer correctement. Nous verrons le reste la prochaine fois. Juste pour aiguiser votre appétit, nous discuterons de l'utilisation de la vue arborescente, des boîtes de dialogues et nous ajouterons pas mal de code. Alors à la prochaine fois.

Fichier Glade :
http://fullcirclemagazine.pastebin.com/2NLaZ3yc

Source Python :
http://fullcirclemagazine.pastebin.com/dAqvxmba