

# Full Circle

LE MAGAZINE INDÉPENDANT DE LA COMMUNAUTÉ UBUNTU LINUX

ÉDITION SPÉCIALE SÉRIE PROGRAMMATION



ÉDITION SPÉCIALE  
SÉRIE PROGRAMMATION

# PROGRAMMER EN PYTHON

Volume quatre

full circle magazine n'est affilié en aucune manière à Canonical Ltd

Spécial Full Circle Magazine

# Full Circle

LE MAGAZINE INDÉPENDANT DE LA COMMUNAUTÉ UBUNTU LINUX

## Au sujet du Full Circle

Le Full Circle est un magazine gratuit, libre et indépendant, consacré à toutes les versions d'Ubuntu, qui fait partie des systèmes d'exploitation Linux. Chaque mois, nous publions des tutoriels, que nous espérons utiles, et des articles proposés par des lecteurs. Le Podcast, un complément du Full Circle, parle du magazine même, mais aussi de tout ce qui peut vous intéresser dans ce domaine.

## Clause de non-responsabilité :

Cette édition spéciale vous est fournie sans aucune garantie ; les auteurs et le magazine Full Circle déclinent toute responsabilité pour des pertes ou dommages éventuels si des lecteurs choisissent d'en appliquer le contenu à leurs ordinateurs et matériel ou à ceux des autres.

## Bienvenue dans une nouvelle édition spéciale consacrée à un seul sujet !

Il ne s'agit de rien d'autre qu'une reprise de la série Programmer en Python, parties 22 à 26, numéros 48 à 52 ; pas de chichis, juste les faits.

Gardez à l'esprit la date de publication ; les versions actuelles du matériel et des logiciels peuvent être différentes de celles illustrées. Il vous est recommandé de bien vérifier la version de votre matériel et des logiciels avant d'essayer d'émuler les tutoriels dans ces numéros spéciaux. Il se peut que vous ayez des logiciels plus récents ou disponibles dans les dépôts de votre distribution.

**Amusez-vous !**

## Nos coordonnées

**SiteWeb :**

<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  
(pseudo : RonnieTucker)

[ronnie@fullcirclemagazine.org](mailto:ronnie@fullcirclemagazine.org)

Webmaster : Rob Kerfia  
(pseudo : admin / linuxgeekery-  
[admin@fullcirclemagazine.org](mailto:admin@fullcirclemagazine.org))

Podcast : Robin Catling  
(pseudo : RobinCatling)  
[podcast@fullcirclemagazine.org](mailto:podcast@fullcirclemagazine.org)

Dir. comm. : Robert Clipsham  
(pseudo : mrmonday) -  
[mrmonday@fullcirclemagazine.org](mailto:mrmonday@fullcirclemagazine.org)



Les articles contenus dans ce magazine sont publiés sous la licence Creative Commons Attribution-Share Alike 3.0 Unported license. Cela signifie que vous pouvez adapter, copier, distribuer et transmettre les articles mais uniquement sous les conditions suivantes : vous devez citer le nom de l'auteur d'une certaine manière (au moins un nom, une adresse e-mail ou une URL) et le nom du magazine (« Full Circle Magazine ») ainsi que l'URL [www.fullcirclemagazine.org](http://www.fullcirclemagazine.org) (sans pour autant suggérer qu'ils approuvent votre utilisation de l'œuvre). Si vous modifiez, transformez ou adaptez cette création, vous devez distribuer la création qui en résulte sous la même licence ou une similaire.

**Full Circle Magazine est entièrement indépendant de Canonical, le sponsor des projets Ubuntu. Vous ne devez en aucun cas présumer que les avis et les opinions exprimés ici aient reçus l'approbation de Canonical.**

**CORRECTION**

Dans la partie 21, il vous était dit de sauvegarder ce que vous aviez dans un fichier nommé « PlaylistMaker.glade » alors que, dans le code, il était indiqué « playlistmaker.glade ». Je suis sûr que vous aviez remarqué que l'un contenait des majuscules et l'autre non. Le code s'exécutera seulement si il y a concordance entre le nom du fichier et l'appel, avec ou sans majuscule.

Pour bien commencer, vous devez avoir les fichiers playlistmaker.glade et playlistmaker.py du mois dernier. Si ce n'est pas le cas, sautez sur le numéro précédent pour les récupérer. Avant de passer au code, nous allons jeter un œil à ce qu'est un fichier de liste de lecture. Il y a plusieurs versions des listes de lecture, qui ont toutes des extensions différentes. Le fichier que nous allons créer sera de type \*.m3u. Dans sa forme la plus simple, c'est juste un fichier texte qui commence par « #EXTM3U » et qui contient une entrée pour chaque fichier audio que vous voulez écouter - avec le chemin

d'accès complet. Il y a aussi une extension qui peut être ajoutée avant chaque entrée contenant la longueur de la chanson, le nom de l'album d'où vient la chanson, le numéro de piste et le nom du morceau. Nous allons ignorer l'extension pour l'instant et nous concentrer uniquement sur la version de base. Voici un exemple d'un fichier de liste de lecture M3U :

```
#EXTM3U
Adult Contemporary/Chris
Rea/Collection/02 - On The
Beach.mp3
Adult Contemporary/Chris
Rea/Collection/07 - Fool (If
You Think It's Over).mp3
Adult Contemporary/Chris
Rea/Collection/11 - Looking
For The Summer.mp3
```

Tous les noms de chemins sont relatifs à l'emplacement du fichier de liste de lecture.

Bien... Maintenant passons au code. Vous voyez à droite le début du code source du mois dernier.

Maintenant, nous devons créer une routine de gestion d'événement pour chacun des événements que nous avons

```
#!/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)
```

puis la définition de la classe

```
class CreateurListeDeLecture:
    def __init__(self):
        self.gladefile = "CreateurListeDeLecture.glade"
        self.wTree = gtk.glade.XML(self.gladefile, "FenetrePrincipale")
```

et la routine principale

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

Ensuite, nous avons le dictionnaire qui devrait se trouver après la routine `__init__`.

```
def DicoEvenements(self):
    dict = {"on_FenetrePrincipale_destroy": gtk.main_quit,
           "on_boBtnQuitter_clicked": gtk.main_quit,
           "on_boBtnAjouter_clicked": self.on_boBtnAjouter_clicked,
           "on_boBtnSupprimer_clicked": self.on_boBtnSupprimer_clicked,
           "on_boBtnEffacer_clicked": self.on_boBtnEffacer_clicked,
           "on_boBtnHaut_clicked": self.on_boBtnHaut_clicked,
           "on_boBtnMonter_clicked": self.on_boBtnMonter_clicked,
           "on_boBtnDescendre_clicked": self.on_boBtnDescendre_clicked,
           "on_boBtnBas_clicked": self.on_boBtnBas_clicked,
           "on_boBtnAPropos_clicked": self.on_boBtnAPropos_clicked,
           "on_btnNomRepertoire_clicked": self.on_btnNomRepertoire_clicked,
           "on_btnSauvegarderListe_clicked":
           self.on_btnSauvegarderListe_clicked}
    self.wTree.signal_autoconnect(dict)
```

mis en place. Notez que `on_FenetrePrincipale_destroy` et `on_boBtnQuitter_clicked` sont déjà faits pour nous, il n'en reste donc que dix autres à écrire (voir en haut à droite). Écrivons juste des ébauches pour l'instant.

Nous modifierons ces ébauches de routines dans quelques minutes. Pour l'instant, cela devrait nous permettre de démarrer l'application ; nous pourrions tester les choses au fur et à mesure que nous avançons. Nous devons quand même ajouter une ligne supplémentaire à la routine `__init__` avant de pouvoir démarrer l'application. Après la ligne `self.wTree`, ajouter :

```
self.DicoEvenements()
```

Maintenant, vous pouvez exécuter l'application, voir la fenêtre, puis cliquer sur le bouton « Quitter de la barre d'outils » pour quitter l'application correctement. Enregistrez le code sous le nom « `CreateurListeDeLecture-1a.py` » et essayez-le. Souvenez-vous qu'il faut l'enregistrer dans le même dossier que le fichier glade que nous avons créé la dernière fois ou bien copier le fichier glade dans le dossier dans lequel vous avez enregistré ce code.

Nous avons également besoin de définir quelques variables pour une utilisation

future. Ajoutez ceci après l'appel à `DicoEvenements()` dans la fonction `__init__`.

```
self.CheminCourant = ""  
self.LigneCourante = 0  
self.NombreDeLignes = 0
```

Maintenant, nous allons créer une fonction qui nous permet d'afficher une boîte de dialogue à chaque fois que nous avons besoin de donner des informations à l'utilisateur. Il existe un ensemble de routines toutes faites que nous allons utiliser, mais nous allons faire une routine à nous pour nous faciliter les choses. C'est la routine `gtk.MessageDialog` et la syntaxe est la suivante :

```
gtk.MessageDialog (parent, drapeaux, MessageType, boutons, message)
```

Une discussion est nécessaire avant d'aller trop loin. Le type de message peut être l'un des suivants :

```
GTK_MESSAGE_INFO - message d'information  
GTK_MESSAGE_WARNING - message d'avertissement  
GTK_MESSAGE_QUESTION - question nécessitant un choix  
GTK_MESSAGE_ERROR - message d'erreur fatale
```

Et les types de boutons sont :

```
def on_boBtnAjouter_clicked(self,widget):  
    pass  
def on_boBtnSupprimer_clicked(self,widget):  
    pass  
def on_boBtnEffacer_clicked(self,widget):  
    pass  
def on_boBtnHaut_clicked(self,widget):  
    pass  
def on_boBtnMonter_clicked(self,widget):  
    pass  
def on_boBtnDescendre_clicked(self,widget):  
    pass  
def on_boBtnBas_clicked(self,widget):  
    pass  
def on_boBtnAPropos_clicked(self,widget):  
    pass  
def on_btnNomRepertoire_clicked(self,widget):  
    pass  
def on_btnSauvegarderListe_clicked(self,widget):  
    pass
```

**GTK\_BUTTONS\_NONE** - aucun bouton

**GTK\_BUTTONS\_OK** - un bouton OK

**GTK\_BUTTONS\_CLOSE** - un bouton

Fermer

**GTK\_BUTTONS\_CANCEL** - un bouton Annuler

**GTK\_BUTTONS\_YES\_NO** - boutons Oui et Non

**GTK\_BUTTONS\_OK\_CANCEL** - boutons OK et Annuler

Normalement, vous utiliseriez le code suivant, ou du code similaire, pour créer la boîte de dialogue, l'afficher, attendre une réponse, puis la détruire.

```
dlg = gtk.MessageDialog (None, 0, gtk.MESSAGE_INFO, gtk.BUT-
```

**TONS\_OK, "Ceci est un message de test ...")**

```
reponse = dlg.run ()
```

```
dlg.destroy ()
```

Toutefois, si vous voulez afficher une boîte de message plus d'une ou deux fois, c'est beaucoup de dactylographie. La règle générale est que si vous écrivez une série de lignes de code plus d'une ou deux fois, il est généralement préférable de créer une fonction puis de l'appeler. Pensez-y de cette manière : si nous voulons afficher un message de dialogue pour l'utilisateur, disons dix fois dans l'application, cela représente 10 x 3 (soit 30) lignes de code.

# TUTORIEL - PROGRAMMER EN PYTHON - PARTIE 22

En faisant une fonction pour faire cela pour nous (en utilisant l'exemple que je viens de présenter), nous aurions 10 + 3 (soit 13) lignes de code à écrire. Plus nous appelons une boîte de dialogue, moins cela fait de code à taper, et plus lisible est notre code. Notre fonction (page suivante en haut à droite) nous permettra d'appeler l'un des quatre types de message de dialogue avec une seule routine en utilisant différents paramètres. C'est une fonction très simple que nous pourrons ensuite appeler comme suit :

```
self.MessageBox("info", "Le bouton QUITTER a été cliqué")
```

Notez que si nous choisissons d'utiliser le type de dialogue MESSAGE\_QUESTION, il y a deux réponses possibles qui seront retournées par la fenêtre de dialogue - un « oui » ou un « non ». Quel que soit le bouton cliqué par l'utilisateur, nous allons recevoir les informations de retour dans notre code. Pour utiliser la boîte de dialogue de question, l'appel ressemblera à ceci :

```
reponse = self.MessageBox("question", "Êtes-vous sûr de vouloir faire cela maintenant ? ")
```

```
if reponse == gtk.RESPONSE_YES:
```

```
def MessageBox(self,niveau,texte):
    if niveau == "info":
        dlg = gtk.MessageDialog(None,0,gtk.MESSAGE_INFO,gtk.BUTTONS_OK,texte)
    elif niveau == "warning":
        dlg = gtk.MessageDialog(None,0,gtk.MESSAGE_WARNING,gtk.BUTTONS_OK,texte)
    elif niveau == "error":
        dlg = gtk.MessageDialog(None,0,gtk.MESSAGE_ERROR,gtk.BUTTONS_OK,texte)
    elif niveau == "question":
        dlg = gtk.MessageDialog(None,0,gtk.MESSAGE_QUESTION,gtk.BUTTONS_YES_NO,texte)
    if niveau == "question":
        resp = dlg.run()
        dlg.destroy()
        return resp
    else:
        resp = dlg.run()
        dlg.destroy()
```

```
print "clic sur oui"
```

```
elif reponse == gtk.RESPONSE_NO:
```

```
print "clic sur non"
```

Vous voyez comment vous pouvez vérifier la valeur du bouton cliqué. Alors maintenant, remplacez l'appel à « pass » dans chacune de nos routines de gestion d'événement par ce que vous voyez ci-dessous à droite.

Nous n'allons pas le garder comme ça, mais cela vous donne une indication visuelle que les boutons fonctionnent comme nous le voulons. Enregistrez maintenant le code sous « Créateur-s'agit du widget treeview. Nous allons ListeDeLecture-1b.py » et testez votre programme. Maintenant nous allons

```
def on_boBtnAjouter_clicked(self,widget):
    self.MessageBox("info","Clic sur bouton Ajouter...")
def on_boBtnSupprimer_clicked(self,widget):
    self.MessageBox("info","Clic sur bouton Supprimer...")
def on_boBtnEffacer_clicked(self,widget):
    self.MessageBox("info","Clic sur bouton Effacer...")
def on_boBtnHaut_clicked(self,widget):
    self.MessageBox("info","Clic sur bouton Haut...")
def on_boBtnMonter_clicked(self,widget):
    self.MessageBox("info","Clic sur bouton Monter...")
def on_boBtnDescendre_clicked(self,widget):
    self.MessageBox("info","Clic sur bouton Descendre...")
def on_boBtnBas_clicked(self,widget):
    self.MessageBox("info","Clic sur bouton Bas...")
def on_boBtnAPropos_clicked(self,widget):
    self.MessageBox("info","Clic sur bouton À propos...")
def on_btnNomRepertoire_clicked(self,widget):
    self.MessageBox("info","Clic sur bouton NomRepertoire...")
def on_btnSauvegarderListe_clicked(self,widget):
    self.MessageBox("info","Clic sur bouton SauvegarderListe...")
```

créer une fonction pour définir nos références de widgets. Cette routine va être appelée une seule fois, mais

elle rendra notre code beaucoup plus maniable et lisible. En fait, nous voulons créer des variables locales

qui font référence à des widgets dans la fenêtre glade - afin que nous puissions faire appel à eux chaque fois que (et si jamais) nous en avons besoin. Mettez cette fonction (page suivante en haut à droite) en dessous de la fonction DicoEvenements.

Remarquez qu'il y a une chose qui n'est pas référencée dans notre routine. Il s'agit du widget treeview. Nous allons créer cette référence lorsque nous créerons l'arborescence elle-même. Notez également la dernière ligne de notre routine. Pour utiliser la barre d'état, il faut s'y référer par son id de contexte. Nous allons utiliser cela plus loin. Ensuite, nous allons mettre en place la fonction qui affiche le dialogue « à propos » quand on clique sur le bouton À propos de la barre d'outils. Encore une fois, ceci est une routine intégrée fournie par la bibliothèque GTK. Placez ceci après la fonction MessageBox. Voici le code, en bas à droite. Sauvegardez votre code, puis faites un essai. Vous devriez voir une fenêtre pop-up, centrée dans notre application, qui affiche ce que nous avons prévu. Il y a plusieurs attributs que vous pouvez définir pour la boîte à propos (qui peuvent être trouvés sur <http://www.pygtk.org/docs/pygtk/class-gtkaboutdialog.html>), mais ceux-ci sont

```
def ReferencesWidgets(self):
    self.txtNomFicher = self.wTree.get_widget("txtNomFicher")
    self.txtChemin = self.wTree.get_widget("txtChemin")
    self.boBtnAjouter = self.wTree.get_widget("boBtnAjouter")
    self.boBtnSupprimer = self.wTree.get_widget("boBtnSupprimer")
    self.boBtnEffacer = self.wTree.get_widget("boBtnEffacer")
    self.boBtnQuitter = self.wTree.get_widget("boBtnQuitter")
    self.boBtnAPropos = self.wTree.get_widget("boBtnAPropos")
    self.boBtnHaut = self.wTree.get_widget("boBtnHaut")
    self.boBtnMonter = self.wTree.get_widget("boBtnMonter")
    self.boBtnDescendre = self.wTree.get_widget("boBtnDescendre")
    self.boBtnBas = self.wTree.get_widget("boBtnBas")
    self.btnNomRepertoire = self.wTree.get_widget("btnNomRepertoire")
    self.btnSauvegarderListe = self.wTree.get_widget("btnSauvegarderListe")
    self.sbar = self.wTree.get_widget("statusbar3")
    self.context_id = self.sbar.get_context_id("Statusbar")
```

puis ajoutez un appel à ceci juste après l'appel à self.DicoEvenements() dans la routine \_\_init\_\_.

```
self.ReferencesWidgets()
```

ceux que je considère être un ensemble minimal.

Avant de poursuivre, nous devons discuter de ce qui se produira à partir d'ici. L'idée générale est que l'utilisateur clique sur le bouton « Ajouter » de la barre d'outils, nous afficherons alors une boîte de dialogue de fichier pour lui permettre d'ajouter des fichiers à la liste de lecture, puis nous afficherons les informations du fichier dans notre widget treeview. De là, il peut ajouter d'autres fichiers, supprimer un fichier unique, supprimer tous les fichiers, déplacer un fichier vers le haut ou le

```
def AfficherAPropos(self):
    apropos = gtk.AboutDialog()
    apropos.set_program_name("Createur de liste de
lecture")
    apropos.set_version("1.0")
    apropos.set_copyright("(c) 2011 by Greg Walters")
    apropos.set_comments("Ecrit pour le Full Circle
Magazine")
    apropos.set_website("http://thedesignatedgeek.com")
    apropos.run()
    apropos.destroy()
```

Maintenant, commentez (ou retirez simplement) l'appel à MessageBox dans la routine on\_boBtnAPropos\_clicked, et remplacez-le par un appel à la fonction AfficherAPropos. Cela devrait ressembler à :

```
def on_boBtnAPropos_clicked(self,widget):
    #self.MessageBox("info","Clic sur bouton APropos...")
    self.AfficherAPropos()
```

bas, ou bien tout en haut ou tout en bas de l'arborescence. Enfin, il va définir le chemin où le fichier sera enregistré, fournir un nom de fichier avec une extension « m3u », puis cliquer sur le bouton « Sauvegarder ». Bien que cela semble assez simple, il se passe beaucoup de choses en coulisses. La magie se produit dans le widget `treeview`, nous allons donc en discuter. Cela ira assez loin, alors lisez attentivement, car il faut le comprendre pour éviter de commettre des erreurs plus tard. Une arborescence peut être quelque chose d'aussi simple qu'une liste à colonnes de données, comme dans une feuille de calcul ou une base de données, ou bien elle peut être plus complexe, comme une liste de fichiers/dossiers avec des parents et enfants, où le dossier serait le parent et les fichiers de ce dossier seraient les enfants, ou quelque chose d'encore plus complexe. Pour ce projet, nous allons utiliser le premier exemple, une liste à colonnes. Dans la liste, il y aura trois colonnes. Une pour le nom du fichier de musique, une pour l'extension du fichier (mp3, ogg, wav, etc.) et la dernière colonne pour le chemin d'accès. En combinant tout ça dans une chaîne (chemin d'accès, nom de fichier, extension) on obtient l'entrée que nous allons écrire dans la liste de

lecture. Vous pourriez bien sûr ajouter d'autres colonnes si vous le souhaitez, mais pour l'instant nous allons nous contenter de trois. Une arborescence est simplement un conteneur visuel de stockage qui détient et affiche un modèle. Le modèle est le véritable « dispositif » qui contient et manipule nos données. Il existe deux modèles prédéfinis qui sont utilisés avec un `treeview`, mais vous pouvez certainement créer le vôtre. Cela étant dit, pour 98 % de votre travail, l'un des deux modèles prédéfinis fera ce dont vous avez besoin. Les deux types sont `GTKListStore` et `GTKTreeStore`. Comme leur nom l'indique, le modèle `ListStore` est habituellement utilisé pour les listes, le `TreeStore` est utilisé pour les arbres. Pour notre application, nous allons utiliser un `GTKListStore`. Les étapes de base sont les suivantes :

- Créer une référence au widget `TreeView`.
- Ajouter les colonnes.
- Définir le type de moteur de rendu à utiliser.
- Créer le `ListStore`.
- Définir l'attribut de modèle dans

```
def SetupTreeView(self):
    self.cNomFic = 0
    self.cTypeFic = 1
    self.cCheminFic = 2
    self.sNomFic = "NomFichier"
    self.sTypeFic = "Type"
    self.sCheminFic = "Dossier"
    self.treeview = self.wTree.get_widget("treeview1")
    self.AjouterColonne(self.sNomFic, self.cNomFic)
    self.AjouterColonne(self.sTypeFic, self.cTypeFic)
    self.AjouterColonne(self.sCheminFic, self.cCheminFic)
    self.listeLecture = gtk.ListStore(str, str, str)
    self.treeview.set_model(self.listeLecture)
    self.treeview.set_grid_lines(gtk.TREE_VIEW_GRID_LINES_BOTH)
```

l'arborescence de notre modèle.

- Remplir les données.

La troisième étape consiste à mettre en place le type de moteur de rendu que la colonne utilisera pour afficher les données. C'est tout simplement une routine qui est utilisée pour tracer les données dans le modèle de l'arbre. GTK fournit de nombreux moteurs de rendu de cellules différents, mais normalement vous utiliserez le plus souvent `GtkCellRenderText` et `GtkCellRendererToggle`.

Nous allons donc créer une fonction (ci-dessus) qui met en place notre widget `TreeView`. Nous allons l'appeler `SetupTreeView`. Nous allons d'abord définir quelques variables pour nos colonnes, définir la variable de référence du `TreeView` proprement dit,

ajouter les colonnes, mettre en place le `ListStore`, et définir le modèle. Voici le code pour la fonction. Placez-le après la fonction `ReferencesWidget`.

Les variables `cNomFic`, `cTypeFic` et `cCheminFic` définissent les numéros de colonne. Les variables `sNomFic`, `sTypeFic` et `sCheminFic` contiennent les noms de colonnes de notre vue. La septième ligne définit la variable de référence du widget `treeview` tel qu'il figure dans notre fichier `glade`.

Ensuite nous appelons une routine (page suivante, en haut à droite), que nous allons créer dans un instant, pour chaque colonne que nous voulons. Puis, nous définissons notre `GTKListStore` avec trois champs de texte et, enfin, nous utilisons ce `GTKListStore` comme attribut de

modèle de notre widget `TreeView`. Nous allons ensuite créer la fonction `AjouterColonne`. Placez-la après la fonction `SetupTreeview`.

Chaque colonne est créée avec cette fonction. Nous lui passons le titre de la colonne (ce qui est affiché sur la première ligne de chaque colonne) et un `idColonne`. Dans ce cas, nous utilisons les variables que nous avons créées plus tôt (`sNomFic` et `cNomFic`). Nous créons ensuite une colonne dans notre widget `TreeView` donnant le titre, le type de rendu de cellule et enfin l'id de la colonne. Nous indiquons ensuite que la colonne est redimensionnable, nous définissons l'id de tri et ajoutons enfin la colonne dans le `TreeView`.

Ajoutez ces deux fonctions à votre code. J'ai choisi de les mettre tout de suite après la fonction `ReferencesWidget`, mais vous pouvez les mettre n'importe où dans la classe `CreateurListeDeLecture`. Ajoutez la ligne suivante après l'appel à `ReferencesWidget()` dans la fonction `__init__` pour appeler la fonction :

```
self.SetupTreeview ()
```

Enregistrez et exécutez votre programme et vous verrez que nous

avons maintenant trois colonnes avec en-têtes dans notre widget `TreeView`.

Il reste tellement de choses à faire. Nous devons avoir un moyen d'obtenir les noms de fichiers de musique de l'utilisateur et un moyen de les mettre dans le `TreeView` sous forme de lignes de données. Nous devons créer nos fonctions `Supprimer`, `Effacer tout`, les fonctions de déplacement, la routine de sauvegarde et les routines de chemins de fichiers, plus quelques « jolies » choses qui donneront à notre application un aspect plus professionnel. Commençons par la routine « Ajouter ». Après tout, c'est le premier bouton sur notre barre d'outils. Lorsque l'utilisateur clique sur le bouton `Ajouter`, nous voulons faire apparaître une fenêtre de dialogue « standard » d'ouverture de fichier, qui permet des sélections multiples. Une fois que l'utilisateur a fait son choix, nous voulons ensuite prendre ces données et les ajouter dans l'arborescence, comme je l'ai indiqué ci-dessus. Ainsi, la première chose logique à faire est de travailler sur la boîte de dialogue `Fichier`. Encore une fois, `GTK` nous fournit un moyen d'appeler une boîte de dialogue « standard »

```
def AjouterColonne(self,titre,idColonne):  
    colonne = gtk.TreeViewColumn(titre,gtk.CellRendererText(),text=idColonne)  
    colonne.set_resizable(True)  
    colonne.set_sort_column_id(idColonne)  
    self.treeview.append_column(colonne)
```

de fichiers. Nous pourrions coder ça en dur simplement avec des lignes de code dans le gestionnaire d'événements `on_boBtnAjouter_clicked`, mais nous allons faire une classe distincte pour le gérer. Tant que nous y sommes, nous pouvons faire en sorte que cette classe gère non seulement un dialogue `Ouvrir un fichier`, mais aussi un dialogue `Sélectionner un dossier`. Comme auparavant avec la fonction `MessageBox`, vous pouvez l'extraire dans un fichier qui contient toutes sortes de routines réutilisables pour un usage ultérieur.

Nous allons commencer par définir une nouvelle classe appelée `DialogueFichier` qui a une seule fonction appelée `AfficheDialogue`. Cette fonction prendra deux paramètres, l'un appelé « type » (un '0' ou un '1'), qui précise si nous créons un dialogue d'ouverture de fichier ou de sélection de dossier, et l'autre, qui est le chemin à utiliser pour la vue par défaut de la boîte de dialogue appelée `CheminCourant`. Créez cette classe juste avant notre code principal à la fin du fichier source.

```
class DialogueFichier:  
    def AfficheDialogue(self,  
        type,CheminCourant):
```

La première partie de notre code doit être une instruction `IF`

```
if type == 0: # choix de fichier  
    ...  
else: # choix de dossier  
    ...
```

Avant d'aller plus loin, nous allons voir la façon dont la boîte de dialogue de fichier/dossier est effectivement appelée et utilisée. La syntaxe de la boîte de dialogue se présente comme suit :

```
gtk.FileChooserDia-  
log(titre,parent,action,bou-  
ttons,backend)
```

et retourne un objet fenêtre de dialogue. Notre première ligne (dans le cas où `type` vaut 0) sera la ligne ci-dessous.

Comme vous pouvez le voir, le titre est « Choisir les fichiers à ajouter... », le parent est défini sur `none` (aucun). Nous demandons une fenêtre de type ouverture de fichier (`action`) et nous

voulons des boutons « Annuler » et « Ouvrir », les deux utilisant des icônes de type « stock ». Nous réglons également les codes de retour de `gtk.RESPONSE_CANCEL` et `gtk.RESPONSE_OK` lorsque l'utilisateur fait ses choix. L'appel au sélecteur de dossier dans la clause `else` est similaire.

Fondamentalement, les seules choses qui ont changé entre les deux définitions sont le titre (ci-dessus à droite) et le type d'action. Donc le code de la classe devrait maintenant être le code affiché au milieu à droite.

Nous définissons la réponse par défaut à la touche OK, puis activons la fonctionnalité de sélection multiple pour que l'utilisateur puisse sélectionner (vous l'aurez deviné) plusieurs fichiers à ajouter. Si nous n'avons pas indiqué cela, la boîte de dialogue permettrait seulement de sélectionner un fichier à la fois, car `set_select_multiple` est réglé sur faux par défaut. Nos lignes suivantes règlent le chemin actuel, puis affichent la boîte de dialogue elle-même. Avant de taper le code, je vais vous expliquer pourquoi nous devons nous occuper du chemin courant. À chaque fois que vous faites apparaître une boîte de dialogue de fichier et que vous ne définissez pas

un chemin, la valeur par défaut est le dossier où réside notre application. Ainsi, si les fichiers de musique que l'utilisateur utilise sont dans `/media/musique/` ils sont ensuite triés par genre puis par artiste, et puis après par album. Supposons également que l'utilisateur a installé notre application dans `/home/user2/createurListeDeLecture`. Chaque fois que nous faisons apparaître le dialogue, le dossier de départ serait `/home/user2/createurListeDeLecture`. Rapidement, l'utilisateur devrait se sentir frustré par cela, préférant retrouver le dernier dossier dans lequel il était lorsqu'il démarre la prochaine fois. Vous comprenez ? Bien. Voici donc en bas à droite les lignes de code suivantes. Ici, nous vérifions les réponses renvoyées. Si l'utilisateur a cliqué sur le bouton « Ouvrir » qui renvoie `gtk.RESPONSE_OK`, nous obtenons le nom ou les noms des fichiers que l'utilisateur a sélectionné, on définit le chemin d'accès courant vers le dossier où nous sommes, on détruit la boîte de dialogue, puis on renvoie les données à la routine appelée. Si, en revanche, l'utilisateur a cliqué sur le bouton « Annuler », il suffit de détruire la boîte de dialogue. Je mets l'instruction `print` là juste pour vous montrer que l'appui sur le bouton a fonctionné. Vous pouvez la laisser

ou la retirer. Notez que lorsque nous sortons de la partie concernant le bouton Ouvrir dans cette routine, nous renvoyons deux ensembles de valeurs : `selectionFichiers` qui est une liste des fichiers sélectionnés par l'utilisateur, ainsi que le `CheminCourant`.

Afin que la routine fasse quelque chose, ajoutez la ligne suivante dans la routine `on_boBtnAjouter_clicked` :

```
fd = DialogueFichier ()  
  
fichiersChoisis, self.CheminCourant = fd.AfficheDialogue(0, self.CheminCourant)
```

Ici on récupère les deux valeurs de retour qui sont renvoyées depuis le `return`. Pour le moment, ajoutez le code ci-dessous pour voir à quoi les informations retournées ressemblent :

```
for f in fichiersChoisis:  
    print "Choix utilisateur :  
%s" % f  
  
print "Chemin courant : %s" %  
self.CheminCourant
```

Lorsque vous exécutez le programme, cliquez sur le bouton « Ajouter ». Vous verrez la boîte de dialogue de fichier. Allez maintenant à un endroit où vous avez des fichiers et sélectionnez-les. Vous pouvez appuyer sur la

touche [Ctrl] et cliquer sur plusieurs fichiers pour les sélectionner individuellement, ou sur la touche [Maj] pour sélectionner plusieurs fichiers contigus. Cliquez sur le bouton « Ouvrir », et examinez la réponse dans un terminal. Remarquez que si vous cliquez sur le bouton « Annuler » à ce moment, vous obtiendrez un message d'erreur. C'est parce que le code ci-dessus suppose qu'il n'y a pas de fichiers sélectionnés. Ne vous inquiétez pas pour l'instant, nous allons régler cela sous peu. Je voulais simplement vous permettre de voir ce qui revient si l'on appuie sur le bouton « Ouvrir ». Une chose que nous devrions faire est d'ajouter un filtre à notre fenêtre d'ouverture de fichier. Puisque nous attendons que l'utilisateur sélectionne normalement des fichiers de musique, nous devrions :  
1) donner la possibilité d'afficher des fichiers de musique uniquement et,  
2) donner la possibilité d'afficher tous les fichiers au cas où. Nous faisons cela en utilisant les attributs `FileFilter` de la boîte de dialogue. Voici le code pour cela, qu'il faut placer dans la partie « `type == 0` » juste après la ligne créant le dialogue.

```
filtre = gtk.FileFilter()  
filtre.set_name("Fichiers musicaux")  
filtre.add_pattern("*.mp3")
```

```
filtre.add_pattern("*.ogg")
filtre.add_pattern("*.wav")
dialogue.add_filter(filtre)
filtre =
gtk.FileFilter()filtre.set_name("Tous les fichiers")
filtre.add_pattern("*")dialogue.add_filter(filtre)
```

Nous mettons en place deux « groupes », l'un pour les fichiers de musique (filtre.set\_name("Fichiers musicaux")), et l'autre pour tous les fichiers. Nous utilisons un motif pour définir les types de fichiers que nous voulons. J'ai défini trois motifs, mais vous pouvez ajouter ou supprimer tous ceux que vous souhaitez. Je mets le filtre pour la musique en premier, puisque c'est ce qui intéresse principalement l'utilisateur. Ainsi, les étapes sont :

- Définir une variable de filtre.
- Régler le nom.
- Ajouter un motif.
- Ajouter le filtre à la boîte de dialogue.

Vous pouvez avoir autant ou aussi peu de filtres que vous le souhaitez. Notez également qu'une fois que vous avez ajouté le filtre à la boîte de dialogue, vous pouvez réutiliser la variable de filtre. Retournez dans la routine on\_boBtnAjouter\_clicked, com-

mentez les dernières lignes que nous avons ajoutées et remplacez-les par cette seule ligne :

```
self.AjouterFichiers(fichiersChoisis)
```

Notre routine ressemble maintenant au code affiché ci-contre.

Ainsi, lorsque nous aurons la réponse au retour de la fenêtre de sélection de fichiers, nous enverrons la liste contenant les fichiers sélectionnés à cette routine. Une fois ici, nous créons une variable de compteur (le nombre de fichiers que nous ajoutons), puis analysons la liste. Rappelez-vous que chaque entrée contient le nom de fichier complet avec le chemin et l'extension. Nous allons devoir fractionner le nom du fichier en chemin, nom de fichier et extension. Nous récupérerons d'abord le tout dernier « . » dans le nom de fichier et supposons que c'est le début de l'extension, et nous affectons sa position dans la chaîne à debutExt. Nous trouvons ensuite le tout dernier « / » dans le nom du fichier pour déterminer le début du nom de fichier. Puis, nous découpons la chaîne en extension, nom de fichier et chemin du fichier. Nous plaçons ensuite ces valeurs dans une liste nommée « data » et ajoutons

```
def on_boBtnAjouter_clicked(self,widget):
    fd = DialogueFichier()
    fichiersChoisis,self.CheminCourant = fd.AfficheDialogue(0,self.CheminCourant)
    self.AjouterFichiers(fichiersChoisis)
```

Nous devons maintenant créer la fonction à laquelle nous venons de faire appel. Placez cette fonction après la routine on\_btnSauvegarderListe\_clicked.

```
def AjouterFichiers(self,ListeFichiers):
    compteur = 0
    for f in ListeFichiers:
        debutExt = f.rfind(".")
        debutnomFic = f.rfind("/")
        extension = f[debutExt+1:]
        nomFic = f[debutnomFic+1:debutExt]
        cheminFic = f[:debutnomFic]
        data = [nomFic,extension,cheminFic]
        self.listeLecture.append(data)
        compteur += 1
    self.NombreDeLignes += compteur
    self.sbar.push(self.context_id,"%s fichiers ajoutés sur un total de %d" % (compteur,self.NombreDeLignes))
```

ceci dans listeLecture. Nous incrémentons le compteur puisque nous avons fait tout le travail. Enfin on incrémente la variable NombreDeLignes qui contient le nombre total de lignes dans listeLecture et nous affichons un message dans la barre d'état.

Maintenant vous pouvez lancer l'application et voir les données dans l'arborescence. Comme toujours, le code complet peut être trouvé ici :

<http://pastebin.com/wTccGDSW>.

La prochaine fois, nous allons finaliser notre application, en remplissant les routines manquantes, etc.

Cette fois-ci, nous allons terminer notre programme de création de liste de lecture. La dernière fois, nous avions bien avancé, mais nous n'avons pas terminé certaines parties. Nous ne pouvons pas encore sauvegarder la liste de lecture, les fonctions de déplacement ne sont pas implémentées, nous ne pouvons pas choisir le chemin vers lequel on veut sauvegarder, etc. Cependant, nous devons faire certaines choses avant de commencer à coder. Tout d'abord, nous devons trouver une image pour le logo de notre application dans la boîte « À propos » et lorsque l'application est minimisée. Vous pouvez chercher une icône qui vous plaît dans le répertoire `/usr/share/icons`, ou aller sur le web en chercher une ou encore en créer une vous-même. Quel que soit le choix, placez cette image dans le répertoire contenant le code source et le fichier glade du mois dernier. Nommez-la `logo.png`. Ensuite, nous devons ouvrir le fichier glade du mois dernier et faire quelques changements.

Tout d'abord, avec la FenetrePrin-

cipale, allez dans l'onglet Général et descendez jusqu'à trouver Icône. En utilisant l'outil de parcours de fichiers, trouvez votre icône et sélectionnez-la. Maintenant le champ de texte devrait contenir « `logo.png` ». Puis, dans la boîte de hiérarchie, choisissez `treeview1`, allez dans l'onglet Signaux et ajoutez un gestionnaire pour `on_treeview1_cur-sor_changed` dans la partie `GtkTreeView | cursor-changed`. Souvenez-vous que nous avons vu le mois dernier que vous devez cliquer à côté pour conserver vos modifications. Enfin, toujours dans la boîte hiérarchie, choisissez `txtNomFichier` et allez dans l'onglet Signaux. Descendez jusqu'à trouver `GtkWidget` et descendez encore jusqu'à `key_press_event`. Ajoutez un gestionnaire d'événement pour `on_txtNomFichier_key_press_event`. Sauvegardez votre projet glade et fermez glade.

Maintenant il est temps de terminer notre projet. Nous commencerons à coder là où nous en étions restés le mois dernier.

La première chose que je veux faire

```
elif response == gtk.RESPONSE_CANCEL:
    print 'Annulation, aucun fichier choisi'
    dialog.destroy()
```

Remarquez que nous ne renvoyons rien. C'est ce qui causait l'erreur. Pour réparer cela, ajoutez la ligne suivante après la ligne `dialog.destroy()` :

```
Return ([], "")
```

Ainsi il n'y aura plus d'erreur. Ensuite, ajoutons le gestionnaire d'événement que nous avons créé dans glade pour le champ de texte. Dans notre dictionnaire, ajoutez la ligne suivante :

```
"on_txtNomFichierFilename_key_press_event":
self.txtNomFichierKeyPress,
```

Vous vous souvenez que cela crée une fonction pour gérer l'appui sur les touches du clavier. Créons maintenant la fonction :

```
def txtNomFichierKeyPress(self, widget, data):
    if data.keyval == 65293: # valeur de la touche Entree
        self.SauveListeLecture()
```

est modifier le code de la classe `DialogueFichier`. Si vous vous souvenez de la dernière fois, si l'utilisateur cliquait le bouton « Annuler », il se produisait une erreur. Nous allons commencer par corriger ça. À la fin de la routine, vous avez le code ci-dessus.

Comme vous pouvez le supposer, cela regarde simplement la valeur de

chaque touche enfoncée lorsque l'utilisateur se trouve dans le champ de texte `txtNomFichier` et la compare à la valeur 65293, qui est le code attribué à la touche Entrée. Si cela correspond, alors il appelle la fonction `SauvegarderListe`. L'utilisateur n'a même pas besoin de cliquer sur le bouton.

Maintenant passons au code. Occu-

pons-nous du bouton « Effacer » de la barre d'outils. Lorsque l'utilisateur clique sur ce bouton, on veut effacer la liste arborescente et ListStore. Cela se fait en une ligne, que l'on peut placer dans la routine `on_boBtnEffacer_clicked`.

```
def
on_boBtnEffacer_clicked(self
,widget) ::

    self.playList.clear()
```

Nous disons simplement à la liste de lecture ListStore de s'effacer. C'était facile. Maintenant occupons-nous du bouton « Supprimer » de la barre d'outils. C'est plus difficile, mais une fois terminé vous allez comprendre.

D'abord nous devons parler de la façon dont nous récupérons une sélection depuis la liste arborescente et ListStore. C'est un peu compliqué, alors allons doucement. Pour récupérer des données depuis ListStore, nous devons d'abord récupérer un objet `gtk.TreeSelection` qui nous aidera à gérer la sélection à l'intérieur d'un `treeview`. Ensuite, on utilise cet objet pour récupérer le type de modèle et un itérateur qui contient les lignes sélectionnées.

Je sais que vous pensez : « Mais bon sang, qu'est-ce qu'un itérateur ? ». Eh bien, vous en avez déjà utilisé sans même le savoir. Regardez le code suivant (ci-dessus à droite) provenant de la fonction `AjouterFichiers` du mois dernier.

Regardez la boucle `for`. On utilise un itérateur pour parcourir la liste `ListeFichiers`. Dans ce cas, l'itérateur passe tout simplement d'une entrée de la liste à la suivante, renvoyant chaque élément séparément. Nous allons créer un itérateur, le remplir avec les lignes de la vue arborescente sélectionnées et l'utiliser comme une liste. Voici donc le code (au milieu à droite) pour `on_boBtnSupprimer`.

La première ligne crée l'objet `TreeSelection`. On l'utilise pour récupérer les lignes sélectionnées (il n'y en a qu'une car notre modèle n'est pas réglé pour offrir la sélection multiple), remplir une liste nommée `iter` avec, et la parcourir en enlevant chaque élément (comme la méthode `.clear`). On décrémente également la variable `NombreDeLignes`, puis on affiche le nombre de fichiers dans la barre d'état.

Maintenant, avant de passer aux fonc-

```
def AjouterFichiers(self,ListeFichiers):
    compteur = 0
    for f in ListeFichiers:
        debutExt = f.rfind(".")
        debutnomFic = f.rfind("/")
        extension = f[debutExt+1:]
        nomFic = f[debutnomFic+1:debutExt]
        cheminFic = f[:debutnomFic]
        data = [nomFic,extension,cheminFic]
        self.listeLecture.append(data)
        compteur += 1
```

```
def on_boBtnSupprimer_clicked(self,widget):
    sel = self.treeview.get_selection()
    (modele,lignes) = sel.get_selected_rows()    iter=[]
    for ligne in lignes:
        iter.append(self.listeLecture.get_iter(ligne))
    for i in iter:
        if i is not None:
            self.listeLecture.remove(i)
            self.NombreDeLignes -= 1
    self.sbar.push(self.context_id,"%d fichiers dans la
liste." % (self.NombreDeLignes))
```

```
def on_btnNomRepertoire_clicked(self,widget):
    fd = DialogueFichier()
    cheminFichier,self.CheminCourant =
fd.AfficheDialogue(1,self.CheminCourant)
    self.txtChemin.set_text(cheminFichier[0])
```

tions de déplacement, occupons-nous de la fonction de sauvegarde du chemin des fichiers. On utilisera notre classe `DialogueFichier` comme précédemment. On placera tout le code pour faire cela (en bas à droite) dans la routine `on_boBtnNomRepertoire_clicked`.

La seule chose vraiment différente

par rapport à avant est la dernière ligne de ce code. On place le nom du chemin retourné par la fenêtre de dialogue dans le champ de texte que l'on a précédemment initialisé avec la méthode `set_text`. Souvenez-vous que les données nous sont renvoyées sous forme de liste, même s'il n'y a qu'un seul élément. C'est pourquoi

on utilise `chemin[0]`.

Écrivons la fonction de sauvegarde de fichier. On peut faire ça avant de passer aux fonctions de déplacement. Nous allons créer une fonction `SauvegarderListe`. La première chose à faire (ci-dessus à droite) est de vérifier s'il y a quelque chose dans le champ de texte `txtChemin`. Ensuite nous devons vérifier s'il y a un nom de fichier dans le champ de texte `txtNomFichier`. Pour ces deux valeurs, on utilise la méthode `get_text()` du champ de texte.

Maintenant que l'on a un chemin (`cf`) et un nom de fichier (`nf`), on peut ouvrir le fichier, imprimer notre en-tête M3U et parcourir la liste de lecture. Le chemin est stocké (si vous vous souvenez) dans la colonne 2, le nom du fichier dans la colonne 0 et l'extension dans la colonne 1. On crée simplement (à droite) une chaîne, puis on l'écrit dans le fichier et enfin on ferme le fichier.

On peut maintenant commencer à travailler sur les fonctions de déplacement. Commençons par la routine `Haut`. Comme nous l'avons fait en écrivant la fonction `Supprimer`, on

```
def SauveListeLecture(self):
    cf = self.txtChemin.get_text()      # recuperer le chemin dans le champ de texte
    nf = self.txtNomFichier.get_text()  # recuperer le nom du fichier dans le champ de
    texte
```

Maintenant on vérifie les valeurs :

```
        if cf == "":                    # SI le chemin est vide
            self.MessageBox("erreur","Veuillez fournir un chemin pour la liste de lecture.")
        elif nf == "":                  # SI le nom de fichier est vide
            self.MessageBox("erreur","Veuillez fournir un nom pour le fichier liste de
lecture.")
        else:                           # Sinon, on peut continuer
```

```
        fic = open(cf + "/" + nf,"w")  # ouvrir le fichier
        fic.writelines('#EXTM3U\n')     # afficher l'en-tete M3U
        for ligne in self.listeLecture:
            fic.writelines("%s/%s.%s\n" % (ligne[2],ligne[0],ligne[1])) # ecrit les donnees
        fic.close                       # referme le fichier
```

Enfin, on affiche un message informant l'utilisateur que le fichier est sauvegardé.

```
self.MessageBox("info","La liste de lecture est sauvegardee !")
```

On doit maintenant appeler cette routine depuis notre routine de gestion d'événement `on_btnSauvegarderListe_clicked`.

```
def on_btnSauvegarderListe_clicked(self,widget):
    self.SauveListeLecture()
```

Sauvegardez votre code et testez-le. Votre liste de lecture devrait être sauvegardée correctement et ressembler à l'exemple que je vous ai montré le mois dernier.

récupère la sélection puis la ligne sélectionnée. Ensuite on doit parcourir les lignes pour récupérer 2 variables. Nous les appellerons `chemin1` et `chemin2`. `chemin2` sera réglé à 0 dans ce cas, car c'est la ligne de «\_destination». `chemin1` est la ligne

```
def on_boBtnHaut_clicked(self,widget):
    sel = self.treeview.get_selection()
    (modele,lignes) = sel.get_selected_rows()
    for chemin1 in lignes:
        chemin2 = 0
    iter1=modele.get_iter(chemin1)
    iter2 = modele.get_iter(chemin2)
    modele.move_before(iter1,iter2)
```

que l'utilisateur a sélectionnée. On utilise enfin la méthode `modele.move_before()` pour déplacer la ligne sélectionnée sur la ligne 0, en poussant d'office tout vers le bas. Nous placerons le code (ci-contre à droite) directement dans la routine `on_boBtnHaut_clicked`.

Pour la fonction Bas, nous utiliserons presque le même code que pour la routine Haut, mais au lieu d'utiliser la méthode `modele.moveBefore()`, nous utiliserons la méthode `modele.moveAfter()` et, au lieu de régler `chemin2` à 0, on le réglera à `self.NombreDeLignes-1`. Maintenant vous comprenez à quoi sert la variable `NombreDeLignes`. Souvenez-vous que les lignes sont numérotées à partir de 0, donc il faut utiliser `NombreDeLignes-1` (en haut à droite).

Maintenant regardons ce que donne la fonction Monter. À nouveau, elle est très ressemblante aux deux fonctions que nous venons de créer. Cette fois-ci, on a `chemin1` qui contient la ligne sélectionnée, et on règle `chemin2` à `NumeroLigne-1`. Ensuite, Si `chemin2` (la ligne de destination) est supérieur ou égal à 0, on utilise la méthode `mo-`

`dele.swap()` (au milieu à droite).

C'est la même chose pour la fonction Descendre. Cette fois-ci, on vérifie que `chemin2` est plus PETIT ou égal à `self.NombreDeLignes-1` (en bas à droite).

Maintenant, modifions quelques fonctionnalités de notre liste de lecture. Dans l'article du mois dernier, je vous ai montré le format de base d'une liste de lecture (en bas).

Cependant, je vous ai indiqué qu'il y avait aussi un format étendu. Dans le format étendu, il y a une ligne supplémentaire que l'on peut ajouter au fichier avant chaque chanson, contenant des informations supplémentaires sur la chanson. Le format de cette ligne est le suivant :

```
#EXTINF:[longueur de la  
chanson en secondes],[Nom de  
l'artiste] - [Titre de la  
chanson]
```

Vous vous demandiez peut-être pourquoi on a inclus la bibliothèque

```
#EXTM3U
```

```
Adult Contemporary/Chris Rea/Collection/02 - On The Beach.mp3  
Adult Contemporary/Chris Rea/Collection/07 - Fool (If You Think It's Over).mp3  
Adult Contemporary/Chris Rea/Collection/11 - Looking For The Summer.mp3
```

```
def on_boBtnBas_clicked(self,widget):  
    sel = self.treeview.get_selection()  
    (modele,lignes) = sel.get_selected_rows()  
    for chemin1 in lignes:  
        chemin2 = self.NombreDeLignes-1  
        iter1=modele.get_iter(chemin1)  
        iter2 = modele.get_iter(chemin2)  
        modele.move_after(iter1,iter2)
```

```
def on_boBtnMonter_clicked(self,widget):  
    sel = self.treeview.get_selection()  
    (modele,lignes) = sel.get_selected_rows()  
    for chemin1 in lignes:  
        chemin2 = (chemin1[0]-1,)  
        if chemin2[0] >= 0:  
            iter1=modele.get_iter(chemin1)  
            iter2 = modele.get_iter(chemin2)  
            modele.swap(iter1,iter2)
```

```
def on_boBtnDescendre_clicked(self,widget):  
    sel = self.treeview.get_selection()  
    (modele,lignes) = sel.get_selected_rows()  
    for chemin1 in lignes:  
        chemin2 = (chemin1[0]+1,)  
        iter1=modele.get_iter(chemin1)  
        if chemin2[0] <= self.NombreDeLignes-1:  
            iter2 = modele.get_iter(chemin2)  
            modele.swap(iter1,iter2)
```

mutagen depuis le début alors qu'on ne l'a jamais utilisée. Eh bien, nous allons l'utiliser maintenant. Pour vous rafraîchir la mémoire, la bibliothèque mutagen permet d'avoir accès aux

informations des balises ID3 des fichiers MP3. Pour lire la discussion complète là-dessus, reportez-vous au numéro 35 du Full Circle qui contient la partie 9 de cette série. Nous crée-

rons une fonction pour gérer la lecture d'un fichier MP3 et renvoyer le nom de l'artiste, le titre de la chanson et sa longueur en secondes, qui sont les trois informations dont nous avons besoin pour la ligne des informations étendues. Placez cette fonction après la fonction APropos dans la classe CreateurListeDeLecture (page suivante, en haut à droite).

À nouveau, pour vous rafraîchir la mémoire, je vais parcourir le code. Tout d'abord nous effaçons les trois variables de retour pour qu'elles soient renvoyées vides si quelque chose se passe de travers. Ensuite on passe le nom du fichier MP3 que nous allons examiner. Puis on place les clés dans (vous l'avez deviné) un itérateur et on parcourt cet itérateur en cherchant les deux balises spécifiques. Ce sont TPE1 pour le nom de l'artiste et TIT2 pour le titre de la chanson. Si jamais la clé n'existe pas, on obtiendra une erreur, donc on entoure chaque appel avec une instruction try/except. Ensuite on va chercher la longueur de la chanson dans l'attribut audio.info.length et on retourne tout ça.

On va maintenant modifier la fonction SauvegarderListe pour qu'elle supporte la ligne d'informations étendues.

Tant que nous y sommes, vérifions si le nom de fichier existe et, si c'est le cas, prévenons l'utilisateur et sortons de la routine. Aussi, pour rendre les choses un peu plus faciles pour l'utilisateur et, puisqu'on ne supporte aucun autre type de fichier, ajoutons automatiquement l'extension .m3u au chemin et au nom de fichier si elle n'y est pas déjà. Commençons par ajouter une ligne «import os.path» au début du code entre les import de sys et de mutagen (à droite).

```
def RecupererInfoMP3(self,nomFichier):
    artiste = ''
    titre = ''
    longueurChanson = 0
    audio = MP3(nomFichier)
    cles = audio.keys()
    for cle in cles:
        try:
            if cle == "TPE1":          # Artiste
                artiste = audio.get(cle)
        except:
            artiste = ''
        try:
            if cle == "TIT2":          # Titre de la chanson
                titre = audio.get(cle)
        except:
            titre = ''
            longueurChanson = audio.info.length # longueur de la
chanson
    return (artiste,titre,longueurChanson)
```

```
import os.path
```

Ensuite continuez et commentez la fonction SauveListeLecture actuelle et nous allons la remplacer.

```
def SavePlaylist(self):
    fp = self.txtPath.get_text()      # Get the file path from the text box
    fn = self.txtFilename.get_text() # Get the filename from the text box
    if fp == "": # IF filepath is blank...
        self.MessageBox("error","Please provide a filepath for the playlist.")
    elif fn == "": # IF filename is blank...
        self.MessageBox("error","Please provide a filename for the playlist file.")
    else: # Otherwise
```

Jusqu'ici la routine est la même. Voici où les changements commencent.

```
        debutExt = nf.rfind(".") # cherche le debut de l'extension
    if debutExt == -1:
        nf += '.m3u' # ajouter une extension s'il n'y en a pas
        self.txtNomFichier.set_text(nf) # remplace le nom de fichier dans le champ de texte
```

Tout comme pour la fonction AjouterFichiers, nous utiliserons la méthode rfind pour trouver la position du dernier point (« . ») dans le nom du fichier nf. S'il n'y en a pas, la valeur renvoyée sera -1. Donc nous vérifions si la valeur retournée est -1 et si c'est le cas on ajoute l'extension et on remplace le nom du fichier dans le champ de texte pour être sympa.

```
if os.path.exists(fp + "/" + fn):
```

```
self.MessageBox("erreur", "Le fichier existe déjà. Choisissez un autre nom.")
```

Ensuite on veut entourer le reste de la fonction dans une clause IF|ELSE (en haut à droite) pour que, si le

```
else:
```

```
    fic = open(cf + "/" + nf, "w") # ouvre le fichier
    fic.writelines('#EXTM3U\n')   # affiche l'en-tete M3U
    for ligne in self.listeLecture:
        nomFic = "%s/%s.%s" % (ligne[2], ligne[0], ligne[1])
        artiste, titre, longueurChanson = self.RecupererInfoMP3(nomFic)
        if longueurChanson > 0 and (artiste != '' and titre != ''):
            fic.writelines("#EXTINF:%d,%s - %s\n" %
                (longueurChanson, artiste, titre))
            fic.writelines("%s\n" % nomFic)
    fic.close # referme le fichier
    self.MessageBox("info", "Liste de lecture sauvegardee !")
```

fichier existe déjà, on puisse simplement sortir de la routine. On utilise os.path.exists(nom du fichier) pour cette vérification.

Le reste du code sert principalement à sauvegarder comme précédemment, mais regardons-le quand même.

La ligne 2 ouvre le fichier dans lequel nous allons écrire. La ligne 3 y place l'en-tête M3U. La ligne 4 règle un parcours à travers la liste de lecture ListStore. La ligne 5 crée le nom du fichier à partir des trois colonnes de ListStore. La ligne 6 appelle RecupererInfoMP3 et stocke les valeurs renvoyées dans des variables. La ligne 7 vérifie ensuite si nous avons

des valeurs dans toutes ces variables. Si c'est le cas, on écrit la ligne d'informations étendues à la ligne 8, sinon on n'essaie pas. La ligne 9 écrit la ligne du nom du fichier comme précédemment. La ligne 10 ferme gentiment le fichier et la ligne 11 affiche un message à l'utilisateur indiquant que tout est terminé.

```
def SetupBullesAide(self):
```

```
    self.boBtnAjouter.set_tooltip_text("Ajoute un ou des fichier(s) a la liste de lecture.")
    self.boBtnAPropos.set_tooltip_text("Affiche les informations sur le programme.")
    self.boBtnSupprimer.set_tooltip_text("Supprime l'entree selectionnee de la liste.")
    self.boBtnEffacer.set_tooltip_text("Supprime toutes les entrees de la liste.")
    self.boBtnQuitter.set_tooltip_text("Quitte le programme.")
    self.boBtnHaut.set_tooltip_text("Deplace l'entree selectionne tout en haut de la liste.")
    self.boBtnMonter.set_tooltip_text("Remonte l'entree selectionnee dans la liste.")
    self.boBtnDescendre.set_tooltip_text("Descend l'entree selectionnee dans la liste.")
    self.boBtnBas.set_tooltip_text("Deplace l'entree selectionnee tout en bas de la liste.")
    self.btnNomRepertoire.set_tooltip_text("Choisis le repertoire de sauvegarde de la liste.")
    self.btnSauvegarderListe.set_tooltip_text("Sauvegarde la liste.")
    self.txtNomFichier.set_tooltip_text("Entrez ici le nom du fichier a sauvegarder. L'extension .m3u sera ajoutee pour vous si vous l'oubliez.")
```

Allez, sauvegardez votre code et essayez-le.

À ce stade, la seule chose qu'on pourrait encore ajouter serait des bulles d'aide lorsque l'utilisateur survole nos contrôles avec sa souris. Cela y ajoute un air professionnel (ci-dessous). Créons maintenant une fonction pour faire cela.

Nous utilisons le widget `references` que nous avons réglé plus haut, puis on règle le texte pour la bulle d'aide avec (vous l'aurez deviné) l'attribut `set_tooltip_text`. Ensuite on doit ajouter l'appel à la routine. Retournez dans la routine `__init__`, après la ligne `self.ReferencesWidgets`, ajoutez :

```
self.SetupBullesAide()
```

Enfin et surtout (!), on veut placer notre logo dans la boîte `APropos`. Comme tout le reste ici, il y a un attribut pour faire cela. Ajoutez la ligne suivante à la routine `APropos` :

```
apropos.set_logo(gtk.gdk.pixbuf_new_from_file("logo.png"))
```

Et voilà. Vous avez maintenant une application complète, fonctionnelle

et jolie, qui fait un travail merveilleux de création de liste de lecture pour vos fichiers de musique.

Le code complet, incluant le fichier glade que nous avons créé le mois dernier, est disponible ici : <http://pastebin.com/ZfZ69zVJ>

Profitez des nouveaux talents que vous vous êtes découverts, jusqu'à la prochaine fois.

## 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.

**W**aouh ! Il est difficile de croire que ceci est déjà le 24e numéro. Cela fait deux ans que nous apprenons le Python ! Vous avez parcouru un très long chemin.

Cette fois-ci, nous allons traiter deux sujets. Le premier est l'impression sur une imprimante, le second est la création de fichiers RTF (Rich Text Format, ou Format de Texte Riche) comme sortie.

## Impression générique sous Linux

Commençons donc avec l'impression sur une imprimante. L'idée de parler de cela provient d'un courriel envoyé par Gord Campbell. Il est réellement facile de faire la plupart des impressions depuis Linux ; plus facile qu'avec cet autre système d'exploitation qui commence par « WIN » - et dont je ne parlerai pas.

Tout est plutôt facile tant que vous ne souhaitez imprimer que du texte simple, sans gras, italique, changements de polices, etc. Voici une

application simple qui permet d'imprimer directement sur votre imprimante :

```
import os

pr = os.popen('lpr', 'w')

pr.write('Test imprimante
depuis linux via python\n')

pr.write('Impression
terminee\n')

pr.close()
```

C'est assez facile à comprendre si vous élargissez un peu votre esprit. Dans le code ci-dessus, « lpr » est le spooler d'impression. Le seul prérequis est que nous ayons déjà configuré « lpd » et qu'il fonctionne. C'est très probablement déjà fait pour vous si vous utilisez une imprimante sous Ubuntu. « lpd » est généralement considéré comme un « filtre magique » qui permet de convertir automatiquement différents types de documents en quelque chose que l'imprimante peut comprendre. Nous allons imprimer sur le périphérique/objet « lpr ». Pensez-y comme à un simple fichier. Nous ouvrons le fichier ; nous devons im-

porter « os ». Puis à la ligne 2, nous avons ouvert « lpr » avec un accès en écriture, en l'assignant à la variable objet « pr ». Nous procédons alors à une écriture « pr.write » avec tout ce que nous voulons imprimer. Enfin (ligne 5), nous fermons le fichier ce qui va envoyer les données vers l'imprimante.

Nous pouvons également créer un fichier texte puis l'envoyer à l'imprimante comme ceci...

```
import os

filename = 'fichier.bidon'

os.system('lpr %s' %
filename)
```

Dans ce cas, nous utilisons toujours l'objet lpr mais avec la commande « os.system » qui sert simplement à envoyer à Linux une commande comme si on l'avait saisie depuis un terminal.

Je vous laisserai vous amuser un peu avec cela.

“ **Waouh ! Il est difficile de croire que ceci est déjà le 24<sup>e</sup> numéro. Cela fait deux ans que nous apprenons le Python !**

## PyRTF

Maintenant occupons-nous des fichiers RTF. Le format RTF (c'est comme quand on dit le numéro PIN puisque PIN signifie Numéro d'Identification Personnel et que ça revient à dire le Numéro Numéro d'Identification Personnel [Ndt : en français on n'a pas ce problème de redondance puisqu'on parle de code PIN] : ça dépend du département Département des Redondances, non ?) a été créé à l'origine par Microsoft en 1987 et sa syntaxe s'est inspirée du langage de composition de texte TeX. PyRTF est une merveilleuse bibliothèque qui facilite la création de fichiers RTF. Cela nécessite de réfléchir en amont à ce à quoi le fichier doit ressembler, mais le résultat en vaut vraiment la peine.

Tout d'abord il faut télécharger et installer le paquet pyRTF. Allez sur <http://pyrtf.sourceforge.net> et récupérez le paquet PyRTF-0.45.tar.gz. Sauvegardez-le quelque part et utilisez le gestionnaire d'archives pour le décompresser. Puis ouvrez un terminal et déplacez-vous à l'endroit où vous l'avez décompressé. Tout d'abord il faut installer le paquet, avec la commande « `sudo python setup.py install` ». Remarquez qu'il y a un répertoire d'exemples, qui contient de bonnes informations pour faire des choses un peu compliquées.

Nous y voilà. Commençons comme d'habitude en créant le canevas de notre programme que vous pouvez voir en haut à droite. Avant d'aller plus loin, parlons de ce qui se passe. La ligne 2 importe la bibliothèque pyRTF. Remarquez que nous utilisons un format d'importation différent des autres fois : cette fois-ci nous importons tout ce qui se trouve dans la bibliothèque. Notre routine principale s'appelle FabriqueExemple et ne fait rien pour le moment. La routine OuvreFichier crée un fichier avec pour nom celui passé en argument, lui ajoute l'extension .rtf, le place en mode écriture et retourne un pointeur sur ce fichier.

Nous avons déjà parlé de la routine `__name__` précédemment, mais pour vous rafraîchir la mémoire je vous rappelle que si nous exécutons le programme en mode autonome la variable interne `__name__` est réglée à « `__main__` » ; par contre, si on l'appelle comme « `import` » depuis un autre programme, cette portion de code sera ignorée.

Nous créons là une instance de l'objet `Renderer`, appelons la routine `FabriqueExemple` et récupérons l'objet retourné `docu`. Puis nous écrivons le fichier (`docu`) en utilisant la routine `OuvreFichier`.

Passons maintenant au contenu de la routine principale `FabriqueExemple`. Remplacez l'instruction `pass` par le code ci-dessous.

```
docu = Document()
ss = doc.StyleSheet
section = Section()
docu.Sections.append(section)

p = Paragraph(ss.ParagraphStyles.Normal)
p.append('Voici notre premier exemple de création de fichier RTF. '
        'Ce premier paragraphe est dans le style prédéfini appelé normal '
        'et tous les paragraphes suivants utiliseront ce style sauf si on le change.')
section.append(p)

return docu
```

Regardons ce que nous avons fait. La première ligne crée une instance de `Document`. Puis on crée une instance de `feuille de style`. Ensuite nous créons une instance de l'objet `section` et on l'ajoute au document. Imaginez une section comme un chapitre dans un livre. Ensuite nous créons un paragraphe en utilisant le style `Normal`. L'auteur de pyRTF a pré-régulé ce style avec une police `Arial` en 11 points. Ensuite on écrit le texte qu'on veut dans ce paragraphe, on l'ajoute à la section et on retourne notre document `docu`.

C'est vraiment facile. Encore une fois, vous devez réfléchir soigneusement en amont à la sortie désirée, mais ce n'est pas très compliqué. Sauvegardez ce programme en tant que « `rtftesta.py` » et exécutez-le. Enfin, utilisez `OpenOffice` (ou

`LibreOffice`) pour ouvrir le fichier et l'examiner.

Maintenant faisons quelques modifications sympathiques. Tout d'abord, ajoutons un en-tête. Là encore l'auteur de pyRTF nous fournit un style prédéfini appelé `Header1`, que nous allons utiliser pour notre en-tête. Ajoutez ce qui suit entre les lignes `docu.Sections.append` et `p = Paragraph`.

```
p = Paragraph(ss.ParagraphStyles.Header1)
```

```
p.append('Exemple d'en-tete')
```

```
section.append(p)
```

Modifiez le nom du fichier en « `_rtftestb` » ; cela devrait donner ceci :

```
DR.Write(docu, OuvreFichier('rtftestb'))
```

Sauvegardez-le sous le nom `rtftestb.py` et exécutez-le. Maintenant nous avons un en-tête. Je suis sûr que votre esprit est en train de se demander tout ce qu'on peut faire encore. Voici une liste des styles prédéfinis que l'auteur nous fournit.

Normal, Normal Short, Heading 1, Heading 2, Normal Numbered, Normal Numbered 2. Il y a également un

```
p = Paragraph(ss.ParagraphStyles.Normal)
p.append( 'Il est aussi possible de passer outre les elements d'un style. ',
        'Par exemple vous pouvez modifier seulement la ',
        TEXT(' taille de la police a 24 points', size=48),
        ' ou',
        TEXT(' son type a Impact', font=ss.Fonts.Impact),
        ' ou meme d'autres attributs comme',
        TEXT(' LA GRAISSE',bold=True),
        TEXT(' ou l'italique',italic=True),
        TEXT(' ou LES DEUX',bold=True,italic=True),
        '.')
section.append(p)
```

**Voyons maintenant comment modifier les polices, leur taille et leurs attributs (gras, italique, etc.) à la volée.**

style List que je vous laisserai découvrir. Si vous voulez en voir davantage, sur ça et sur d'autres sujets, les styles sont définis dans le fichier `Elements.py` que vous avez installé tout à l'heure.

Ces styles prédéfinis sont utiles pour beaucoup de choses, mais on peut avoir besoin d'en créer d'autres. Voyons maintenant comment modifier les polices, leur taille et leurs attributs (gras, italique, etc.) à la volée. Après notre paragraphe, et avant de retourner l'objet `document`, insérez le code ci-dessus à droite et modifiez le nom du fichier de sortie en `rtftestc`.

Sauvegardez le fichier sous le nom `rtftestc.py` et exécutez-le. La nouvelle portion du document devrait ressembler à ceci...

Il est également possible de passer outre les éléments d'un style. Par exemple vous pouvez modifier seulement la taille de la police à 24 points, ou son type à Impact ou même modifier d'autres attributs comme la graisse ou l'italique ou les deux.

Bon, qu'avons-nous fait ? La ligne 1 crée un nouveau paragraphe. On commence comme auparavant à l'ajouter au texte. Regardez la ligne 4 (`TEXT(' taille de la police a 24 points', size=48),`) : en utilisant le qualificatif `TEXT` on indique à `pyRTF` qu'il faut faire quelque chose de différent au milieu de la phrase, dans ce cas on modifie la taille de la police (Arial) à 24 points, en précisant

à la suite la commande « `size =` ». Mais attendez une minute : on indique 48 comme taille alors qu'on veut écrire en 24 points ; et la sortie est réellement en 24 points. Que se passe-t-il ici ? Eh bien, la commande de taille est en demi-points ; ainsi si on veut écrire en police 8 points, on doit utiliser « `size = 16` ». Vous comprenez ?

Ensuite, on continue le texte et on modifie la police avec la commande « `_font =` ». Cette fois encore, tout ce qui est dans l'instruction en ligne `TEXT` entre les guillemets sera affecté, mais pas le reste.

Bien. Si vous avez compris tout cela, que peut-on faire d'autre ?

On peut aussi régler la couleur du texte avec l'instruction en ligne `TEXT` de cette façon :

```
p = Paragraph()
p.append('Voici un nouveau
paragraphe avec le mot ',
        TEXT('ROUGE',colour=ss.Colours.Red),
        ' ecrit en rouge.')
section.append(p)
```

Remarquez que nous n'avons pas eu à préciser que le style de paragraphe est Normal, puisqu'il ne change pas tant qu'on ne lui dit pas. Remarquez également que si vous habitez aux États-Unis vous devez utiliser la bonne orthographe pour « `Colours_` » [Ndt : les Américains utilisent souvent l'orthographe « `impropre` » `Color`].

Voici les couleurs prédéfinies : Black,

Blue, Turquoise, Green, Pink, Red, Yellow, White, BlueDark, Teal, GreenDark, Violet, RedDark, YellowDark, GreyDark et Grey.

Et voici une liste de toutes les polices prédéfinies (ce sont les notations pour les utiliser) :

Arial, ArialBlack, ArialNarrow, BitstreamVeraSans, BitstreamVeraSerif, BookAntiqua, BookmanOldStyle, Castellar, CenturyGothic, ComicSansMS, CourierNew, FranklinGothicMedium, Garamond, Georgia, Haettenschweiler, Impact, LucidaConsole, LucidaSansUnicode, MicrosoftSansSerif, PalatinoLinotype, MonotypeCorsiva, Papyrus, Sylfaen, Symbol, Tahoma, TimesNewRoman, TrebuchetMS et Verdana.

Maintenant vous devez penser que tout cela est bien joli, mais comment peut-on créer ses propres styles ? C'est assez simple. Retournez en haut de notre fichier et ajoutez le code qui suit avant la ligne d'en-tête.

```
result = doc.StyleSheet
```

```
NormalText =  
TextStyle(TextPropertySet  
(result.Fonts.CourierNew,16)  
)
```

```
ps2 =  
ParagraphStyle('Courier',
```

```
p = Paragraph(ss.ParagraphStyles.Courier)  
p.append('Now we are using the Courier style at 8 points. '  
        'All subsequent paragraphs will use this style automatically. '  
        'This saves typing and is the default behaviour for RTF documents.',LINE)  
section.append(p)  
p = Paragraph()  
p.append('Also notice that there is a blank line between the previous paragraph ',  
        'and this one. That is because of the "LINE" inline command.')
```

```
section.append(p)
```

```
NormalText.Copy()
```

```
result.ParagraphStyles.append(ps2)
```

Avant d'écrire le code pour l'utiliser, regardons ce que nous avons fait. Nous créons une nouvelle instance de feuille de style nommée result. À la deuxième ligne nous réglons la police à CourierNew en 8 points puis « enregistrons » le style comme Courier. Souvenez-vous que nous devons indiquer 16 comme taille puisque ce sont des demi-points.

Maintenant, ajoutons un nouveau paragraphe en utilisant le style Courier, avant la ligne return en bas de la routine.

Maintenant que vous avez un nouveau style, vous pouvez l'utiliser quand vous le souhaitez. Vous pouvez utiliser n'importe quelle police de la liste ci-dessus et créer vos propres

styles. Recopiez simplement le code du style et remplacez les informations de police et de taille comme vous le voulez. On peut aussi faire cela :

```
NormalText =  
TextStyle(TextPropertySet  
(result.Fonts.Arial,22,bold=  
True,colour=ss.Colours.Red))
```

```
ps2 =  
ParagraphStyle('ArialGrasRouge',  
NormalText.Copy())
```

```
result.ParagraphStyles.append(ps2)
```

Et ajouter le code suivant :

```
p =  
Paragraph(ss.ParagraphStyles.  
ArialGrasRouge)
```

```
p.append(LINE, 'Et  
maintenant on  
utilise le style  
ArialGrasRouge.',LINE)
```

```
section.append(p)
```

pour afficher en style ArialGrasRouge.

## Tableaux

Souvent, la seule manière de présenter correctement des données dans un document est d'utiliser un tableau. Faire des tableaux dans un texte est plutôt difficile, mais PARFOIS c'est plutôt facile avec pyRTF. J'expliquerai cela plus tard dans cet article.

Regardons un tableau standard (ci-dessous) dans OpenOffice/LibreOffice. Cela ressemble à une feuille de calcul, où tout est placé dans des colonnes.

Des lignes horizontales, et des colonnes verticales. Un concept simple.

Commençons une nouvelle application nommée `rtfTableau-a.py`. Démarons avec notre code standard (page suivante) et construisons à partir de là.

Pas besoin d'explications ici puisque c'est à peu près le même code que nous avons utilisé précédemment. Maintenant, écrivons la routine `ExempleTableau`. J'utilise en partie l'exemple fourni par l'auteur de `pyRTF`. Remplacez l'instruction `pass` dans la routine par le code suivant :

```
docu = Document()
ss = docu.StyleSheet
section = Section()
docu.Sections.append(section)
```

Cette partie est la même que précédemment, passons à la suite.

```
tableau =
Table(TabPS.DEFAULT_WIDTH * 7,
      TabPS.DEFAULT_WIDTH * 3,
      TabPS.DEFAULT_WIDTH * 3)
```

Cette ligne (oui, il ne s'agit que d'une ligne, mais découpée pour plus de clarté) crée notre tableau basique.

Nous créons un tableau à trois colonnes, la première contient 7 cellules, les deux suivantes en contiennent 3. Nous n'avons pas que des cellules uniques à notre disposition, car on pourra saisir les largeurs en « twips » [Ndt : ce sont des unités de mesure, utilisées en LibreOffice et pour le RTF, équivalentes à 1/1440 d'un pouce (2,54 cm). Cf [http://en.wikipedia.org/wiki/Twip#In\\_computing](http://en.wikipedia.org/wiki/Twip#In_computing).] Nous y reviendrons dans un moment.

```
c1 = Cell(Paragraph('ligne 1, cellule 1'))
c2 = Cell(Paragraph('ligne 1, cellule 2'))
c3 = Cell(Paragraph('ligne 1, cellule 3'))
tableau.AddRow(c1,c2,c3)
```

Ici nous réglons les données qui vont dans chaque cellule de la première ligne.

```
c1 = Cell(Paragraph(ss.ParagraphStyles.Heading2, 'Style entete 2'))
c2 = Cell(Paragraph(ss.ParagraphStyles.Normal, 'Retour au style Normal'))
c3 = Cell(Paragraph('Encore du style Normal'))
tableau.AddRow(c1,c2,c3)
```

```
#!/usr/bin/env python
from PyRTF import *

def ExempleTableau():
    pass

def OuvreFichier(nom):
    return file('%s.rtf' % nom, 'w')

if __name__ == '__main__':
    DR = Renderer()
    docu = ExempleTableau()
    DR.Write(docu, OuvreFichier('rtftable-a'))
    print "Fini"
```

Ce morceau de code règle les données pour la deuxième ligne. Remarquez que nous pouvons régler des styles différents pour une seule ou plusieurs cellules.

```
c1 = Cell(Paragraph(ss.ParagraphStyles.Heading2, 'Style entete 2'))
c2 = Cell(Paragraph(ss.ParagraphStyles.Normal, 'Retour au style Normal'))
c3 = Cell(Paragraph('Encore du style Normal'))
tableau.AddRow(c1,c2,c3)

Ceci règle la dernière ligne.
section.append(tableau)
return docu
```

Ceci ajoute le tableau dans la section et retourne le document pour affichage.

Sauvegardez et exécutez l'application. Vous remarquerez que tout ressemble à ce que vous attendiez, mais qu'il n'y a pas de bordures pour le tableau. Ceci peut rendre les choses difficiles à lire : réglons ce problème. À nouveau, j'utilise en grande partie le code de l'exemple fourni par l'auteur de `pyRTF`.

Sauvegardez votre fichier sous le nom `rtfTableau-b.py`, puis effacez tout ce qui est entre « `docu.Sections.append(section)` » et « `return docu` » dans la routine `ExempleTableau`, et remplacez-le par ce qui suit :

```
cote_fin = BorderPS( width=20,
style=BorderPS.SINGLE )
cote_epais = BorderPS( width=80,
```

```
style=BorderPS.SINGLE )
```

```
bord_fin = FramePS( cote_fin,
cote_fin, cote_fin, cote_fin )
bord_epais = FramePS( cote_epais,
cote_epais, cote_epais, cote_epais )
```

```
bord_mixte = FramePS( cote_fin,
cote_epais, cote_fin, cote_epais )
```

Ici nous réglons les définitions des côtés et des bords pour les encadrements.

```
tableau = Table(
TabPS.DEFAULT_WIDTH * 3,
TabPS.DEFAULT_WIDTH * 3,
TabPS.DEFAULT_WIDTH * 3 )
```

```
c1 = Cell( Paragraph( 'L1C1' ),
bord_fin )
```

```
c2 = Cell( Paragraph( 'L1C2' ) )
```

```
c3 = Cell( Paragraph( 'L1C3' ),
bord_epais )
```

```
tableau.AddRow( c1, c2, c3 )
```

Dans la première ligne, les cellules de la colonne 1 (bord\_fin) et de la colonne 3 (bord\_epais) auront une bordure.

```
c1 = Cell( Paragraph( 'L2C1' ) )
```

```
c2 = Cell( Paragraph( 'L2C2' ) )
```

```
c3 = Cell( Paragraph( 'L2C3' ) )
```

```
tableau.AddRow( c1, c2, c3 )
```

Aucune des cases n'aura de bordure dans la ligne 2.

```
c1 = Cell( Paragraph( 'L3C1' ),
bord_mixte )
```

```
c2 = Cell( Paragraph( 'L3C2' ) )
```

```
c3 = Cell( Paragraph( 'L3C3' ),
bord_mixte )
```

```
tableau.AddRow( c1, c2, c3 )
```

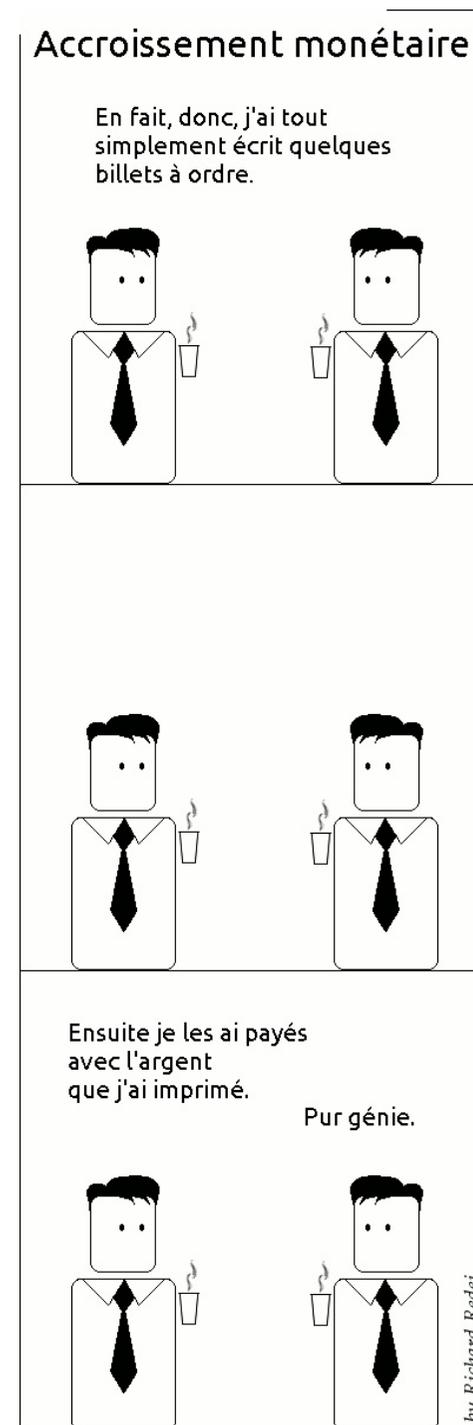
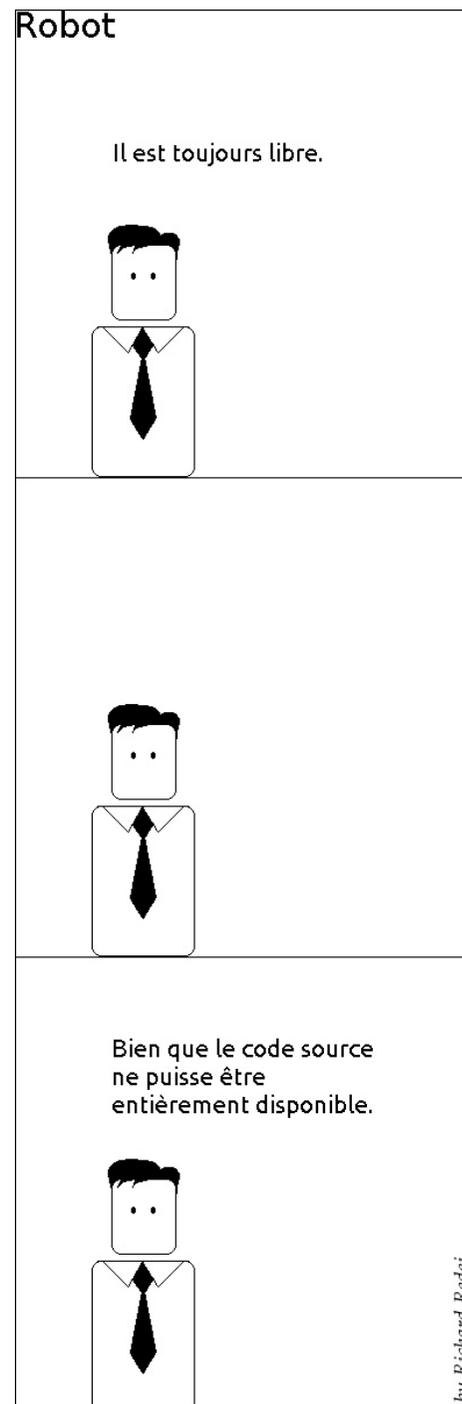
À nouveau, les cases des colonnes 1 et 3 auront une bordure mixte dans la troisième ligne.

```
section.append( tableau )
```

Et voilà. Vous avez maintenant les bases pour créer des documents RTF avec du code.

À la prochaine fois !

Le code source est disponible sur pastebin comme d'habitude. La première partie est ici : <http://pastebin.com/uRVrGjkV> et contient le résumé de rtfTest.py (a à e), la seconde partie rtfTableau.py (a et b) est ici : <http://pastebin.com/L8DGU7Lz>.



Un certain nombre d'entre vous ont commenté les articles de programmation graphique et dit combien vous les avez appréciés. En réponse à cela, nous allons commencer à jeter un œil à un autre outil d'interfaces graphiques appelé Tkinter. Ceci est la façon « officielle » de faire de la programmation graphique en Python. Tkinter existe depuis longtemps et a une assez mauvaise réputation pour son côté « démodé ». Ceci a changé récemment, alors j'ai pensé que nous pourrions nous battre contre ce mauvais processus de réflexion.

N.B. : Tout le code présenté ici est pour Python 2.x seulement. Dans un prochain article, nous allons discuter de la façon d'utiliser Tkinter avec Python 3.x. Si vous DEVEZ utiliser Python 3.x, changez les déclarations d'importation en « `from tkinter import *` ».

## Un peu d'histoire et un peu de contexte

Tkinter est l'abréviation de « Tk interface ». Tk est un langage de programmation à lui tout seul, et le module Tkinter nous permet d'uti-

liser les fonctions de l'interface graphique de ce langage. Il y a un certain nombre de widgets qui viennent nativement avec le module Tkinter. Parmi eux, on trouve des conteneurs de haut niveau (des fenêtres principales), des boutons, des étiquettes, des cadres, des zones de saisie de texte, des cases à cocher, des boutons radio, des canevas, des entrées de texte multilignes, et bien plus encore. Il y a aussi de nombreux modules qui ajoutent des fonctionnalités par dessus Tkinter. Ce mois-ci, nous allons nous concentrer sur quatre widgets. Un conteneur de haut niveau (à partir d'ici je vais essentiellement l'appeler la fenêtre racine), un cadre, des étiquettes et des boutons. Dans le prochain article, nous verrons plus de widgets plus en profondeur.

Fondamentalement, nous avons le widget conteneur de haut niveau qui contient d'autres widgets. Il s'agit de la fenêtre racine ou principale. Dans cette fenêtre racine, nous plaçons les widgets que nous voulons utiliser dans notre programme. Chaque widget, à l'exception du conteneur racine principal, a un parent. Le parent n'est pas forcément la fenêtre racine ; ça

programmer en python

peut être un autre widget. Nous verrons cela le mois prochain. Pour ce mois-ci, tous les widgets auront pour parent la fenêtre racine.

Afin de placer et d'afficher les widgets enfants, nous devons utiliser ce qu'on appelle la « gestion de géométrie ». C'est la façon dont les choses se placent dans la fenêtre racine principale. La plupart des programmeurs utilisent un de ces trois types de gestion de géométrie : Packer, Grid, ou Gestion de la place. À mon humble avis, la méthode Packer est très maladroite. Je vous laisse l'explorer par vous-même. La méthode de gestion de la place permet un placement extrêmement précis des widgets, mais ça peut être compliqué. Nous en reparlerons dans un futur article. Cette fois-ci, nous allons nous concentrer sur la méthode de la grille.

Pensez à un tableur. Il y a des lignes et des colonnes. Les colonnes sont verticales, les lignes sont horizontales. Voici une représentation texte simple des adresses de cellule d'une grille

	COLUMNS				-	>
ROWS	0,0	1,0	2,0	3,0	4,0	
	0,1	1,1	2,1	3,1	4,1	
	0,2	1,2	2,2	3,2	4,2	
	0,3	1,3	2,3	3,3	4,3	

simple de 5 colonnes sur 4 lignes (en haut à droite). Le parent possède la grille, les widgets vont dans les positions de la grille. Au premier regard, vous pourriez penser que cela est très limitatif. Toutefois, les widgets peuvent s'étendre sur plusieurs positions sur la grille, soit dans le sens des colonnes, soit dans celui des lignes, ou les deux à la fois.

## Notre premier exemple

Notre premier exemple est SUPER simple (seulement quatre lignes), mais explicite.

```
from Tkinter import *
racine = Tk()
bouton = Bouton(racine, text = "Bonjour FullCircle").grid()
racine.mainloop()
```

# TUTORIEL - PROGRAMMER EN PYTHON - PARTIE 25

Bon, qu'est-ce qui se passe ici ? La première ligne importe la bibliothèque Tkinter. Ensuite, on instancie l'objet Tk racine (Tk est une partie de Tkinter). Voici la ligne trois :

```
bouton = Button(racine, text = "Bonjour FullCircle").grid()
```

Nous créons un bouton appelé bouton, définissons son parent à la fenêtre racine, réglons son texte à « Bonjour FullCircle » et le plaçons dans la grille. Enfin, nous appelons la boucle principale de la fenêtre. Ça paraît très simple quand on regarde le code, mais beaucoup de choses se passent dans les coulisses. Heureusement, nous n'avons pas besoin de comprendre tout cela pour l'instant.

Exécutez le programme et nous allons voir ce qui se passe. Sur ma machine, la fenêtre principale apparaît en bas à gauche de l'écran. Elle pourrait apparaître ailleurs sur le vôtre. Cliquer sur le bouton ne fait rien. Réparons cela dans notre prochain exemple.

## Notre deuxième exemple

Cette fois, nous allons créer une classe appelée App. Ce sera la classe qui détient effectivement notre fenêtre.

```
class App:
    def __init__(self, principale):
        cadre = Frame(principale)
        self.lblTexte = Label(cadre, text = "Voici un widget label")
        self.btnQuitter = Button(cadre, text="Quitter", fg="red", command=cadre.quit)
        self.btnBonjour = Button(cadre, text="Bonjour", command=self.DitUnTruc)
        cadre.grid(column = 0, row = 0)
        self.lblTexte.grid(column = 0, row = 0, columnspan = 2)
        self.btnBonjour.grid(column = 0, row = 1)
```

Commençons :

```
from Tkinter import *
```

C'est la déclaration d'importation pour la bibliothèque Tkinter.

Nous définissons notre classe, et dans la routine `__init__`, nous mettons en place nos widgets et les plaçons dans la grille.

La première ligne dans la routine `__init__` crée un cadre qui sera le parent de tous nos autres widgets. Le parent de ce cadre est la fenêtre racine (widget de plus haut niveau). Ensuite, nous définissons un label et deux boutons. Regardons la ligne de création de l'étiquette.

```
self.lblTexte = Label(cadre, text = "Ceci est un widget label")
```

Nous créons le widget étiquette et

l'appelons `self.lblTexte`. Il hérite de l'objet widget `Label`. Nous réglons son parent (le cadre) et définissons le texte à afficher (`text = "Ceci est un widget label"`). C'est aussi simple que cela. Bien sûr, nous pouvons faire beaucoup mieux, mais pour l'instant c'est tout ce dont nous avons besoin. Ensuite, nous mettons en place les deux boutons que nous allons utiliser :

```
self.btnQuitter = Button(cadre, text="Quitter", fg="red", command=cadre.quit)
self.btnBonjour = Button(cadre, text="Bonjour", command=self.DitUnTruc)
```

Nous nommons les widgets, fixons leur parent (cadre) et définissons le texte à afficher. Maintenant `btnQuitter` a un attribut marqué `fg` que nous avons réglé à « red ». Vous avez deviné que cela définit la couleur d'avant-plan ou la couleur du texte à la couleur rouge. Le dernier attribut sert à définir la commande que nous

voulons utiliser lorsque l'utilisateur clique sur le bouton. Dans le cas de `btnQuitter`, c'est `cadre.quit`, qui termine le programme. C'est une fonction intégrée, donc nous n'avons pas besoin de la créer. Dans le cas de `btnBonjour`, c'est une routine appelée `self.DitUnTruc`. Nous devons créer celle-ci, mais auparavant nous avons encore quelque chose à faire.

Nous devons placer nos widgets dans la grille. Voici les lignes à nouveau :

```
cadre.grid(column = 0, row = 0)
```

```
self.lblTexte.grid(column = 0, row = 0, columnspan = 2)
```

```
self.btnBonjour.grid(column = 0, row = 1)
```

```
self.btnQuitter.grid(column = 1, row = 1)
```

Tout d'abord, nous attribuons une grille au cadre. Ensuite, nous réglons l'attribut de grille de chaque widget

selon l'endroit où nous voulons placer le widget. Notez la ligne « colspan » pour l'étiquette (self.lblTexte). Cela indique que nous voulons que l'étiquette s'étende sur deux colonnes de la grille. Puisque nous avons seulement deux colonnes, il s'agit de toute la largeur de l'application. Maintenant nous pouvons créer notre fonction de rappel :

```
def DitUnTruc(self):
    print "Bonjour lecteur
du Magazine FullCircle!!"
```

Cela affiche simplement dans la fenêtre du terminal le message "Bonjour lecteur du Magazine FullCircle!!" Enfin, on instancie la classe Tk - notre classe App - et exécutons la boucle principale :

```
class Calculator():
    def __init__(self, root):
        master = Frame(root)
        self.CurrentValue = 0
        self.HolderValue = 0
        self.CurrentFunction = ''
        self.CurrentDisplay = StringVar()
        self.CurrentDisplay.set('0')
        self.DecimalNext = False
        self.DecimalCount = 0
        self.DefineWidgets(master)
        self.PlaceWidgets(master)
```

```
racine = Tk()
app = App(racine)
racine.mainloop()
```

Essayez le programme. Maintenant, il fait vraiment quelque chose. Mais là encore, la position de la fenêtre est très gênante. Corrigons cela dans notre prochain exemple.

## Notre troisième exemple

Enregistrez l'exemple précédent sous le nom exemple3.py. Tout est exactement pareil, sauf une seule ligne qui se trouve en bas de la routine principale. Je vais vous montrer ces lignes avec la nouvelle :

```
root = Tk()
root.geometry('150x75+550+150')
```

```
app = App(root)
root.mainloop()
```

Ceci force notre fenêtre initiale à une taille de 150 pixels de large sur 75 pixels de haut. Nous voulons aussi que le coin supérieur gauche de la fenêtre soit placé à une position hori-

zontale de 550 pixels (depuis la droite) et à une position verticale de 150 pixels (depuis le haut). Comment suis-je arrivé à ces chiffres ? J'ai commencé avec des valeurs raisonnables et les ai peaufiné à partir de là. C'est un peu difficile de faire de cette façon, mais les résultats sont meilleurs que si on ne fait rien du tout.

## Notre quatrième exemple - Une calculatrice simple

Maintenant, regardons quelque chose d'un peu plus compliqué. Cette fois, nous allons créer une calculatrice simple à 4 boutons, pour les 4 opérations : addition, soustraction, multiplication et division. À droite vous voyez à

quoil elle ressemblera, sous forme de texte simple.

Nous allons y plonger tout de suite et je vous expliquerai le code (au milieu à droite) au fur et à mesure. À part la déclaration de la géométrie, ceci devrait être assez facile pour vous de comprendre maintenant (à gauche). Rappelez-vous, prenez des valeurs raisonnables, modifiez-les, puis

```
-----
|                                     0 |
-----
| 1 | 2 | 3 | + |
-----
| 4 | 5 | 6 | - |
-----
| 7 | 8 | 9 | * |
-----
| - | 0 | . | / |
-----
|                                     = |
-----
|                                     CLEAR |
-----
```

```
from Tkinter import *

def StartUp():
    global val, w, root
    root = Tk()
    root.title('Easy Calc')
    root.geometry('247x330+469+199')
    w = Calculator(root)
    root.mainloop()
```

continuez.

Nous commençons notre définition de la classe en mettant en place notre fonction \_\_init\_\_. Nous réglons trois variables comme suit :

- ValeurCourante - Contient la valeur actuelle qui a été entrée dans la calculatrice.
- ValeurAncienne - Contient la valeur

qui existait avant que l'utilisateur ne clique sur une touche de fonction.

- Fonction Courante - C'est tout simplement pour se souvenir quelle fonction est traitée.

Ensuite, nous définissons la variable `AffichageCourant` et l'attribuons à l'objet `StringVar`. C'est un objet spécial qui fait partie de la trousse Tkinter. Quel que soit le widget auquel vous l'attribuez, cela met automatiquement à jour la valeur dans le widget. Dans ce cas, nous allons l'utiliser pour contenir ce que nous voulons que le widget d'affichage label... euh... eh bien... affiche. Nous devons l'instancier avant de pouvoir l'assigner au widget. Ensuite, nous utilisons la fonction « set » fournie par Tkinter. Nous définissons ensuite une variable booléenne appelée `PartieDecimale` et une variable `CompteDecimales`, puis nous appelons la fonction `DefinirWidgets` qui crée tous les widgets et ensuite nous appelons la fonction `PlacerWidgets`, qui les place réellement dans la fenêtre racine.

```
def  
DefinirWidgets(self, principale):
```

```
self.lblAffichage =
```

```
Label(principale,  
anchor=E, relief =  
SUNKEN, bg="white",  
height=2, textvar  
iable=self.Affic  
hageCourant)
```

Bon, nous avons déjà défini un label auparavant. Cependant, cette fois, nous ajoutons un certain nombre d'autres attributs. Notez que nous n'utilisons pas l'attribut « text ». Ici, nous assignons l'étiquette au parent (la fenêtre principale), puis nous définissons l'ancre (ou, pour nos fins, la justification) pour le texte lorsqu'il est écrit. Dans ce cas, nous précisons à l'étiquette de justifier tout le texte à l'est, c'est-à-dire sur le côté droit du widget. Il existe un attribut de justification, mais il sert lorsqu'il y a plusieurs lignes de texte. L'attribut d'ancrage a les options suivantes : N, NE, E, SE, S, SW, W, NW et CENTER. La valeur par défaut est de centrer. Vous devriez penser à des points cardinaux. Dans des circonstances normales, les valeurs réellement utilisables sont E (à droite), W (à gauche), et CENTER (pour centrer).

Ensuite, nous réglons le relief, qui est le style visuel de l'étiquette. Les options autorisées sont FLAT (à

```
self.btn1 = Button(master, text = '1', width = 4, height=3)  
self.btn1.bind('<ButtonRelease-1>', lambda e: self.funcNumButton(1))  
self.btn2 = Button(master, text = '2', width = 4, height=3)  
self.btn2.bind('<ButtonRelease-1>', lambda e: self.funcNumButton(2))  
self.btn3 = Button(master, text = '3', width = 4, height=3)  
self.btn3.bind('<ButtonRelease-1>', lambda e: self.funcNumButton(3))  
self.btn4 = Button(master, text = '4', width = 4, height=3)  
self.btn4.bind('<ButtonRelease-1>', lambda e: self.funcNumButton(4))
```

plat), SUNKEN (en creux), RAISED (en relief), GROOVE (en strie) et RIDGE (en arête). La valeur par défaut est à plat si vous ne spécifiez rien. N'hésitez pas à essayer les autres combinaisons par vous-mêmes lorsque nous aurons fini. Ensuite, nous définissons le fond (bg) à blanc afin de le démarquer un peu du reste de la fenêtre. Nous fixons la hauteur à 2 (qui signifie deux lignes de texte de haut, et non pas 2 pixels) et enfin nous assignons la variable que nous venons de définir juste avant (`self.AffichageCourant`) à l'attribut `textvariable`. À chaque fois que la valeur de `self.AffichageCourant` changera, le label modifiera son texte pour correspondre automatiquement.

Ci-dessus, nous allons créer quelques-uns des boutons.

J'ai montré seulement 4 boutons ici. C'est parce que, comme vous pouvez le voir, le code est presque

exactement le même. Encore une fois, nous avons créé des boutons plus tôt dans ce tutoriel, mais nous allons regarder de plus près ce que nous faisons ici.

Nous commençons par définir le parent (la fenêtre principale), le texte que nous voulons sur le bouton, et la largeur et la hauteur. Notez que la largeur est en caractères et la hauteur est en lignes de texte. Si vous vouliez un graphique dans le bouton, vous utiliseriez des pixels pour définir la hauteur et la largeur. Cela peut devenir un peu confus jusqu'à ce que vous le compreniez sans faille. Ensuite, nous réglons l'attribut « bind ». Quand nous avons fait des boutons dans les exemples précédents, nous avons utilisé l'attribut « command= » pour définir quelle fonction serait appelée lorsque l'utilisateur clique sur le bouton. Cette fois, nous utilisons l'attribut « .bind ». [Ndt : « relier »]. C'est presque la

même chose, mais c'est un moyen plus facile de le faire et de transmettre des informations à la routine de rappel qui est statique. Notez que nous utilisons ici « <ButtonRelease-1> » comme l'élément déclencheur de la liaison. Dans ce cas, nous voulons nous assurer que l'appel de la fonction se fait seulement après que l'utilisateur clique ET relâche le bouton gauche de la souris. Enfin, nous définissons la fonction de rappel que nous voulons utiliser et ce que nous allons lui envoyer. Maintenant, ceux d'entre vous qui sont astucieux (ce qui est bien sûr votre cas à tous) noteront quelque chose de nouveau : L'appel « lambda e: ».

En Python, nous utilisons Lambda pour définir des fonctions anonymes qui apparaîtront à l'interpréteur comme des instructions valides. Cela nous permet de mettre plusieurs morceaux dans une seule ligne de code. Pensez-y comme à une mini-fonction. Dans ce cas, nous mettons en place le nom de la fonction de rappel et la valeur que nous voulons lui envoyer, ainsi que la balise événement (e:). Nous parlerons plus en détail de Lambda dans un article ultérieur. Pour l'instant, il suffit de suivre l'exemple.

Je vous ai donné les quatre premiers

```
self.btnDash = Button(principale, text = '-',width = 4,height=3)
self.btnDash.bind('<ButtonRelease-1>', lambda e: self.foncBoutonFonction('SIGNE'))
self.btnDot = Button(principale, text = '.',width = 4,height=3)
self.btnDot.bind('<ButtonRelease-1>', lambda e: self.foncBoutonFonction('Dec'))
```

Le bouton « btnDash » change le signe de la valeur affichée. 523 devient -523 et -523 devient 523. Le bouton btnDot saisit un point décimal. Ces exemples, ainsi que les suivants, utilisent la fonction foncBoutonFonction.

```
self.btnPlus = Button(principale,text = '+', width = 4, height=3)
self.btnPlus.bind('<ButtonRelease-1>', lambda e: self.foncBoutonFonction('Ajouter'))
self.btnMinus = Button(principale,text = '-', width = 4, height=3)
self.btnMinus.bind('<ButtonRelease-1>', lambda e: self.foncBoutonFonction('Soustraire'))
self.btnStar = Button(principale,text = '*', width = 4, height=3)
self.btnStar.bind('<ButtonRelease-1>', lambda e: self.foncBoutonFonction('Multiplier'))
self.btnDiv = Button(principale,text = '/', width = 4, height=3)
self.btnDiv.bind('<ButtonRelease-1>', lambda e: self.foncBoutonFonction('Diviser'))
self.btnEqual = Button(principale, text = '=')
self.btnEqual.bind('<ButtonRelease-1>', lambda e: self.foncBoutonFonction('Egal'))
```

Voici les quatre boutons pour les fonctions mathématiques.

```
self.btnClear = Button(principale, text = 'EFFACER')
self.btnClear.bind('<ButtonRelease-1>', lambda e: self.foncEffacer())
```

Enfin, voici le bouton Effacer. Il efface bien sûr les variables et l'affichage. Maintenant nous plaçons les widgets avec la routine PlacerWidgets. D'abord nous initialisons la grille, puis nous plaçons les widgets dedans. Voici la première partie de la routine :

```
def PlacerWidgets(self,principale):
    principale.grid(column=0,row=0)
    self.lblAffichage.grid(column=0,row=0,columnspan = 4,sticky=EW)
    self.btn1.grid(column = 0, row = 1)
    self.btn2.grid(column = 1, row = 1)
    self.btn3.grid(column = 2, row = 1)
    self.btn4.grid(column = 0, row = 2)
    self.btn5.grid(column = 1, row = 2)
    self.btn6.grid(column = 2, row = 2)
    self.btn7.grid(column = 0, row = 3)
    self.btn8.grid(column = 1, row = 3)
    self.btn9.grid(column = 2, row = 3)
    self.btn0.grid(column = 1, row = 4)
```

boutons. Copiez et collez le code ci-dessus pour les boutons de 5 à 9 et pour le bouton 0. Ils sont tous identiques, à l'exception du nom du bouton et de la valeur que nous envoyons au rappel. Les prochaines étapes sont indiquées à droite.

La seule chose dont nous n'avons pas parlé pour l'instant, ce sont les attributs « colspan » et « sticky ». Comme je l'ai mentionné auparavant, un widget peut s'étendre sur plus d'une colonne ou une ligne. Dans ce cas, nous « étirons » le widget étiquette sur les quatre colonnes. C'est ce que fait l'attribut « colspan ». Il existe également un attribut « rowspan ». L'attribut « sticky » [Ndt : « collant »] indique au widget où aligner ses bords. Pensez-y comme la manière dont le widget se remplit au sein de la grille. En haut à gauche vous voyez le reste de nos boutons.

Avant d'aller plus loin nous allons jeter un œil à la façon dont les choses vont fonctionner quand l'utilisateur appuiera sur les boutons.

Disons que l'utilisateur veut saisir  $563 + 127$  et obtenir la réponse. Il appuiera ou cliquera (logiquement) sur 5, puis 6, puis 3, puis le « + », puis 1, puis 2, puis 7, puis le bouton « = ».

```
self.btnDash.grid(column = 0, row = 4)
self.btnDot.grid(column = 2, row = 4)
self.btnPlus.grid(column = 3, row = 1)
self.btnMinus.grid(column = 3, row = 2)
self.btnStar.grid(column = 3, row = 3)
self.btnDiv.grid(column=3, row = 4)
self.btnEqual.grid(column=0,row=5,columnspan = 4,sticky=NSEW)
self.btnClear.grid(column=0,row=6,columnspan = 4, sticky = NSEW)
```

```
def funcNumButton(self, val):
    if self.DecimalNext == True:
        self.DecimalCount += 1
        self.CurrentValue = self.CurrentValue + (val * (10**(-self.DecimalCount)))
    else:
        self.CurrentValue = (self.CurrentValue * 10) + val
    self.DisplayIt()
```

Comment pouvons-nous gérer cela dans le code ? Nous avons déjà réglé les rappels pour les touches numériques à la fonction `foncBoutonNumerique`. Il y a deux façons de gérer cela. Nous pouvons conserver les informations saisies comme une chaîne et puis la convertir en nombre quand nous avons besoin, ou bien nous pouvons le garder comme un nombre tout le temps. Nous allons utiliser cette dernière méthode. Pour ce faire, nous allons conserver la valeur qui est déjà là (0 quand nous commencerons) dans une variable appelée « `self.ValeurCourante` », puis quand un nouveau chiffre arrive, nous prenons la variable, la multiplions par 10 et ajoutons la nouvelle valeur. Ainsi, lorsque l'utilisateur entre 5, 6

et 3, nous faisons les choses suivantes :

L'utilisateur clique 5 :  $0 * 10 + 5$  (5)

L'utilisateur clique 6 :  $5 * 10 + 6$  (56)

L'utilisateur clique 3 :  $56 * 10 + 3$  (563)

Bien sûr, nous devons ensuite afficher la variable « `self.ValeurCourante` » dans l'étiquette.

Ensuite, l'utilisateur clique sur le bouton « + ». Nous prenons la valeur de « `self.ValeurCourante` » et la plaçons dans la variable « `self.ValeurAncienne` » et réinitialisons « `self.ValeurCourante` » à 0. Nous devons ensuite répéter le

processus pour les clics sur 1, 2 et 7. Lorsque l'utilisateur clique sur la touche « = », nous devons ensuite ajouter les valeurs de « `self.ValeurCourante` » et « `self.ValeurAncienne` », les afficher, puis effacer les deux variables pour continuer.

Ci-dessus, voici le code pour commencer à définir nos fonctions de rappel.

La routine « `foncBoutonNumerique` » reçoit la valeur que nous lui passons en appuyant sur un bouton. La seule chose qui diffère de l'exemple ci-dessus est lorsque l'utilisateur appuie sur le bouton de décimale (« . »). Ci-dessous, vous verrez que nous utilisons une variable booléenne pour retenir le fait qu'il a déjà appuyé sur

le bouton décimal, et, au prochain clic, on s'en occupe. D'où la ligne « if self.PartieDecimale == True: ». Nous allons procéder pas à pas.

L'utilisateur clique sur 3, puis 2, puis le point décimal, puis 4 pour créer « 32.4 ». Nous traitons les clics sur 3 et 2 grâce à la routine « foncBoutonNumerique ». Nous vérifions pour voir si self.PartieDecimale est vrai (ce qu'il n'est pas tant que l'utilisateur n'a pas cliqué sur le bouton « . »). Sinon, nous multiplions simplement la valeur de self.ValeurCourante par 10 et ajoutons la nouvelle valeur. Lorsque l'utilisateur clique sur le « . », la fonction de rappel « foncBoutonFonction » est appelée avec la valeur « Dec ». Tout ce que nous faisons est de régler la variable booléenne « self.PartieDecimale » à vrai (True). Lorsque l'utilisateur clique sur le 4, nous allons tester la valeur de « self.PartieDecimale » et, puisqu'elle est vraie, nous faisons un peu de magie. Premièrement, on incrémente la variable self.CompteDecimales, qui nous indique le nombre de décimales avec lequel nous travaillons. Nous prenons ensuite la nouvelle valeur entrante, la multiplions par  $(10^{**</nowiki>-self.CompteDecimales})$ . En

```
def foncBoutonFonction(self, fonction):
    if fonction == 'Dec':
        self.PartieDecimale = True
    else:
        self.PartieDecimale = False
        self.CompteDecimales = 0
        if fonction == 'SIGNE':
            self.ValeurCourante *= -1
            self.Rafraichir()4
```

La fonction ABS fournit simplement la valeur actuelle et la multiplie par -1.

```
elif fonction == 'Ajouter':
    self.ValeurAncienne = self.ValeurCourante
    self.ValeurCourante = 0
    self.FonctionCourante = 'Ajouter'
```

La fonction Add copie “self.CurrentValue” dans “self.HolderValue”, nettoie “self.CurrentValue”, and sets the “self.CurrentFunction” to “Add”. Les fonctions Soustraire, Multiplier et Diviser font la même chose avec the proper keyword being set in “self.CurrentFunction”.

```
elif fonction == 'Soustraire':
    self.ValeurAncienne = self.ValeurCourante
    self.ValeurCourante = 0
    self.FonctionCourante = 'Soustraire'
elif fonction == 'Multiplier':
    self.ValeurAncienne = self.ValeurCourante
    self.ValeurCourante = 0
    self.FonctionCourante = 'Multiplier'
elif fonction == 'Diviser':
    self.ValeurAncienne = self.ValeurCourante
    self.ValeurCourante = 0
    self.FonctionCourante = 'Diviser'
```

La fonction “Eq” (Egal) is where the “magic” happens. Il sera facile pour vous de comprendre le code suivant maintenant.

```
elif fonction == 'Egal':
    if self.FonctionCourante == 'Ajouter':
        self.ValeurCourante += self.ValeurAncienne
    elif self.FonctionCourante == 'Soustraire':
        self.ValeurCourante = self.ValeurAncienne - self.ValeurCourante
    elif self.FonctionCourante == 'Multiplier':
        self.ValeurCourante *= self.ValeurAncienne
    elif self.FonctionCourante == 'Diviser':
        self.ValeurCourante = self.ValeurAncienne / self.ValeurCourante
    self.Rafraichir()
    self.ValeurCourante = 0
    self.ValeurAncienne = 0
```

utilisant cet opérateur magique, nous obtenons une simple fonction « élévation à la puissance ». Par exemple `10<nowiki>**2` renvoie 100. `10<nowiki></nowiki>-2` retourne 0.01. Parfois, en utilisant cette routine, cela conduit à un problème d'arrondi, mais pour notre calculatrice simple, cela fonctionnera pour la plupart des nombres décimaux raisonnables. Je vais vous laisser le soin de travailler à une meilleure fonction. Prenez cela comme vos devoirs pour ce mois-ci.

```
def funcClear(self):  
  
self.CurrentValue = 0  
  
self.HolderValue = 0  
  
self.DisplayIt()
```

La routine « `foncEffacer` » efface simplement les deux variables mémoire, puis rafraîchit l'affichage. `def foncEffacer(self): self.ValeurCourante = 0 self.ValeurAncienne = 0 self.Rafraichir()` Maintenant les fonctions. Nous avons déjà discuté de ce qui se passe avec la fonction « `Dec` ». Nous l'avons traitée en premier avec l'instruction « `if` ». Nous allons passer au « `else` » et, dans le cas où la fonction est autre, nous effaçons les variables « `self.PartieDecimale` » et « `self.CompteDecimales` ».

Les prochaines étapes sont indiquées sur la page précédente (encadré de droite).

La routine « `Rafraichir` » règle simplement la valeur de l'étiquette d'affichage. N'oubliez pas que nous avons dit à l'étiquette de « `surveiller` » la variable « `self.AffichageCourant` ». À chaque fois que cette variable change, l'étiquette change automatiquement d'affichage pour correspondre. Nous utilisons la méthode « `.set` » pour changer la valeur.

```
def Rafraichir(self):  
  
print('ValeurCourante = {0}  
- ValeurAncienne =  
{1}'.format(self.ValeurCourante,  
self.ValeurAncienne))  
  
self.AffichageCourant.set(self.ValeurCourante)
```

Enfin, nous avons nos lignes de démarrage.

```
if __name__ == '__main__':  
  
Demarrage()
```

Maintenant, vous pouvez exécuter le programme et l'essayer.

Comme toujours, le code de cet article peut être trouvé sur PasteBin.

Les exemples 1, 2 et 3 sont ici :

<http://pastebin.com/RAF4KK6E>

et l'exemple `Calc.py` est ici :

<http://pastebin.com/Pxr0H8FJ>

Le mois prochain, nous allons continuer à explorer Tkinter et la richesse de ses widgets. Dans un prochain article, nous verrons un concepteur d'interface graphique pour Tkinter appelé PAGE. En attendant, amusez-vous bien. Je pense que vous apprécierez Tkinter.

Le mois dernier, nous avons parlé de Tkinter et de quatre des widgets disponibles : la fenêtre principale, les fenêtres, les boutons et les étiquettes (ou labels). Je vous ai également dit le mois dernier que je parlerais de la façon d'avoir un widget autre que le widget de premier niveau comme parent. Aussi, ce mois-ci, nous allons approfondir les fenêtres, les boutons et les étiquettes, et introduire les cases à cocher, les boutons radio, les zones de texte (ou widgets Entry), les listes avec une barre de défilement verticale (ListBox) et les fenêtres de message. Avant de commencer, examinons certains de ces widgets.

Les cases à cocher servent à faire plusieurs choix parmi plusieurs propositions et ont deux états : cochée ou non cochée, ou on pourrait dire aussi oui ou non. Elles sont généralement utilisées pour fournir une série d'options où une, quelques-unes ou toutes peuvent être sélectionnées. Vous pouvez définir un événement pour vous informer quand la case a changé d'état ou tout simplement pour interroger la valeur du widget à tout moment.

Les boutons radio servent à faire un choix parmi plusieurs propositions. Ils ont aussi deux états, oui ou non. Cependant, ils sont groupés ensemble pour fournir un groupe d'options dont une seule peut être choisie. Vous pouvez avoir plusieurs groupes de boutons radio qui, s'ils sont bien programmés, n'interféreront pas entre eux.

Une ListBox fournit une liste d'éléments parmi lesquels l'utilisateur peut choisir. La plupart du temps, vous voulez que l'utilisateur sélectionne un seul des éléments à la fois, mais, parfois, vous pouvez vouloir permettre à l'utilisateur de sélectionner plusieurs éléments. Une barre de défilement peut être placée horizontalement ou verticalement afin de permettre à l'utilisateur de parcourir facilement tous les éléments disponibles.

Notre projet consistera en une fenêtre principale et sept cadres principaux qui regrouperont visuellement nos ensembles de widgets :

1. Le premier cadre sera très basique : il contient simplement différents labels, montrant les différentes options de relief.

2. Le second contiendra des boutons - c'est plutôt simple aussi - qui utilisent différentes options de relief.

3. Dans ce cadre, nous aurons deux cases à cocher et un bouton qui peut les activer/désactiver et qui enverront leur état (1 ou 0) à la fenêtre du terminal lorsqu'on clique dessus ou les active/désactive.

4. Ensuite, nous aurons deux groupes de trois boutons radio, chacun envoyant un message à la fenêtre du terminal lorsqu'on clique dessus. Chaque groupe est indépendant de l'autre.

5. Celui-ci contient des champs de texte qui ne sont pas nouveaux pour vous, mais il y a aussi un bouton pour activer et désactiver l'un d'eux. Lorsqu'il est désactivé, aucune saisie ne peut y être faite.

6. Celui-ci contient une liste avec une barre de défilement verticale qui envoie un message au terminal chaque fois qu'un élément est sélectionné ; il aura deux boutons. Un

```
# widgetdemo1.py
# Labels
from Tkinter import *

class Demo:
    def __init__(self, principale):
        self.DefinirVariables()
        f = self.ConstruireWidgets(principale)
        self.PlacerWidgets(f)
```

bouton va effacer la zone de liste et l'autre la remplira avec des valeurs fictives.

7. Le dernier cadre contient une série de boutons qui appellent les différents types de boîtes de message.

Bon, maintenant nous allons commencer notre projet. Nommons-le « wid-getdemo1.py ». Assurez-vous de le sauvegarder, car nous allons écrire notre projet par petits morceaux et construire notre application complète petit à petit. Chaque morceau tourne autour de l'un des cadres. Vous remarquerez que j'intègre un certain nombre de commentaires au fur et à mesure, pour que vous puissiez suivre ce qui se passe. Voici les premières lignes (voir encadré ci-dessus).

Les deux premières lignes (commentaires) sont le nom de l'application et le thème de cette partie. La ligne trois est notre déclaration d'importation. Ensuite, nous définissons notre classe. La ligne suivante commence notre routine `__init__`, avec laquelle vous devriez tous être familiers maintenant ; mais si vous venez juste de nous rejoindre, c'est le code qui est exécuté quand on instancie la routine dans la partie principale du programme. Nous lui passons la fenêtre racine (ou `toplevel`), qui s'appelle « principale » ici. Les trois dernières lignes (jusqu'à présent) appellent trois routines différentes. La première (`DefinirVariables`) réglera différentes variables dont nous aurons besoin plus tard. La suivante (`ConstruireWidgets`) sera l'endroit où nous définissons nos widgets, et la dernière (`PlacerWidgets`) est celle où nous allons placer les widgets dans la fenêtre racine. Comme nous l'avons fait la dernière fois, nous allons utiliser le gestionnaire de géométrie « grille ». Notez que `ConstruireWidgets` retournera l'objet « f » (qui est notre fenêtre racine) et que nous le passerons à la routine `PlacerWidgets`.

Voici notre routine `ConstruireWidgets` (ci-contre, en haut à droite). Les lignes qui commencent par « self. »

```
def ConstruireWidgets(self, principale):
    # définition de nos widgets
    fenetre = Frame(principale)
    # labels (ou étiquettes)
    self.fenetreLabels = Frame(fenetre, relief = SUNKEN, padx = 3, pady = 3,
                               borderwidth = 2, width = 500)
    self.lbl1 = Label(self.fenetreLabels, text="Label plat", relief = FLAT,
                     width = 13, borderwidth = 2)
    self.lbl2 = Label(self.fenetreLabels, text="Label creux", relief = SUNKEN,
                     width = 13, borderwidth = 2)
    self.lbl3 = Label(self.fenetreLabels, text="Label arete", relief = RIDGE, width = 13,
                     borderwidth = 2)
    self.lbl4 = Label(self.fenetreLabels, text="Label souleve", relief = RAISED,
                     width = 13, borderwidth = 2)
    self.lbl5 = Label(self.fenetreLabels, text="Label rainure", relief = GROOVE,
                     width = 13, borderwidth = 2)
    return fenetre
```

ont été coupées pour deux raisons. Tout d'abord, c'est une bonne pratique de garder la longueur de la ligne à moins de 80 caractères. Deuxièmement, cela facilite les choses pour notre merveilleux éditeur. Vous avez deux possibilités : soit écrire des lignes longues, soit les garder comme ça. Python nous permet de couper les lignes tant qu'elles sont dans des parenthèses ou des crochets. Comme je l'ai dit précédemment, nous définissons les widgets avant de les placer dans la grille. Quand nous écrirons la routine suivante, vous remarquerez que nous pouvons aussi définir un widget au moment où nous le plaçons dans la grille, mais le définir avant de le

mettre dans la grille dans une routine comme celle-ci facilite les choses, puisque nous faisons (la plupart) des définitions dans cette routine.

Nous définissons donc d'abord notre fenêtre principale. C'est là que nous mettrons le reste de nos widgets. Ensuite, nous définissons une fenêtre fille (de la fenêtre principale), qui contiendra cinq étiquettes, et l'appelons `fenetreLabels`. Nous réglons les différents attributs de la fenêtre ici. Nous réglons le relief à « en creux » (« `SUNKEN` »), un remplissage de 3 pixels à gauche et à droite (`padx`), et de 3 pixels en haut et en bas (`pady`). Nous avons également mis la largeur de bordure à 2 pixels de telle sorte

que son relief en creux soit perceptible. Par défaut, la largeur de bordure vaut 0 et l'effet de creux ne serait pas visible. Enfin, nous avons mis la largeur totale de la fenêtre à 500 pixels.

Ensuite, nous définissons chaque widget étiquette que nous allons utiliser. Nous fixons le parent à `self.fenetreLabels`, et non pas `fenetre`. De cette façon, toutes les étiquettes sont des enfants de `fenetreLabels` et `fenetreLabels` est un enfant de `fenetre`. Remarquez que chaque définition est à peu près semblable pour l'ensemble des cinq étiquettes, sauf le nom du widget (`lbl1`, `lbl2`, etc), le texte et le relief ou l'effet visuel. Enfin, nous retour-

nous la fenêtre à la routine appelante (`_init_`).

Voici notre routine `PlacerWidgets` (page suivante, en haut à droite).

Nous récupérons l'objet fenêtre en tant que paramètre appelé « principale ». Nous l'assignons à « fenetre » simplement pour être cohérent avec ce que nous avons fait dans la routine `ConstruireWidgets`. Ensuite, nous mettons en place la grille principale (`fenetre.grid(column=0, row=0)`). Si nous ne faisons pas cela, rien ne fonctionnera correctement. Ensuite, nous commençons à mettre nos widgets dans les emplacements de la grille. D'abord nous mettons la fenêtre (`fenetreLabels`) qui contient toutes nos étiquettes et définissons ses attributs. Nous la plaçons colonne 0, ligne 1, réglons le remplissage à 5 pixels sur tous les côtés, lui disons de s'étaler sur 5 colonnes (à droite et à gauche), et enfin utilisons l'attribut « sticky » [Ndt : collant] pour forcer la fenêtre à s'étendre complètement à gauche et à droite (« WE » pour Ouest et Est). Maintenant vient la partie qui enfreint la règle dont je vous ai parlé. Nous mettons une étiquette comme premier widget dans la fenêtre, mais nous ne l'avons pas défini à l'avance : nous le définissons

maintenant. Nous avons mis comme parent `fenetreLabels`, tout comme les autres étiquettes. Nous réglons le texte à « 'La-bels |' », la largeur à 15, et l'ancre à Est ('e'). Si vous vous souvenez de la dernière fois, en utilisant l'attribut d'ancrage, nous pouvons choisir où le texte s'affiche dans le widget. Dans ce cas, c'est le long du bord droit. Maintenant la partie amusante. Ici, nous définissons l'emplacement de la grille (et tous les autres attributs de la grille dont nous avons besoin), simplement en ajoutant « `.grid` » à la fin de la définition des étiquettes.

Ensuite, nous plaçons toutes nos autres étiquettes dans la grille, à partir de la colonne 1, ligne 0.

Voici notre routine `DefinirVariables`. Notez que nous utilisons simplement l'instruction `pass` pour l'instant. Nous la remplissons plus tard, car nous n'en avons pas besoin pour cette partie :

```
def DefinirVariables(self):  
    # Definit nos ressources  
    pass
```

Et enfin nous plaçons notre code pour la routine principale :

programmer en python

```
def PlacerWidgets(self, principale):  
    fenetre = principale  
    # place les widgets  
    fenetre.grid(column = 0, row = 0)  
    # place les labels  
    self.fenetreLabels.grid(column = 0, row = 1, padx = 5, pady = 5,  
                             columnspan = 5, sticky='WE')  
    l = Label(self.fenetreLabels, text='Labels |', width=15,  
              anchor='e').grid(column=0, row=0)  
    self.lb11.grid(column = 1, row = 0, padx = 3, pady = 5)  
    self.lb12.grid(column = 2, row = 0, padx = 3, pady = 5)  
    self.lb13.grid(column = 3, row = 0, padx = 3, pady = 5)  
    self.lb14.grid(column = 4, row = 0, padx = 3, pady = 5)  
    self.lb15.grid(column = 5, row = 0, padx = 3, pady = 5)
```

```
root = Tk()  
root.geometry('750x40+150+150')  
root.title("Widget Demo 1")  
demo = Demo(root)  
root.mainloop()
```

D'abord, on initialise une instance de Tk. Puis nous définissons la taille de la fenêtre principale à 750 pixels de large sur 40 pixels de haut et la localisons à 150 pixels de la gauche et du haut de l'écran. Puis nous réglons le titre de la fenêtre etinstancions

notre objet `Demo` et, enfin, appelons la boucle principale de Tk.

Essayez. Vous devriez voir les cinq étiquettes ainsi que l'étiquette de « dernière minute » avec divers effets magnifiques.

## Les boutons

Maintenant, enregistrez ce que vous avez en tant que `widgetde-mo1a.py`

```
# place les boutons  
self.fenetreBoutons.grid(column=0, row = 2, padx = 5,  
                           pady = 5, columnspan = 5, sticky = 'WE')  
l = Label(self.fenetreBoutons, text='Boutons |', width=15,  
          anchor='e').grid(column=0, row=0)  
self.btn1.grid(column = 1, row = 0, padx = 3, pady = 3)  
self.btn2.grid(column = 2, row = 0, padx = 3, pady = 3)  
self.btn3.grid(column = 3, row = 0, padx = 3, pady = 3)  
self.btn4.grid(column = 4, row = 0, padx = 3, pady = 3)  
self.btn5.grid(column = 5, row = 0, padx = 3, pady = 3)
```

et nous allons ajouter quelques boutons. Puisque nous avons construit notre programme de base ainsi, nous allons simplement pouvoir y ajouter les parties qui manquent. Commençons par la routine `ConstruireWidgets`. Après les définitions des étiquettes, et avant le « `return fenetre` », ajoutez ce qui se trouve en haut de la page suivante.

Rien de bien nouveau ici. Nous avons défini les boutons avec leurs attributs et avons fixé leurs fonctions de rappel avec un « `.bind` ». Notez que nous utilisons `lambda` pour envoyer les valeurs 1 à 5 suivant le bouton sur lequel on clique. Dans la fonction de rappel, nous allons utiliser cela afin de savoir quel bouton on doit gérer. Maintenant, nous allons travailler dans la routine `PlacerWidgets`. Placez le code (page précédente, en bas à droite) juste après l'emplacement de la dernière étiquette.

Une fois de plus, rien de vraiment nouveau ici, donc nous allons continuer. Voici notre routine de rappel (ci-contre, en bas à droite). Placez-la après la routine `DefinirVariables`.

Encore une fois, rien de vraiment sensationnel ici. Nous utilisons simplement une série de routines IF/ELIF pour afficher quel bouton a été

cliqué. La principale chose à regarder ici (lorsque nous exécutons le programme) est que le bouton « en creux » ne bouge pas lorsqu'on clique dessus. En général on n'utilise pas le relief « en creux », sauf si vous souhaitez un bouton qui reste enfoncé lorsque vous cliquez dessus. Enfin, nous avons besoin d'ajuster la déclaration de la géométrie à cause des widgets supplémentaires que nous avons ajoutés :

```
root.geometry('750x110+150+150')
```

Ok. C'est terminé pour celui-ci. Enregistrez-le et lancez-le.

Maintenant sauvegardez ceci comme `widgetdemo1b.py` et nous allons passer aux cases à cocher.

## Les cases à cocher

Comme je l'ai dit précédemment, cette partie de la démo a un bouton normal et deux cases à cocher. L'apparence de la première case est celle,

```
# boutons
self.fenetreBoutons = Frame(fenetre, relief = SUNKEN, padx = 3,
                             pady = 3, borderwidth = 2, width = 500)
self.btn1 = Button(self.fenetreBoutons, text="Bouton plat",
                   relief = FLAT, borderwidth = 2)
self.btn2 = Button(self.fenetreBoutons, text="Bouton creux",
                   relief = SUNKEN, borderwidth = 2)
self.btn3 = Button(self.fenetreBoutons, text="Bouton arete",
                   relief = RIDGE, borderwidth = 2)
self.btn4 = Button(self.fenetreBoutons, text="Bouton souleve",
                   relief = RAISED, borderwidth = 2)
self.btn5 = Button(self.fenetreBoutons, text="Bouton rainure",
                   relief = GROOVE, borderwidth = 2)
self.btn1.bind('<ButtonRelease-1>', lambda e: self.clicBouton(1))
self.btn2.bind('<ButtonRelease-1>', lambda e: self.clicBouton(2))
self.btn3.bind('<ButtonRelease-1>', lambda e: self.clicBouton(3))
self.btn4.bind('<ButtonRelease-1>', lambda e: self.clicBouton(4))
self.btn5.bind('<ButtonRelease-1>', lambda e: self.clicBouton(5))
```

normale, à laquelle vous pouvez vous attendre. La seconde est plus comme un « bouton collant » - quand elle n'est pas sélectionnée (ou cochée), elle ressemble à un bouton normal. Lorsque vous la sélectionnez, elle ressemble à un bouton qui reste

enfoncé. Nous pouvons faire cela simplement en définissant l'attribut `indicatoron` à `False`. Le bouton « normal » permet de basculer les cases de « cochées » à « décochées » et vice-versa, à chaque fois que l'on clique dessus. Nous arrivons à pro-

```
def clicBouton(self, val):
    if val == 1:
        print("Clic bouton plat...")
    elif val == 2:
        print("Clic bouton creux...")
    elif val == 3:
        print("Clic bouton arete...")
    elif val == 4:
        print("Clic bouton souleve...")
    elif val == 5:
        print("Clic bouton rainure...")
```

grammer cela en appelant la méthode `.toggle` liée à la case à cocher. Nous relierons l'événement clic gauche de la souris (lorsque le bouton est relâché) à une fonction afin de pouvoir envoyer un message (dans notre cas) au terminal. En plus de tout cela, nous mettons en place deux variables (une pour chacune des cases à cocher) que l'on peut interroger à tout moment. Ici, nous interrogeons ces valeurs et les affichons à chaque fois qu'une case est cliquée. Faites attention à la partie variable du code : elle est utilisée dans de nombreux widgets. Dans la routine `ConstruireWidgets`, après le code des boutons que nous venons d'ajouter et avant l'instruction de retour, placez le code (ci-contre, en haut à droite).

Encore une fois, vous avez vu tout cela avant. Nous créons la fenêtre pour contenir nos widgets. Nous créons un bouton et deux cases à cocher. Plaçons-les maintenant (ci-contre, au milieu à droite).

Maintenant, nous définissons les deux variables que nous allons utiliser pour surveiller la valeur de chaque case à cocher. Sous `DefinirVariables`, commentez l'instruction `pass` et ajoutez ceci :

```
self.Chk1Val =
IntVar()
self.Chk2Val =
IntVar()
```

Après la fonction de rappel des boutons, placez ce qui suit (ci-contre, en bas à droite).

Et enfin remplacez l'instruction de géométrie par ceci :

```
root.geometry('75
0x170+150+150')
```

Enregistrez et exécutez. Enregistrez-le comme `wid-getdemo1c.py` et continuons avec les boutons radio.

```
def btnInverser(self,p1):
self.chk1.toggle()
self.chk2.toggle()
print("Valeur de la case à cocher 1 : {0}".format(self.Chk1Val.get()))
print("Valeur de la case à cocher 2 : {0}".format(self.Chk2Val.get()))
```

```
# checkbox (ou cases a cocher)
self.fenetreCases = Frame(fenetre, relief = SUNKEN, padx = 3, pady = 3,
borderwidth = 2, width = 500)

self.chk1 = Checkbutton(self.fenetreCases, text = "Case a cocher normale",
variable=self.Chk1Val)

self.chk2 = Checkbutton(self.fenetreCases, text = "Case a cocher",
variable=self.Chk2Val,indicatoron = False)

self.chk1.bind('<ButtonRelease-1>',lambda e: self.clicCases(1))
self.chk2.bind('<ButtonRelease-1>',lambda e: self.clicCases(2))
self.btnInverserCases = Button(self.fenetreCases,text="Inverser cases")
self.btnInverserCases.bind('<ButtonRelease-1>',self.btnInverser)
```

```
# place les cases à cocher et le bouton d'inversion
self.fenetreCases.grid(column = 0, row = 3, padx = 5, pady = 5,
columnspan = 5,sticky = 'WE')

l = Label(self.fenetreCases,text='Cases à cocher |',width=15,
anchor='e').grid(column=0,row=0)
self.btnInverserCases.grid(column = 1, row = 0, padx = 3, pady = 3)

self.chk1.grid(column = 2, row = 0, padx = 3, pady = 3)
self.chk2.grid(column = 3, row = 0, padx = 3, pady = 3)
```

## Les boutons radio

Si vous êtes assez vieux pour vous souvenir des autoradios avec boutons poussoirs pour sélectionner les stations pré-réglées, vous compren-

rez pourquoi on appelle cela des boutons radio. Lorsque vous utilisez des boutons radio, l'attribut variable est très important. C'est ce qui regroupe les boutons radio ensemble. Dans cette démo, le premier groupe de boutons est formé avec la va-

riable nommée self.RBVal. Le second groupe est formé par la variable self.RBVal2. Nous devons également définir l'attribut « value » au moment de la conception, afin de garantir que les boutons retourneront une valeur qui a du sens quand ils seront

cliqués.

Retournez dans ConstruireWidgets, et ajoutez le code (ci-dessous), juste avant l'instruction de retour.

```
# boutons radio
self.fenetreBoutonsRadio = Frame(fenetre, relief = SUNKEN, padx = 3, pady = 3, borderwidth = 2, width = 500)
self.rb1 = Radiobutton(self.fenetreBoutonsRadio, text = "Radio 1", variable = self.RBVal, value = 1)
self.rb2 = Radiobutton(self.fenetreBoutonsRadio, text = "Radio 2", variable = self.RBVal, value = 2)
self.rb3 = Radiobutton(self.fenetreBoutonsRadio, text = "Radio 3", variable = self.RBVal, value = 3)
self.rb1.bind('<ButtonRelease-1>', lambda e: self.clicBoutonRadio())
self.rb2.bind('<ButtonRelease-1>', lambda e: self.clicBoutonRadio())
self.rb3.bind('<ButtonRelease-1>', lambda e: self.clicBoutonRadio())
self.rb4 = Radiobutton(self.fenetreBoutonsRadio, text = "Radio 4", variable = self.RBVal2, value = "1-1")
self.rb5 = Radiobutton(self.fenetreBoutonsRadio, text = "Radio 5", variable = self.RBVal2, value = "1-2")
self.rb6 = Radiobutton(self.fenetreBoutonsRadio, text = "Radio 6", variable = self.RBVal2, value = "1-3")
self.rb4.bind('<ButtonRelease-1>', lambda e: self.clicBoutonRadio2())
self.rb5.bind('<ButtonRelease-1>', lambda e: self.clicBoutonRadio2())
self.rb6.bind('<ButtonRelease-1>', lambda e: self.clicBoutonRadio2())
```

Ajoutez ceci dans PlacerWidgets :

```
# place les boutons radio et selectionne le premier
self.fenetreBoutonsRadio.grid(column = 0, row = 4, padx = 5, pady = 5, columnspan = 5, sticky = 'WE')
l = Label(self.fenetreBoutonsRadio,
          text='Boutons radio |',
          width=15, anchor='e').grid(column=0, row=0)
self.rb1.grid(column = 2, row = 0, padx = 3, pady = 3, sticky = 'EW')
self.rb2.grid(column = 3, row = 0, padx = 3, pady = 3, sticky = 'WE')
self.rb3.grid(column = 4, row = 0, padx = 3, pady = 3, sticky = 'WE')
self.RBVal.set("1")
l = Label(self.fenetreBoutonsRadio, text='| Un autre groupe |',
          width = 15,
          anchor = 'e').grid(column = 5, row = 0)
self.rb4.grid(column = 6, row = 0)
self.rb5.grid(column = 7, row = 0)
self.rb6.grid(column = 8, row = 0)
self.RBVal2.set("1-1")
```

Une chose à noter ici. Remarquez les définitions de « dernière minute » pour les étiquettes dans la routine `PlacerWidgets`. Ces lignes longues sont coupées pour montrer comment utiliser les parenthèses pour permettre à nos longues lignes d'être formatées correctement dans notre code, et de fonctionner toujours correctement.

Dans `DefinirVariables`, ajoutez :

```
self.RBVal = IntVar()
```

Ajoutez les routines de clics :

```
def clicBoutonRadio(self):  
    print("Clic bouton  
radio - Valeur :  
{0}".format(self.RBVal.get()))
```

```
def clicBoutonRadio2  
(self):  
    print("Clic bouton  
radio - Valeur :  
{0}".format(self.RBVal2.get()))
```

et enfin modifiez à nouveau la géométrie comme ceci :

```
root.geometry('750x220+150+150')
```

Enregistrez le projet sous `widgetdemo1d.py` et exécutez-le. Maintenant, nous allons travailler sur les champs de texte standard (ou widgets de saisie).

## Les champs de texte

Encore une fois, nous avons déjà utilisé des champs de texte (ou widgets de saisie) dans diverses interfaces graphiques auparavant. Mais cette fois-ci, comme je l'ai dit précédemment, nous allons montrer comment empêcher l'utilisateur de faire des changements dans le champ de texte en le désactivant. Cela s'avère utile si vous affichez certaines données et permettez à l'utilisateur de les modifier seulement quand il est dans un mode d'édition. Maintenant vous devriez savoir que la première chose que nous devons faire est d'ajouter du code à la routine `ConstruireWidgets` (ci-contre, à droite).

Comme d'habitude, nous créons notre fenêtre. Puis nous créons notre

```
# champs de texte  
self.fenetreChampsTexte = Frame(fenetre, relief = SUNKEN, padx  
= 3, pady = 3, borderwidth = 2, width = 500)  
self.txt1 = Entry(self.fenetreChampsTexte, width = 10)  
self.txt2 = Entry(self.fenetreChampsTexte,  
disabledbackground="#cccccc", width = 10)  
self.btnDesactiver = Button(self.fenetreChampsTexte, text =  
"Activer/Desactiver")  
self.btnDesactiver.bind('<ButtonRelease-1>',  
self.clicBoutonDesactiver)
```

Ensuite, ajoutez ces lignes de code à la routine `PlacerWidgets` :

```
# place les champs de texte  
self.fenetreChampsTexte.grid(column = 0, row = 5, padx = 5,  
pady = 5, columnspan = 5, sticky = 'WE')  
l = Label(self.fenetreChampsTexte, text='Champs de texte  
' ,width=15, anchor='e').grid(column=0,row=0)  
self.txt1.grid(column = 2, row = 0, padx = 3, pady = 3)  
self.txt2.grid(column = 3, row = 0, padx = 3, pady = 3)  
self.btnDesactiver.grid(column = 1, row = 0, padx = 3, pady = 3)
```

Ajoutez cette ligne en bas de la routine `DefinirVariables` :

```
self.Disabled = False
```

Maintenant ajoutez la fonction qui répond au clic sur le bouton :

```
def clicBoutonDesactiver(self,p1):  
    if self.Disabled == False:  
        self.Disabled = True  
        self.txt2.configure(state='disabled')  
    else:  
        self.Disabled = False  
        self.txt2.configure(state='normal')
```

Enfin, relancez la ligne de géométrie ::

```
root.geometry('750x270+150+150')
```

Sauvegardez-le sous le nom `widgetdemo1d.py` et exécutez-le.

barre de défilement verticale. Nous faisons cela avant de créer la liste, parce que nous devons faire référence à la méthode « .set » de la barre de défilement. Remarquez l'attribut « height = 5 ». Cela force la liste à montrer 5 éléments à la fois. Dans la déclaration .bind, nous utilisons « "ListboxSelect" » comme événement. C'est ce qu'on appelle un événement virtuel, puisque ce n'est pas vraiment un événement « officiel ».

Maintenant, nous allons nous occuper du code supplémentaire dans la routine PlacerWidgets (page suivante, encadré de gauche).

## Les boîtes de dialogue

Cette section est tout simplement une série de boutons « normaux » qui appellent les différents types de boîtes de dialogue. Nous les avons déjà rencontrés avec une boîte à outils différente. Nous allons explorer seulement 5 types différents, mais il y en a plus. Dans cette section, nous allons regarder Information, Avertissement, Erreur, Question, et les dialogues Oui/Non. Ils sont très utiles lorsque vous avez besoin de faire passer des informations à votre utilisateur d'une manière assez importante. Dans la routine ConstruireWidgets,

ajoutez (ci-contre, en bas à droite).

Voici la routine d'appui (ci-contre, en haut à droite). Pour les trois premiers (Info, Avertissement et Erreur), il suffit d'appeler « tkMessageBox.show-info », ou celui dont vous avez besoin, avec deux paramètres. Le premier est le titre de la boîte de message et le second est le message réel que vous voulez montrer. L'icône est gérée pour vous par Tkinter. Pour les dialogues qui fournissent une réponse (Question, Oui/Non), nous fournissons une variable qui reçoit la valeur correspondant au bouton cliqué. Dans le cas de la boîte de dialogue Question, la réponse est « Oui » ou « Non », et dans le cas du dialogue Oui/Non, la réponse est « True »

```
# champs de texte
self.fenetreChampsTexte = Frame(fenetre, relief = SUNKEN, padx = 3, pady = 3, borderwidth = 2, width = 500)
self.txt1 = Entry(self.fenetreChampsTexte, width = 10)
self.txt2 = Entry(self.fenetreChampsTexte, disabledbackground="#cccccc", width = 10)
self.btnDesactiver = Button(self.fenetreChampsTexte, text = "Activer/Desactiver")
self.btnDesactiver.bind('<ButtonRelease-1>', self.clicBoutonDesactiver)
```

Ensuite, ajoutez ces lignes de code à la routine PlacerWidgets :

```
# place les champs de texte
self.fenetreChampsTexte.grid(column = 0, row = 5, padx = 5, pady = 5, columnspan = 5, sticky = 'WE')
l = Label(self.fenetreChampsTexte, text='Champs de texte |', width=15, anchor='e').grid(column=0, row=0)
self.txt1.grid(column = 2, row = 0, padx = 3, pady = 3)
self.txt2.grid(column = 3, row = 0, padx = 3, pady = 3)
self.btnDesactiver.grid(column = 1, row = 0, padx = 3, pady = 3)
```

Ajoutez cette ligne en bas de la routine DéfinirVariables :

```
self.Disabled = False
```

Maintenant ajoutez la fonction qui répond au clic sur le bouton :

```
def clicBoutonDesactiver(self, p1):
    if self.Disabled == False:
        self.Disabled = True
        self.txt2.configure(state='disabled')
    else:
        self.Disabled = False
        self.txt2.configure(state='normal')
```

Enfin, relancez la ligne de géométrie ::

```
root.geometry('750x270+150+150')
```

Sauvegardez-le sous le nom widgetdemo1d.py et exécutez-le.

```
# place la liste et les boutons associes
self.fenetreListe.grid(column = 0, row = 6, padx = 5,
pady = 5, columnspan = 5, sticky = 'WE')
l = Label(self.fenetreListe, text='Liste |', width=15,
anchor='e').grid(column=0, row=0, rowspan=2)
self.liste.grid(column = 2, row = 0, rowspan=2)
self.defilementV.grid(column = 3, row = 0, rowspan =
2, sticky = 'NSW')
self.btnEffacerListe.grid(column = 1, row = 0, padx = 5)
self.btnRemplirListe.grid(column = 1, row = 1, padx = 5)
```

Ajoutez ceci dans DéfinirVariables :

```
# les elements pour notre liste
self.exemples = ['Element un', 'Element deux', 'Element
trois', 'Element quatre']
```

Et ajoutez les routines de support suivantes :

```
def effacerListe(self):
    self.liste.delete(0,END)

def remplirListe(self):
    # Note : effacer d'abord la liste ; aucune
    verification n'est faite
    for ex in self.exemples:
        self.liste.insert(END,ex)
    # insert([0,ACTIVE,END],element)

def listeSelection(self,pl):
    print("Clic sur un élément de la liste")
    items = self.liste.curselection()
    selitem = items[0]
    print("Index de l'élément choisi :
{0}".format(selitem))
    print("Texte de l'élément choisi :
{0}".format(self.liste.get(selitem)))
```

Enfin, mettez à jour la ligne de géométrie :

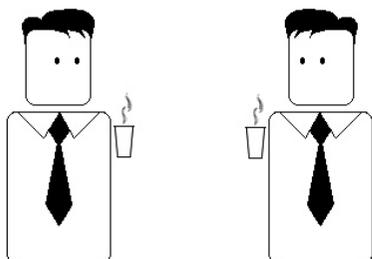
```
root.geometry('750x370+150+150')
```

Sauvegarder cela comme widgetdemo1e.py et exécutez-le. Maintenant, nous allons faire les dernières modifications à notre application.

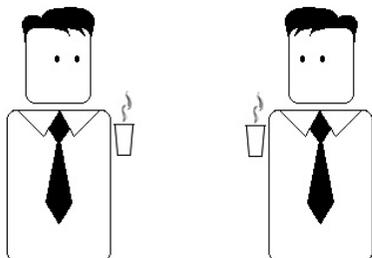
```
# des choses pour la liste
self.fenetreListe = Frame(fenetre,
relief = SUNKEN,
padx = 3,
pady = 3,
borderwidth = 2,
width = 500
)
# boite avec barre de défilement pour la
liste
self.defilementV =
Scrollbar(self.fenetreListe)
self.liste = Listbox(self.fenetreListe,
height = 5,
yscrollcommand =
self.defilementV.set)
# hauteur par défaut = 10

self.liste.bind('<<ListboxSelect>>',self.listeSelec
tion)
self.defilementV.config(command =
self.liste.yview)
self.btnEffacerListe = Button(
self.fenetreListe,
text = "Effacer liste",
command = self.effacerListe,
width = 11
)
self.btnRemplirListe = Button(
self.fenetreListe,
text = "Remplir liste",
command = self.remplirListe,
width = 11
)
# <<ListboxSelect>> est un evenement virtuel
# remplit la liste
self.remplirListe()
```

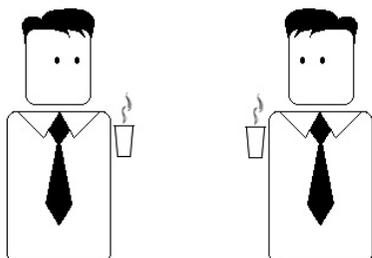
Je sais que c'est le contraire de tout ce que je représente. Je me détesterais si je le fais.



Mais j'en ai besoin.



J'ai besoin d'un iPod Touch.



by Richard Redei

ou « False ».

Enfin, modifiez la ligne de géométrie :

```
root.geometry('750x490+550+150')
```

Sauvegardez ceci sous le nom wid-

getdemo1f.py et amusez-vous avec.

J'ai placé le code de widgetdemo1f.py sur pastebin ici :

<http://pastebin.com/TQgppVnF>

C'est tout pour cette fois. J'espère que ceci vous aura inspiré à explorer toutes les fonctionnalités proposées par tkinter. À la prochaine fois.

```
def afficheFenetreMessage(self,which):
    if which == 1:
        tkMessageBox.showinfo('Demo','Voici un message INFO')
    elif which == 2:
        tkMessageBox.showwarning('Demo','Voici un message WARNING (avertissement)')
    elif which == 3:
        tkMessageBox.showerror('Demo','Voici un message ERREUR')
    elif which == 4:
        rep = tkMessageBox.askquestion('Demo','Voici une QUESTION ?')
        print('clic sur {0}...'.format(rep))
    elif which == 5:
        rep = tkMessageBox.askyesno('Demo','Voici un message OUI/NON')
        print('clic sur {0}...'.format(rep))
```

```
# boutons pour afficher les fenetres de messages et de dialogues
self.fenetreMessages = Frame(fenetre,relief = SUNKEN,padx = 3, pady = 3, borderwidth = 2)
self.btnMBInfo = Button(self.fenetreMessages,text = "Info")
self.btnMBWarning = Button(self.fenetreMessages,text = "Avertissement")
self.btnMBError = Button(self.fenetreMessages,text = "Erreur")
self.btnMBQuestion = Button(self.fenetreMessages,text = "Question")
self.btnMBYesNo = Button(self.fenetreMessages,text = "Oui/Non")
self.btnMBInfo.bind('<ButtonRelease-1>', lambda e: self.afficheFenetreMessage(1))
self.btnMBWarning.bind('<ButtonRelease-1>', lambda e: self.afficheFenetreMessage(2))
self.btnMBError.bind('<ButtonRelease-1>', lambda e: self.afficheFenetreMessage(3))
self.btnMBQuestion.bind('<ButtonRelease-1>', lambda e: self.afficheFenetreMessage(4))
self.btnMBYesNo.bind('<ButtonRelease-1>', lambda e: self.afficheFenetreMessage(5))
```

Maintenant ajoutez le code dans la routine PlacerWidgets :

```
# boutons de boîtes de messages et de dialogues
self.fenetreMessages.grid(column = 0,row = 7, columnspan = 5, padx = 5, sticky = 'WE')
l = Label(self.fenetreMessages,text='Messages |',width=15, anchor='e').grid(column=0,row=0)
self.btnMBInfo.grid(column = 1, row = 0, padx= 3)
self.btnMBWarning.grid(column = 2, row = 0, padx= 3)
self.btnMBError.grid(column = 3, row = 0, padx= 3)
self.btnMBQuestion.grid(column = 4, row = 0, padx= 3)
self.btnMBYesNo.grid(column = 5, row = 0, padx= 3)
```