tutoriel

Créer un GUI avec Tkinter

Introduction

Les 4 tâches de base en programmation d'interfaces GUI

Pour développer une interface utilisateur (GUI), il faut écrire :

le code qui détermine ce que l'utilisateur voit sur son écran

le code qui accomplit les tâches de ce programme

le code qui associe l'apparence (ce que l'utilisateur voit) avec l'action (les routines qui exécutent les tâches du programme.

le code qui attend les entrées de l'utilisateur.

Glossaire de la programmation d'un Gui

La programmation avec un GUI (interface utilisateur graphique) a un jargon spécifique

widgets
: élément de l'interface graphique : fenêtres, boutons, menus, éléments de menus, icônes, listes déroulantes, barres de défilement etc.
Définis dans l'espace (c'est-à-dire, si un widget est au-dessus ou en dessous d'un autre, ou à droite ou à gauche d'autres widgets)

callback handlers
event handlers
: les routines qui font le travail du GUI. Ces routines sont appelées des *handlers* parce qu'elles répondent à ces événements.

Events
: événements comme un clic de souris ou l'appui sur un touche du clavier.

binding
: le fait d'associer un *event handler* avec un *widget*. En gros, le *binding* implique :

un type d'événement (par exemple un clic gauche sur la souris ou appuyer sur la touche ENTER du clavier),

un widget (c'est-à-dire un bouton)

et une routine de gestion d'événement.

Par exemple, on peut associer

un (seul) clic gauche de la souris

au bouton/widget "CLOSE" sur l'écran

à la routine *closeProgram*, qui ferme la fenêtre et arrête le programme.

event loop

le code qui tourne et attend une saisie

L'**event loop** passe tout son temps à regarder ce qui se passe.

La plupart des événements sont sans intérêt et il ne fait rien quand il les voit.

Mais s'il voit un événement rattaché à un **event handler**, il informe immédiatement l'event handler de ce qui s'est passé.

Pré-requis

Première étape

Premier exemple : tt000.py

Ce premier programme très simple montre l'implémentation de trois concepts de base dans un programme.

Il n'utilise pas Tkinter ni une forme de programmation de GUI.

Il se contente d'afficher un menu sur la console et fait une saisie au clavier.

Il exécute les quatre tâches de base de la programmation d'un GUI.

tt000.py

```
#----- tâche 2:  définir les routines de l'event handler----------
-----------
def handle_A():
    print "Faux ! Essayez encore !"

def handle_B():
    print "Tout à fait exact !  Trillium est une variété de fleur
!"

def handle_C():
    print "Faux ! Essayez encore !"

# ------------ tâche 1: définir l'apparence de l'écran -----------
-
print "\n"*100   # nettoyer l'écran
```

```
print "            Jeu de devinette très difficile"
print "============================================================"
print "Tapez la lettre de votre réponse, puis appuyez sur la
touche ENTREE."
print
print "    A.  Animal"

print "    B.  Légume"
print "    C.  Minéral"
print
print "    X.  Quitter ce programme"
print
print "============================================================"
print "Dans quelle catégorie est 'Trillium'?"

print

# ---- tâche 4: l'event loop.  Nous bouclons en permanence, en
scrutant les évènements. ---
while 1:

    # Nous attendons le prochain évènement.
    answer = raw_input().upper()

    # ------------------------------------------------------
    # Tâche 3: Associer des évènements claviers intéressants avec
leurs
    # event handlers.  Un forme simple d'association.
    # ------------------------------------------------------
    if answer == "A": handle_A()
    if answer == "B": handle_B()
    if answer == "C": handle_C()
    if answer == "X":
        # nettoyer l'écran et sortir de l'event loop
        print "\n"*100
        break

    # Notez que les autres évènements ne sont pas intéressants et
sont donc ignorés.
```

Deuxième exemple (tt010.py) : le plus court programme Tkinter (3 Instructions !)

Ce programme n'exécute qu'une des quatre tâches de base d'un GUI présentées plus haut, l'event loop.

La première instruction importe Tkinter, pour le rendre disponible.

la syntaxe *from Tkinter import* * évite l'obligation de préfixer

ce qui vient de Tkinter par un préfixe “Tkinter.”.

La deuxième instruction crée une fenêtre “toplevel”, une instance de la classe “Tkinter.Tk”.
La fenêtre d'avant-plan est le composant de plus haut niveau du GUI d'une application Tkinter.
Par convention, on nomme cette fenêtre “root”.

La troisième instruction est la boucle principale (mainloop) (c'est-à-dire, l'event loop), qui est une méthode de l'objet “root”.
La boucle principale attend qu'un événement se produise dans la fenêtre root.
Un événement qui arrive est traité et la boucle continue, en attente de l’événement suivant.
La boucle continue à s'exécuter jusqu'à ce qu'un événement “destroy” arrive dans le fenêtre d'avant-plan, ce qui ferme la fenêtre et sort de l'event loop.

Pendant l'exécution de ce programme, des widgets de la fenêtre d'avant-plan permettent de réduire ou augmenter la taille de la fenêtre, ou la fermer.

Un clic sur le widget “close” (le “x” de la barre de titre) déclenche un évènement “destroy” qui termine la boucle principale d'évènement ; comme il n'y a pas d'instructions après “root.mainloop()”, le programme se termine.

```
from Tkinter import *

root = Tk()
root.mainloop()
```

Troisième exemple (tt020.py) : indiquer à quoi le GUI doit ressembler

Avec ce programme, nous allons introduire trois concepts importants de la programmation Tkinter :

créer un objet GUI et l'associer avec ses parents
le **packing**
les **containers** par opposition aux **widgets**

Glossaire :

widget
: un composant du GUI qui est (en général) visible

container component
: conteneur contenant des widgets

Tkinter propose de nombreux conteneurs.

- **Canvas** est un container pour des applications de dessin.
- Le conteneur **Frame** (cadre) est le plus utilisé.

Les Frames sont fournis par Tkinter dans une classe appelée **Frame**. Une expression comme :

```
Frame(myParent)
```

crée une instance de la classe **Frame** et l'associe avec son parent, myParent. En d'autres termes, cette expression ajoute un frame enfant au composant myParent.

Dans ce programme, l'instruction (1)

```
myContainer1 = Frame(myParent)
```

crée un conteneur nommé **myContainer1**, de type Frame, dont le parent est myParent (c'est-à-dire, root) et dans lequel on peut placer des widgets. (Ici, nous ne plaçons pas de widget, nous le ferons plus loin).

<note>La relation parent/enfant est ici une relation LOGIQUE, pas une relation visuelle. Cette relation existe pour supporter des choses comme l'événement destroy – afin que, quand un composant parent (comme root) est détruit, le parent détruit ses enfants avant de se détruire lui-même.</note>

L'instruction suivante (2) **packe** (remplit) myContainer1 :

```
myContainer1.pack()
```

L'instruction "pack" rend visible un composant du GUI en mettant en place une relation visuelle entre ce composant et son parent. Il faut faire un pack (insertion) d'un composant, sinon il reste invisible.

Pack appelle le **geometry manager pack** de Tkinter.

geometry manager
: une API – une façon de communiquer avec Tkinter – pour indiquer à Tkinter comment les containers et widgets doivent être visuellement présentés.
Tkinter supporte trois geometry managers : pack, grid, et place.
Pack et grid sont les plus utilisés, parce qu'ils sont les plus faciles à utiliser.
Tous les exemples dans "Penser en Tkinter" utilisent le pack geometry manager.

Nous avons donc ici un motif de base pour la programmation Tkinter que nous retrouverons souvent.

une instance (d'un widget ou d'un container) est créée, et associée avec son parent
l'instance est packée (insérée).

L'exécution ce programme ressemble beaucoup au précédent, à la différence qu'il affiche moins de choses. C'est parce que les Frames (cadres) Sont élastiques.

frame
: = un cadre. C'est un container.

cavity
: espace intérieur du container

stretchy
: élastique : La *cavity* (creux) est élastique. A moins d'indiquer un taille minimum ou maximum pour le frame (cadre), la *cavity* s'étire ou se réduit pour s'adapter au frame (cadre).

Dans le programme précédent, le *root* s'est affiché avec la taille par défaut car nous n'avions rien

précisé.

Mais dans ce programme, nous avons placé quelque chose dans la cavity de root (nous y avons placé Container1). Le cadre root se réduit donc pour s'adapter à la taille de Container1. Comme nous n'avons pas mis de widget dans Container1, ni indiqué une taille minimum pour Container1, la cavity de root se réduit à rien. Voilà pourquoi il n'y a rien à voir en dessous de la title bar.

Dans les programmes suivants, nous placerons des widgets et autres containers dans Container1 ; nous verrons comment Container1 s'étire pour s'adapter.

tt020.py

```
from Tkinter import *

root = Tk()

myContainer1 = Frame(root) # (1)
myContainer1.pack() # (2)

root.mainloop()
```

Internationalisation : traduire un programme Python

Voir Internationalisation : traduire un programme Python

Autres étapes

Conclusion

Problèmes connus

Voir aussi

- **(fr)** http://

Basé sur « Article » par Auteur.