Table des matières

fiche 1 : Python, l'interface graphique tkinter

1. Une interface graphique.

1.1. de TK à tkinter.

Tk est une bibliothèque d'interfaces graphiques multi plate-forme et extensible. Conçu à l'origine pour le langage de script Tcl, il s'interface avec Python. Le module s'appelle **`tkinter`**.

Python est un langage de script qu'on exécute en principe dans une console (console DOS sous Windows, console Linux …) en mode texte. On peut lui adjoindre des modules qui peuvent réaliser une interface graphique (c'est aissi le cas de Java par exemple, avec les librairies **`awt`** et **`swing`**).

Cette interface graphique n'appartient pas au langage ; il existe d'ailleurs sous Python d'autres interfaces graphiques mais tkinter a été la première, et elle reste le module de référence pour les réalisations d'applications avec une interface graphique. Le module **`tkinker`** appartient au standard de Python, dans la mesure où il est fourni avec l'interpréteur, et que la documentation de Python documente **`tkinter`** comme si c'était un module intégré.

1.2. Les widgets.

Le mot **widget** est une contraction de "window gadget", qu'on peut traduire par "accessoire pour applications fenêtrées". Ces accessoires se retrouvent dans toutes les applications en interface graphique : applications HTML , Java, Delphi, Javascript etc.

Tk propose nativement une collection de composants d'interface graphique (widgets) :

```
* button (bouton poussoir)
* checkbutton (case à cocher)
* radiobutton (bouton radio)
* label (étiquette)
* entry (champ de texte en entrée)
* listbox (liste défilante)
* tk_optionMenu (liste)
* menu (menu déroulant)
* menubutton (menu déroulant à partir d'un bouton)
* scale (curseur horizontal et vertical)
* spinbox (zone de sélection numérique)
* frame (cadre)
* labelframe (cadre avec titre)
* scrollbar (barre de défilement)
* panedwindow (panneau coulissant)
* text (conteneur hypertexte évolué)
* canvas (conteneur d'objets graphiques 2D évolué)
* tk_chooseColor (sélecteur de couleur)
* tk_chooseDirectory (sélecteur de répertoire)
* tk_dialog (boîte de dialogue modale)
* tk_getOpenFile (sélecteur de fichier)
* tk_messageBox (boîte de message)
* tk_popup (menu contextuel)
```

Chaque widget possède des propriétés modifiables selon le type (taille, relief, couleur, contenu, état, événement).

1.3. La géométrie des fenêtres.

Pour contrôler la dimension et agencer graphiquement les widgets, il existe trois gestionnaires de géométrie :

* **`grid`** (dispose les widgets selon une grille)
* **`pack`** (empile ou dispose côte-à-côte les widgets selon un ordre relatif)
* **`place`** (dispose les widgets de manière absolue)

Tant qu'un widget n'est pas associé à un gestionnaire de géométrie, il n'apparaît pas à l'écran.

Les gestionnaires de géométrie (**layers** en Java) sont incompatibles entre eux : dans une fenêtre ou un cadre, on ne peut utiliser qu'un seul type de gestionnaire.

1.4. Gestion des événements.

À la différence d'un programme en ligne de commande où l'interaction avec l'utilisateur est séquentielle, **l'interface graphique fait intervenir la notion de programmation événementielle** C'est une autre logique ! À tout moment, chaque widget est susceptible d'être affecté par l'action de l'utilisateur (l'événement). Il existe des événements simples (clic de souris sur un bouton, saisie au clavier dans un champ) et des événements plus complexes (navigation dans un menu ou une liste déroulante).

À chaque widget est attaché par défaut un certain nombre de réponses automatiques à des événements. Celles-ci correspondent à une gestion des événements de bas niveau où le programmeur n'a que très peu à intervenir. Une boucle événementielle (la méthode mainloop()) les prend en charge et les répartit. Dans d'autres langages (Java, HTML), la boucle de capture des événements est lancée par défaut et ne s'arrête que par fermeture de la fenêtre d'accueil. En Python la méthode **`quit()`** fait sortir de la boucle initiée par **`mainloop()`** et continue le programme en séquence.

Par l'intermédiaire de l'option **`command`** d'un widget, on peut lier le widget à un appel de fonction, de méthode ou de commande extérieure (callback).

2. Un message et un bouton.

```
# ref. Swinnen, Apprendre à programmer avec Python, page 81
#
# un label et un bouton dans une fenêtre

from tkinter import Tk, Label, Button

# définir la fenêtre
laFenetre = Tk()
# définir l'étiquette (Label)
texteLabel = "bonjour tout le monde"
etiquette = Label (laFenetre, text = texteLabel, fg = "red")
etiquette.pack(padx = 10, pady = 5)
# définir le bouton (Button)
texteBouton = " Quitter "
commandeBouton = laFenetre.quit   # sortir de la boucle
bouton = Button (laFenetre, text = texteBouton, command = commandeBouton)
bouton.pack(pady = 5)
```

```
# lancer l'application
laFenetre.mainloop()

# on vient de sortir de la boucle de capture d'événement
laFenetre.destroy ()
```

*** from tkinter import Tk, Label, Button**

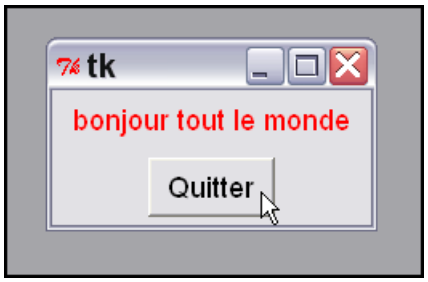
On importe du module graphique les trois classes **Tk()** qui crée une fenêtre, **Label** qui écrit un message dans une fenêtre, **Button** qui crée un bouton réactif dans ue fenêtre.

*** etiquette.pack(padx = 10, pady = 5)**

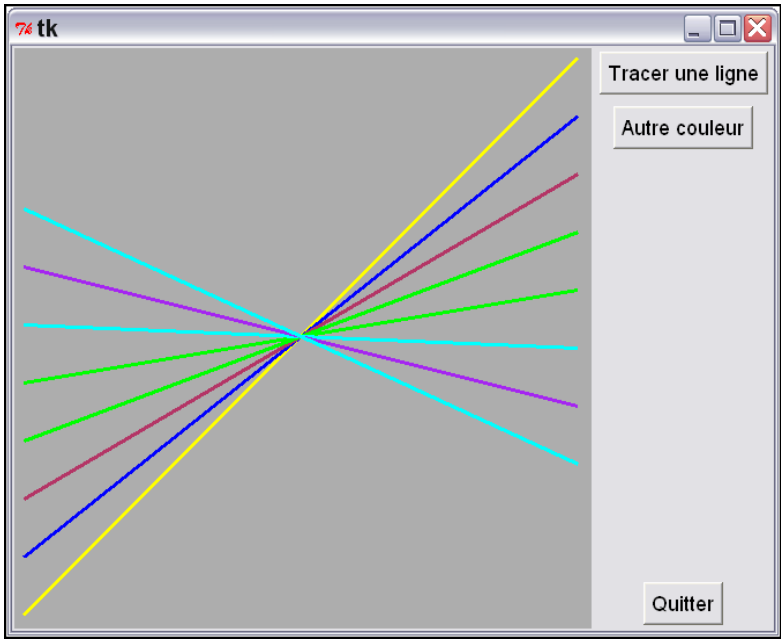
La méthode **pack()** est le layer de la fenêtre maître qui a été retenue ici. Appliquée à tous les widgets esclaves de la fenêtre, la méthode de layer les dispose "au mieux", par colonnes, à partir du haut. Les deux widgets sont donc superposés dans l'ordre de déclaration. Les paramètres **padx** et **pady**, facultatifs (valeur par défaut : 0), sont des paramètres de padding (rembourrage) ; ils définissent un entourage supplémentaire à l'espace minimum qu'occuperait le widget, lui-même calculé en fonction du "contenu" du widget (ici, des textes).

* **laFenetre.quit :** la méthode **quit()** termine **mainloop()**.

L'insruction laFenetre.destroy() tue l'aplication.



3. Un canevas et des boutons.



Il s'agit ici de définir une fenêtre, comportant une surface dans lequel on peut dessiner de droites de diverses couleurs définies aléatoirement. Une telle surface où l'on peut dessiner est appelée un canevas (**canvas** dans le langage). La commande de dessin est réalisée par bouton ; de même pour le changement de couleur. Un bouton est destiner à tuer l'application.

```
# ref. Swinnen, Apprendre à programmer avec Python, page 87
#
# importations

from tkinter import Tk, Canvas, Button
from random import randrange

palette=['purple','cyan','maroon','green','red',\
         'blue','orange','yellow','dark grey']

# --- définition des attributs de ligne et
#     des fonctions gestionnaires d'événements : ---
class Ligne () :
    # coordonnées de la ligne
    x1, y1 = 10, 490
    x2, y2 = 490, 10
    # couleur de la ligne
    c = randrange(8)        # génère un nombre aléatoire de 0 à  7
    couleur = palette [c]   # couleur de la ligne

    def tracerLigne():
        "Tracé d'une ligne dans le canevas cannevas1"
        cannevas.create_line(Ligne.x1,Ligne.y1,\
                             Ligne.x2,Ligne.y2,\
                             width=3,fill=Ligne.couleur)
        # modification des coordonnées pour la ligne suivante :
        Ligne.y2, Ligne.y1 = Ligne.y2+50, Ligne.y1-50

    def changerCouleur():
        "Changement aléatoire de la couleur du tracé"
        c = randrange(8)
        Ligne.couleur = palette[c]

#------ Programme principal -------

# Création du widget principal ("maître") :
laFenetre = Tk()

# création des widgets "esclaves" :
cannevas = Canvas(laFenetre,bg=palette[8],height=500,width=500)
cannevas.pack(side= "left")

boutonQuitter = Button(laFenetre, text='Quitter', command=laFenetre.quit)
boutonQuitter.pack(side="bottom", padx=5, pady=5)
```

```
boutonTracer = Button(laFenetre, text='Tracer une ligne',\
                      command=Ligne.tracerLigne)
boutonTracer.pack(padx=5, pady=5)

boutonColorier = Button(laFenetre, text='Autre couleur',\
                        command=Ligne.changerCouleur)
boutonColorier.pack (padx=5,pady=5)

laFenetre.mainloop () # démarrage du réceptionnaire d'événements
laFenetre.destroy () # destruction (fermeture) de la fenêtre
```

* `from random import randrange`

`random` est le module des "nombres au hasard". la fonction **`randrange()`** fournit des nombres aléatoires dans un intervalle 0..n.

* le principe du script est dans la définition d'une classe **`Ligne`** qui comporte en attribut de classe les paramètres variables d'une ligne (début, fin, couleur). Les méthodes de la classe définissent les deux fonctions principale : changer la couleur par défaut et tracer une ligne.

La classe Ligne n'est pas instanciée ; c'est l'un des usages des classes en Python de servir de "boîte à outils" pour le reste du script et d'encapsulation de données.

`* cannevas.pack(side= "left")`

`boutonQuitter.pack(side="bottom", padx=5, pady=5)`

le paramètre **`side`** situe l'objet dans la fenêtre : le canevas occupe le côté gauche de la fenêtre et le bouton pour quitter est collé au bas de la fenêtre : il suscite la création d'une nouvelle colonne. En l'absence du paramètre side pour les deux autre boutons, ceux-ci cherchent le premier espace libre à partir du haut (le haut par défaut) ; il le trouvent dans la colonne qui vient d'être créée.

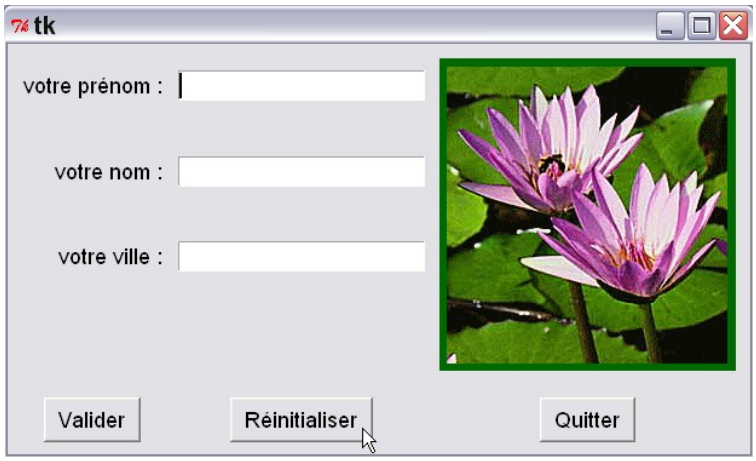
4. Les Widgets Grid et Entry.

4.1. Le gestionnaire de géométrie Grid.

Le gestionnaire de géométrie **`Grid`** rappelle une vieille méthode du HTML qui consiste à placer les éléments constitutifs d'une page dans un tableau. Les lignes (row) sont numérotées 1, 2, 3 … et de même pour les colonnes. La hauteur et la largeur des lignes sont tels que chacun des widgets programés puissent s'insérer dans une case. Le plus grand widget impose sa loi !

Au besoin, on peut fondre plusieurs lignes d'une colonne ou plusieurs colonnes d'un ligne (ou les deux) pour insérer un widget plus encombrant. Par défaut, les widgets sont placés au centre de la case, et on dispose d'une option supplémentaire d'alignement (**`sticky`**) et de l'option de rembourrage classique (padding).

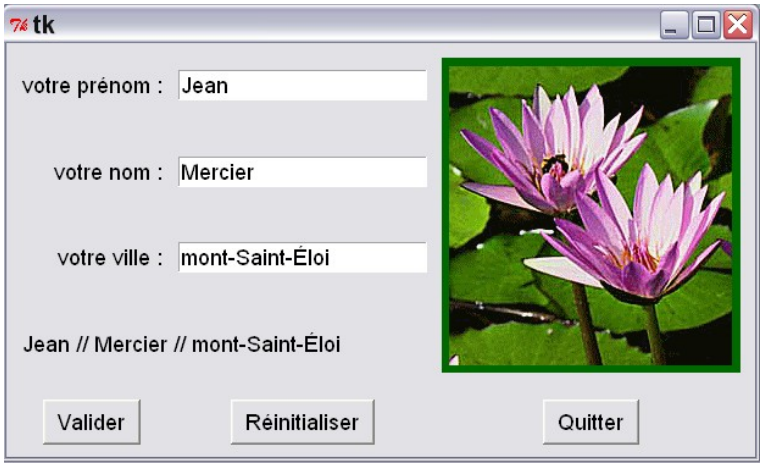
4.2. Une fenêtre de saisie de texte.



La fenêtre vide (démarrage de l'application ou réinitialisation) permet de saisir nom, prénom et villle. La validation affiche les trois valeurs saisie.

Les zones de saisie de texte appartiennent à la classe Entry.

L'image est chargée depuis un fichier dans un widget de la classe PhotoImage qui la stocke. Pour l'afficher on la "dessine" dans un canevas (widget **`Canvas`**).



4.3. Le script.

```
# ref. Swinnen, Apprendre à  programmer avec Python, page 97
#      Introduction to TKinter, page 84 ...
#
# le layer grid ; le widget Entry
#
from tkinter import *

fen = Tk () # la fenêtre de l'application

# les labels
labelUn = Label (fen, text="votre prénom :")
labelUn.grid (row = 1,column = 1, sticky = "E", padx = 10)
labelDeux = Label (fen, text="votre nom :")
```

```
labelDeux.grid (row = 2, column = 1, sticky = "E", padx = 10)
labelTrois = Label (fen, text="votre ville :")
labelTrois.grid (row = 3, column = 1, sticky = "E", padx = 10)

labelValider = Label (fen, text="") # label de la chaîne de validation
labelValider.grid (row = 4, column = 1, columnspan = 2,\
                   sticky ="W",padx = 10)

# les entrées
entreeUn = Entry (fen)
entreeUn.grid (row = 1, column = 2)
entreeUn.focus_set()
entreeDeux = Entry (fen)
entreeDeux.grid (row = 2, column = 2)
entreeTrois = Entry (fen)
entreeTrois.grid (row = 3, column = 2)

# le cannevas et son image (232x245)
cannevasImg = Canvas (fen, width =245, height = 258, bg = "dark green")
photo = PhotoImage (file="a1_lotus.gif")
image = cannevasImg.create_image(124, 131, image = photo)
cannevasImg.grid(row = 1, column = 3, rowspan = 4, padx = 10, pady = 10)

# les boutons et leur gestion

def valider () :
    chn = entreeUn.get()+" // "+ entreeDeux.get()+ " // "+ \
          entreeTrois.get()
    labelValider.config(text = chn)

def initialiser ():
    labelValider.config(text = "")
    entreeTrois.delete (0, END)
    entreeDeux.delete (0, END)
    entreeUn.delete (0, END)
    entreeUn.focus_set()


boutonQuitter = Button (fen, text=' Quitter ',command=fen.quit)
boutonQuitter.grid (row = 5,  column = 3, pady = 10)

boutonValider = Button (fen, text=' Valider ',command=valider)
boutonValider.grid (row = 5, column = 1, pady = 10)

boutonInitialiser = Button (fen, text=' Réinitialiser ',\
                            command=initialiser)
boutonInitialiser.grid (row = 5, column = 2, pady = 10)
```

```
fen.mainloop() # boucle de capture des événements
fen.destroy()  # fin de l'application
```

* **la grille.**

<table>
<tr><td>labelUn</td><td>entreeUn</td><td rowspan="4">le cannevas

et la photo</td></tr>
<tr><td>labelDeux</td><td>entreeDeux</td></tr>
<tr><td>labelTrois</td><td>entreeTrois</td></tr>
<tr><td colspan="2">labelValider</td></tr>
<tr><td>boutonValider</td><td>boutonInitialiser</td><td>boutonQuitter</td></tr>
</table>

* `photo = PhotoImage (file="a1_lotus.gif")`

Cette instruction créer une représentation interne de la photo. Cette représentation permet l'accès aux paramètres de la photo. tkinter accepte le format gif, mais pas les formats usuels jpeg, png, tif, wmf, emf, svg. Pour utiliser d'autres formats, il faut enrichir le module…

* `image = cannevasImg.create_image(124, 131, image = photo)`

Cette instruction dessine l'image sur le cannevas en mettant son centre aux coordonnées (124, 131) du cannevas. Il n'ya pas redimensionnement de l'image.

* `entreeTrois.get()`

La méthode `get()` d'un objet de la classe `Entry` retourne la chaîne frappée dans le widget.

* `entreeTrois.delete (0, END)`

La méthode `delete(n,p)` efface les caractères entre les emplacement n (inclus) et p. Si on remplace la seconde coordonnée par l'identificateur prédéfini `END`, toute la chaîne est effacée.

* `entreeUn.focus_set()`

Cette intruction utilise une méthode qui donne le focus au widget qualifié. Ici, le curseur de saisie de texte est actif dans l'entrée du haut.

* `config(text = chn)`

La méthode `config()` permet de changer la valeur d'un attribut du widget qu'elle qualifie (les attributs sont les paramètres utilisables lors de la réation d'un widget).

* `sticky = "E"` : le widget est placé dans la case du tableau en utilisant comme repères les points cardinaux les valeurs possibles sont N, S, E, W (bords) NE, NW, SE, SW (coins).

fiche 2 : fenêtres et frame

1. Une question de documentation sur tkinter

1.1. La documentation Python.

On continue à travailler avec IDLE. On a vu que IDLE permet d'avoir accès à la documentation officielle de Python, en langue anglaise. La traduction n'a été faite que partiellement. Il se peut que l'adresse du fichier de documentation doivent être changé, soit que l'on veuille avoir accès à cete documentation en mode local, soit que les distributeurs de Python n'aient pas fait toutes les mises à jour. La référence se trouve dans le fichier `C:\Python30\Lib\idlelib\EditorWindow.py`, ligne : `EditorWindow.help_url = "http://docs.python.org/3.0/"`. Le fichier python est recompilé automatiquement au redémarrage de IDLE.

La partie consacrée à tkinter est restreinte ; il convient de consulter le site en ligne ou le fichier pdf **"An Introduction to Tkinter"** de Fredrik Lundh. **C'est le passage obligé**, même si le texte qui remonte à 1999 a été un peu mis à jour. Il convient évidemment d'adapter les exemples et certaines annotations en Python 3.0 (`print` est une fontion, les retours de fonctions de type `list` sont devenus des types `map (dictionnaire)`, et il faut alors caster avec `list()` ; les noms de modules sont écrits en minuscules …). Le site `http://www.pythonware.com/library/` contient beaucoup de documentation sur les modules Python. Le site `http://infohost.nmt.edu/tcc/help/pubs/tkinter/` (en anglais) est à jour, précis, mais ne traite pas `tkinter` sous tous ses aspects.

1.2. Exemple pour le Widget Frame (Cadre)

Chapter 26. The Frame Widget

A frame is rectangular region on the screen. The frame widget is mainly used as a geometry master for other widgets, or to provide padding between other widgets.

When to use the Frame Widget

Frame widgets are used to group other widgets into complex layouts. They are also used for padding, and as a base class when implementing compound widgets.

Patterns

The frame widget can be used as a place holder for video overlays and other external processes.

To use a frame widget in this fashion, set the background color to an empty string (this prevents updates, and leaves the color map alone), pack it as usual, and use the window_id method to get the window handle corresponding to the frame.

```
frame = Frame(width=768, height=576, bg="", colormap="new")
frame.pack()
video.attach_window(frame.window_id())
```

Methods

Except for the standard widget interface (config, etc), the Frame widget has no methods.

Options

The Frame widget supports the following options:

Table 26-1. Frame Options

Option	Type	Description
height, width	distance	Frame size.
background (bg)	color	The background color to use in this frame. This defaults to the application background color. To prevent updates, set the color to an empty string.
colormap	window	Some displays support only 256 colors (some use even less). Such displays usually provide a color map to specify which 256 colors to use. This option allows you to specify which color map to use for this frame, and its child widgets. By default, a new frame uses the same color map as its parent. Using this option, you can reuse the color map

Option	Type	Description
		of another window instead (this window must be on the same screen and have the same visual characteristics). You can also use the value “new” to allocate a new color map for this frame. You cannot change this option once you've created the frame.
cursor	cursor	The cursor to show when the mouse pointer is placed over the button widget. Default is a system specific arrow cursor.
relief	constant	Border decoration. The default is FLAT. Other possible values are SUNKEN, RAISED, GROOVE, and RIDGE. Note that to show the border, you need to change the borderwidth from it's default value of 0.
borderwidth (bd)	distance	Border width. Defaults to 0 (no border).
takefocus	flag	Indicates that the user can use the **Tab** key to move to this widget. Default is an empty string, which means that the frame accepts focus only if it has any keyboard bindings (default is off, in other words).
highlightback-ground, highlightcolor	color	Controls how to draw the focus highlight border. When any child to the frame has focus, the border is drawn in the highlightcolor color. Otherwise, it is drawn in the highlightbackground color. The defaults are system specific.
highlight-thickness	distance	Controls the width of the focus highlight border. Default is 0 (no border).

2. Un problème avec Frame.

2.1. Création d'une application maître avec Frame.

On vient de voir la fiche de documentation de l'objet **`Frame`**. Les options de configuration sont des caractéristiques qui peuvent être fixées par le programme (en général lors de la création de l'instance du "cadre" ; la méthode **`configure()`** permet de fixer ou changer les options). Les options ont des

valeurs par défaut. Les options explicitées sont accessibles, pas les valeurs par défaut.

Tout serait bien sans la possibilité de créer une fenêtre "maître" à l'aide de Frame, sans passer par Tk(). Même si le code développant cette possibilité est peu explicite, et qu'il vaut mieux ne pas l'utiliser, il faut cependant en connaître l'existence, car on le trouve très souvent dans les exemples de programmation d'interfaces graphiques, et même parfois comme exemple type !

Quand on crée un widget, le premier argument (sauf pour l'objet maître de classe Tk) est l'identificateur du "propriétaire" (on utilisera le mot "propriétaire" ou "conteneur" plutôt que "parent", pour éviter la confusion avec le "parent" au sens hiérarchique). Si en début de programme, on déclare une instance de **`Frame`** sans argument, ou ce qui est la même chose, l'argument **`None`**, **`tkinter`** crée une enveloppe fenêtrée. Pour accéder à cette enveloppe, on utilise le procédé classique avec l'attribut "**`master`**", qui permet pour tous les widgets, l'accès d'une instance à l'instance de son propriétaire.

2.2. Le programme d'illustration.

L'application fenêtrée comporte un cadre (rouge) propriétaire d'un label de même couleur qui y est "placé". On peut quitter l'application et détruire la fenêtre enveloppante soit en cliquant la zone rouge, soit en cliquant le bouton.

On peut voir sur la copie d'écran les messages affichés dans la console lors d'une précédente exécution du programme.

2.3. Le fichier.

fichier : a2_prg3_master.py

```
from tkinter import *
#
# création de la classe de l'application fenêtrée
#
class App(Frame):
    def __init__(self):
        Frame.__init__(self, None)
        self.pack()
#
# création de l'instance racine
monApp = App()
#
# la fonction pour quitter la "mainloop"
#
def quitter(event):
    print ("on va quitter")
    monApp.quit()
#
def bquitter():
    quitter(None)
#
# appel à la classe du manager de fenêtre
#
monApp.master.title("application qui ne fait rien")
#
# le cadre à cliquer
#
cadre = Frame (monApp, border=2, width=500, height = 400,\
               bg="red", relief="groove")
```

```
cadre.bind ("<Button-1>", quitter)
cadre.grid(row=1)
#
# le label, lui aussi cliquable
#
message = Label (cadre, text = "cliquer la zone rouge pour quitter")
message.bind ("<Button-1>", quitter)
message.configure (bg=message.master.cget("bg"))
message.place(x=120, y=300)
#
# le bouton de fermeture
#
b = Button (monApp, text="quitter par bouton", command=bquitter)
b.grid(row=2, pady=20)
#
# le programme démarre
#
monApp.mainloop()
#
# on vient de quitter la boucle principale par monApp.quit()
#
print ("destruction")
monApp.master.destroy()
print ("fin de l'application")
```

* `Frame.__init__(self, None)`

Il faut toujours appeler le constructeur de la classe parent (hiérarchique) dans le constructeur de la nouvelle classe.

*
```
def bquitter():
    quitter(None)
```

La fonction identifiée à **`command`** ne peut avoir de paramètre ; on a donc créé une fonction sans argument pour uniformiser la fermeture. On verra ultérieurement une méthode différente, plus intéressante, avec utilisation de fonction anonyme (lambda fonction).

* `monApp.master.title("application qui ne fait rien")`

L'attribut **`master`** permet d'accéder au propriétaire (anonyme) de **`monApp`** (héritière du type **`Frame`**). C'est également ce propriétaire qu'il faut détruire pour effacer le graphisme de l'application !!! La boucle principale appartient, elle, à l'application.

* **note :** les commandes **`print()`** sont utiles pour l'illustration et le débuggage. Elles affichent dans la console de lancement du programme. Pour afficher dans la fenêtre, on a utilisé un label.

* `cadre.bind ("<Button-1>", quitter)`

On a un exemple de liaison entre le bouton de la souris **`"<Button-1>"`** qui, quand il est cliqué dans **`cadre`** appelle la fonction **`quitter`**. Un fonction ainsi "liée" prend nécessairement en premier, un argument "événement", qui n'est pas utilisé ici.

2.4. Une présentation plus classique (et recommandée).

`fichier : a2_prg3_tk.py`

```
from tkinter import *
#
# création de l'instance racine
monApp = Tk()
#
# la fonction pour quitter la "mainloop"
#
def quitter(evenement=None): # commun à Button et .bind()
    print ("on va quitter")
    monApp.quit()
#
# utilisation du manager de fenêtre
#
monApp.title("application qui ne fait rien")
#
# le cadre à cliquer
#
cadre = Frame (monApp, border=2, width=500, height = 400,\
               bg="red", relief="groove")
cadre.bind ("<Button-1>", quitter)
cadre.grid(row=1)
#
# le label lui aussi cliquable
#
message = Label (cadre, text = "cliquer la zone rouge pour quitter")
message.bind ("<Button-1>", quitter)
message.configure (bg=message.master.cget("bg"))
message.place(x=120, y=300)
#
# le bouton de fermeture
#
b = Button (monApp, text="quitter par bouton",\
            command=lambda e=None : quitter(e))
b.grid(row=2, pady=20)
#
# le programme démarre
#
monApp.mainloop()
#
# on vient de quitter la boucle principale par monApp.quit()
#

print ("destruction")
monApp.destroy()
print ("fin de l'application")
```

* `message.configure (bg=message.master.cget("bg"))`

cget() permet de récupérer la couleur de l'arrière plan de message.master. La méthode configure() applique ensuite cette couleur au label. C'est évidemment la bonne façon de programmer la contrainte : "*de la même couleur*"

* **`command=lambda e=None : quitter(e))`**

On affecte à **`command`** une fonction anonyme qui prend un argument **`e`** de valeur None ; on peut procéder ainsi puisque l'événement créé par le clic n'est pas utilisé.

* **`message.place(x=120, y=300)`**

Les gestionnaires de géométrie (**`pack(), grid(), place()`**) sont incompatibles entre eux : pour un propriétaire donné qui sert de conteneur, il n'y a qu'un type de gestionnaire autorisé. Mais cette limitation ne dépasse pas les widgets d'un propriétaire ; on peut donc utiliser un gestionnaire différent pour chaque propriétaire. Pour le gestionnaire **`place()`** qui "place" un widget en coordonnées absolues, **`x`** et **`y`** sont les coordonnées en pixel du coin supérieur gauche du widget, l'origine étant le coin supérieur gauche du propriétaire.

* **`def quitter(evenement=None):`**

`quitter()` st utilisé à la fois par un bouton et la méthode **`.bin()`**. Or le premier ne demande pas d'argument et le second en exige un ; on résout le problème en utilisant un paramètre initialisé.

3. Centrer une application fenêtrée.

3.1. objectifs.

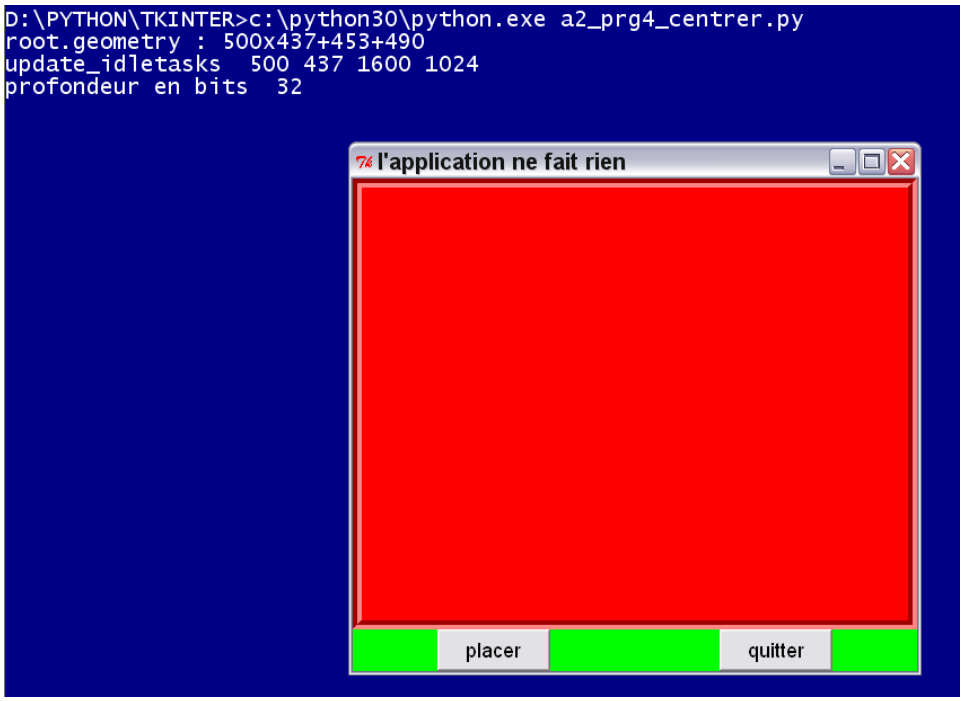
Les interfaces graphiques associées aux systèmes d'exploitation (parfois appelées x-windows) comportent un gestionnaire de fenêtre (windows manager). Les affichages de widgets impliquent soit un calcul par le widget de ses caractéristiques graphiques (encombrement, place, couleurs, curseur…), soit un calcul par le gestionnaire de fenêtre de dites caractéristiques. En principe, c'est le gestionnaire qui fait les calculs, et les paramètres ne sont définis qu'une fois l'affichage réalisé, c'est à dire après la mise en action de la boucle principale.

Dans la boucle principale, on ne peut accéder aux données d'un widget qu'en déclenchant un événement et en interrogeant le widget ou son gestionnaire de fenêtre dans le traitement de cet événement. Sinon, il faut simuler ce gestionnaire, ce que sait réaliser chaque widget.

3.2. une application de démonstration.

Voici un programme de démonstration, où l'objectif visé est d'abord de centrer la fenêtre de l'application sur l'écran. Pour retrouver les caractéristiques d'un widget, on va utiliser les trois démarches possibles :

- demander des données à un widget, avant son affichage. Ceci ne fonctionne que pour de données indépendantes de l'affichage (par exemple les dimensions de l'écran)
- demander à un widget des données après affichage, à travers le traitement d'un événement déclenché par un bouton
- demander à un widget des données qui ne sont connues qu'après affichage et qui pour être saisies avant affichage, demandent une simulation d'affichage par une méthode de widget, la méthode **`winfo_idletasks()`** (informations sur des tâches non accomplies)



La saisie d'écran (partielle) est faite au cours de l'exécution du script dans une console DOS. La fenêtre de l'application est bien au milieu de l'écran.

3.3. Le source.

fichier a2_prg4_centrer.py

```
from tkinter import *
#
root = Tk()
root.title("l'application ne fait rien")
flag = False # flag d'existence de root
# cadre rouge
cadre = Frame (root, border=8, width=500, height = 400,\
               bg="red", relief="groove")
cadre.grid()
#
# affichage de données dans la console
def placer():
     print ("root.geometry :",root.geometry())
     print ("update_idletasks ", wh, ht, swh, sht)
     print ('profondeur en bits ', root.winfo_screendepth())
#
def quitter() :
    global flag # attention !!!
    flag = True
    root.quit()
#
# les boutons
#
```

```
cadreBoutons = Frame (root, bg="green")
cadreBoutons.grid()
#
# attention : la largeur des boutons est donnée en caractères
bPlacer = Button (cadreBoutons, text="placer",width=9, command=placer)
bQuitter = Button (cadreBoutons, text="quitter",width=9, command=quitter)

p=(250 - bPlacer.winfo_reqwidth()) // 2 # deux boutons de même largeur
bPlacer.grid(row=1, column=1, padx=p)
bQuitter.grid(row=1, column=2, padx=p)

# calcul des coordonnnées du sommet top/left
swh=root.winfo_screenwidth()
sht=root.winfo_screenheight()

root.update_idletasks() # nécessaire pour winfo_width()
wh = root.winfo_width()
ht = root.winfo_height()
xtl=(swh-wh) // 2
ytl=(sht - ht) // 2
# placement de l'application
root.geometry('+'+str(xtl)+'+'+str(ytl))
#
root.mainloop()
#
# destruction de l'application si ce n'est déjà  fait
if flag :
    print ("destruction")
    root.destroy()
# fin de l'application
```

*** swh=root.winfo_screenwidth()**

Les dimensions de l'écran sont une donnée accessible par tout widget, et ce, avant l'affichage.

*** wh = root.winfo_width()**

La largeur effective d'un widget est une donnée après exécution, en principe après affichage. On simule donc cet affichage avec la méthode **winfo_idletasks()**.

* La géométrie d'un widget n'est connue qu'après affichage. On peut la fixer avant affichage. C'est un chaîne faite de sept éléments sur le mode suivant : "300x500+450+80". Le premier nombre est la largeur du widget, le. second sa hauteur, le troisième et le quatrième, les coordonnées de son coin top-left. On peut ne pas metttre les deux premiers lors de la définition de la géométrie :

```
root.geometry('+'+str(xtl)+'+'+str(ytl))
```

```
* flag = False # flag d'existence de root
if flag :
    root.destroy()
```

Le dispositif avec flag permet de détruire la fenêtre avec l'icône de fermeture sans problème.

* **`p=(250 - bPlacer.winfo_reqwidth()) // 2`**

Pour placer correctement les deux boutons, on les définit avec un padding de façon à remplir l'espace libre sous le cadre rouge. Mais comme la largeur des boutons est donnée en caractères, et dépend donc des ressources système, on récupère le largeur vraisemblable calculée par le widget avant exécution, par la méthode **`winfo_reqwidth()`** qui ne nécessite pas la simulation.

fiche 3 : fenêtres de dialogue

introduction.

Une fenêtre de dialogue est une fenêtre qui s'ouvre pour saisir une donnée de peu de longueur ou afficher un avertissement. Une fenêtre de dialogue a les propriétés suivantes :

- sa création revient à une fenêtre d'une application existante, sa fenêtre maître. Elle n'est donc pas une fenêtre racine d'une application. En langage Python elle est **`Toplevel`**.

- elle s'affiche au dessus des autres fenêtres de l'application ; c'est un attribut des fenêtres **`transient.`**

- elle détourne à son seul profit tous les événements dont l'application est la cible ; ceci a pour conséquence que l'on ne peut plus accéder au reste de l'application, puisque le focus reste nécessairement sur la fenêtre.

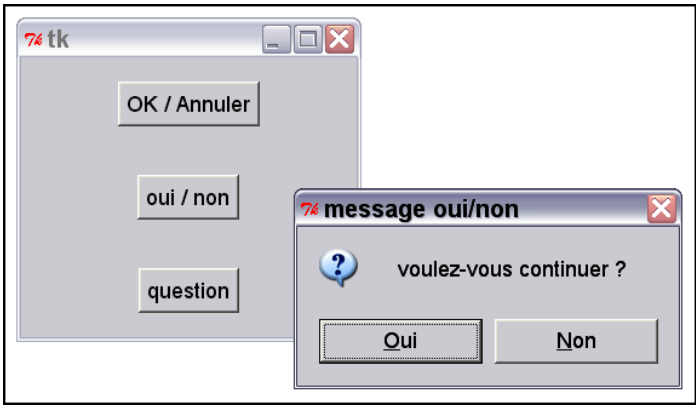
- on ne peut donc retourner à l'application qu'en la détruisant ; lors de sa fermeture, elle peut transmettre des données à sa fenêtre maître. La fenêtre est **`modale`**.

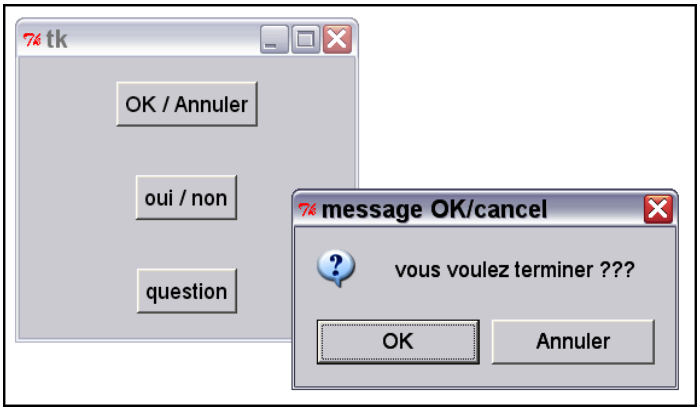
- si par un procédé quelconque externe, la fenêtre maître est iconifiée, la fenêtre dialogue est effacée (**`withdraw`**), mais elle reste présente et réapparaît dans la restitution de la fenêtre maître. Si la fenêtre maître est détruite, la fenêtre de dialogue est détruite.

Comme tous les langages fenêtrés (Java, C++, Delphi, Javascript etc), Python offre une petite famille de fenêtres de dialogues toutes prêtes, répondant aux besoins fréquents d'un acquiescement (oui/non, où OK/annuler) ou d'un avertissement à valider.

1. Dialogues d'acquiescement.

1.1. Copies d'écran.





1.2. Le fichier source.

fichier : a3_prg2_ok.py

```
from tkinter import Tk, Button
from tkinter.messagebox import askokcancel, askyesno, askquestion


maFenetre = Tk()

def comOK() :
    rep = askokcancel ("message OK/cancel",\
                       "vous voulez terminer ???")
    if rep :
        maFenetre.quit()

def comOui() :
    rep = askyesno ("message oui/non",\
                       "voulez-vous continuer ?")
    if not rep :
        maFenetre.quit()

def comQuestion() :
    rep = askquestion ("message question",\
                       "voulez-vous continuer ?")
    if rep=="no" :
        maFenetre.quit()

monBoutonOK = Button (maFenetre, text = "OK / Annuler", command=comOK)
monBoutonOK.pack(padx=80 , pady=20)

monBoutonOui = Button (maFenetre, text = "oui / non", command=comOui)
monBoutonOui.pack(padx=80 , pady=20)

monBoutonQuestion = Button (maFenetre, text = "question",\
                            command=comQuestion)
monBoutonQuestion.pack(padx=80 , pady=20)


maFenetre.mainloop()
maFenetre.destroy()
```

* noter l'importation des trois fenêtres de dialogues possibles dans **`tkinter.messagebox`**

* les trois "messagebox" se ressemblent ; noter cependant que les deux premières retournent un booléen, et la dernière **`"yes"`** ou **`"no"`**.

2. Le protocole.

On a pris soin dans l'exemple ci-dessus de fermer proprement l'application : appel de la méthode **`quit()`** après confirmation dans une fenêtre de dialogue. On sort alors de la boucle principale, et on

appelle **destroy()** qui supprime la fenêtre principale.

Mais qu'en est-il lorsque l'on utilise la fermeture de fenêtre iconifiée ?

On a une erreur, évidemment sans conséquence pratique, mais que l'on va éviter. Plus gênant, la demande de confirmation de la fermeture n'est pas appelée !

Voici le message d'erreur : la fenêtre a été détruite (et donc la boucle principale), puis le script a continué en demandant une destruction déjà réalisée !

```
Traceback (most recent call last):
  File "D:/python/tkinter/a3_prg1_protocole.py", line 18, in <module>
    maFenetre.destroy()
  File "C:\Python30\lib\tkinter\__init__.py", line 1672, in destroy
    self.tk.call('destroy', self._w)
_tkinter.TclError: can't invoke "destroy" command:  application has been destroyed
```

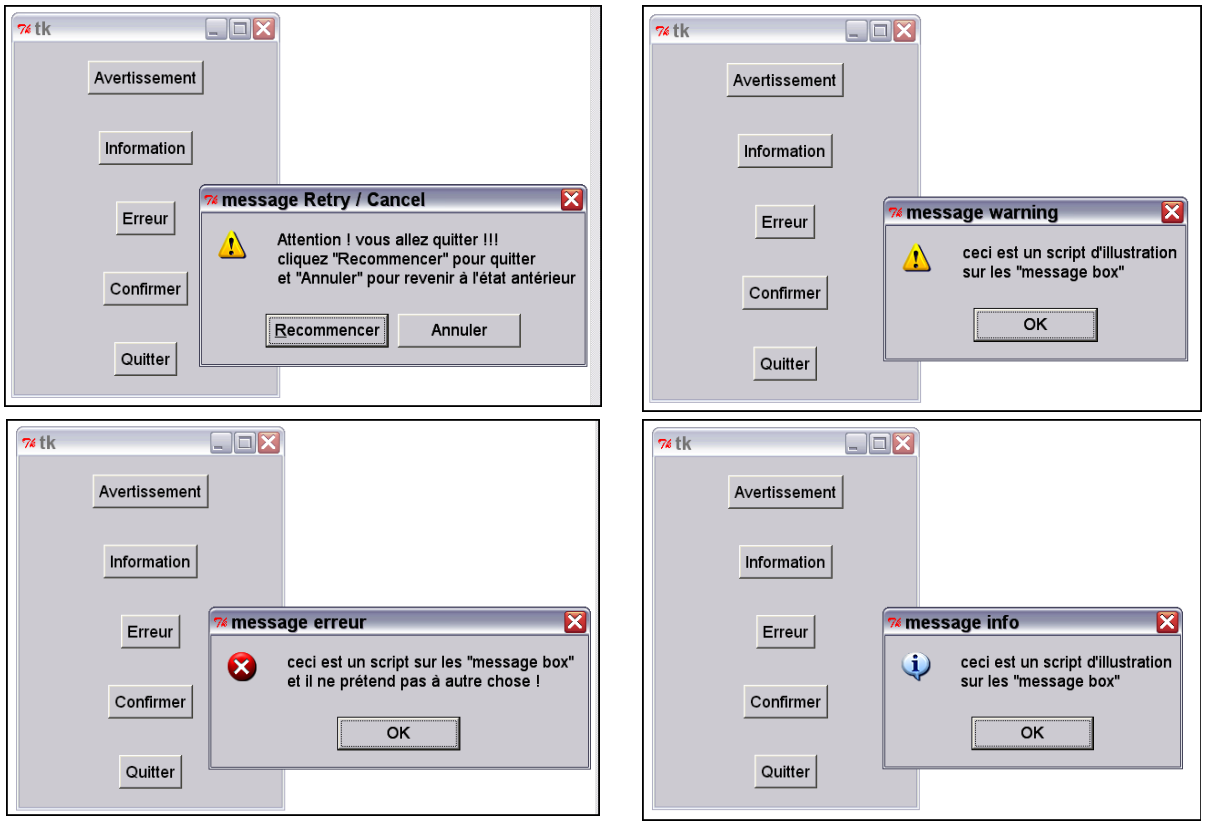
Il existe un moyen de capturer l'appel de fermeture par l'icône appelé "**protocol**", qui est une méthode de Tk. Le programme qui suit est en quelque sorte la base de toute définition de fenêtre principale d'une application :

fichier : a3_prg1_protocole.py

```
from tkinter import Tk, Button
from tkinter.messagebox import askokcancel


maFenetre = Tk()

def comOK() :
    rep = askokcancel ("confirmer la fermeture ",\
                       "vous voulez terminer ?")
    if rep :
        maFenetre.quit()


monBoutonOK = Button (maFenetre, text = "quitter", command=comOK)
monBoutonOK.pack(padx=20 , pady=20)

maFenetre.protocol ("WM_DELETE_WINDOW", comOK)
maFenetre.mainloop()
maFenetre.destroy()
```

3. Avertissements.

3.1. quatre exemples avec un ou deux boutons.



3.2. Le fichier source.

fichier a3_prg3_warning.py

```
from tkinter import Tk, Button
from tkinter.messagebox import askretrycancel, showwarning,\
                               showinfo, showerror


maFenetre = Tk()

def comOK() :
    rep = showwarning ("message warning",\
                       "ceci est un script d'illustration\n"+\
                       "sur les \"message box\"")

def comInfo() :
    rep = showinfo ("message info",\
                       "ceci est un script d'illustration\n"+\
                       "sur les \"message box\"")

def comErreur() :
```

```
    rep = showerror ("message erreur",\
                         "ceci est un script sur les \"message box\" \n"+\
                         "et il ne prétend pas à autre chose ! ")

def comRetry() :
    rep = askretrycancel ("message Retry / Cancel",\
                    "Attention ! vous allez quitter !!!\n"\
                    + "cliquez \"Recommencer\" pour quitter\n"\
                    + "et \"Annuler\" pour revenir à  l'état antérieur")
    if rep :
        maFenetre.quit()


monBoutonOK = Button (maFenetre, text = "Avertissement", command=comOK)
monBoutonOK.pack(padx=80 , pady=20)

monBoutonInfo = Button (maFenetre, text = "Information", command=comInfo)
monBoutonInfo.pack(padx=80 , pady=20)

monBoutonErreur = Button (maFenetre, text = "Erreur", command=comErreur)
monBoutonErreur.pack(padx=80 , pady=20)

monBoutonRetry = Button (maFenetre, text = "Confirmer", command=comRetry)
monBoutonRetry.pack(padx=80 , pady=20)

monBoutonQuit = Button (maFenetre, text = "Quitter", command=maFenetre.quit)
monBoutonQuit.pack(padx=80 , pady=20)


maFenetre.mainloop()
maFenetre.destroy()
```

4. Un module utilitaire.

4.1. cahier des charges.

Un premier module permet de définit une fenêtre de dialogue générique : cette fenêtre n'a pas de contenu ni de retour. Un second module montre comment utiliser la classe définie dans le premier module en l'étendant de façon à lui donner un contenu (la saisie de deux chaînes avec les widget de saisie) et un retour utilisé l'application principale (les deux chaînes).

Ce cahier des charges implique trois sortes d'exigences :

- d'une part, la classe de la fenêtre de dialogue doit avoir les caractéristiques d'une fenêtre de dialogue telles qu'on les a définies dans l'introduction.
- d'autre part, il faut que cette classe fonctionne sans modification (sinon ce n'est plus un "module") quel que soit l'habillage qu'on va lui donner.
- enfin, de façon plus anecdotique, on peut imposer des contingences moins nécessaires, comme par exemple celle de placer la fenêtre de dialogue. Ici on va imposer de pouvoir choisir entre une place centrale (attribut **`centrer`**) et une place relative à la fenêtre maître (**`offx, offy`**).

4.2. Code source pour la fenêtre générique.

fichier : a3_module_dialogue.py /// module : a3_module_dialogue

```
from tkinter import *

# ****************************************************************
# ce module définit une classe, celle d'une fenêtre de dialogue  *
# ****************************************************************
class Dialogue (Toplevel) :
    # le constructeur
    # ###############
    def __init__ (self, conteneur, titre = None, centrer= False,\
                  offx=250, offy=250) :
        # associer la fenête de dialogue et son conteneur
        Toplevel.__init__(self, conteneur)
        self.protocol("WM_DELETE_WINDOW", self.annuler)

        # le dialogue apparaît au dessus de son conteneur
        self.transient (conteneur)
        #
        if titre :
            self.title(titre) # title() est hérité de Toplevel

        # conteneur = fenêtre qui ouvre le Dialogue
        self.conteneur = conteneur
        self.resultat = None

        # définition d'un  cadre d'habillage
        cadre = Frame(self)
        self.initial_focus = self.habillage (cadre)
        cadre.pack (padx=10, pady = 10)

        # créer la boîte bouton
        focusDefaut = self.boiteBoutons()

        # rendre la fenêtre modale
        self.grab_set()

        # cas où on n'a pas surchargé la fonction habillage()
        if not self.initial_focus :
            self.initial_focus = focusDefaut # bouton OK

        if centrer :
            # centrer la fenêtre de dialogue
            self.update_idletasks() # nécessaire pour winfo_width()
            wh = self.winfo_width()
            ht = self.winfo_height()
            swh= self.winfo_screenwidth()
```

```
            sht= self.winfo_screenheight()
            xtl=(swh-wh) // 2
            ytl=(sht - ht) // 2
            self.geometry('+'+str(xtl)+'+'+str(ytl))
        else:
            # afficher la fenêtre de dialogue par rapport au conteneur
            self.geometry ("+"+str(conteneur.winfo_rootx()+offx)+\
                           "+"+str(conteneur.winfo_rooty()+offy))

        # porter le focus sur le cadre ou la fenêtre d'entrée
        self.initial_focus.focus_set ()

        # boucle locale de la fenêtre de dialogue
        self.wait_window (self)
    #
    # construire
    def habillage (self, master) :
        pass   # méthode qui doit être surchargée

    # définition de la boîte à boutons
    # ################################
    def boiteBoutons (self) :
        boite = LabelFrame (self, text="Valider")
        w1 = Button (boite, text = "O.K.", width = 10,\
                    command = self.ok, default = ACTIVE)
        w1.pack (side=LEFT, padx = 5, pady = 5)

        w2 = Button (boite, text = "Annuler", width = 10,\
                    command = self.annuler)
        w2.pack (side=LEFT, padx = 5, pady = 5)
        self.bind("<Return>", self.ok)
        self.bind("<Escape>", self.annuler)
        boite.pack ()
        return w1

    def ok (self, evenement = None) :
        self.initial_focus.focus_set()

        # effacement avant de supprimer (pour le rendu)
        self.withdraw ()

        # nécessaire si dans apply() on utilise des éléments
        # qui doivent être visibles pour fournir des données
        self.update_idletasks()

        self.apply()
        self.annuler()
```

```
    def annuler (self, evenement = None) :
        self.conteneur.focus_set()
        self.destroy()

    def apply (self) :
        pass # méthode qui doit être surchargée
```

*** une fenêtre modale**

- On a jusque là construit explicitement que des fenêtres **`Tk`**. Or ans une application il ne peut y avoir qu'une fenêtre **`Tk`**, la fenêtre "racine" de l'application. Toutes les autres fenêtres son appelées Toplevel et sont dépendantes d'une autre fenêtre, son conteneur (maître, propriétaire). Cette dépendance implique que sa durée de vie, sa visualisation ou son iconisation dépendent de son conteneur. Ceci explique que si une fenêtre est déclarée Toplevel, ou hérite de Toplevel, il faut exécuter l'initialisation de la classe Toplevel, en passant en paramètre la fenêtre maître:

```
Toplevel.__init__(self, conteneur)
```

- Il faut ensuite placer la fenêtre au-dessus des autres fenêtres de l'application ; la méthode **`transient()`** le fait (elle ne fait pas que cela, mais c'est sans utilité ici) :

```
self.transient (conteneur)
```

- Une fenêtre modale est une fenêtre qui détourne vers elle tous les événements. La méthode "attrape-tout" (grab) s'appelle **`grab_set()`** :

```
self.grab_set()
```

- Il n'y a pour l'instant pas de boucle de surveillance des événements (**`mainloop()`**). Une boucle locale est créée, qui attend la destruction de la fenêtre (**`destroy()`**) pour se terminer.

- Il faut donc prévoir un événement qui détruit la fenêtre de dialogue. Il y a deux boutons qui y concourent : le bouton **`ok`** et le bouton **`annuler`**. La méthode **`annuler()`** se contente de donner le focus au conteneur et de détruire la fenêtre de dialogue. La méthode **`ok()`** appelle la méthode apply() qui gère le retour de la fonction le cas échéant.

*** une fenêtre utile.**

Une fenêtre de dialogue est sensée apporter un plus dans l'application. On a rencontré les cas communs dans la première partie : confirmation, avertissement, erreur… Une fenêtre de dialogue peut réaliser une initialisation à la demande de l'usager, demander d'entrer une ou plusieurs valeurs (par exemple un login/password) etc. C'est cet ensemble de fonctions que nous appelons l'habillage de la fenêtre de dialogue. Ceci est réalisé dans la fonction **`habillage()`** qui doit en principe être surchargée par l'utilisateur. La partie graphique est réalisée dans un cadre. Dans le cas où la surcharge n'est pas réalisée, la méthode retourne None (casté en False). Il faut prévoir ce cas : **`self.initial_focus`** peut être None ! Sinon c'est l'erreur !

```
cadre = Frame(self)
self.initial_focus = self.habillage (cadre)
cadre.pack (padx=10, pady = 10)
```

D'autre part, il se peut que la fenêtre de dialogue retourne des données à son conteneur. Ce n'est pas le cas d'une fenêtre d'avertissement, mais c'est celui d'une fenêtre oui/non. Ici on peut être plus exigeant ; la méthode **`apply()`**, dans sa version surchargée doit réaliser les "finitions", avec un résultat dans **`self.resultat`**. Ce peut être long, et alors, la fenêtre est toujours affichée, avec le risque que l'on clique de nouveau et inconsidérément sur les boutons : il faut effacer la fenêtre. Mais certaines données ne sont accessibles que sur une fenêtre effectivement affichée (on a rencontré ce cas dans la fiche 2, dans le chapitre "centrage d'une fenêtre" qui a été repris ici). On a donc les deux appels de méthodes :

```
self.withdraw ()
```

```
        self.update_idletasks()
```

* **un plus : le positionnement.**

Usuellement, une fenêtre de dialogue est centrée sur l'écran ; elle peut aussi être positionnée par rapport à sa fenêtre conteneur. On a gardé les dux possibilités : si on renseigne **centrer** à **True**, il y a centrage de la fenêtre ; sinon la fenêtre est placée avec un offset de (250, 250), à moins que l'on ne redéfinisse **offx** et **offy**.

4.3. Un exemple d'application d'appel.

fichier : a._main_dialogue.py

```
from tkinter import *
from tkinter.messagebox import askyesno
from a3_module_dialogue import Dialogue

class MonDialogue (Dialogue) :

     def habillage (self, cadreConteneur) :
         Label(cadreConteneur, text = "Nom").grid (row=0)
         Label(cadreConteneur, text = "Prénom").grid (row=1)

         self.saisieNom = Entry(cadreConteneur)
         self.saisiePrenom = Entry(cadreConteneur)
         self.saisieNom.grid(row=0, column = 1)
         self.saisiePrenom.grid (row=1, column = 1)
         return self.saisieNom

     def apply (self) :
         un = self.saisieNom.get()
         deux = self.saisiePrenom.get()
         self.resultat = un+" "+deux
         labelResultat.configure(text = self.resultat)

# programme principal

def popDialogue ():
    d= MonDialogue(racine, titre = "module d'essais",
                   offx=-50, offy=-100)

def quitter () :
    if askyesno ("quitter l'application", "voulez-vous quitter ?") :
        racine.quit()


racine = Tk ()
racine.geometry("+300+200")

cadreRacine = Frame(racine)
cadreRacine.pack(padx=150, pady=100)
```

```
labelResultat = Label (cadreRacine, text="RESULTAT")
labelResultat.pack()

Button(cadreRacine, text = "Hello", \
       command=popDialogue).pack(padx=10,pady=10)

Button(cadreRacine, text = "stop", \
       command=quitter).pack(padx=10,pady=10)

# capture la demande de suppression
racine.protocol("WM_DELETE_WINDOW", quitter)
racine.mainloop()

racine.destroy()
```

* **from a3_module_dialogue import Dialogue**

La classe **Dialogue** est générique, et on l'importe pour l'étendre par la classe **MonDialogue**. La classe fille n'a pas de méthode __init__() ; c'est donc la méthode __init__() de la classe **Dialogue** qui est appelée sans qu'on ait besoin de l'invoquer. Mais à la création de l'instance, il faut paramétrer correctement.

* on notera l'utilisation d'un protocole pour la capture des commandes de destruction de l'application. Pour le dialogue, on remarque qu'il n'y a que l'icône de fermeture, que l'on fait jouer comme une annulation.

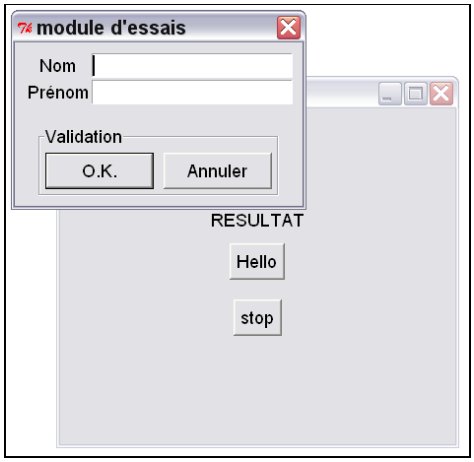
```
racine.protocol("WM_DELETE_WINDOW", quitter)
self.protocol("WM_DELETE_WINDOW", self.annuler)
```

Les copies d'écran montrent les fenêtres sur trois états :

* à droite, le clic sur Hello affiche la fenêtre de dialogue, avec ses zones de saisie.

* ci-dessous, la fenêtre principale après fermeture du dialogue.

* en bas à droite, la fenêtre oui/non pour quitter.




tk
Mercier Jean
Hello
stop


tk
quitter l'application
Mercier Jea
voulez-vous quitter ?
Hello
stop
Oui
Non

fiche 4 : fontes

introduction.

Tout système sur lequel on peut exécuter une application Python, principalement Window, Mac ou Linux a installé des fichiers de fontes (polices de caractères) disponibles pour les programmes qui peuvent être exécutés. Ceci concerne les programmes utilisant l'interface graphique plus spécialement, où les polices utilisées, avec les caractéristiques qu'on leur a affectées, comme la taille, l'épaisseur du trait, l'inclinaison, vont faire sens et participer à l'ergonomie. C'est à ces polices de caractères appartenant au système que l'on s'intéresse dans cette fiche (pas aux polices dont on peut disposer des fichiers, mais ne sont pas installées).

1. fonte dans les widgets.

1.1. configuration d'un widget.

La configuration des widgets comporte en général une clef **`font`**. Il a une valeur par défaut, est sert à tout l'affichage de txte du widget ; le cas le plus évident est celui des widgets **`Label`**. Les widgets **`Entry, Listbox, Menu, Message, Button`** etc utilisent également un affichage de texte.

La clef **`font`** peut être associée indifféremment à divers objets : soit des descripteurs de fonte, soit une instance de la classe **`Font`**. Il y a là une diversité liée à l'histoire du module graphique.

1.2. Descripteurs de fontes.

Décrire une fonte, c'est donner son nom (Arial, Comic Sans Ms, Courier New, Times New Roman, Helvetica), sa taille, ses attributs (bold, italic).

Deux problèmes : le nom peut comporter des espaces ; la taille peut être exprimée en pixels ou en points (le point vaut 1/72 de pouce, soit environ 0,35 mm). Pour la taille, si elle est en point, elle s'exprime par un entier positif ; une taille négative exprime une taille en pixels.

Un descripteur peut être une chaîne :

```
"{Comic Sans Ms} -24 bold"
"Arial 20 "
"Arial"
```

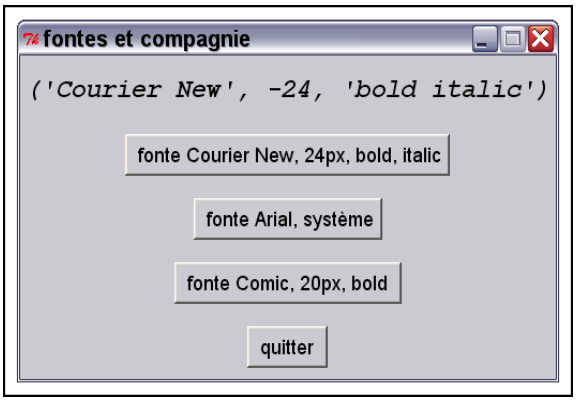
L'espace sert de séparateur : les accolades sont nécessaires les noms à espaces. Le descripteur n'est pas nécessairement complet.

Un descripteur peut être un nuplet :

```
("Comic Sans Ms" -24 "bold italique"
("Arial", 20)
```

Il existe encore d'autres façons de décrire une fonte … Mais les deux méthodes ci-dessus sont recommandables.

1.3. Un exemple.



fichier source :

fichier : a4_prg1_fontes.py

```
from tkinter import *

def quitter():
    racine.quit()

descCourier = ("Courier New", -24, "bold italic")
descArial = "Arial"
descComic = ("Comic Sans MS",-20,"bold underline")

def chgFonte(descripteur = descArial) :
    t = str(descripteur)
    label.configure(font=descripteur, text=t, width=len (t))

racine = Tk()
racine.title("fontes et compagnie")
racine.protocol("WM_DELETE_WINDOW", quitter) # capture de l'icone quitter

# Frame impose sa dimension s'il ne contient pas de pack() ni de grid()
cadre = Frame (racine, width=500, height=60)
cadre.pack()

label = Label(cadre)
chgFonte ()

label.place(relx= 0.5, rely=0.5, anchor=CENTER) # ne modifie pas le cadre

Button (racine, text = " fonte Courier New, 24px, bold, italic" ,\
        command = lambda x=descCourier : chgFonte(x)).\
        pack(padx=10, pady=10)

Button (racine, text = " fonte Arial, système" ,\
        command = lambda : chgFonte()).\
        pack(padx=10, pady=10)

Button (racine, text = " fonte Comic, 20px, bold " ,\
        command = lambda x = descComic : chgFonte(descComic)).\
        pack(padx=10, pady=10)

Button (racine, text = " quitter " ,\
        command = quitter).pack(padx=10, pady=10)

racine.resizable(0,0)
racine.mainloop()

racine.destroy()
```

* `def chgFonte(descripteur = descArial) :`
  `t = str(descripteur)`
  `label.configure(font=descripteur, text=t, width=len (t))`

Le texte du label est le descripteur de fonte. La largeur d'un label est exprimée en caractères !

* `command = lambda x=descCourier : chgFonte(x)`

L'utilisation d'une fonction **lambda** permet d'appeler une fonction avec paramètres comme argument de `command`.

* `racine.resizable(0,0)`

La fenêtre n'est pas modifiale, ni en hauteur, ni en largeur.

2. fonctions et méthodes du module "font".

2.1. le module font.

Dans la section précédente, on a utilisé les fontes, sans avoir à utiliser explicitement la classe `tkinter.font`. Les descripteurs appartiennent à `tkinter`, et il n'y a rien de plus à importer que ce module. On va ici utiliser le module `tkinter.font`, qui comporte des fonctions spécifiques et la classe `Font`, qu'il faut instancier pour créer un objet fonte.

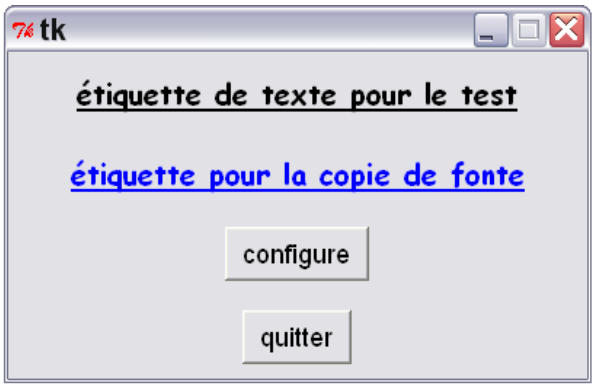
2.2. un script d'illustration.

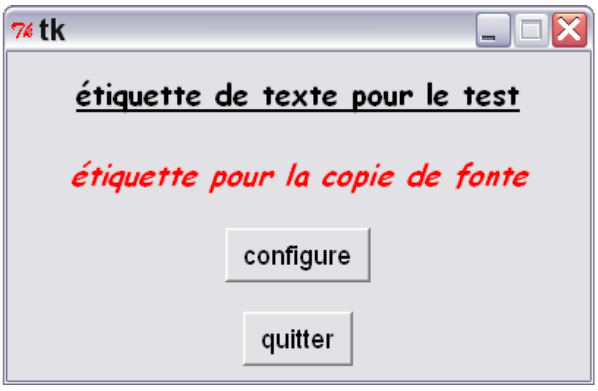
fichier a4_fontes_methodes.py

```
from tkinter import *
from tkinter.font import families, Font

def quitter():
    racine.quit()

def configurer () :
    # confiLabel["font"] ou confiLabel.cget("font")
    if confiLabel.cget("font") == str(ftComic) :
        copieFtComic = ftComic.copy() # change de fonte
        copieFtComic.configure (slant="italic", underline = False)
        confiLabel.configure (fg="red",\
                              font = copieFtComic) # reconfigurer !!!
    else :
        confiLabel.configure (fg="blue",font = ftComic)

racine = Tk() # obligatoire pour families()
racine.protocol("WM_DELETE_WINDOW", quitter)

rc = racine.__dict__
for x in rc.keys() :
    print (x,"===>>>",rc[x])  # le champ tk est utilisé dans families()
print ()

# famille de fontes disponibles dans l'application
# families() est une fonction du module tkinter.font
famille = families(racine)
```

```
famille = list(famille)
famille.sort() # nécesairement une liste, pas un tuple
for x in famille :
    print (x)
print()
# création d'une fonte par Font()

ftComic = Font (family = "Comic Sans MS", size = -20,\
                underline = True, weight = "bold")

Label (racine, text = "étiquette de texte pour le test",\
       font = ftComic).pack (padx= 40, pady = 10)
confiLabel = Label (racine, text = "étiquette pour la copie de fonte",\
       fg="blue", font = ftComic)
confiLabel.pack (padx= 40, pady = 10)
confi =Button (racine, text = " configure " ,\
        command = configurer)
confi.pack(padx=10, pady=10)
Button (racine, text = " quitter " ,\
        command = quitter).pack(padx=10, pady=10)

# propriétés de ftComic
print ("clef <font> de confiLabel :", confiLabel["font"])
# print ("fonte de confiLabel :",confiLabel.font)
print ("\n paramètres de ftComic ")
print (Font.actual(ftComic)) # retourne un dict
print ("largeur :",ftComic.measure ("papa est en voyage"), "pixels")
print ("hauteur :",Font.metrics(ftComic)["linespace"], "pixels")

racine.resizable(0,0)
racine.mainloop()
racine.destroy()
```

2.3. Copies d'écran.





2.4. Commentaires.

* **families()**

La fonction est importée de **tkinter.font** ; **comme son nom ne l'indique pas**, cette fonction recherche le nom des fontes installées dans le système. Elle les retourne sous forme d'un tuple. Comme on veut afficher par ordre alphabétique, il faut caster la réponse en liste.

Il faut passer en paramètre un objet Tk(), car c'est cette instance qui a le champ (le champ tk) donnant l'accès au fontes. Pour s'en assurer, on a affiché le dictionnaire de racine :

```
rc = racine.__dict__
for x in rc.keys() :
   print (x,"===>>>", rc[x])
```

```
_tclCommands ===>>> ['tkerror', 'exit', '15238088destroy', '15237568quitter']
master ===>>> None
children ===>>> {}
_tkloaded ===>>> 1
tk ===>>> <tkapp object at 0x00E73B48>
```

```
for x in famille :
    print (x)
```

```
8514oem
Arial
Arial Baltic
Arial Black
Arial Narrow
Blackadder ITC
Bookman Old Style
Bookshelf Symbol 1
Bookshelf Symbol 2
Bradley Hand ITC
Cartoon
Century Gothic
Comic Sans MS
Copperplate Gothic Bold
Copperplate Gothic Light
Courier
Courier New
.....
```

* **création d'instance.**

```
ftComic = Font (family = "Comic Sans MS", size = -20,\
                underline = True, weight = "bold")
```

On crée ici une instance de **Font**, et c'est elle qui va servir pour le label affiché. Mais attention, si pour l'usage usuel cela est équivalent à utiliser un descripteur, la représentation interne n'est pas identique et il faut y prendre garde si par exemple on compare deux attributs fontes, comme dans l'exemple suivant :

```
if confiLabel.cget("font") == str(ftComic) :
```

Il faut être cohérent : soit instance, soit descripteurs des deux côtés. Noter que **cget()** retourne la représentation en chaîne de caractères de l'instance, d'où le cast sur **ftComic**.

* **sur quelques méthodes :**

```
print (Font.actual(ftComic)) # retourne un dict
print ("largeur :",ftComic.measure ("papa est en voyage"), "pixels")
print ("hauteur :",Font.metrics(ftComic)["linespace"], "pixels")
```

```
 paramètres de ftComic
{'family': 'Comic Sans MS', 'weight': 'bold', 'slant': 'roman',
'overstrike': 0, 'underline': 1, 'size': 12}
largeur : 187 pixels
hauteur : 29 pixels
```

* **la méthode copy()**

On veut réaliser un second affichage (label), avec soit la même fonte que le premier, soit une seconde fonte obtenue par copie, puis modification. On vérifie en même temps que `copy()` crée une autre fonte. **Note :** La couleur appartient au widget, pas à la fonte.

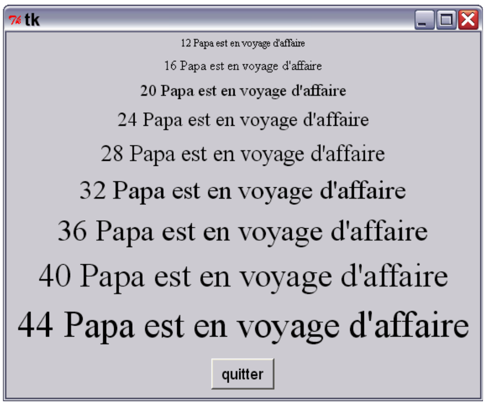
3. Fontes standard.

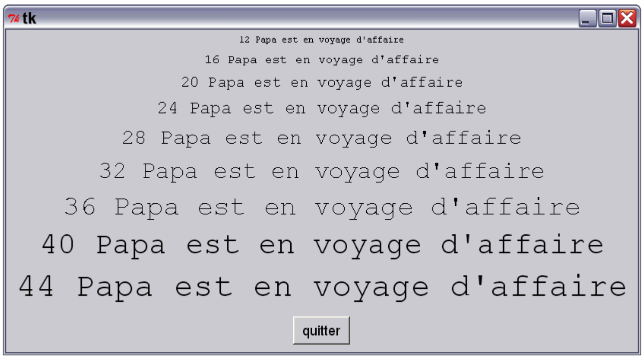
3.1. Une procédure sans erreur.

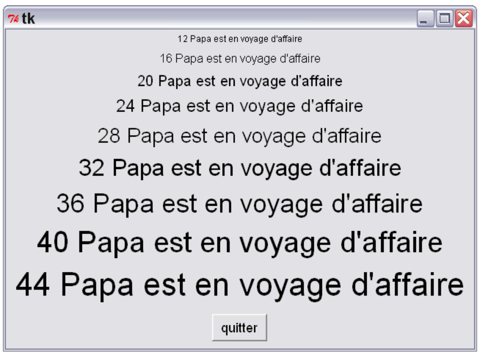
Que se passe-t-il si on fait appel à une fonte qui n'existe pas dans le système ? Ceci n'est pas un cas d'école, puisque autant qu'il est possible, une application Python est multi-plateforme.

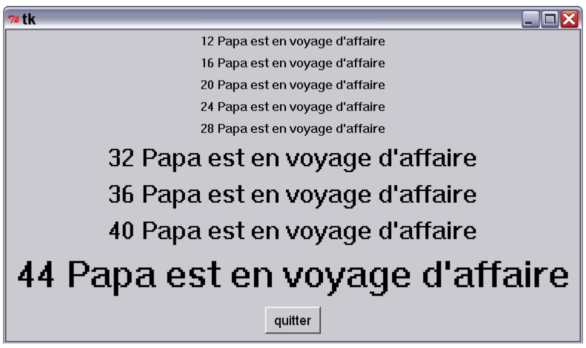
La règle est que Python ne déclenche pas d'erreur mais "tente" de remplacer la fonte inconnue par une autre "qui lui ressemble". On comprend les limites de ce procédé. Aussi Python a-t-il, comme Java, défini des fontes "passe-partout". En Java, il s'agit de noms génériques Serif, SansSerif, Monospaced… qui correspondent en Python à Times, Helvetica et Courier. Si une fonte de ce nom existe sur le système, elle est utilisée. Sinon, elle est remplacée, par une fonte "ressemblante", qui existe en standard sur tous les systèmes d'exploitation.

3.2. Illustrations.









Les saisies d'écran sont faites sous Windows. En haut : Times, Courier ; en bas : Helvetica et Sytem.

Les fontes réelles sont : **Times New Roman, Courier New, Arial** et **System.** On peut remarquer que bien qu'il existe une fonte **Courier** (fonte bitmap), c'est **Courier New** (front TTF) qui est choisie. Pour **System**, il s'agit de fontes bitmap ; seules les dimensions disponibles sont utilisées.

3.3. Le script.

fichier : a4_courier_helvetica_times.py

```
# équivalents pour le fontes
from tkinter import *

def quitter():
    racine.quit()

#police ="System"
#police ="Helvetica"
#police ="Times"
police ="Courier"
racine = Tk()
racine.protocol("WM_DELETE_WINDOW", quitter)
for i in range (12,48,4) :
    ft = (police, -i )
    testLabel = Label (racine, text =str(i)+\
                       " Papa est en voyage d'affaire", font=ft)
    testLabel.pack(padx=10, pady=2)

Button (racine, text = " quitter " ,\
        command = quitter).pack(padx=10, pady=10)
racine.mainloop()
racine.destroy()
```

4. Exercice utilisant une règle à curseur.

4.1. Un utilitaire.

Ce script est un utilitaire qui permet d'afficher un label avec une fonte donnée, mais en variant la dimension à l'aide d'une règle à curseur (widget Scale en Python).

4.2. Copie d'écran.

4.3. Le script.

fichier a4_scale_fonte.py

```
#!/usr/bin/python3

from tkinter import *

def quitter():
    racine.quit()

nomFonte ="Courier" # "Helvetica" "Times"
tailleFonte = -12

racine = Tk()
racine.protocol("WM_DELETE_WINDOW", quitter)
racine.title("Scale et Fonte")

def taille (event = None) :
    tailleFonte = - echelle.get()
    labelHello.config (font = nomFonte +" "+ str(tailleFonte) + " bold")

cadre = Frame (racine, width = 520, height = 250)
cadre.pack()

echelle = Scale (cadre, from_ = 10, to = 40, command=taille, \
                 orient=HORIZONTAL, length = 470, width =40,
                 resolution = 1, label="taille en pixels",showvalue=1,
                 tickinterval=5, sliderlength=20)
echelle.set (- tailleFonte)

echelle.place(anchor=CENTER, x=260, y=120)

labelHello = Label (cadre, text = "Bonjour tout le monde")
taille()
labelHello.place(anchor=CENTER, x=260, y=30)

Button (racine, text = " quitter " , \
        command = quitter).place(x=260, y=210, anchor=CENTER)

racine.mainloop()
racine.destroy()
```

```
* echelle = Scale (cadre, from_ = 10, to = 40, command=taille, \
                   orient=HORIZONTAL, length = 470, width =40,
                   resolution = 1, label="taille en pixels", showvalue=1,
                   tickinterval=5, sliderlength=20)
```

- `from_ = 10, to = 40` : les bornes de la règle
- `command=taille` : traitement de l'événement provoqué par le déplacement du curseur.
- `resolution = 1` : pas du curseur.

- **showvalue=1** : les valeurs de repérage sont inscrites le long de la règle.
- **tickinterval=5** : intervalle pour les repères de valeur : 10, 15, 20 etc.
- **sliderlength=20** : largeur du curseur.

* **echelle.set (- tailleFonte)** : fixe la taille initiale de la fonte (négative : en pixels, positive : en points).

* **tailleFonte = - echelle.get():** récupère la position du curseur après un déplacement et la transforme en taille de la fonte en pixels.

fiche 5 : quelques widgets

introduction.

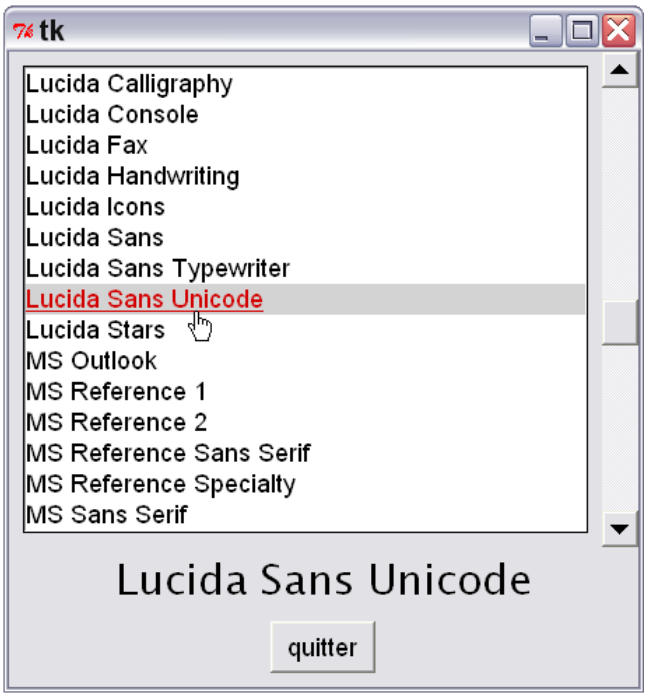
Il n'est pas question ici de détailler tous les widgets de tkinter. Un certain nombre ont été rencontrés : frame, canevas, label, bouton, règle à curseur, widgets de géométrie (pack, grid, place)… On va en voir encore quelques-uns ; pour une vue plus complète, on renvoie à la documentation, en priorité à "*an introduction to Tkinter*" de Frédéric Lundh (disponible sur internet en pdf).

1. Une boîte de liste avec un ascenseur.

1.1. Sujet.

On veut réaliser le script suivant, qui pourrait être intégré à un script plus "utile" par exemple un script de configuration de choix des polices Serif, SansSerif et Monospaced d'un logiciel où on a défini les fontes par ces trois identificateurs (comme en Java). Le script charge tous les noms de fonte du système, les affiches comme une liste à ascenseur et par un clic sélectionne un nom que, pour mémoire, on affiche avec la fonte donnée dans un label.

1.2. Copie d'écran.



2.3. Le script.

fichier : a5_listbox.py

```
#!/usr/bin/python3
# listbox, ascenseurs, fontes

from tkinter import *
from tkinter.font import families
```

```
def quitter():
    racine.quit()

def valider(event=None):
    # retour d'un-uplet de chaînes, ici un singleton
    index = listBoxFontes.curselection()
     if (index == ()):
        return
    index = int (index[0])
    nomDeFonte = listBoxFontes.get(index)
    # pour les noms avec espace : "{Courier New} -30"
    fonte = "{" + nomDeFonte + "} -30"
    affLabel.configure (text=nomDeFonte, font = fonte)

racine = Tk()
racine.protocol("WM_DELETE_WINDOW", quitter)

cadre = Frame (racine)
cadre.pack()
ascenseur = Scrollbar (cadre)
ascenseur.pack(side=RIGHT, fill=Y)


listBoxFontes = Listbox \
                (cadre, yscrollcommand=ascenseur.set,\
                 width = 40, height = 15, activestyle = "none",\
                 selectbackground="#d0d0d0",selectforeground="#d00000",\
                 selectmode=SINGLE, cursor="hand2")

fontes= list(families (racine))
fontes.sort()
for item in fontes :
    listBoxFontes.insert(END, item)

listBoxFontes.bind("<ButtonRelease-1>", valider) # relaché
listBoxFontes.pack(padx=10, pady=10)
ascenseur.configure(command=listBoxFontes.yview)

affLabel = Label (racine, text = "")
affLabel.pack()

Button (racine, text = " quitter " ,\
        command = quitter).pack(padx=10, pady=10)
racine.mainloop()
racine.destroy()
```

```
* listBoxFontes = Listbox \
```

```
(cadre, yscrollcommand=ascenseur.set,\
 width = 40, height = 15, activestyle = "none",\
 selectbackground="#d0d0d0",selectforeground="#d00000",\
 selectmode=SINGLE, cursor="hand2")
```

- **cadre** : La listbox et son ascenseur appartiennent à la frame nommée cadre.

- **width = 40, height = 15** : dimensions de la listbox en caractères (ici, le caractère par défaut, puisqu'on n'a rien redéfini en matière de fonte).

- **activestyle = "none"** : la ligne active est celle qui était active au moment du bouton pressé ; quand le bouton est relâché, la ligne active est celle qui contient le curseur au moment du relâchement (ou la dernière ligne marquée active si le relâchement se fait hors listbox). Pour éviter cette ambiguité, on a mis le paramètre à "none" (chaîne de caractères !) ; ce pourrait être souligné (valeur par défaut) ou encadré par des pointillés. Comme on a **selectmode=SINGLE**, c'est à dire "choisir un seul item", on peut supprimer cette marque.

- **selectbackground="#d0d0d0",selectforeground="#d00000"**, : couleur du fond et des caractère de la ligne choisie.

- **yscrollcommand=ascenseur.set**, : poser l'ascenseur (méthode : ascenseur.set()) comme commande de déplacement vertical de la liste dans sa listbox.

Note : cette opération de liaison d'un widget graphique et d'un ascenseur est possible pour certains widget seulement, Listbox, Canvas, Entry, mais pas pour Frame.

* **listBoxFontes.bind("<ButtonRelease-1>", valider)**

Python ne connaît pas le clic simple de souris. On doit donc faire attention au processus : le bouton de souris enfoncé (**<Button -1>** ou **<ButtonPress-1>**) déplace la sélection dans la liste. Si on se contente d'un clic, comme celui-ci est équivalent à "bouton enfoncé", la sélection nouvelle n'est pas détectée ; il faut donc attendre la fin de la sélection pour la traiter et donc utiliser la "bouton relâché".

* **index = listBoxFontes.curselection()**

La méthode retourne le numéro de la ligne sélectionnée dans la liste. En fait, cette méthode, prévue pour les sélection multiples, retourne un tuple dont les éléments sont le numéros sous forme de chaîne. Ce qui explique la forme de la condition **index == ()**, vérifiée si aucune ligne n'est sélectionnée, et le cast d'une valeur du tuple undex : **index = int (index[0]).**

* **nomDeFonte = listBoxFontes.get(index)**

On trouve ici le nom sélectionné sous forme d'une chaîne. Attention : Un nom de fonte n'est en général pas un descripteur ; d'où la fabrication du descripteur :

```
fonte = "{" + nomDeFonte + "} -30"
```

```
* ascenseur = Scrollbar (cadre)
  ascenseur.pack(side=RIGHT, fill=Y)
```

L'ascenseur appartient au widget cadre. Il est placé à droite (**side=RIGHT**) et occupe tout l'espace vertical de son propriétaire (**fill=Y**).

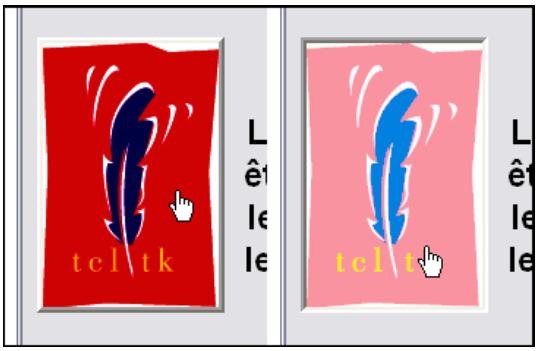
2. Un bouton avec icone.

2.1. Sujet.

Il s'agit de réaliser un bouton à trois états : inactif, survolé, pressé à l'aide d'un label ; ce bouton appelle une fenêtre de dialogue définie dans la fiche 2. Cette fenêtre est centrée, elle n'a pas de focus par défaut car elle présente en habillage une icone et un label qui sont inerte. Le but de ce dialogue est de demander s'il faut ou non remplacer l'image du bouton/label par celle présentée sur la fenêtre de dialogue. Puis d'effectuer ce changement le cas échéant.

2.2. Copies d'écran.

* **le bouton/label en survol et avec bouton de souris pressé**



* **la fenêtre de dialogue et ouverte.**

2.3. Le script.

fichier a5_label_img.py

```
#!/usr/bin/python3
# utilisation de la fenêtre popup définie dans la fiche 2
# gros bouton construit.

from tkinter import *
from tkinter.messagebox import askyesno
from a3_module_dialogue import Dialogue

# classe de la fenêtre de dialogue
class PopUp (Dialogue) :
    def habillage (self, cadreConteneur) :
        Label(cadreConteneur, text = "voici le logo proposé :")\
              .grid (padx=10, pady=10, row=0)
```

```
        if flagRB :
            limage = logoB
        else :
            limage = logoR
        Label(cadreConteneur, image = limage)\
                                .grid (padx=10, pady=10, row=1)
        return None # pas de focus par défaut sur le cadre

    def apply (self) : # exécuté uniquement si OK
        global flagRB, logo, logoP, logoO # il y a effet de bord
        if flagRB :
            logo = logoB
            logoP = logoBP
            logoO = logoBO
        else :
            logo = logoR
            logoP = logoRP
            logoO = logoRO
        labelImg.configure(image=logo)
        flagRB = not flagRB

# fonctions de l'application

def quitter():
    rep = askyesno ("message oui/non", "voulez-vous quitter ?")
    if rep :
        racine.quit()

def logoPress(event=None):
    labelImg.configure(relief=SUNKEN, image=logoP)

def logoRelease(event=None):
    labelImg.configure(relief=RAISED, image=logo)
    PopUp (racine, titre = "choix d'icone", centrer = 1)

def logoSurvol(event=None):
    labelImg.configure(image=logoO)

def logoLaisse(event=None):
    labelImg.configure(relief=RAISED, image=logo)

# l'application
racine = Tk()
racine.protocol("WM_DELETE_WINDOW", quitter)

# un cadre pour le bouton et le texte
cadre = Frame(racine)
cadre.pack(padx=10, pady=10)
```

```
# chargement des logos
logoR = PhotoImage (file="logo_tk.gif")
logoRP = PhotoImage (file="logo_tkpress.gif")
logoRO = PhotoImage (file="logo_tkover.gif")
logoB = PhotoImage (file="logobleu_tk.gif")
logoBP = PhotoImage (file="logobleu_tkpress.gif")
logoBO = PhotoImage (file="logobleu_tkover.gif")

#initialisation
flagRB = True
logo = logoR
logoP = logoRP
logoO = logoRO

# bouton construit
labelImg = Label (cadre, image = logoR, borderwidth=4, \
                  relief=RAISED, cursor="hand2")
labelImg.bind ("<ButtonPress-1>", logoPress)
labelImg.bind ("<ButtonRelease-1>", logoRelease)
labelImg.bind ("<Enter>", logoSurvol)
labelImg.bind ("<Leave>", logoLaisse)
labelImg.pack( pady=10, side=LEFT)

# message
message = "\nLe bouton avec une icone doit \n\
être cliqué si on veut remplacer\n\
le logo par celui proposé dans \n\
le fenêtre popup qui apparaîtra."
fonte = ("Arial", -24, "bold")
Label(cadre, text=message, font=fonte ).pack(padx=10, side=RIGHT)

# quitter
Button (racine, text = " quitter " ,\
        command = quitter, cursor="hand2").pack(padx=10, pady=10)

racine.mainloop()
racine.destroy()
```

```
* logo = logoR
  logoP = logoRP
  logoO = logoRO
Pour réaliser les effets d'icône, on remplace l'icône de départ par deux
autres, adaptées pour le survol et l'activation.
```

Important : Les images sont obligatoirement créées après la définition de la fenêtre principale. C'est une exigence de tkinter. Mais si on essaie de les charger dans une fenêtre popup, on risque d'éprouver des difficultés : le chargement est incomplet car le fichier n'est pas complètement lu au moment de la routine d'affichage. Une bonne précaution consiste à charger les images dès le départ de l'application, dans le programme principal.

```
* labelImg.bind ("<Enter>", logoSurvol)
```

Les événements qui peuvent se produire sur le bouton/label sont : souris pressée, souris relâchée, souris entrante (Enter), souris sortante.

```
*global flagRB, logo, logoP, logoO # il y a effet de bord
```

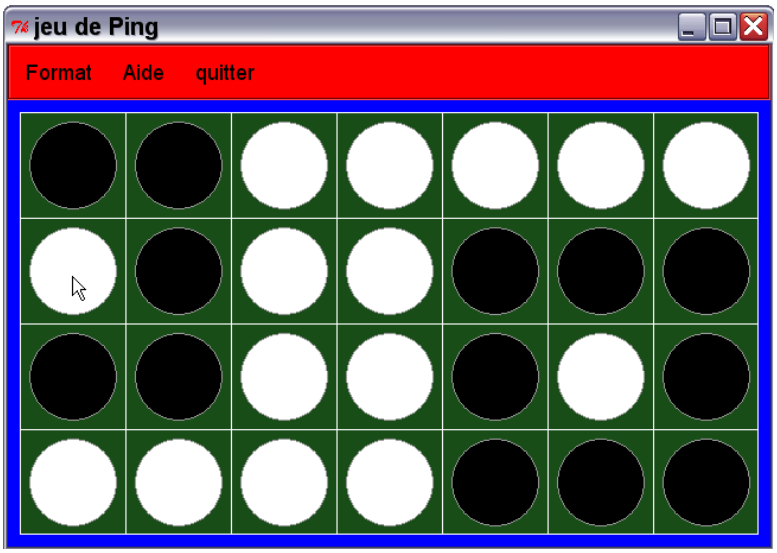
Ne pas oublier la déclaration `global` **sur les variables globales modifiées.**

3. Une application complète : le jeu de Ping.

3.1. Cahier des charges.

On se propose de réaliser un jeu de Ping (pour les règles, voir la copie d'écran ci-dessous) avec les contrainte suivantes :

- Le nombre lignes et de colonnes peut être choisi .
- La fenêtre principale est redimensionnable.
- La grille du jeu est automatiquement mise à l'échelle lorsque l'on change de partie et lorsque l'on redimensionne la fenêtre principale.
- Une aide popup et disponible pour donner la règle du jeu.
- Les divers services (nouvelle partie, reset, aide, copyright, fin de session) sont accessibles par menus



3.2. Redimensionner.

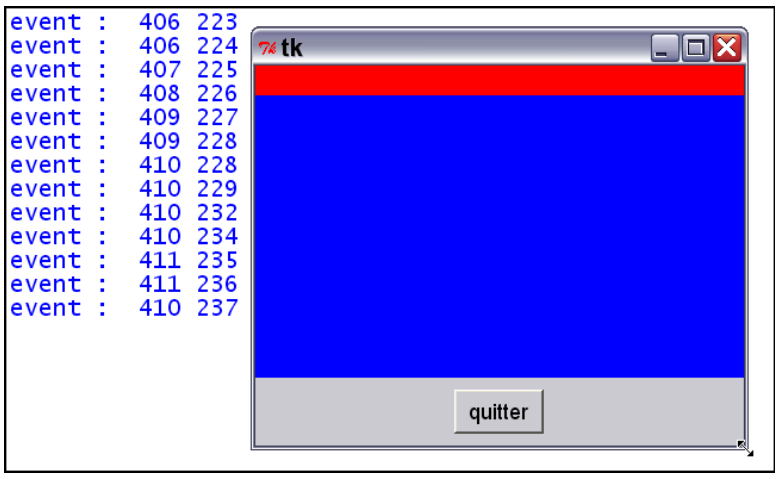
Dans toute les interfaces graphiques, le redimensionnement se fait en "tirant" la fenêtre de l'application à l'aide la souris pressée sur un de ses bords (quand il y a mise à l'échelle, un coin).

Voici le principe du redimensionnement pour l'application ci-dessus :

```
from tkinter import *

def quitter():
    racine.quit()
```

```
def aff(event) :
    print ("event : ", event.width, event.height)

racine = Tk()
racine.protocol("WM_DELETE_WINDOW", quitter)
racine.geometry("400x300")

haut = Frame (racine, bg ="red", height = 25, width =100)
haut.pack(fill=X)

fond = Frame (racine, bg ="blue")
fond.pack(fill=BOTH, expand=1)
fond.bind ("<Configure>", aff)

b1=Button (racine, text = " quitter ", command = quitter)
b1.pack(padx=10, pady=10)

racine.mainloop()
racine.destroy()
```

Ce fichier correspond à l'affichage qui suit, où la fenêtre principale est à côté de la console qui permet de repérer les affichages du **`print()`**.



* la zone rouge correspond à un cadre (**`Frame`**), de dimensions **`height`** et **`width`** pécisées. Tkinter ne respecte ce type d'attribut que si une commande ultérieure n vient pas le perturber. Or ici, on a la commande **`pack(fill=X)`** qui stipule que le cadre doit remplir tout l'espace horizontal allouable ; l'attribut **`height`** est respecté dans tout redimensionement (voit le curseur dans le coin bas/droite), mais pas nécessairement width.

* **`racine.geometry("400x300")`** : la dimension de la fenêtre est imposée (les valeurs 400x300 concernent la zone "utile" de la fenêtre, sans encadrement ni barre de titre). Elle est respectée tant qu'une commande ne vient pas la contrarier. Ce qui sera le cas quand on va redimensionner, mais pas au départ.

* En effet, le cadre bleu n'a aucune dimension imposée. Seule sa commande de géométrie lui

donnera ses dimensions : **`fond.pack(fill=BOTH, expand=1)`**.

L'attribut **`fill`** a pour valeur **`BOTH (X et Y)`**. Pour le remplissage horizontal, il n'y pas de concurrence : tout l'espace horizontal est occupé. Ce n'est pas le cas pour l'espace vertical et l'attribut **`fill`** tout seul n'impose pas d'autre contrainte : le gestionnaire accorde 1 pixel à 'espace vertical. A moins que pack comporte un attribut qui contraigne à faire le forcing : **`expand = True`** (ou tout équivalent à **`True`**). Le cadre bleu prend alors tout ce qui reste une fois servis les deux autres widgets, le cadre rouge et le bouton "quitter".

* **`fond.bind ("<Configure>", aff)`**

Un événement est lié au cadre bleu : chaque fois qu'il est redimensionné, cet événement nommé "configure" se produit. Parmi les valeurs constitutives de cet événement, on trouve longueur et largeur de l'élément appelant, soit ici le cadre bleu. Ce sont ces dimensions qui sont affichées dans la console par la fonction **`print()`**.

3.3. Des fenêtres popup.

Un fenêtre d'aide avec les règles du jeu et un fenêtre "à propos" doivent apparaître en tant que fenêtre modale. Il en est de même avec les paramètres (nombre de lignes et de colonnes). Le problème a été rencontré dans la fiche 2 ; on avait créé un module qui comportait une classe de dialogue.

Ce module est tout à fait adapté à la saisie des paramètres. L'habillage est fait de deux règles à curseur, avec une échelle allant de 1 à 12. Comme il n'y a pas nécessité de fixer le focus sur un élément particulier, la méthode habillage() doit retourner **`None`**. Le retour de la fenêtre de dialogue peut se produire sur OK ; la méthode apply() est exécutée et donne un tuple, comportant les deux paramètres, affecté à **`self.resultat`**. En cas d'abandon, self résultat est initialisé aux valeurs du jeu lors de l'appel.

Par contre même si le type de fenêtre est semblable (fenêtre popup modale), il y a quelques différences qui rendent plus simple le module nécessaire : pas de focus, un seul bouton, pas de résultat. Il est don intéressant de définir un nouveau module, calqué sur le module de dialogue mais simplifié.

fichier du module : a5_module_aide.py

```
from tkinter import *

# ****************************************************************
# ce module définit une classe, celle d'une fenêtre de dialogue  *
# ****************************************************************
class PopAide (Toplevel) :
    # le constructeur
    # ###############
    def __init__ (self, conteneur, titre = "PopAide", offx=200, offy=0) :

        # associer la fenêtre d'aide et son conteneur
        # conteneur = fenêtre qui ouvre le Dialogue
        Toplevel.__init__(self, conteneur)

        self.protocol("WM_DELETE_WINDOW", self.terminer)

        self.title(titre)  # title() est hérité de Toplevel

        # le dialogue apparaît au dessus de son conteneur
        self.transient (conteneur)
        # détourne tous les événements vers la fenêtre de dialogue
```

```
        self.grab_set()

        # afficher la fenêtre d'aide par rapport au conteneur
        self.geometry ("+"+str(conteneur.winfo_rootx()+offx)+\
                       "+"+str(conteneur.winfo_rooty()+offy))

        frameContenu = Frame (self, border=2, relief=GROOVE)

        # définition d'un contenu du cadre frameContenu
        self.habillage(frameContenu)
        frameContenu.pack ()

        # bouton d'effacement
        boutonTerminer = Button (self, text = "terminer", width = 12,\
                                 command = self.terminer)
        boutonTerminer.pack (pady=5)
        self.bind("<Return>", self.terminer)
        self.bind("<Escape>", self.terminer)

        # porter le focus sur  la fenêtre d'appel
        self.focus_set ()
        # boucle locale de la fenêtre
        self.wait_window (self)

    def habillage (self, frame) :
        pass

    def terminer (self, evenement = None) :
        self.master.focus_set()
        self.destroy()
```

3.4. Le fichier source.

introduction.

fichier : a5_ping.py

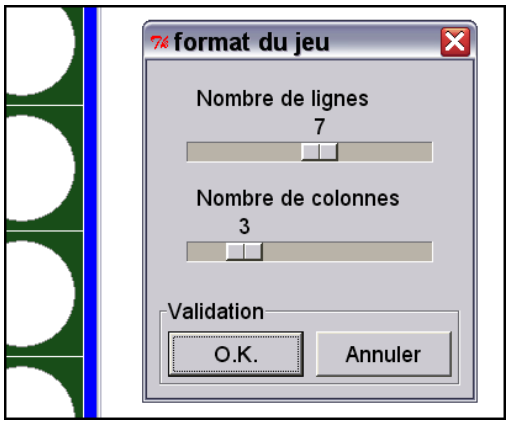
```
# jeu de ping (cf Swinnen 15.2)

from tkinter import *
from tkinter.messagebox import askyesno
from a3_module_dialogue import Dialogue
from a5_module_aide import PopAide

# globales
initLg, initCl = 4, 4 # lignes et colonnes du début/reset
appli = None # l'application TK
jeu = None # le jeu de Ping (c'est un Frame)
logoImg = None # PhotoImage(file="logo_colombe.gif")
```

```
pad = 10 # pad de la grille sur son cadre
marge = 2 * pad
bgMenu ="red"         # les couleurs sont criardes
bgPing = "blue"       # mais d'une part elles sont faites
bgGrille = "#408040" # pour être adaptée, et d'autre part
bgAide= "green"       # leur choix relève du repérage des
bgGlose ="yellow"     # composants sur le source

def quitter(event=None):
    confirmer = askyesno ("confirmer", "voulez-vous quitter ?")
    if confirmer :
        appli.quit()
def popAide () : # menu AIDE
    pageAide = Aide (appli)
def popGlose () : # menu A PROPOS
    pageGlose = Glose (appli, offx=50, offy=-20)
```

* Les valeurs qui seront utilisées dans tout le script sont précisées dès l'entrée ; leur initialisation est réalisée au cours du développement. On insiste sur le fait que la définition de l'image du logo est globale, initialisée dans le programme principal (voir § 2).

saisie du nombre de lignes et colonnes

```
# classe de la fenêtre de saisie du nombre de lignes et colonnes
# ---------------------------------------------------------------
class PopOptions (Dialogue) :
    def habillage (self) : # self = le Dialogue
        chb = self.cadreHabillage
        self.lignes = Scale (chb, length = 200,\
                        label = "Nombre de lignes", orient = HORIZONTAL,
                        from_ = 1, to = 12 )
        self.lignes.set (jeu.lignes) # position du curseur lignes
        self.lignes.pack(padx=10, pady=5)

        self.colonnes = Scale (chb, length = 200,\
                        label = "Nombre de colonnes", orient = HORIZONTAL,
                        from_ = 1, to = 12 )
        self.colonnes.set (jeu.colonnes)  # position du curseur colonnes
        self.colonnes.pack(padx=10, pady=5)

        # pour le cas "abandonner" : initialiser self.resultat :
        self.resultat = ( jeu.lignes , jeu.colonnes )
        return None # pas de focus sur l'habillage

    def apply (self) :
        self.resultat = (self.lignes.get(), self.colonnes.get())
```



la barre de menu.

```
# définition de la barre de menu
# ------------------------------
class BarreDeMenu (Frame) :
    def __init__ (self, fenApp) :  # self = barre de menu
        Frame.__init__ (self, borderwidth=2, relief = GROOVE, bg=bgMenu)
        ### menu Format
        menuFormat = Menubutton (self, text = "Format",
                                 bg = self.cget("bg"))
        menuFormat.pack(side = LEFT, padx=5, pady=5)
        ####### les items du menu Format
        itemsMenuFormat = Menu (menuFormat, tearoff=False)
        itemsMenuFormat.add_command (label = "options",
                                     command=self.options )
        itemsMenuFormat.add_command (label = "reset",
                                     command=self.reset )
        itemsMenuFormat.add_command (label = "quitter", command=quitter)
        menuFormat.configure (menu = itemsMenuFormat)

        ### menu Aide
        menuAide = Menubutton (self, text = "Aide", bg = self.cget("bg") )
        menuAide.pack(side = LEFT, padx=5, pady=5)
        ####### les items du menu Aide
        itemsMenuAide = Menu (menuAide, tearoff=False)
        itemsMenuAide.add_command (label="règles du jeu", command=popAide)
        itemsMenuAide.add_command (label="à propos...", command=popGlose)
        menuAide.configure (menu = itemsMenuAide)

        ### bouton quitter ; seulement le bouton
        menuQuitter = Menubutton (self, text="quitter", bg=self.cget("bg"))
        menuQuitter.bind("<Button-1>", quitter)
        menuQuitter.pack(side = LEFT, padx=5, pady=5)

    def options (self) :
        jeu.formatJeu ()
```

```
        jeu.initEtat()
        jeu.redimensionner()

    def reset (self) :
        jeu.lignes = initLg
        jeu.colonnes = initCl
        jeu.initEtat()
        jeu.redimensionner()
```

* La barre de menu ne présente pas de grosse difficulté. Il faut noter une particularité : **tearoff=False**. L'attribut **tearoff** donne la possibilité de transformer le menu fixe en fenêtre de menu déplaçable. La valeur par défaut est **True**.

La grille du jeu (début)

```
# définition de la grille de jeu
# ------------------------------
class Ping (Frame) :
    # constructeur
    def __init__ (self, fenApp) : # self = le jeu ;
        Frame.__init__ (self)
        appli.title("jeu de Ping")
        self.lignes = initLg
        self.colonnes = initCl
        self.configure (bg=bgPing)

        # événement lié à la variation du cadre du jeu
        self.bind ("<Configure>", self.redimensionner)

        # le canevas sur son cadre
        self.grille = Canvas (self, background = bgGrille,
                              borderwidth = 0, highlightthickness = 1,
                              highlightbackground = "white")
        self.grille.pack(padx=pad, pady = pad)

        # événement : clic sur les pions
        self.grille.bind ("<Button-1>", self.clic)

        # état initial du jeu
        self.initEtat()
```

```
        def formatJeu (self) :
        j = PopOptions (self.master, centrer = 1, titre = "format du jeu")
        self.lignes = j.resultat [0]
        self.colonnes = j.resultat [1]

    def initEtat (self) :
        # tableau qui mémorise l'état du jeu
        self.etat = []
        for i in range(0, self.lignes):      # lignes
            u = []
            for j in range(0, self.colonnes): # colonnes
                u.append(0)
            self.etat.append(u)
```

* Le "jeu", instance de la classe **`Ping`** est un cadre (en bleu sur les saisies d'écran) avec de champs adjoints : le nombre de lignes et de colonnes, un canevas (la grille du jeu), un champ **`etat.`**

* le champ **`etat`** décrit sur un simulation de tableau d'entiers (0 ou 1) réalisé par une liste de liste. On renvoie à la fiche sur les listes pour l'initialisation. L'erreur classique consiste à définir une ligne sous la forme u=[0,0,0,0,0,0,0,0,0,0,0,0] puis à la dupliquer 12 fois :

```
u = u=[0,0,0,0,0,0,0,0,0,0,0,0]
etat = []
for j in range (12) :
    etat.append (u)
```

* **`self.bind ("<Configure>", self.redimensionner)`**

Le "jeu" suit le redimensionnement de la fenêtre principale. La grille incluse dans le cadre "jeu" se recalcule en fonction des dimensions actuelles du cadre.

La grille du jeu (suite)

```
    def redimensionner (self, event=None) :
        if event :
            self.width = event.width
            self.height = event.height

        # dimension de la case et du cannevas
        longMax = (self.width - pad) // self.colonnes
        hautMax = (self.height - pad) // self.lignes
        case = self.case = min (longMax, hautMax)
        large = self.large = case * self.colonnes
        haut = self.haut = case * self.lignes

        self.grille.configure (width = self.large, height =self.haut)

        # remettre l'application à dimensions
        self.unbind ("<Configure>")
        g = str(self.large + marge)+"x" \
```

```
                str(menu.winfo_height()+self.haut+marge)
        appli.geometry (g)
        appli.update() # application immédiate de la géométrie
        self.bind ("<Configure>", self.redimensionner)

        self.traceGrille()

        # fin de redimensionnement de l'application

    def traceGrille(self) :
        # alias
        case = self.case
        large = self.large
        haut = self.haut
        grille = self.grille
        lignes = self.lignes
        colonnes = self.colonnes

        # tout effacer
        grille.delete(ALL)

        # tracé du quadrillage
        for k in range (1, lignes) :
            grille.create_line (0, case*k, large, case*k, fill = "white")
        for k in range (1, colonnes) :
            grille.create_line (case*k, 0, case*k, haut, fill = "white")

        # tracer les pions
        for lg in range (lignes) :
            for cl in range (colonnes) :
                x1 = cl * case + 8
                x2 = (cl + 1) * case - 8
                y1 = lg * case + 8
                y2 = (lg + 1) * case - 8
                couleur = ["white", "black"][self.etat[lg][cl]]
                grille.create_oval (x1, y1, x2, y2, \
                       outline="grey",width = 1, fill = couleur)

        # fin de tracerGrille()
```

```
*       self.unbind ("<Configure>")
        g = str(self.large + marge)+"x" \
                str(menu.winfo_height()+self.haut+marge)
        appli.geometry (g)
        appli.update() # application immédiate de la géométrie
        self.bind ("<Configure>", self.redimensionner)
```

Une fois calculée la grille, il faut adapter son cadre ; en effet, seule une des dimensions du cadre a été prise en considération ! Il faut donc déconnecter le gestionnaire d'événements, sinon c'est la boucle sans fin. La dimension de la fenêtre est calculée et appliquée. Seulement la réalisation de l'affichage

n'est jamais prioritaire dans une application graphique : il faut la forcer par le `update()`
On peut alors reconnecter le gestionnaire d'événements.

* `couleur = ["white", "black"][self.etat[lg][cl]]`

On voit ici comment le tableau d'état permet l'affichage.

La grille du jeu (fin)

```
    def clic(self, event) :
        # trouver la ligne et la colonne du clic
        ligne = event.y // self.case
        colonne = event.x // self.case
        maxLg = self.lignes - 1
        maxCl = self.colonnes - 1

        # les 8 cases adjacentes (ou moins...)
        for lg in range (ligne - 1, ligne + 2) :
            if (lg < 0) or (lg > maxLg ) :
                continue

            for cl in range (colonne-1, colonne+2) :
                if (cl< 0) or (cl > maxCl) :
                   continue
                if (lg==ligne) and (cl==colonne) :
                    continue
                # retourner
                self.etat[lg][cl] = (self.etat[lg][cl]+1) % 2

        # retracer la grille
        self.traceGrille ()

        # fin du gestionnaire de clic sur les pions
```

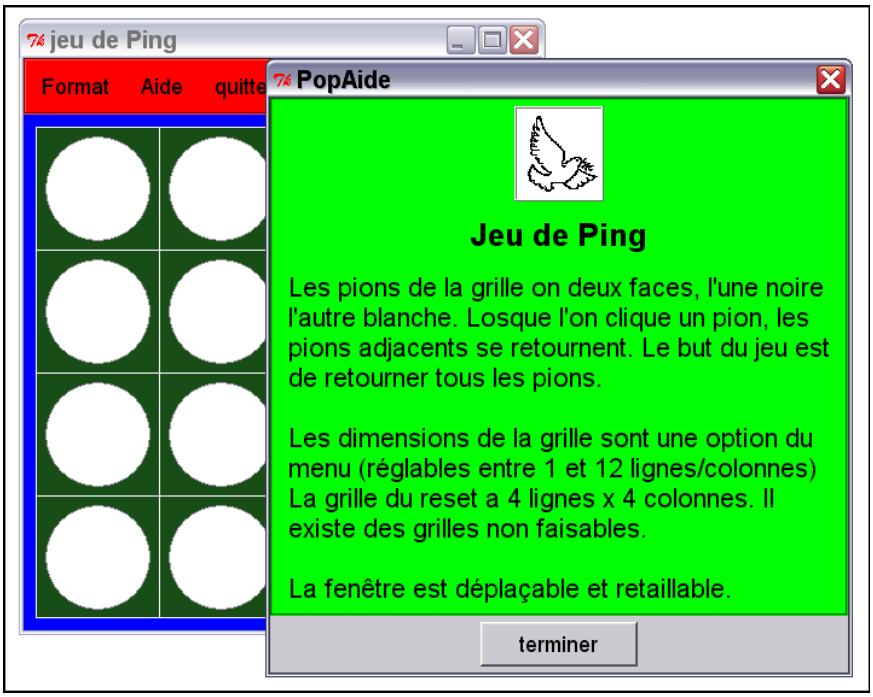
* `self.etat[lg][cl] = (self.etat[lg][cl]+1) % 2`

On trouve ici la mise à jour du tableau d'état. On aurait pu faire un tableau de booléens avec la meme logique.

La fenêtre d'aide.

```
class Aide (PopAide) :
    # l'aide apparaît dans une fenêtre pop, issue de la classe PopAide

    def habillage (self, frame) : # surcharge de la classe parent
        # couleur de fond
        frame.configure (bg=bgAide)

        # image d'illustration
        labelImage = Label(frame, relief=RIDGE, borderwidth=2,\
                           image = logoImg)
```

```
        labelImage.pack (pady=5)

        # titre de l'application
        labelTitre = Label(frame,  text="Jeu de Ping",\
                           bg=frame.cget("bg"),\
                           font=("helvetica",-26, "bold"))
        labelTitre.pack (pady=5)

        # texte de l'AIDE
        regles ="Les pions de la grille on deux faces, l'une noire\n"+\
                "l'autre blanche. Losque l'on clique un pion, les\n"+\
                "pions adjacents se retournent. Le but du jeu est\n"+\
                "de retourner tous les pions.\n\n"+\
                "Les dimensions de la grille sont une option du\n"+\
                "menu (réglables entre 1 et 12 lignes/colonnes)\n"+\
                "La grille du reset a 4 lignes x 4 colonnes. Il \n"+\
                "existe des grilles non faisables.\n\n"+\
                "La fenêtre est déplaçable et retaillable."

        labelTexte = Label(frame,  text=regles, justify=LEFT,\
                     bg=frame.cget("bg"), font=("helvetica",-22))
        labelTexte.pack (padx=10, pady =5)

        # fin de l'AIDE
```



La grille de l'à propos

```
class Glose (PopAide) :
    # l'aide apparaît dans une fenêtre pop, issue de la classe PopAide

    def habillage (self, frame) : # surcharge de la classe parent
        # couleur de fond
        frame.configure (bg=bgGlose)

        # image d'illustration
        labelImage = Label(frame, relief=RIDGE, borderwidth=2,\
                           image = logoImg)
        labelImage.pack (pady=5)

        # titre de l'application
        labelTitre = Label(frame,  text="Jeu de Ping",\
                           bg=frame.cget("bg"),\
                           font=("helvetica",-26, "bold"))
        labelTitre.pack (pady=5)

        # texte de l'A PROPOS
        glose ="Le jeu de Ping est un classique. Une étude complète\n"+\
                "peut être trouvée à l'adresse suivante : \n"+\
                "www2.lifl.fr/~delahaye/dnalor/JeuAEpisodes.pdf \n\n"+\
                "Réalisé le 20 août 2009 pour l'atelier Python  \n"+\
                "du club AmiPoste-Télécom de Mont-Saint-Éloi \n\n"+\
                "Le livre de  Gérard Swinnen, \"Apprendre à\n"+\
                "programmer avec Python\" comporte une autre \n"+\
                "réalisation du jeu.\n\n - - - - Jean Mercier - - - -"

        labelGlose = Label(frame,  text=glose,\
                     bg=frame.cget("bg"), font=("helvetica",-18))
        labelGlose.pack (padx=10, pady =5)

        # fin de l'A PROPOS
```

Le programme

```
# ##################################################################
#                        APPLICATION
# ##################################################################
appli = Tk()
appli.protocol("WM_DELETE_WINDOW", quitter)
appli.geometry ("420x500+200+200")

logoImg = PhotoImage(file="logo_colombe.gif")

menu = BarreDeMenu(appli)
menu.pack(fill=X, side=TOP)

jeu = Ping(appli)
jeu.pack(fill = BOTH, expand = 1)

appli.mainloop()
appli.destroy()
```

* `appli.geometry ("420x500+200+200")`

On rappelle que les dimensions 420x500 s'appliquent à la partie "utile" de la fenêtre. Les valeurs +200+200 sont les coordonnées, en pixels, du coin supérieur gauche de la fenêtre (barre de titre).

atelier 6 : le gestionnaire de géométrie "place"

introduction

Cette fiche a pour objet d'illustrer l'utilisation du gestionnaire de géométrie **`place`**. Une annexe est consacrée à la réalisation d'une distribution exécutable sous Windows sur une machine où il est impossible d'installer Python (pour des raisons de droits par exemple).

1. L'exercice.

1.1. cahier des charges.

Le problème est de simuler un grand nombre d'expériences dans une situation où la probabilité théorique d'obtenir un certain résultat permet d'exhiber une approximation de pi. Historiquement, on connaît le problème de Buffon, qui consiste à faire tomber des aiguilles sur un parquet latté ; la probabilité qu'une aiguille coupe un raccord s'exprime comme une fraction de pi. Un grand nombre d'expériences donne un résultat tel que si on divise le nombre de cas favorables (l'aiguille chevauche le raccord) par le nombre d'essai, on approxime le résultat théorique. Une telle méthode d'approximation est appelée une méthode de Monte-Carlo.

la simulation :

On suppose que l'on dispose d'une cible carrée dans laquelle est inscrit un cercle. Le tir dans la cible est supposé aléatoire et uniforme, c'est-à-dire que tout point de la cible peut être atteint avec la même probabilité. Tout tir atteint la cible. Dans ces conditions, la probabilité d'atteindre le cercle est proportionnelle à sa surface, sachant que la probabilité d'atteindre le carré est 1 (événement certain). Si on suppose un cercle de rayon unité, sa surface est **pi** unités de surface, alors que celle du carré est 4 unités de surface. La probabilité d'atteindre le cercle est pi/4.

le cahier des charges :

* on demande de réaliser une simulation des tirages et en déduire une approximation de pi.
* il faut se donner la possibilité de faire un nombre de tirages quelconque (raisonnable),
* et faire une interface graphique avec la possibilité de choisir 1, 10, 100, 1000 tirages en cumulant les résultats.
* l'interface graphique doit montrer la simulation d'une séquence de tirs par des points sur une figure carrée inscrivant un cercle et en afficher le score après chaque "salve".
* on doit quitter proprement le logiciel.
* on doit pouvoir effectuer un reset.
* la mise à dimensions de l'interface en fonction de l'écran est automatique.
* la fenêtre de l'interface graphique est centrée sur l'écran.
* une partie de l'espace de la fenêtre est réservée pour spécifier l'origine du logiciel (logo).

note : le cahier des charges est adapté d'un exercice (numéro t4q7) proposé dans le challenge 2010 des clubs Microtel par le club organisateur Caen-Hérouville.

1.2. Plan de travail.

* L**a partie simulation** et la partie interface graphique sont largement indépendantes ; on peut même réaliser la simulation sans l'interface. Aussi une première partie comporte uniquement la simulation. C'est la classe **`PiSimulation`**.

* **L'interface graphique** est la classe **`Application`**. C'est elle qui règle toute l'interface et la sortie du logiciel.

* L'interface graphique comporte **quatre panneaux** (**`Frame`**).

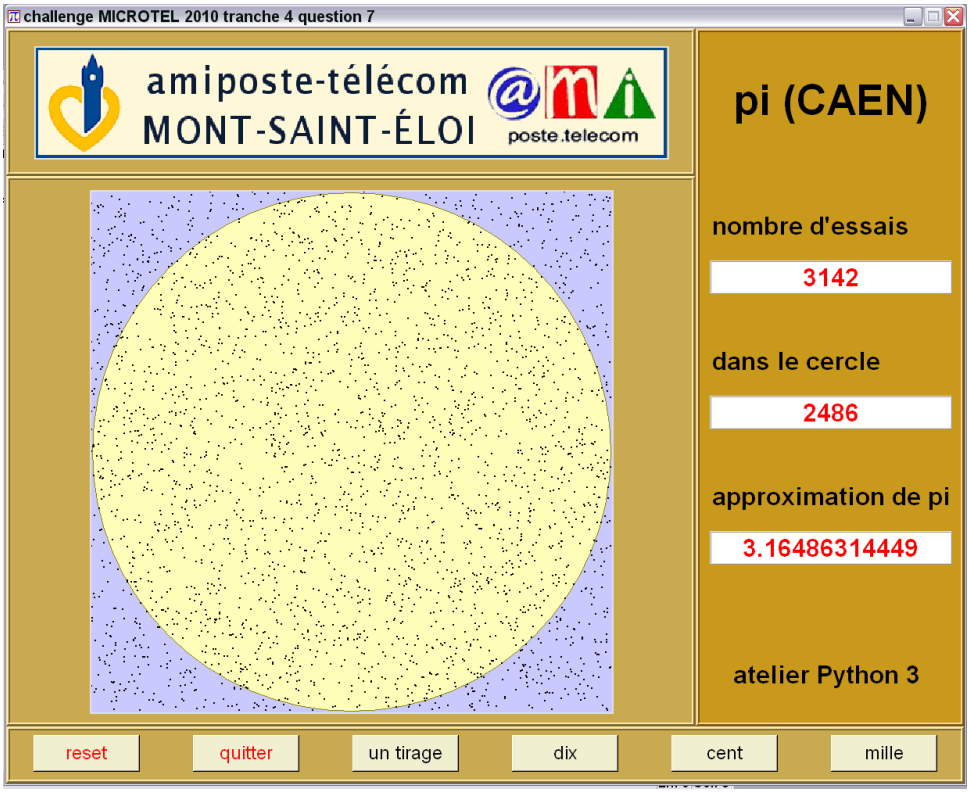
- le panneau où est spécifiée l'origine du logiciel (une image **gif**) : Classe **FrameLogo**.
- le panneau des scores : Classe **FrameScore.**
- le panneau du carré avec le cercle inscrit : Classe **FrameCible**.
- le panneau des boutons de commandes : Classe **FrameCommande**.

* **Les constantes**. Pour les gérer proprement, toutes les constantes sont regroupées dans un seul objet. La classe **Constantes** permet de spécifier les constantes ; une instance de cette classe, **ksT,** est affectée comme champ de l'instance unique (**top level**) **application** de la classe **Application**,. Pour simplifier les notations, on crée un alias **kT** de **application.ksT** dans chaque Frame. Ainsi une constante s'écrit **kT.id_constante**. Sans cela on aurait eu systématiquement **self.ksT.id_constante.**

1.3. Le plan du script.

```
from random import uniform, seed
from tkinter import *
from tkinter.messagebox import askyesno
import sys

class PiSimulation () : # une classe conteneur
    def fleche():
    def rafale(n) :
    def setInit():
    def getScore():

class Constantes () : # regrouper / initialiser les constantes
    def __init__ (self, num, den) :

class FrameScore(Frame):
    def __init__(self,pApplication) :
    def afficherScore(self) :

class FrameCommande(Frame):
    def __init__(self, pApplication) :
    def reset(self):
    def un(self):
    def dix(self):
    def cent(self):
    def mille(self):

class FrameCible(Frame):
    def __init__(self,pApplication) :
    def setCanvasCible(self):
    def dot(self, xy) :

class FrameLogo(Frame):
    def __init__(self,pApplication) :

class Application(Tk) :
    def __init__(self) :
```

```
    def dimensionner (self):
    def quitter(self) :

# *************** main ****************************
application = Application()
application.mainloop()
application.destroy()
```



2. La classe PiSimulation.

2.1. une classe conteneur.

Si on n'avait pas à créer l'interface graphique, la simulation aurait pu faire l'objet d'un script simple, avec ses fonctions, et un programme réduit à sa plus simple expression, un ordre de réaliser une salve de n tirages et l'envoi sur la sortie standard du résultat !

Dans le logiciel, on encapsule simplement dans une classe conteneur tout ce qui concerne la simulation : pas besoin d'instanciation. Dans le jargon de la programmation objet, on appelle cela une classe **`static`**. Jusque la version 3 de Python, il fallait déclarer **`static`** les méthodes de classe .

2.2. la classe et ses méthodes.

```
class PiSimulation () : # une classe conteneur
    __total = 0
    __bon = 0

    def fleche():
```

```
        PiSimulation.__total += 1
        x = uniform(-1.0, 1.0)
        y = uniform(-1.0, 1.0)
        if (x*x+y*y < 1.0) :
            PiSimulation.__bon += 1
        return (x, y)

    def rafale(n) :
        seed()
        resultat = []
        for i in range(n) :
            xy = PiSimulation.fleche()
            resultat.append(xy)
        return (resultat)

    def setInit():
        PiSimulation.__total = 0
        PiSimulation.__bon = 0

    def getScore():
        try : # demander une fois pardon ...
          return (PiSimulation.__total, PiSimulation.__bon,
                  PiSimulation.__bon / PiSimulation.__total * 4 )
        except :
          return (0, 0,"")
```

* La classe comporte deux variables de classe protégées : **__total** qui est un compteur des tirages et **__bon** qui est un compteur des "flèches" qui sont arrivées dans le cercle.

* la méthode **fleche()** simule un essai de tir. La fonction **uniform(-1.0, 1.0)** retourne un nombre flottant pseudo-aléatoire compris entre -1 et +1. Cette fonction est importée du module **random**, qui est un module inclus dans Python. **x** et **y** sont les coordonnées d'un point du carré de côté 2 unités, centré à l'origine d'un repère euclidien. Le cercle centré de même, de rayon unité a pour équation : $x^2 + y^2 = 1$ (théorème de Pythagore) : la condition **(x*x+y*y < 1.0)** exprime donc le fait que le point de coordonnées (x, y) est à l'intérieur du cercle.

Les deux compteurs sont mis à jour, et la fonction **fleche** retourne les coordonnées du point d'impact sous forme d'un **tuple**

* la méthode rafale() prend un argument, **n**, qui est le nombre d'essais effectués ; la boucle simule n essais successifs. Mais auparavant, le générateur de nombres aléatoires est relancé "aléatoirement" par la fonction **seed()** qui est importée du module **random**. La fonction retourne une liste de tous les impacts (liste de tuples).

* la méthode **setInit()** est une fonction de rénitialisation des compteurs.

* la méthode **getScore()** fournit les résultats après une ou plusieurs salves (en effet, si il y a plusieurs salves, les résultats s'ajoutent dans les compteurs).

La méthode getScore() utilise une exception comme principe de navigation. Plutôt que de faire une conditionnelle à chaque appel pour vérifier qu'on ne divise par par zéro, c'est la division par zéro qui provoque la seconde forme de retour qui aurait été la partie **else** de la conditionnelle !

3. L'application graphique.

attention : il faut situer cette section du code à sa place pour respecter les dépendances.! Voir le plan dans la section 1.3.

3.1. Le principe.

Cette fiche est consacrée en premier lieu au gestionnaire de géométrie **`place()`**. Il s'agit en autres de tester l'adaptabilité de ce gestionnaire à la mise à l'échelle de l'interface graphique. La partie essentielle du test est la mise à dimensions automatique de la fenêtre principale en fonction de la résolution de l'écran.

Le principe consiste à raisonner comme si on était dans une application de dimensions prédéfinies (920 pixels x 725 pixels) et donc de définir toutes les données dimensionnelles en fonction de ces dimensions originelles. Ces dimensions sont adaptées à un écran récent, dont la définition verticale est inférieure à 1024 pixels ordinateur portable par exemple). Les dimensions des sous-fenêtres (frames), des caractères, du logo sont dans un premier temps, **données en valeurs absolues, adaptées au "cadre"** de la fenêtre. On aurait pu procéder autrement, en travaillant en pourcentage par exemple (**exemple** : la hauteur de la frame est 80% de la hauteur du cadre...).

Dans un deuxième temps, on applique à chaque dimension un coefficient qui dépend de la dimension verticale de l'écran ; ce coefficient est donné sous forme de fraction (**exemple** : 3/4 pour une définition verticale de 600 pixels). Le premier intérêt de la fraction est de pouvoir travailler uniquement en nombres entiers, alors que le travail avec un coefficient d'échelle (flottant) aurait obligé à travailler en flottants, avec constamment des questions d'arrondis.

Ce n'est pas la seule raison (voir dans le paragraphe 4 la mise à dimensions du logo).

3.2. structuration de la fenêtre

Il y a quatre fonctions à remplir : afficher le logo qui spécifie l'origine du logiciel ; afficher le simulation sous forme graphique (le carré, le cercle, les impacts) ; afficher les résultats sans décalage visible sur la simulation ; proposer un tableau de commandes (des boutons) permettant de lancer une salve (4 boutons), de quitter l'application (1 bouton), d'effectuer une réinitialisation (1 bouton).

Chaque fonction est réalisée dans un cadre (**`frame`**) : **`FrameLogo, FrameCible, FrameScore`** et **`FrameCommande.`**

La fenêtre principale de l'application doit être mise à dimension et centrée sur l'écran.

3.3. la classe Application : le code __init__().

```
class Application(Tk) :
    def __init__(self) :
        # self est la racine de l'application
        Tk.__init__(self)
        self.ksT = self.dimensionner() # ici sont définies les constantes
        kT=self.ksT # un alias

        # la fenêtre ne peut être redimensionnée
        self.resizable(False, False)

        # titre / icone dans la barre de la fenêtre
        self.title(kT.titre)
        win = sys.platform
        if win[0:3] == 'win' :
            self.iconbitmap(kT.pictoIcone) # uniquement windows
        # sortir proprement par clic sur l'icone <<croix>>
        self.protocol(kT.fermeIcone, self.quitter)
```

```
        # le logo
        self.frameLogo=FrameLogo(self)
        self.frameLogo.place(x=0,y=0, anchor=NW)

        # score
        self.frameScore=FrameScore(self)
        self.frameScore.place(x=kT.posXFrameScore,
                              y=kT.posYFrameScore,
                              anchor=NW)
        # cible
        self.frameCible=FrameCible(self)
        self.frameCible.place(x=kT.posXFrameCible,
                              y=kT.posYFrameCible,
                              anchor=NW)

        # commande / attention à l'ordre des frames
        self.frameCommande=FrameCommande(self)
        self.frameCommande.place(x=kT.posXFrameCommande,
                                 y=kT.posYFrameCommande,
                                 anchor=NW)
```

*** `self.ksT = self.dimensionner()` `# ici sont définies les constantes`**
`kT=self.ksT` `# un alias`

Les constantes sont définies comme attributs d'une instance de la classe **`Constantes`** dans la méthode **`dimensionner()`**. Cette instance est un attribut de l'application. Pour aérer le code, on crée un alias **`kT`** de cet attribut dans toutes les classes et méthodes qui le requièrent.

* **`self.iconbitmap(kT.pictoIcone)`** une application fenêtrée peut contenir un icone qui s'affiche dans la fenêtre de titre ; cet icone doit être au format spécial **`ico`**. Ce format peut être obtenu avec Gimp. Fonctionne sous Windows, d'où la nécessité de définir la plateforme.

3.4. le programme d'appel.

```
# *************** main ****************************
application = Application()
application.mainloop()
application.destroy()
```

3.5. la classe Application : les méthodes.

```
    def dimensionner (self):
        hs = self.winfo_screenheight()
        ws = self.winfo_screenwidth()

        if (hs > 1023) :
            num = 4
```

```
            den = 3
        elif (hs > 800):
           num =1
           den = 1
        else :
            num = 3
            den = 4

        cst = Constantes (num, den)
        cst.l=(ws - cst.w - cst.bordure*2) // 2
        cst.t=(hs - cst.h - cst.bandeau) // 2
        geometrie = str(cst.w)+"x"+str(cst.h)+"+"+str(cst.l)
+"+"+str(cst.t)
        self.geometry(geometrie)
        return cst


    def quitter(self) :
        kT=self.ksT # un alias
        confirmer = askyesno (kT.terme, kT.confirmeQ)
        if (confirmer) :
            self.quit()
```

* **`quitter()`** : une fenêtre de confirmation est invoquée ; elle a été importée avec le sous-module **`messagebox`** du module **`tkinter`**.

* **`dimensionner()`** On rappelle que cette méthode doit retourner une instance de la classe des constantes. Pour définir cette instance, il est nécessaire de connaître la fraction/échelle qui permettra de moduler tous les paramètres de placement et de dimension. La recherche des dimensions d'écran a été rencontrée dans la fiche 2, ainsi que la méthode de centrage sur l'écran. Il faut simplement noter que l'on a distingué 3 cas qui couvrent les situations suivantes : écrans HD et portables récents, écrans classiques, vieux matériels.

Comme on utilise un placement calculé, celui-ci doit être défini avant d'appeler les méthodes de frame. On avait procédé autrement avec les méthodes **`pack()`** et **`grid()`** (voir fiche 4)

4. La classe des constantes.

4.1. intérêt de regrouper les constantes

Le regroupement des constantes facilite le "réglage" des constantes lors de la mise au point. Le code est plus lisible. Cette méthode pourrait permettre d'en fixer un certain nombre par un fichier de configuration : les couleurs, les messages, le type de bordure. Dans cet esprit, on se contente ici de charger une ressource externe (le logo) dans cette classe.

On retrouve la mise à dimension par utilisation d'une fraction dans cette classe et uniquement dans cette classe.

4.2. le code

```
class Constantes () : # regrouper / initialiser les constantes
    def __init__ (self, num, den) :
        self.num=num
        self.den=den
```

```
        self.w=920 * num // den
        self.h=725 * num // den
        self.t=0
        self.l=0
        self.relief = "ridge"
        self.bordure = 4 * num // den # bordure des frames
        self.bandeau = 35 # hauteur maximum du bandeau haut + bordure

        # logo
        refLogoImg = "./picto_logo.gif"
        img= PhotoImage (file=refLogoImg)
        img = img.zoom(num)
        self.logoImg = img.subsample(den)

        self.bordX = 20 * num // den + self.bordure
        self.bordY = 10 * num // den + self.bordure
        self.wCanLogoImg = 606 * num // den # dimension du logo
        #           # note : 12 pixels extra du canvas
        self.wFrameLogo= self.wCanLogoImg + self.bordX*2 + 12
        self.posXLogoImg = self.bordX
        self.hCanLogoImg = 106 * num // den # dimension verticale du logo
        self.hFrameLogo= self.hCanLogoImg + self.bordY*2 + 12
        self.posYLogoImg = self.bordY

        # score
        self.wFrameScore = self.w-self.wFrameLogo
        self.hFrameScore = 670 * num // den
        self.posXFrameScore = self.wFrameLogo
        self.posYFrameScore = 0
        self.posXTitre=30 * num // den
        self.posXValeur= 10 * num // den
        self.posYTitre = 40 * num // den
        self.delta =130 * num // den
        self.decale = 50 * num // den
        self.wLabValeur = 16 # nombre de caractères
        self.posYCopy = self.hFrameScore- (70 * num // den)

        # commande
        self.wFrameCommande = self.w
        self.hFrameCommande = self.h-self.hFrameScore
        self.posXFrameCommande = 0
        self.posYFrameCommande = self.hFrameScore
        self.distribution =self.w//6
        self.btWidth= 9 # unité : le caractère
        self.pad = 4 * num // den + 1

        # cible
        self.wFrameCible = self.wFrameLogo
```

```
        self.hFrameCible = self.hFrameScore - self.hFrameLogo
        self.posXFrameCible = 0
        self.posYFrameCible = self.hFrameLogo
        self.dCanCible = 500 * num // den  # dimensions du canevas cible
        self.cCanCible = self.dCanCible//2  # centre du canvas cible
        self.posXCanCible = (self.wFrameCible - self.dCanCible-12) // 2
        self.posYCanCible = (self.hFrameCible - self.dCanCible-12) // 2
        self.point = 2

        # couleurs
        self.couleurLogo = "#c7a04b"
        self.couleurScore = "#c7901b"
        self.couleurCible = "#c7a04b"
        self.couleurCommande = "#c7a04b"
        self.couleurCarre = "#c0c0ff"
        self.couleurCercle = "#ffffb0"
        self.couleurBord = "#808000"
        self.couleurBgValeur = "white"
        self.couleurFgValeur = "red"
        self.bgBt = "#e8e8c8"

        # affichage
        self.descTitre=("Helvetica", -40 * num // den, "bold")
        self.descInfo=("Helvetica", -24 * num // den, "bold")
        self.descVal=("Helvetica", -28 * num // den)
        self.desBt = ("Helvetica", -18 * num // den)

        # textes
        self.fermeIcone = "WM_DELETE_WINDOW"
        self.pictoIcone ="./picto_icon.ico"
        self.terme = "terminer ?"
        self.confirmeQ = "voulez-vous quitter ?"
        self.reset = "réinitialiser"
        self.confirmeR = "voulez-vous réinitialiser ?"
        self.titre = "challenge MICROTEL 2010 tranche 5 question 7"
        self.textTitre="pi (CAEN)"
        self.textCopy ="atelier Python 3"
        self.textTotal="nombre d'essais"
        self.textCercle="dans le cercle"
        self.textApprox="approximation de pi"
```

* **mise à dimension du logo** : le logo doit être lui aussi mis à dimensions. On dispose pour cela de deux méthodes de la classe **`PhotoImage`** : **`zoom()`** et **`subsample()`**. Ces deux méthodes prennent **un argument entier** : le facteur de zoom pour la première, le facteur de réduction pour la seconde. Ces méthodes fonctionnent par duplication et réduction de pixels. Ce qui implique que si on met **`subsample()`** en premier, on a un fort effet de grain. Mais si on met **`zoom()`** en premier avec un facteur de zoom élevé, il se crée une buffération qui peut être trop chère pour le système. D'où l'intérêt

d'optimiser les deux facteurs à des valeurs les plus petites possibles. Si on dispose uniquement d'un facteur d'échelle (pourcentage) en nombre flottant, on a intérêt à le remplacer par une fraction d'approximation à numérateur faible. Une théorie mathématique comme celle des fractions continues est utile dans ce cas. Par **exemple**, supposons que le facteur d'échelle soit 3,72 :

3,72 = 4 - 0,28. La première fraction utile serait 4/1 ; c'est peu satisfaisant. Mais 0,28 = 1 / 3.5714.... On peut prendre comme fraction la valeur 4 - 1/4 = 15/4, ce qui l'est davantage (3,75). Il vaut mieux cependant avoir une image assez petite, car le facteur de zoom est 15.

5. La classe FrameLogo.

```
class FrameLogo(Frame):
    def __init__(self,pApplication) :
        Frame.__init__(self)
        kT= pApplication.ksT # un alias
        self.configure(border=kT.bordure, relief=kT.relief,
                       height=kT.hFrameLogo, width=kT.wFrameLogo,
                       bg=kT.couleurLogo)
        canvasLogo = Canvas (self,
                             width=kT.wCanLogoImg,
                             height=kT.hCanLogoImg,
                             bg=kT.couleurLogo)
        image = canvasLogo.create_image(1, 1, anchor=NW, image=kT.logoImg)
        canvasLogo.place(x=kT.posXLogoImg, y=kT.posYLogoImg, anchor=NW)
```

* la seule difficulté est liée au fait que pour évaluer l'encombrement du canevas il faut tenir compte des 6 pixels des bordures divers.

6. La classe FrameScore.

```
class FrameScore(Frame):
    def __init__(self) :
        Frame.__init__(self, pApplication)
        kT=pApplication.ksT # un alias
        self.configure(border=kT.bordure, relief=kT.relief,
                       height=kT.hFrameScore, width=kT.wFrameScore,
                       bg=kT.couleurScore)
        titre = Label(self, font=kT.descTitre, text=kT.textTitre,
                      bg=kT.couleurScore)
        titre.place(x=kT.posXTitre, y=kT.posYTitre, anchor=NW)

        copyr = Label(self, font=kT.descInfo, text=kT.textCopy,
                      bg=kT.couleurScore)
        copyr.place(x=kT.posXTitre, y=kT.posYCopy, anchor=NW)

        labelInfoTotal = Label(self, font=kT.descInfo, text=kT.textTotal,
                               bg=kT.couleurScore)
        labelInfoTotal.place(x=kT.posXValeur,
                                    y=kT.posYTitre+kT.delta, anchor=NW)
        labelInfoCercle = Label(self, font=kT.descInfo,
```

```
                          text=kT.textCercle, bg=kT.couleurScore)
        labelInfoCercle.place(x=kT.posXValeur,
                              y=kT.posYTitre+kT.delta*2, anchor=NW)

        labelInfoApprox = Label(self, font=kT.descInfo,
text=kT.textApprox,
                                bg=kT.couleurScore)
        labelInfoApprox.place(x=kT.posXValeur,
                                  y=kT.posYTitre+kT.delta*3, anchor=NW)

        self.labelValeurTotal = Label(self, font=kT.descInfo,
                          text="",
                          width=kT.wLabValeur,borderwidth=2,relief=RIDGE,
                          fg= kT.couleurFgValeur,
                          bg=kT.couleurBgValeur)
        self.labelValeurTotal.place(x=kT.posXValeur,
                    y=kT.posYTitre+kT.delta+kT.decale, anchor=NW)

        self.labelValeurCercle = Label(self, font=kT.descInfo, text="",
                           width=kT.wLabValeur,borderwidth=2,relief=RIDGE,
                           fg= kT.couleurFgValeur,
                           bg=kT.couleurBgValeur)
        self.labelValeurCercle.place(x=kT.posXValeur,
                            y=kT.posYTitre+kT.delta*2+kT.decale,anchor=NW)

        self.labelValeurApprox = Label(self, font=kT.descInfo, text="",
                        width=kT.wLabValeur,borderwidth=2,relief=RIDGE,
                        fg= kT.couleurFgValeur, bg=kT.couleurBgValeur)
                        self.labelValeurApprox.place(x=kT.posXValeur,
                        y=kT.posYTitre+kT.delta*3+kT.decale, anchor=NW)
        self.afficherScore()

    def afficherScore(self) :
        r = PiSimulation.getScore()
        self.labelValeurTotal.configure(text = str(r[0]))
        self.labelValeurCercle.configure(text = str(r[1]))
        self.labelValeurApprox.configure(text = str(r[2]))
```

5. La classe FrameCommande.

```
class FrameCommande(Frame):
    def __init__(self, pApplication) :
        Frame.__init__(self, pApplication)
        kT=pApplication.ksT # un alias
        self.configure(border=kT.bordure, relief=kT.relief,
                        height=kT.hFrameCommande, width=kT.wFrameCommande,
```

```
                    bg=kT.couleurCommande)
        self.frameSc = pApplication.frameScore  # *********
        self.frameCn = pApplication.frameCible  # *********

        btReset = Button(self, text="reset",
                         command=self.reset,
                         width=kT.btWidth, font=kT.desBt, bg=kT.bgBt,
                         fg="red")
        btReset.place(x=kT.pad*5, y=kT.pad, anchor=NW)

        btQuitter = Button(self, text="quitter", fg="red",
                           command=pApplication.quitter,
                           width=kT.btWidth, font=kT.desBt, bg=kT.bgBt)
        btQuitter.place(x=kT.pad*5+kT.distribution, y=kT.pad, anchor=NW)

        btUn = Button(self, text="un tirage",command=self.un,
                      width=kT.btWidth, font=kT.desBt, bg=kT.bgBt)
        btUn.place(x=kT.pad*5+kT.distribution*2, y=kT.pad, anchor=NW)
        btDix = Button(self, text="dix",command=self.dix,
                               width=kT.btWidth, font=kT.desBt, bg=kT.bgBt)
        btDix.place(x=kT.pad*5+kT.distribution*3, y=kT.pad, anchor=NW)
        btCent = Button(self, text="cent",command=self.cent,
                               width=kT.btWidth, font=kT.desBt, bg=kT.bgBt)
        btCent.place(x=kT.pad*5+kT.distribution*4, y=kT.pad, anchor=NW)
        btMille = Button(self, text="mille",command=self.mille,
                               width=kT.btWidth, font=kT.desBt, bg=kT.bgBt)
        btMille.place(x=kT.pad*5+kT.distribution*5, y=kT.pad, anchor=NW)

    def reset(self):
        kT = self.master.ksT # alias
        confirmer = askyesno (kT.reset, kT.confirmeR)
        if (confirmer) :
            PiSimulation.setInit()
            self.frameSc.afficherScore()
            self.frameCn.setCanvasCible()

    def un(self):
        res=PiSimulation.rafale(1)
        self.master.frameCible.dot (res[0])
        self.frameSc.afficherScore()

    def dix(self):
        res=PiSimulation.rafale(10)
        self.frameSc.afficherScore()
        for i in range(10) :
            self.master.frameCible.dot (res[i])

    def cent(self):
```

```
        res = PiSimulation.rafale(100)
        self.frameSc.afficherScore()
        for i in range(100) :
            self.master.frameCible.dot (res[i])

    def milleself):
        for i in range(10) :
            res = PiSimulation.rafale(100)
            for i in range(100) :
                self.master.frameCible.dot (res[i])
        self.frameSc.afficherScore()
```

* avec les méthodes **un()**, **dix()**, **cent()** on voit l'intérêt qu'il y a dans **PiSimulation** à ce que la méthode **rafale()** retourne les coordonnées sous forme d'une liste puisque la méthode **dot()** qui ajoute des point sur la cible les utilise. On peut remarquer l'utilisation du champ **master** qui est systématiquement présent comme champ d'instance et désigne le parent immédiat (parfois appelé **super** dans d'autres langages).

* Pour la méthode **mille()**, on a itéré sur des rafales de 100 tirages. Cela évite une trop forte bufférisation et améliore les performances.

* la méthode **reset()** est assez brutale ; en effet, pour redessiner la cible, on redéfinit tout le canevas ! Pour cela, on utilise la même méthode qui a servi à la créer.

6. La classe FrameCible.

```
class FrameCible(Frame):
    def __init__(self,pApplication) :
        Frame.__init__(self, pApplication)
        kT=pApplication.ksT # un alias
        self.configure(border=kT.bordure, relief=kT.relief,
                       height=kT.hFrameCible, width=kT.wFrameCible,
                       bg=kT.couleurCible)
        self.setCanvasCible()

    def setCanvasCible(self):
        kT=self.master.ksT # un alias
        self.canvasCible = Canvas (self, width = kT.dCanCible,
                                   height = kT.dCanCible, bg=kT.couleurCarre)
        cercle = self.canvasCible.create_oval(3, kT.dCanCible,
                                              kT.dCanCible, 3,
                                              fill=kT.couleurCercle,
                                              outline=kT.couleurBord)

        self.canvasCible.place(x=kT.posXCanCible, y=kT.posYCanCible,
anchor=NW)

    def dot(self, xy) :
        kT = self.master.ksT # alias
        x = int((1.0+xy[0])*kT.cCanCible)
```

```
        y = int((1.0+xy[1])*kT.cCanCible)
        point = self.canvasCible.create_line(x, y,x+kT.point, y,
                                             width=kT.point)
```

* **setCanvasCible() :** la création du canevas est une méthode qui est utilisée plusieurs fois ; le fait que le champ **canvasCible** soit réaffecté avec **self.canvasCible** invalide l'ancien canevas, qui est rendu inaccessible, et le remplace par le nouveau. Lors de la mis à jour de l'écran c'est le canevas accessible qui est pris en compte.

* dans la méthode **dot()**, on aurait pu s'attendre à ce qu'un impact soit représenté par un simple pixel. Mais avec les écrans modernes, le pixel est minuscule et peu lisible. Tant pour les essais que pour une bonne lisibilité de l'interface, on a paramétré la dimension du point d'impact.