8 Interfaces graphiques

1 Interfaces graphiques (GUI)

Si vous ne le saviez pas encore, apprenez dès à présent que le domaine des interfaces graphiques (en anglais **GUI : Graphical User Interfaces**) est extrêmement complexe. Chaque système d'exploitation peut en effet proposer plusieurs « bibliothèques » de fonctions graphiques de base, auxquelles viennent fréquemment s'ajouter de nombreux compléments, plus ou moins spécifiques au langage de programmation utilisé. Tous ces composants sont généralement présentés comme des **classes d'objets**, dont il faudra étudier les **attributs** et les **méthodes**.

Avec Python, la bibliothèque graphique la plus simple d'utilisation est la bibliothèque **tkinter**, qui est une adaptation de la bibliothèque **Tk** développée à l'origine pour le langage **Tcl**. Plusieurs autres bibliothèques graphiques fort intéressantes ont été proposées pour Python : **wxPython**, **pyQT**, **pygame**, etc. Il existe également des possibilités d'utiliser les bibliothèques de **widgets Java** et les **MFC** de **Windows**. Dans le cadre de ce cours, nous nous limiterons cependant à **tkinter**, dont il existe fort heureusement des versions similaires pour les plates-formes **Linux**, **Windows** et **Mac OS**.

1.1 Premiers pas avec tkinter

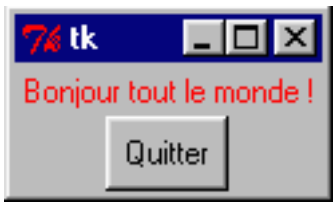
Pour la suite des explications, nous supposerons bien évidemment que le module **tkinter** a déjà été installé sur votre système d'exploitation. Pour pouvoir en utiliser les fonctionnalités dans un script Python, il faut que l'une des premières lignes de ce script contienne l'instruction d'importation suivante :

```
from tkinter import *
```

Comme toujours sous Python, il n'est même pas nécessaire d'écrire un script. Vous pouvez faire un grand nombre d'expériences directement en lignes de commande, en lançant simplement Python en mode interactif.

Dans l'exemple qui suit, nous allons créer une fenêtre très simple, et y ajouter deux **widgets**[1] typiques : un bout de texte (ou **label**) et un bouton (ou **button**).

```
>>> from tkinter import *
>>> fen = Tk()
>>> txt = Label(master=fen, text='Bonjour tout le monde !', fg='red')
>>> txt.pack()
>>> bou = Button(master=fen, text='Quitter', command = fen.destroy)
>>> bou.pack()
>>> fen.mainloop()
```



Examinons à présent plus en détail chacune des lignes de commandes exécutées :

(1) Comme cela a déjà été expliqué dans le chapitre 4, il est aisé de construire différents modules Python, qui contiendront des scripts, des définitions de fonctions, des classes d'objets, etc. On peut alors importer tout ou une partie de ces modules dans n'importe quel programme, ou même dans l'interpréteur fonctionnant en mode interactif. C'est ce que nous faisons à la première ligne de notre exemple : `from tkinter import *` consiste à importer toutes les classes contenues dans le module **tkinter**.

1. « **widget** » est le résultat de la contraction de l'expression « *window gadget* ». Dans certains environnements de programmation, on appellera cela plutôt un « composant graphique ». Ce terme désigne en fait toute entité susceptible d'être placée dans une fenêtre d'application, comme par exemple un bouton, une case à cocher, une image, etc., et parfois aussi la fenêtre elle-même.

(2) À la deuxième ligne de notre exemple : `fen = Tk()`, nous recourons à l'une des classes du module **tkinter**, la classe `Tk()`, et nous en créons une **instance**, à savoir la fenêtre `fen`.

La classe `Tk()`, qui est l'une des classes les plus fondamentales de la bibliothèque **tkinter**, contient tout ce qu'il faut pour engendrer différents types de fenêtres d'application, de tailles ou de couleurs diverses, avec ou sans barre de menus, etc. Nous nous en servons ici pour créer notre objet graphique de base, à savoir la fenêtre qui contiendra tout le reste. Dans les parenthèses de `Tk()`, nous pourrions préciser différentes options en arguments, mais nous laisserons cela pour un peu plus tard.

Rappelons que l'instruction d'**instanciation** ressemble à une simple affectation de variable. Comprenons bien cependant qu'il se passe ici deux choses à la fois :

- la **création** d'un nouvel objet ;
- l'**affectation** d'une variable, qui va désormais servir de **référence** pour manipuler l'objet.

(3) À la troisième ligne : `txt = Label(master=fen, text='Bonjour tout le monde !', fg='red')`, nous créons un autre objet (un **widget**), cette fois par **instanciation** de la classe `Label()`. Comme son nom l'indique, cette classe définit toutes sortes d'**étiquettes** (ou de **libellés**). En fait, il s'agit tout simplement de fragments de texte quelconques, utilisables pour afficher des informations et des messages divers à l'intérieur d'une fenêtre.

Remarquons que des arguments ont été fournis au constructeur de la classe `Label` lors de l'instanciation :

- Le premier argument transmis (`master=fen`), indique que le nouveau **widget** que nous sommes en train de créer sera contenu **dans un autre widget préexistant**, que nous désignons donc ici comme son « maître » : l'objet `fen` est le **widget maître** de l'objet `txt`. On pourra dire aussi que l'objet `txt` est un **widget esclave** de l'objet `fen`.
- Les deux arguments suivants servent à préciser la forme exacte que doit prendre notre **widget**. Ce sont en effet deux options de création, chacune fournie sous la forme d'une chaîne de caractères : d'abord le texte de l'étiquette, ensuite sa couleur d'avant-plan (ou `foreground`, en abrégé `fg`). Ainsi le texte que nous voulons afficher est bien défini, et il doit apparaître coloré en rouge.
  Nous pourrions encore préciser bien d'autres caractéristiques : la police à utiliser, ou la couleur d'arrière-plan, par exemple. Toutes ces caractéristiques ont cependant une valeur par défaut dans la définition interne du constructeur de la classe `Label()`. Nous ne devons indiquer des options que pour les caractéristiques que nous souhaitons différentes du modèle standard.

(4) À la quatrième ligne de notre exemple : `txt.pack()`, nous activons une **méthode** associée à l'objet `txt` : la méthode `pack()`. Cette méthode fait partie d'un ensemble de méthodes qui sont applicables non seulement aux widgets de la classe `Label()`, mais aussi à la plupart des autres widgets **tkinter**, et qui agissent sur leur disposition géométrique dans la fenêtre. Comme vous pouvez le constater par vous-même si vous entrez les commandes de notre exemple une par une, la méthode `pack()` réduit automatiquement la taille de la fenêtre « maître » afin qu'elle soit juste assez grande pour contenir les widgets « esclaves » définis au préalable.

(5) À la cinquième ligne : `bou = Button(master=fen, text='Quitter', command = fen.destroy)`, nous créons notre second widget « esclave » : un bouton. Comme nous l'avons fait pour le widget précédent, nous appelons la classe `Button()` en fournissant entre parenthèses un certain nombre d'arguments. Étant donné qu'il s'agit cette fois d'un objet interactif, nous devons préciser avec l'option `command` ce qui devra se passer lorsque l'utilisateur effectuera un clic sur le bouton. Dans ce cas précis, nous actionnerons la méthode `destroy` associée à l'objet `fen`, ce qui devrait provoquer l'effacement de la fenêtre [2].

(6) La sixième ligne utilise la méthode `pack()` pour adapter la géométrie de la fenêtre au nouvel objet que nous venons d'y intégrer.

(7) La septième ligne : `fen.mainloop()` est très importante, parce que c'est elle qui provoque le démarrage du **réceptionnaire d'événements** associé à la fenêtre. Cette instruction est nécessaire pour que votre application soit « à l'affût » des clics de souris, des pressions exercées sur les touches du clavier, etc. C'est donc cette instruction qui « la met en marche », en quelque sorte.

Comme son nom l'indique (**mainloop**), il s'agit d'une méthode de l'objet `fen`, qui active une **boucle** de programme, laquelle « tournera » en permanence en tâche de fond, dans l'attente de messages émis par le système d'exploitation de l'ordinateur. Celui-ci interroge en effet sans cesse son environnement, notamment au niveau des périphériques d'entrée (souris, clavier, etc.). Lorsqu'un événement quelconque est détecté, divers **messages** le décrivant sont expédiés aux programmes qui souhaitent en être avertis.

2. L'appel de la méthode `destroy` n'a pas lieu dans l'instruction décrivant le bouton. Il ne faut donc pas accoler des parenthèses à son nom. C'est tkinter qui se chargera d'effectuer l'appel `destroy()`, lorsqu'un utilisateur cliquera sur ce bouton.

2 Programmes pilotés par des événements

Vous venez d'expérimenter votre premier programme utilisant une interface graphique. Ce type de programme est structuré d'une manière différente des scripts « textuels » étudiés auparavant.

Tous les programmes comportent **grosso-modo** trois phases principales :

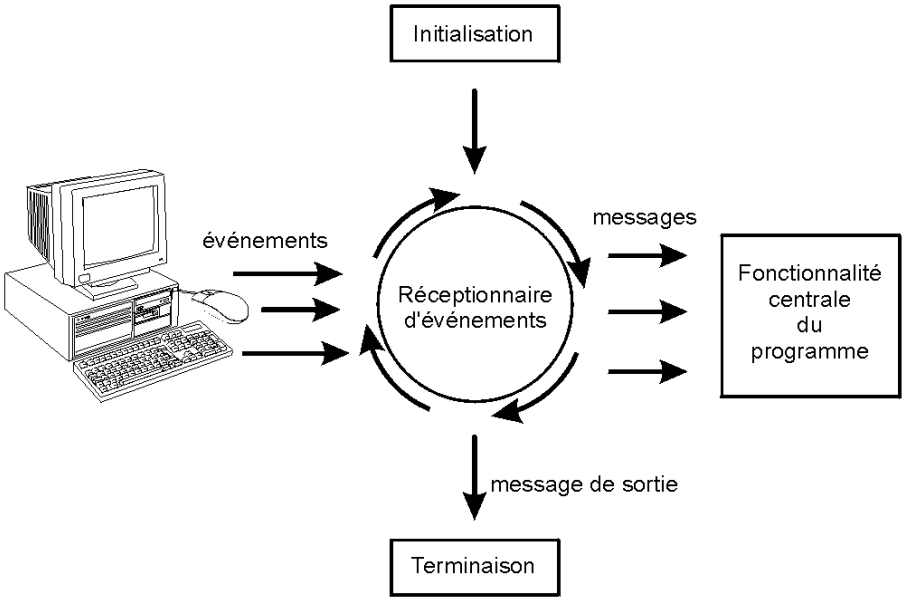
- **une phase d'initialisation**, laquelle contient les instructions qui préparent le travail à effectuer (appel des modules externes nécessaires, ouverture de fichiers, connexion à un serveur de bases de données ou à l'Internet, etc.) ;
- **une phase centrale** où l'on trouve la fonctionnalité du programme (c'est-à-dire tout ce qu'il est censé faire : afficher des données à l'écran, effectuer des calculs, modifier le contenu d'un fichier, imprimer, etc.) ;
- **une phase de terminaison** qui sert à clore « proprement » les opérations (fermer les fichiers restés ouverts, couper les connexions externes, etc.).

Dans un programme « en mode texte », ces trois phases sont organisées suivant un schéma **linéaire** comme dans l'illustration ci-contre. Par conséquent, ces programmes se caractérisent par une **interactivité très limitée** avec l'utilisateur. Celui-ci ne dispose pratiquement d'aucune liberté : il lui est demandé de temps à autre d'entrer des données au clavier, mais toujours dans un ordre prédéterminé correspondant à la séquence d'instructions du programme.

Dans le cas d'un programme qui utilise une interface graphique, par contre, l'organisation interne est différente. On dit d'un tel programme qu'il est **piloté par les événements**. Après sa phase d'initialisation, un programme de ce type se met en quelque sorte « en attente », et passe la main à un autre logiciel, lequel est plus ou moins intimement intégré au système d'exploitation de l'ordinateur et « tourne » en permanence.

Ce **réceptionnaire d'événements** scrute sans cesse tous les périphériques (clavier, souris, horloge, modem, etc.) et réagit immédiatement lorsqu'un événement y est détecté. Un tel événement peut être une action quelconque de l'utilisateur : déplacement de la souris, appui sur une touche, etc., mais aussi un événement externe ou un automatisme (top d'horloge, par exemple).

Lorsqu'il détecte un événement, le réceptionnaire envoie un message spécifique au programme [3], lequel doit être conçu pour réagir en conséquence comme l'illustre le schéma ci-dessous.



3. Ces messages sont souvent notés WM (**Window messages**) dans un environnement graphique constitué de fenêtres (avec de nombreuses zones réactives : boutons, cases à cocher, menus déroulants, etc.). Dans la description des algorithmes, il arrive fréquemment aussi qu'on confonde ces messages avec les événements eux-mêmes.

La phase d'initialisation d'un programme utilisant une interface graphique comporte un ensemble d'instructions qui mettent en place les divers composants interactifs de cette interface (fenêtres, boutons, cases à cocher, etc.). D'autres instructions définissent les messages d'événements qui devront être pris en charge : on peut en effet décider que le programme ne réagira qu'à certains événements en ignorant tous les autres.

Alors que dans un programme « textuel », la phase centrale est constituée d'une suite d'instructions qui décrivent à l'avance l'ordre dans lequel la machine devra exécuter ses différentes tâches (même s'il est prévu des cheminements différents en réponse à certaines conditions rencontrées en cours de route), on ne trouve dans la phase centrale d'un programme avec interface graphique qu'un ensemble de fonctions ou méthodes indépendantes. Chacune de ces fonctions ou méthodes est appelée spécifiquement lorsqu'un événement particulier est détecté par le système d'exploitation : elle effectue alors le travail que l'on attend du programme en réponse à cet événement, et rien d'autre [4].

Il est important de bien comprendre ici que pendant tout ce temps, le réceptionnaire continue à « tourner » et à guetter l'apparition d'autres événements éventuels.

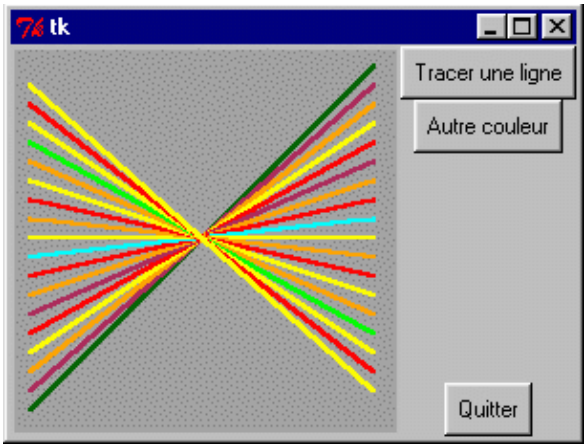
S'il arrive d'autres événements, il peut donc se faire qu'une deuxième fonction ou méthode (ou une 3ème, une 4ème...) soit activée et commence à effectuer son travail « en parallèle » avec la première qui n'a pas encore terminé le sien. Les systèmes d'exploitation et les langages modernes permettent en effet ce parallélisme que l'on appelle aussi **multitâche**.

Au chapitre 4, nous avons déjà remarqué que la structure du texte d'un programme n'indique pas directement l'ordre dans lequel les instructions seront finalement exécutées. Cette remarque s'applique encore bien davantage dans le cas d'un programme avec interface graphique, puisque l'ordre dans lequel les fonctions sont appelées n'est plus inscrit nulle part dans le programme. Ce sont les événements qui pilotent !

Finalement, notons que la programmation orientée objet convient particulièrement bien au développement d'applications avec interface graphique. Dans les exemples qui suivent, nous allons appliquer les concepts décrits dans le chapitre précédent en nous efforçant de mettre en évidence les avantages qu'apporte l'orientation objet. Ces exemples devraient vous aider à ébaucher vos propres projets en vous présentant quelques widgets ainsi que certaines utilisations de ceux-ci. Au-delà de leur visée documentaires, ces exemples sont également destinés à vous apprendre par l'exemple comment s'articule une application construite à l'aide de classes et d'objets.

2.1 Tracé de lignes dans un canevas

Le premier exemple de cette section consiste à créer une fenêtre comportant trois boutons et un **canevas**. Suivant la terminologie de *tkinter*, un **canevas** est une surface rectangulaire délimitée, dans laquelle on peut installer ensuite divers dessins et images à l'aide de méthodes spécifiques [5].



Lorsque l'on clique sur le bouton « Tracer une ligne », une nouvelle ligne colorée apparaît sur le canevas, avec à chaque fois une inclinaison différente de la précédente.

Si l'on actionne le bouton « Autre couleur », une nouvelle couleur est tirée au hasard dans une série limitée. Cette couleur est celle qui s'appliquera aux tracés suivants.

Le bouton « Quitter » sert bien évidemment à terminer l'application en refermant la fenêtre.

4. Au sens strict, une telle fonction qui ne devra renvoyer aucune valeur est donc plutôt une ***procédure*** (cf. chapitre 4).
5. Ces dessins pourront éventuellement être animés dans une phase ultérieure.

Version avec utilisation de variables globales

Le script décrit ci-dessous implémente notre première application en recourant à des variables globales. Comme nous pouvons le constater, les fonctions de ce script peuvent modifier les valeurs de certaines variables qui sont définies au niveau principal du programme, grâce à l'instruction `global` utilisée dans leur définition. Nous procédons ainsi afin de pouvoir distinguer les différents composants d'un programme piloté par des événements. Néanmoins, sachez que **cette pratique n'est vraiment pas recommandable** surtout lorsqu'il s'agit d'écrire de grands programmes ! Nous aborderons par la suite une version orientée objet du même programme permettant de comprendre comment il est possible de traduire des variables globales en attributs de classe ou d'instance.

```
# Version avec variables globales d'un tracé de lignes dans un canevas
from tkinter import *
from random import randrange

# ---- définition des fonctions gestionnaires d'événements : ----
def drawline():
    "Tracé d'une ligne dans le canevas can"
    global x1, y1, x2, y2, coul
    can.create_line(x1,y1,x2,y2,width=2,fill=coul)
    # modification des coordonnées pour la ligne suivante :
    y1, y2 = y1-10, y2+10

def changecolor():
    "Changement aléatoire de la couleur du tracé"
    global coul
    palette = ['purple','cyan','maroon','green','red','blue','orange','yellow']
    c = randrange(8)          # => génère un nombre aléatoire de 0 à 7
    coul = palette[c]

#------- Programme principal --------
# les variables suivantes seront utilisées de manière globale :
x1, y1, x2, y2 = 10, 190, 190, 10      # coordonnées de la ligne
coul = 'dark green'                    # couleur de la ligne

# Création du widget principal ("maître") :
fen = Tk()
# création des widgets "esclaves" :
# canvas:
can = Canvas(master=fen,bg='dark grey',height=200,width=200)
can.pack(side = LEFT)
# boutons:
bou1 = Button(master=fen,text='Quitter',command = fen.quit)
bou1.pack(side = BOTTOM)
bou2 = Button(master=fen,text='Tracer une ligne',command = drawline)
bou2.pack()
bou3 = Button(master=fen,text='Autre couleur',command = changecolor)
bou3.pack()

fen.mainloop()       # démarrage du réceptionnaire d'événements

fen.destroy()        # destruction (fermeture) de la fenêtre
```

Conformément à ce que nous avons expliqué dans le texte des pages précédentes, la fonctionnalité de ce programme est essentiellement assurée par les deux fonctions `drawline()` et `changecolor()`, qui seront activées par des événements, ceux-ci étant eux-mêmes définis dans la phase d'initialisation.

Dans la phase d'initialisation, située dans le corps principal du programme, on commence par importer l'intégralité du module `tkinter` ainsi que la fonction `randrange` du module `random`. On crée ensuite les différents widgets par instanciation à partir des classes `Tk()`, `Canvas()` et `Button()`. Remarquons au passage que la même classe `Button()` sert à instancier plusieurs boutons, qui sont des objets similaires pour l'essentiel, mais néanmoins individualisés grâce aux options de création et qui pourront fonctionner indépendamment l'un de l'autre.

L'initialisation se termine avec l'instruction `fen.mainloop()` qui démarre le réceptionnaire d'événements. Les instructions qui suivent ne seront exécutées qu'à la sortie de cette boucle.

L'option `command` utilisée dans l'instruction d'instanciation des boutons permet de désigner le nom de la fonction qui devra être appelée lorsqu'un événement « **clic gauche de la souris sur le widget** » se produira. Il s'agit en fait d'un raccourci pour cet événement particulier, qui vous est proposé par *tkinter* pour votre facilité parce que cet événement est celui que l'on associe naturellement à un widget de type bouton. Nous verrons qu'il existe d'autres techniques plus générales pour associer n'importe quel type d'événement à n'importe quel widget.

Dans notre fonction `changecolor()`, une couleur est choisie au hasard dans une liste. Nous utilisons pour ce faire la fonction `randrange()` importée du module `random`. Appelée avec un argument n, cette fonction renvoie un nombre entier, tiré au hasard entre 0 et $n-1$.

La commande liée au bouton « Quitter » appelle la méthode `quit()` de la fenêtre `fen`. Cette méthode sert à fermer (quitter) le réceptionnaire d'événements (`mainloop`) associé à cette fenêtre. Lorsque cette méthode est activée, l'exécution du programme se poursuit avec les instructions qui suivent l'appel de `mainloop`. Dans notre exemple, cela consiste donc à effacer (`destroy`) la fenêtre.

Version orientée objet

Le script ci-dessous est une version orientée objet du programme de tracé de lignes dans un canevas :

```
# Version orientée objet d'un tracé de lignes dans un canevas
from tkinter import *
from random import randrange

# ------- Définition de la classe d'applications -------
class Application(Tk):

    # attribut de classe accessible par tous les objects de la classe Application
    palette = ['purple', 'cyan', 'maroon', 'green', 'red', 'blue', 'orange', 'yellow']

    def __init__(self):
        Tk.__init__(self)       # constructeur de la classe parente

        # coordonnées de la prochaine ligne:
        self.x1, self.y1, self.x2, self.y2 = 10, 190, 190, 10
        # couleur de la prochain ligne:
        self.coul = 'dark green'

        # création des widgets 'esclaves':
        # canevas:
        self.can = Canvas(master=self, bg='dark grey', height = 200, width = 200)
        self.can.pack(side=LEFT)
        # boutons:
        bou1 = Button(self, text = 'Quitter', command = self.quit)
        bou1.pack(side=BOTTOM)
        bou2 = Button(self, text = 'Tracer une ligne', command = self.drawline)
        bou2.pack()
        bou3 = Button(self, text = 'Autre couleur', command = self.changecolor)
        bou3.pack()

    # définitions des méthodes gestionnaires d'événements:

    def drawline(self):
        "Tracé d'une ligne dans le canevas self.can"

        self.can.create_line(self.x1, self.y1, self.x2, self.y2, fill = self.coul)
        # modification des coordonnées pour la ligne suivante:
        self.y1, self.y2 = self.y1-10, self.y2+10

    def changecolor(self):
        "Changement aléatoire de la couleur du tracé"

        c = randrange(8)      # => génère un nombre aléatoire de 0 à 7
        self.coul = Application.palette[c]

```

```
46 # ———— Corps principal du programme ————
47
48 fen = Application()        # Création d'une instance de la classe Application
49
50 fen.mainloop()             # Démarrage du réceptionnaire d'événements
51
52 fen.destroy()              # Destruction (fermeture) de la fenêtre
```

La classe `Application()` est obtenue par dérivation de la classe de base `Tk()` de *tkinter*. Son constructeur met en place la fenêtre principale, son canevas, les trois boutons ainsi que les coordonnées des extrémités de la prochaine ligne à tracer ainsi que la couleur de cette dernière. Cette classe sert donc d'*espace de noms commun*, dans lequel il est possible d'*encapsuler* les variables et les fonctions propres à l'application. Le principal intérêt de procéder ainsi est que cela permet d'éviter l'utilisation de variables globales. En effet :

- Mettre en route l'application se résumera à instancier un objet de cette classe ;
- Les fonctions qu'on voudra y mettre en oeuvre seront les `méthodes` de cet objet-application ;
- A l'intérieur de ces méthodes, il suffira de rattacher un nom de variable au paramètre `self` pour que cette variable soit accessible de partout à l'intérieur de l'objet. Une telle *variable d'instance* est donc tout à fait l'équivalent d'une variable globale (mais seulement à l'intérieur de l'objet), puisque toutes les autres méthodes de cet objet peuvent y accéder par l'intermédiaire de `self`. Elle reste néanmoins strictement confinée à l'intérieur de l'objet et il est déconseillé de vouloir y accéder de l'extérieur [6]. D'autre part, tous les objets que vous instancierez à partir d'une même classe posséderont chacun leur propre jeu de ces variables, qui sont bel et bien *encapsulés* dans ces objets. On les appelle pour cette raison des **attributs d'instance**.

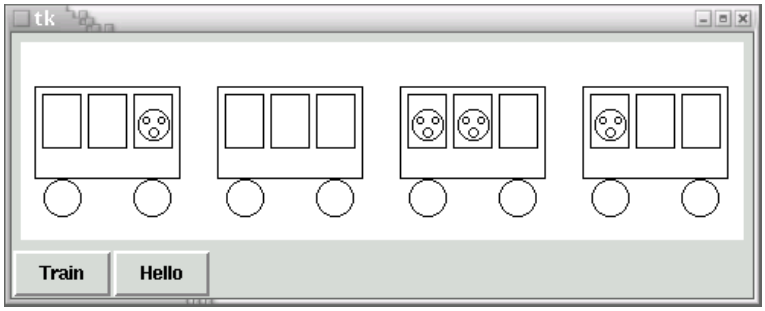
Penchons-nous quelques instants sur le code de cette application :

- Lignes 11 à 29 : Comme nous l'avons expliqué dans le chapitre précédent, le constructeur d'une classe dérivée doit activer lui-même le constructeur de la classe parente, en lui transmettant la référence de l'instance comme premier argument.
  Les lignes 21 à 29 servent à instancier les widgets nécessaires : canevas et boutons. Remarquez que le canevas est référencé à l'aide d'une variable d'instance (`self.can`) accessible de partout à l'intérieur de l'objet. Par contre, la référence des boutons n'étant nécessaire qu'à l'intérieur du constructeur lui-même, les variables `bou1, bou2` et `bou3` ne sont pas précédées du paramètre `self` : il s'agit de variables locales.
- Lignes 33 à 38 : Cette méthode est invoquée lorsqu'on actionne le deuxième bouton et permet de dessiner une ligne à l'aide de la méthode `create_line()` située dans l'objet de la classe `Canvas()`.
- Lignes 40 à 44 : Cette méthode est invoquée lorsqu'on actionne le troisième bouton et permet de modifier la couleur de la prochaine ligne à tracer dans le canevas. Remarquez que la couleur correspond à un *attribut d'instance* (`self.coul`) propre à chaque objet de la classe `Application()` construit alors que l'ensemble des couleurs utilisables est un *attribut de classe* accessible par tous les objets de la classe `Application()`. Ce choix permet d'éviter d'associer une liste de couleurs à chaque objet construit et gaspiller ainsi de la mémoire inutilement.
- Lignes 48 et 50 : Une fois l'instance `fen` de la classe `Application()` construite, celle-ci est lancée lors du démarrage du réceptionnaire d'événement par invocation de `mainloop()`. Remarquons que cette dernière méthode est héritée de la classe `Tk()`, parente de la classe `Application()`.

6. Python vous permet d'accéder aux attributs d'instance en utilisant la qualification des noms par points. D'autres langages de programmation l'interdisent, ou bien ne l'autorisent que moyennant une déclaration particulière des attributs (distinctions entre attributs *privés* et *publics*). Sachez en tout cas que **ce n'est pas recommandé** : le bon usage de la programmation orientée objet stipule que vous ne devez pouvoir accéder aux attributs d'objets que par l'intermédiaire de méthodes spécifiques (l'*interface*).

2.2 Dessin d'un petit train

Cet autre exemple vous montrera comment vous pouvez exploiter les connaissances que vous avez acquises précédemment, concernant les boucles, les listes et les fonctions, afin de réaliser de nombreux dessins avec seulement quelques lignes de code. Il pourra constituer une première étape dans la réalisation d'un logiciel de jeu. Il s'agit d'une fenêtre contenant un canevas et deux boutons. Lorsqu'on actionne le premier de ces deux boutons, un petit train apparaît dans le canevas. Lorsqu'on actionne le second bouton, quelques petits personnages apparaissent à certaines fenêtres des wagons :



Le but de deuxième exemple est de construire une application simple sur la base de plusieurs classes, afin d'examiner comment différents objets peuvent *s'échanger des informations par l'intermédiaire de leurs méthodes.* En effet, l'application ci-dessous comprend deux classes :

- La classe `Application()` sera obtenue par dérivation de la classe `Tk()` et mettra en place la fenêtre principale, son canevas et ses deux boutons.
- La classe `Wagon()`, indépendante de la première, permettra d'instancier dans un canevas 4 objets-wagons similaires, dotés chacun d'une méthode `perso()`. Celle-ci sera destinée à provoquer l'apparition d'un petit personnage à l'une quelconque des trois fenêtres du wagon. L'application principale invoquera cette méthode pour différents objets-wagons, afin de faire apparaître un choix de quelques personnages.

```
1  from tkinter import *
2
3  def cercle(can, x, y, r, coul = 'black'):
4      "dessin d'un cercle de rayon <r> en <x,y> dans le canevas <can>"
5      can.create_oval(x-r, y-r, x+r, y+r, outline = coul)
6
7  # ------- Définition des classes -------
8
9  class Application(Tk):
10     def __init__(self):
11         Tk.__init__(self)          # constructeur de la classe parente
12         self.can = Canvas(master=self, width = 475, height = 130, bg = "white")
13         self.can.pack(side = TOP, padx = 5, pady = 5)
14         bou1 = Button(master=self, text = "Train", command = self.dessine)
15         bou1.pack(side = LEFT)
16         bou2 = Button(master=self, text = "Hello", command = self.coucou)
17         bou2.pack(side = LEFT)
18
19     def dessine(self):
20         "instanciation de 4 wagons dans le canevas"
21         self.w1 = Wagon(self.can, 10, 30)
22         self.w2 = Wagon(self.can, 130, 30)
23         self.w3 = Wagon(self.can, 250, 30)
24         self.w4 = Wagon(self.can, 370, 30)
25
26     def coucou(self):
27         "apparition de personnages dans certaines fenêtres"
28         self.w1.perso(3)          # 1er wagon, 3e fenêtre
29         self.w3.perso(1)          # 3e wagon, 1e fenêtre
30         self.w3.perso(2)          # 3e wagon, 2e fenêtre
31         self.w4.perso(1)          # 4e wagon, 1e fenêtre
32
```

```
33 class Wagon(object):
34     def __init__(self, canevas, x, y):
35         "dessin d'un petit wagon en <x,y> dans le canevas <canev>"
36         # mémorisation des paramètres dans des variables d'instance :
37         self.canevas, self.x, self.y = canevas, x, y
38         # rectangle de base : 95x60 pixels :
39         canevas.create_rectangle(x, y, x+95, y+60)
40         # 3 fenêtres de 25x35 pixels, écartées de 5 pixels :
41         for xf in range(x+5, x+90, 30):
42             canevas.create_rectangle(xf, y+5, xf+25, y+40)
43         # 2 roues de rayon égal à 12 pixels  :
44         cercle(canevas, x+18, y+73, 12)
45         cercle(canevas, x+77, y+73, 12)
46
47     def perso(self, fen):
48         "apparition d'un petit personnage à la fenêtre <fen>"
49         # calcul des coordonnées du centre de chaque fenêtre :
50         xf = self.x + fen*30 -12
51         yf = self.y + 25
52         cercle(self.canevas, xf, yf, 10)       # visage
53         cercle(self.canevas, xf-5, yf-3, 2)    # oeil gauche
54         cercle(self.canevas, xf+5, yf-3, 2)    # oeil droit
55         cercle(self.canevas, xf, yf+5, 3)      # bouche
56
57 # Programme principal:
58
59 monApp =Application()    # instanciation de l'objet-application
60 monApp.mainloop()        # lancement du réceptionnaire d'événements
```

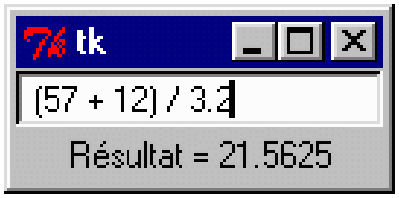
Analysons ce programme :

- Lignes 3 à 5 : Nous projetons de dessiner une série de petits cercles. Cette fonction nous facilitera le travail en nous permettant de définir ces cercles à partir de leur centre et leur rayon.
  Vous savez probablement déjà que la classe `Canvas()` de *tkinter* est dotée d'une méthode `create_oval()` qui permet de dessiner des ellipses quelconques (et donc aussi des cercles), mais cette méthode doit être invoquée avec quatre arguments qui seront les coordonnées des coins supérieur gauche et inférieur droit d'un rectangle fictif, dans lequel l'ellipse viendra alors s'inscrire. Cela n'est pas très pratique dans le cas particulier du cercle : il nous semblera plus naturel de commander ce tracé en fournissant les coordonnées de son centre ainsi que son rayon. C'est ce que nous obtiendrons avec notre fonction `cercle()`, laquelle invoque la méthode `create_oval()` en effectuant la conversion des coordonnées. Remarquez aussi que cette fonction attend un argument facultatif en ce qui concerne la couleur du cercle à tracer (noir par défaut).
- Lignes 9 à 17 : La classe principale de notre application est construite par dérivation de la classe de fenêtres `Tk()` importée du module `tkinter`.
  Les lignes 12 à 17 servent à instancier les widgets nécessaires : canevas et boutons. Remarquez que le canevas est référencé à l'aide d'une variable d'instance (**self.can**) accessible de partout à l'intérieur de l'objet. Comme dans le script précédent, les widgets sont mis en place dans la fenêtre à l'aide de leur méthode `pack()`, mais cette fois nous utilisons celle-ci avec des options :
  — l'option `side` peut accepter les valeurs TOP, BOTTOM, LEFT ou RIGHT, pour « pousser » le widget du côté correspondant dans la fenêtre. Ces noms écrits en majuscules sont en fait ceux d'une série de variables importées avec le module tkinter, et que vous pouvez considérer comme des « pseudo-constantes ».
  — les options `padx` et `pady` permettent de réserver un petit espace autour du widget. Cet espace est exprimé en nombre de pixels : `padx` réserve un espace à gauche et à droite du widget, `pady` réserve un espace au-dessus et au-dessous du widget.
- Lignes 19 à 24 : Cette méthode est invoquée lorsqu'on actionne le premier bouton et permet d'instancier les 4 objets-wagons, produits à partir de la classe correspondante.
  Nous voulons placer nos objets-wagons dans le canevas, à des emplacements bien précis : il nous faut donc transmettre quelques informations au constructeur de ces objets : au moins la référence du canevas, ainsi que les coordonnées souhaitées. Ces considérations nous font également entrevoir que lorsque nous définirons la classe `Wagon()`, un peu plus loin, nous devrons associer à sa méthode constructeur un nombre égal de paramètres afin de réceptionner ces arguments.

- Lignes 26 à 31 : Cette méthode est invoquée lorsqu'on actionne le second bouton. Elle invoque elle-même la méthode `perso()` de certains objets-wagons, avec des arguments différents, afin de faire apparaître les personnages aux fenêtres indiquées. Ces quelques lignes de code vous montrent donc comment un objet peut communiquer avec un autre, en faisant appel à ses méthodes. Il s'agit-là du mécanisme central de la programmation par objets : **Les objets sont des entités programmées qui s'échangent des messages et interagissent par l'intermédiaire de leurs méthodes.**
  La méthode `coucou()` devrait comporter quelques instructions complémentaires, lesquelles vérifieraient d'abord si les objets-wagons concernés existent bel et bien, avant d'autoriser l'activation d'une de leurs méthodes. Nous n'avons pas inclus ce genre de garde-fou afin que l'exemple reste aussi simple que possible, mais cela entraîne la conséquence que vous ne pouvez pas actionner le second bouton avant le premier.
- Ligne 33 : La classe **Wagon()** ne dérive d'aucune autre classe préexistante. Étant donné qu'il s'agit d'une classe d'objets graphiques, nous devons munir sa méthode constructeur de paramètres, afin de recevoir la référence du canevas auquel les dessins sont destinés, ainsi que les coordonnées de départ de ces dessins.
- Lignes 34 à 55 : Ces instructions ne nécessitent guère de commentaires. La méthode `perso()` est dotée d'un paramètre qui indique celle des 3 fenêtres où il faut faire apparaître un petit personnage. Ici aussi nous n'avons pas prévu de garde-fou : vous pouvez invoquer cette méthode avec un argument égal à 4 ou 5, par exemple, ce qui produira des effets incorrects.

2.3 Calculatrice minimaliste

Le petit script ci-dessous implémente une calculatrice complète, avec laquelle vous pourrez même effectuer des calculs comportant des parenthèses et des fonctions scientifiques. N'y voyez rien d'extraordinaire. Toute cette fonctionnalité n'est qu'une conséquence du fait que vous utilisez un interpréteur plutôt qu'un compilateur pour exécuter vos programmes.



Comme vous le savez, le compilateur n'intervient qu'une seule fois, pour traduire l'ensemble de votre code source en un programme exécutable. Son rôle est donc terminé **avant même** l'exécution du programme. L'interpréteur, quant à lui, est toujours actif **pendant** l'exécution du programme, et donc tout à fait disponible pour traduire un nouveau code source quelconque, comme une expression mathématique entrée au clavier par l'utilisateur.

Les langages interprétés disposent donc toujours de fonctions permettant d'évaluer une chaîne de caractères comme une suite d'instructions du langage lui-même. Il devient alors possible de construire en peu de lignes des structures de programmes très dynamiques. Dans l'exemple ci-dessous, nous utilisons la fonction intégrée `eval()` pour analyser l'expression mathématique entrée par l'utilisateur dans le champ prévu à cet effet, et nous n'avons plus ensuite qu'à afficher le résultat.

```
from tkinter import *
from math import *

class Calculatrice(Tk):

    def __init__(self):
        Tk.__init__(self)
        entree = Entry(master=self, bd = 2, relief = GROOVE)
        entree.bind("<Return>", self.evaluer)
        entree.pack(padx = 10, pady = 5)
        self.chaine = Label(master=self)
        self.chaine.pack(padx = 10, pady = 5)

    def evaluer(self, event):
        self.chaine.configure(text = "Résultat = {}".format(eval(self.entree.get())))


# ------ Corps principal du programme: ------

calculette = Calculatrice()
calculette.mainloop()
```

Nous commençons par importer les modules `tkinter` et `math`, ce dernier étant nécessaire afin que la dite calculatrice puisse disposer de toutes les fonctions mathématiques et scientifiques usuelles.

Le constructeur de notre classe `Calculatrice()` contient la phase d'initialisation du programme. On y trouve l'instanciation d'une fenêtre `Tk()` dérivée, un widget `self.chaine` instancié à partir de la classe `Label()` et un widget `entree` instancié à partir de la classe `Entry()`. Ce dernier widget est utilisé pour permettre à l'utilisateur d'entrer une expression numérique quelconque à l'aide de son clavier. Afin que ce widget puisse vraiment faire son travail, c'est-à-dire transmettre au programme l'expression que l'utilisateur y aura encodée, nous lui associons un événement à l'aide de la méthode **bind()**[7] :

```
entree.bind("<Return>",self.evaluer)
```

Cette instruction signifie : « *Lier l'événement <pression sur la touche Return> à l'objet <entree>, le gestionnaire de cet événement étant la méthode <self.evaluer>* ».

L'événement à prendre en charge est décrit dans une chaîne de caractères spécifique (dans notre exemple, il s'agit de la chaîne `<Return>`. Il existe un grand nombre de ces événements (mouvements et clics de la souris, enfoncement des touches du clavier, positionnement et redimensionnement des fenêtres, passage au premier plan, etc.). Vous trouverez la liste des chaînes spécifiques de tous ces événements dans certaines documentations propres à *tkinter* comme celle située à l'adresse http://tkinter.fdex.eu/doc/event.html.

Remarquez bien qu'il n'y a pas de parenthèses après le nom de la méthode `evaluer`. En effet, dans cette instruction, nous ne souhaitons pas déjà invoquer la méthode elle-même (ce serait prématuré) ; ce que nous voulons, c'est établir un lien entre un type d'événement particulier et cette méthode, de manière à ce qu'elle soit invoquée plus tard, chaque fois que l'événement se produira. Si nous mettions des parenthèses, l'argument qui serait transmis à la méthode `bind()` serait la valeur de retour de la méthode `evaluer` et non sa référence.

Finalement, remarquons que la méthode `evaluer()` représente la commande qui sera exécutée par le programme lorsque l'utilisateur actionnera la touche `Return` (ou `Enter`) après avoir entré une expression mathématique quelconque dans le champ d'entrée `self.entree`.

Cette méthode utilise la méthode `configure()` du widget `self.chaine`[8], pour modifier son attribut `text`. L'attribut en question reçoit donc ici une nouvelle valeur, déterminée par ce que nous avons écrit à la droite du signe égale : il s'agit en l'occurrence d'une chaîne de caractères construite dynamiquement, à l'aide d'une fonction intégrée dans Python : `eval()`, et de deux méthodes : `get` et `format`.

`eval()` fait appel à l'interpréteur pour évaluer une expression Python qui lui est transmise dans une chaîne de caractères. Le résultat de l'évaluation est fourni en retour sous forme de float.

Exemple :

```
>>> chaine = "(25 + 8)/3"      # chaîne contenant une expression mathématique
>>> res = eval(chaine)         # évaluation de l'expression contenue dans la chaîne
>>> print(res + 5)              # => le contenu de la variable res est numérique
16.0
```

La conversion en chaîne de caractères de la valeur numérique retournée par la fonction `eval` se fait à l'aide de la méthode `format` qui permet d'incorporer le résultat obtenu au message `Résultat =`.

`get()` est une méthode associée aux widget de la classe `Entry`. Cette méthode permet en quelque sorte « d'extraire » du widget `entree` la chaîne de caractères qui lui a été fournie par l'utilisateur.

7. En anglais, le mot *bind* signifie « lier ».
8. La méthode `configure()` peut s'appliquer à n'importe quel widget préexistant, pour en modifier les propriétés.

2.4 Détection et positionnement d'un clic de souris

Dans la définition de la méthode « evaluer » de l'exemple précédent, vous aurez remarqué que nous avons fourni un argument `event` (entre les parenthèses).

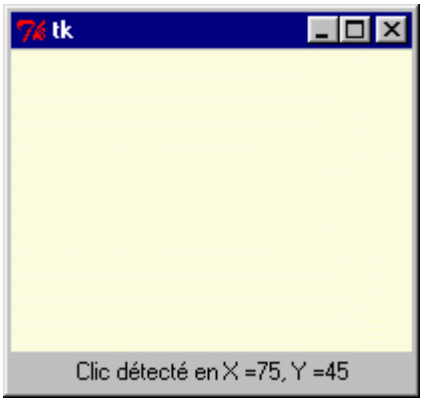
Cet argument est obligatoire[9]. Lorsque vous définissez une méthode gestionnaire d'événement qui est associée à un widget quelconque à l'aide de sa méthode `bind()`, vous devez toujours l'utiliser comme premier argument. Cet argument désigne en effet un objet créé automatiquement par *tkinter*, qui permet de transmettre au gestionnaire d'événement un certain nombre d'attributs de l'événement :

- le type d'événement : déplacement de la souris, enfoncement ou relâchement de l'un de ses boutons, appui sur une touche du clavier, entrée du curseur dans une zone prédéfinie, ouverture ou fermeture d'une fenêtre, etc.
- une série de propriétés de l'événement : l'instant où il s'est produit, ses coordonnées, les caractéristiques du ou des widget(s) concerné(s), etc.

Nous n'allons pas entrer dans trop de détails. Si vous voulez bien encoder et expérimenter le petit script ci-dessous, vous aurez vite compris le principe.

```
from tkinter import *

class Application(Tk):

    def __init__(self):
        Tk.__init__(self)

        cadre = Frame(master = self, width = 200, height = 150,
                      bg = 'light yellow')
        cadre.bind("<Button-1>", self.pointeur)
        cadre.pack()
        self.chaine = Label(self)
        self.chaine.pack()

    # Définition du gestionnaire d'événements:
    def pointeur(self, event):
        self.chaine.configure(text = "Clic détecté en X = {}, Y = {}"\
                              .format(event.x, event.y))

# Partie principale du programme:

fen = Application()

fen.mainloop()
```

Le script fait apparaître une fenêtre contenant un **cadre** (**Frame**) rectangulaire de couleur jaune pâle, dans lequel l'utilisateur est invité à effectuer des clics de souris.



La méthode `bind()` du widget `cadre` associe l'événement <**clic à l'aide du premier bouton de la souris**> au gestionnaire d'événement « self.pointeur ».

Ce gestionnaire d'événement peut utiliser les attributs `x` et `y` de l'objet `event` généré automatiquement par *tkinter*, pour construire la chaîne de caractères qui affichera la position de la souris au moment du clic.

9. La présence d'un argument est obligatoire, mais le nom *event* est une simple convention. Vous pourriez utiliser un autre nom quelconque à sa place, bien que cela ne soit pas recommandé.

2.5 Bouton radio

Les widgets « boutons radio » permettent de proposer à l'utilisateur un ensemble de choix mutuellement exclusifs. On les appelle ainsi par analogie avec les boutons de sélection que l'on trouvait jadis sur les postes de radio. Ces boutons étaient conçus de telle manière qu'un seul à la fois pouvait être enfoncé : tous les autres ressortaient automatiquement.

La caractéristique essentielle de ces widgets est qu'on les utilise toujours par groupes. Tous les boutons radio faisant partie d'un même groupe sont associés à une seule et même **variable tkinter**, mais chacun d'entre eux se voit aussi attribuer une valeur particulière.
Lorsque l'utilisateur sélectionne l'un des boutons, la valeur correspondant à ce bouton est affectée à la variable tkinter commune.

```
1  from tkinter import *
2
3  class RadioDemo(Frame):
4      """Démo : utilisation de widgets 'boutons radio'"""
5      def __init__(self, boss =None):
6          """Création d'un champ d'entrée avec 4 boutons radio"""
7          Frame.__init__(self)
8          self.pack()
9          # Champ d'entrée contenant un petit texte :
10         self.texte = Entry(master=self, width =30, font ="Arial 14")
11         self.texte.insert(END, "La programmation, c'est génial")
12         self.texte.pack(padx =8, pady =8)
13         # Nom français et valeur associée des quatre styles de police :
14         stylePoliceFr =["Normal", "Gras", "Italique", "Gras/Italique"]
15         stylePoliceTk =["normal", "bold", "italic"  , "bold italic"]
16         # Le style actuel est mémorisé dans un 'objet-variable' tkinter ;
17         self.choixPolice = StringVar()
18         self.choixPolice.set(stylePoliceTk[0])
19         # Création des quatre 'boutons radio' :
20         for n in range(4):
21             bout = Radiobutton(master = self,
22                                text = stylePoliceFr[n],
23                                variable = self.choixPolice,
24                                value = stylePoliceTk[n],
25                                command = self.changePolice)
26             bout.pack(side =LEFT, padx =5)
27
28     def changePolice(self):
29         """Remplacement du style de la police actuelle"""
30         police = "Arial 15 " + self.choixPolice.get()
31         self.texte.configure(font =police)
32
33 if __name__ == '__main__':
34     monApp = RadioDemo()
35     monApp.mainloop()
```

Rappelons que les lignes de code situées après l'instruction `if __name__ == '__main__' :` ne sont pas exécutées si le script est importé en tant que module dans une autre application. Si on lance le script comme application principale, par contre, ces instructions s'exécutent. Il s'agit d'un mécanisme intéressant permettant d'intégrer des instructions de test à l'intérieur des modules, même si ceux-ci sont destinés à être importés dans d'autres scripts.

Analysons ce code :

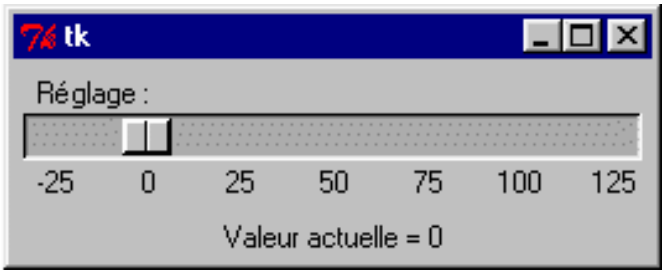
- Ligne 3 : Nous préférons construire ici notre petite application comme une classe dérivée de la classe `Frame()`, ce qui nous permettrait éventuellement plus tard de l'intégrer sans difficulté dans une application plus importante. Notons que *tkinter* autorise de construire la fenêtre principale d'une application par dérivation d'une classe de *widget* (le plus souvent, il s'agira d'un *widget* `Frame()` comme ici). La fenêtre englobant ce *widget* sera alors automatiquement ajoutée.
- Ligne 8 : En général, on applique une méthode de positionnement des widgets comme `pack()` après instanciation de ceux-ci, ce qui permet de choisir librement leur disposition à l'intérieur des fenêtres maîtresses. Comme nous le montrons ici, il est cependant tout à fait possible de déjà prévoir ce positionnement dans le constructeur du widget.
- Ligne 11 : Les widgets de la classe `Entry` disposent de plusieurs méthodes pour accéder à la chaîne de caractères affichée. La méthode `get()` permet de récupérer la chaîne entière. La méthode `insert()` permet d'insérer de nouveaux caractères à un emplacement quelconque (c'est-à-dire au début, à la fin, ou même à l'intérieur d'une chaîne préexistante éventuelle). Cette méthode s'utilise donc avec deux arguments, le premier indiquant l'emplacement de l'insertion (utilisez `0` pour insérer au début, `END` pour insérer à la fin, ou encore un indice numérique quelconque pour désigner un caractère dans la chaîne). La méthode `delete()` permet d'effacer tout ou partie de la chaîne. Elle s'utilise avec les mêmes arguments que la précédente.
- Lignes 14-15 : Plutôt que de les instancier dans des instructions séparées, nous préférons créer nos quatre boutons à l'aide d'une boucle. Les options spécifiques à chacun d'eux sont d'abord préparées dans les deux listes `stylePoliceFr` et `stylePoliceTk` : la première contient les petits textes qui devront s'afficher en regard de chaque bouton, et la seconde les valeurs qui devront leur être associées.
- Lignes 17-18 : Comme expliqué à la page précédente, les quatre boutons forment un groupe autour d'une variable commune. Cette variable prendra la valeur associée au bouton radio que l'utilisateur décidera de choisir. Nous ne pouvons cependant pas utiliser une variable ordinaire pour remplir ce rôle, parce que les attributs internes des objets tkinter ne sont accessibles qu'au travers de méthodes spécifiques. Nous utilisons donc ici un **objet-variable** tkinter, de type chaîne de caractères, que nous instancions à partir de la classe `StringVar()`, et auquel nous donnons une valeur par défaut à la ligne 18. Les *variables tkinter* issues des classes `StringVar(), IntVar(), DoubleVar()` ou `BooleanVar()` sont des objets qui se comportent comme des variables à l'intérieur des widgets *tkinter*. Ainsi, l'objet `self.choixPolice` contient l'équivalent d'une variable de type string, dans un format utilisable par *tkinter*. Pour accéder à sa valeur depuis Python, il faut utiliser des méthodes spécifiques de cette classe d'objets : la méthode `set()` permet de lui assigner une valeur et la méthode `get()` permet de la récupérer (ce qui est mis en pratique à la ligne 30).
- Lignes 20 à 26 : Instanciation des quatre boutons radio. Chacun d'entre eux se voit attribuer une étiquette et une valeur différentes, mais tous sont associés à la même *variable tkinter* commune (`self.choixPolice`). Tous invoquent également la même méthode `self.changePolice()`, chaque fois que l'utilisateur effectue un clic de souris sur l'un ou l'autre.
- Lignes 28 à 31 : Le changement de police s'obtient par re-configuration de l'option `font` du widget `Entry`. Cette option attend un tuple contenant le nom de la police, sa taille, et éventuellement son style. Si le nom de la police ne contient pas d'espaces, le tuple peut aussi être remplacé par une chaîne de caractères :

```
('Arial', 12, 'italic')
('Helvetica', 10)
('Times New Roman', 12, 'bold italic')
"Verdana 14 bold"
"President 18 italic"
```

2.6 Widget Scale

Le widget **Scale** se présente comme un curseur qui coulisse devant une échelle. Il permet à l'utilisateur de choisir rapidement la valeur d'un paramètre quelconque, d'une manière très attrayante.

Le petit script ci-dessous vous montre comment le paramétrer et l'utiliser dans une fenêtre :




```
from tkinter import *

class Curseur(Tk):

    def __init__(self):
        Tk.__init__(self)

        curseur = Scale(master = self, length = 250, orient = HORIZONTAL,
                        label = 'Réglage :', troughcolor = 'dark grey',
                        sliderlength = 20, showvalue = 0, from_ = -25,
                        to = 125, tickinterval = 25, command = self.updateLabel)
        curseur.pack()

        self.lab = Label(master=self)
        self.lab.pack()

    def updateLabel(self, x):
        self.lab.configure(text = 'Valeur actuelle = ' + str(x))

# Partie principale du programme:

root = Curseur()
root.mainloop()
```

Ces lignes ne nécessitent guère de commentaires.

Vous pouvez créer des widgets `Scale` de n'importe quelle taille (option `length`), en orientation horizontale (comme dans notre exemple) ou verticale (option `orient = VERTICAL`).
Les options `from_` (attention : n'oubliez pas le caractère 'souligné', lequel est nécessaire afin d'éviter la confusion avec le mot réservé `from`!) et `to` définissent la plage de réglage. L'intervalle entre les repères numériques est défini dans l'option `tickinterval`, etc.

La méthode désignée dans l'option `command` est appelée automatiquement chaque fois que le curseur est déplacé, et la position actuelle du curseur par rapport à l'échelle lui est transmise en argument. Il est donc très facile d'utiliser cette valeur pour effectuer un traitement quelconque. Considérez par exemple le paramètre `x` de la méthode `updateLabel()`, dans notre exemple.

Le widget `Scale` constitue une interface très intuitive et attrayante pour proposer différents réglages aux utilisateurs de vos programmes.

3 Les classes de widgets tkinter

Il existe 15 classes de base pour les widgets tkinter :

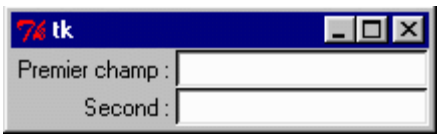
Widget	Description
`Button`	Un bouton classique, à utiliser pour provoquer l'exécution d'une commande quelconque.
`Canvas`	Un espace pour disposer divers éléments graphiques. Ce widget peut être utilisé pour dessiner, créer des éditeurs graphiques, et aussi pour implémenter des widgets personnalisés.
`Checkbutton`	Une case à cocher qui peut prendre deux états distincts (la case est cochée ou non). Un clic sur ce widget provoque le changement d'état.
`Entry`	Un champ d'entrée, dans lequel l'utilisateur du programme pourra insérer un texte quelconque à partir du clavier.
`Frame`	Une surface rectangulaire dans la fenêtre, où l'on peut disposer d'autres widgets. Cette surface peut être colorée. Elle peut aussi être décorée d'une bordure.
`Label`	Un texte (ou libellé) quelconque (éventuellement une image).
`Listbox`	Une liste de choix proposés à l'utilisateur, généralement présentés dans une sorte de boîte. On peut également configurer la `Listbox` de telle manière qu'elle se comporte comme une série de « boutons radio » ou de cases à cocher.
`Menu`	Un menu. Ce peut être un menu déroulant attaché à la barre de titre, ou bien un menu « *pop up* » apparaissant n'importe où à la suite d'un clic.
`Menubutton`	Un bouton-menu, à utiliser pour implémenter des menus déroulants.
`Message`	Permet d'afficher un texte. Ce widget est une variante du widget `Label`, qui permet d'adapter automatiquement le texte affiché à une certaine taille ou à un certain rapport largeur/hauteur.
`Radiobutton`	Représente (par un point noir dans un petit cercle) une des valeurs d'une variable qui peut en posséder plusieurs. Cliquer sur un bouton radio donne la valeur correspondante à la variable, et « vide » tous les autres boutons radio associés à la même variable.
`Scale`	Vous permet de faire varier de manière très visuelle la valeur d'une variable, en déplaçant un curseur le long d'une règle.
`Scrollbar`	Ascenseur ou barre de défilement que vous pouvez utiliser en association avec les autres widgets : Canvas, Entry, Listbox, Text.
`Text`	Affichage de texte formaté. Permet aussi à l'utilisateur d'éditer le texte affiché. Des images peuvent également être insérées.
`Toplevel`	Une fenêtre affichée séparément, au premier plan.

Ces classes de widgets intègrent chacune un grand nombre de méthodes. On peut aussi leur associer (lier) des événements, comme nous venons de le voir dans les pages précédentes. Vous allez apprendre en outre que tous ces widgets peuvent être positionnés dans les fenêtres à l'aide de trois méthodes différentes : la méthode `grid()`, la méthode `pack()` et la méthode `place()`.

L'utilité de ces méthodes apparaît clairement lorsque l'on s'efforce de réaliser des programmes portables (c'est-à-dire susceptibles de fonctionner de manière identique sur des systèmes d'exploitation aussi différents que **Unix**, **Mac OS** ou **Windows**), et dont les fenêtres soient **redimensionnables.**

3.1 Méthode `grid()` pour contrôler la disposition des widgets

Jusqu'à présent, nous avons toujours disposé les widgets dans leur fenêtre à l'aide de la méthode `pack()`. Cette méthode présentait l'avantage d'être extraordinairement simple, mais elle ne nous donnait pas beaucoup de liberté pour disposer les widgets à notre guise. Comment faire, par exemple, pour obtenir la fenêtre ci-contre ?



Nous pourrions effectuer un certain nombre de tentatives en fournissant à la méthode `pack()` des arguments de type « `side =` », comme nous l'avons déjà fait précédemment, mais cela ne nous mène pas très loin. Essayons par exemple :

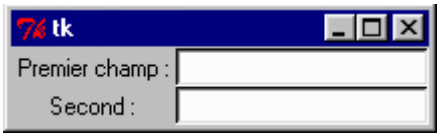
```
from tkinter import *

fen = Tk()
txt1 = Label(fen, text = 'Premier champ :')
txt2 = Label(fen, text = 'Second :')
entr1 = Entry(fen)
entr2 = Entry(fen)
txt1.pack(side = LEFT)
txt2.pack(side = LEFT)
entr1.pack(side = RIGHT)
entr2.pack(side = RIGHT)

fen.mainloop()
```

... mais le résultat n'est pas vraiment celui que nous recherchions !

Pour mieux comprendre comment fonctionne la méthode `pack()`, vous pouvez encore essayer différentes combinaisons d'options, telles que `side=TOP`, `side=BOTTOM`, pour chacun de ces quatre widgets. Mais vous n'arriverez certainement pas à obtenir ce qui vous a été demandé. Vous pourriez peut-être y parvenir en définissant deux widgets `Frame()` supplémentaires, et en y incorporant ensuite séparément les widgets `Label()` et `Entry()`. Cela devient fort compliqué.

Il est temps que nous apprenions à utiliser une autre approche du problème. Veuillez donc analyser le script ci-dessous : il contient en effet (presque) la solution :

```
from tkinter import *

fen = Tk()
txt1 = Label(fen, text = 'Premier champ :')
txt2 = Label(fen, text = 'Second :')
entr1 = Entry(fen)
entr2 = Entry(fen)
txt1.grid(row = 0)
txt2.grid(row = 1)
entr1.grid(row =0, column = 1)
entr2.grid(row =1, column = 1)
fen.mainloop()
```



Dans ce script, nous avons donc remplacé la méthode `pack()` par la méthode `grid()`. Comme vous pouvez le constater, l'utilisation de la méthode `grid()` est simple. Cette méthode considère la fenêtre comme un tableau (ou une grille). Il suffit alors de lui indiquer dans quelle ligne (`row`) et dans quelle colonne (`column`) de ce tableau on souhaite placer les widgets. On peut numéroter les lignes et les colonnes comme on veut, en partant de zéro, ou de un, ou encore d'un nombre quelconque : tkinter ignorera les lignes et colonnes vides. Cependant, si vous ne fournissez aucun numéro pour une ligne ou une colonne, la valeur par défaut sera zéro.

Tkinter détermine automatiquement le nombre de lignes et de colonnes nécessaire. Mais ce n'est pas tout : si vous examinez en détail la petite fenêtre produite par le script ci-dessus, vous constaterez que nous n'avons pas encore tout à fait atteint le but poursuivi. Les deux chaînes apparaissant dans la partie gauche de la fenêtre

sont **centrées**, alors que nous souhaitions les **aligner** l'une et l'autre par la droite. Pour obtenir ce résultat, il nous suffit d'ajouter un argument dans l'appel de la méthode `grid()` utilisée pour ces widgets. L'option `sticky` peut prendre l'une des quatre valeurs `N, S, W, E` (les quatre points cardinaux en anglais). En fonction de cette valeur, on obtiendra un alignement des widgets par le haut, par le bas, par la gauche ou par la droite. Remplacez donc les deux premières instructions `grid()` du script par :

```
txt1.grid(row =0, sticky =E)
txt2.grid(row =1, sticky =E)
```

... et vous atteindrez enfin exactement le but recherché.

Analysons à présent la fenêtre suivante :

Cette fenêtre comporte 3 colonnes : une première avec les 3 chaînes de caractères, une seconde avec les 3 champs d'entrée, et une troisième avec l'image. Les deux premières colonnes comportent chacune 3 lignes, mais l'image située dans la dernière colonne s'étale en quelque sorte sur les trois.

Le code correspondant est le suivant :

```
class Fenetre(Tk):

    def __init__(self):
        Tk.__init__(self)

        # Création de widgets 'Label' et 'Entry':
        self.txt1 = Label(self, text = 'Premier champ : ')
        self.txt2 = Label(self, text = 'Second : ')
        self.txt3 = Label(self, text = 'Troisième : ')
        self.entr1 = Entry(self)
        self.entr2 = Entry(self)
        self.entr3 = Entry(self)

        # Création d'un widget 'Canvas' contenant une image bitmap:
        self.can = Canvas(self, width = 160, height = 160, bg = 'white')
        self.photo = PhotoImage(file = 'martin_p.gif')
        self.item = self.can.create_image(80, 80, image = self.photo)

        # Mise en page à l'aide de la méthode 'grid' :
        self.txt1.grid(row = 1, sticky = E)
        self.txt2.grid(row = 2, sticky = E)
        self.txt3.grid(row = 3, sticky = E)
        self.entr1.grid(row = 1, column = 2)
        self.entr2.grid(row = 2, column = 2)
        self.entr3.grid(row = 3, column = 2)
        self.can.grid(row = 1, column = 3, rowspan = 3, padx = 10, pady = 5)


# démarrage :
fen = Fenetre()
fen.mainloop()
```

Pour pouvoir faire fonctionner ce script, il vous faudra probablement remplacer le nom du fichier image (*martin_p.gif*) par le nom d'une image de votre choix. Attention : la bibliothèque tkinter standard n'accepte qu'un petit nombre de formats pour cette image. Choisissez de préférence le format GIF [10].

Nous pouvons remarquer un certain nombre de choses dans ce script :

(1) La technique utilisée pour incorporer une image :
tkinter ne permet pas d'insérer directement une image dans une fenêtre. Il faut d'abord installer un canevas, et ensuite positionner l'image dans celui-ci. Étant donné qu'il existe de nombreux types d'images, nous devons en outre déclarer l'objet image comme étant un bitmap GIF, à partir de la classe `PhotoImage()`.

(2) La ligne où nous installons l'image dans le canevas est la ligne :
`self.item = self.can.create_image(80, 80, image = self.photo)`
Nous utilisons ici la méthode `create_image()` associée à l'attribut `self.can` (lequel est lui-même une instance de la classe `Canvas`). Les deux premiers arguments transmis (`80, 80`) indiquent les coordonnées x et y du canevas où il faut placer le centre de l'image. Les dimensions du canevas étant de 160x160, notre choix aboutira donc à un centrage de l'image au milieu du canevas.

(3) La numérotation des lignes et colonnes dans la méthode `grid()` :
La numérotation des lignes et des colonnes dans la méthode `grid()` utilisée ici commence cette fois à partir de 1 (et non à partir de zéro comme dans le script précédent). Ce choix de numérotation est tout à fait libre. Numéroter à partir de 1 augmente probablement la lisibilité de notre code.

(4) Les arguments utilisés avec `grid()` pour positionner le canevas :
`self.can.grid(row = 1, column = 3, rowspan = 3, padx = 10, pady = 5)`
Les deux premiers arguments indiquent que le canevas sera placé dans la première ligne de la troisième colonne. Le troisième (`rowspan = 3`) indique qu'il pourra « s'étaler » sur trois lignes. Les deux derniers (`padx = 10, pady = 5`) indiquent la dimension de l'espace qu'il faut réserver autour de ce widget.

3.2 Composition d'instructions pour écrire un code plus compact

Il est souvent possible (et souhaitable) de retravailler un script afin de le rendre plus compact. Vous pouvez par exemple assez fréquemment utiliser la composition d'instructions pour appliquer la méthode de mise en page des widgets au moment même où vous créez ces widgets. Le code correspondant devient alors un peu plus simple, et parfois plus lisible. Vous pouvez par exemple remplacer les deux lignes :

```
txt1 = Label(fen, text = 'Premier champ :')
txt1.grid(row = 1, sticky = E)
```

du script précédent par une seule, telle que :

```
Label(fen, text =' Premier champ :').grid(row = 1, sticky = E)
```

Dans cette nouvelle écriture, vous pouvez constater que nous faisons l'économie de la variable intermédiaire `txt1`. Nous avions utilisé cette variable pour bien dégager les étapes successives de notre démarche, mais elle n'est pas toujours indispensable. Le simple fait d'invoquer la classe `Label()` provoque en effet l'instanciation d'un objet de cette classe, même si l'on ne mémorise pas la référence de cet objet dans une variable (*tkinter* la conserve de toute façon dans sa représentation interne de la fenêtre). Si l'on procède ainsi, la référence est perdue pour le restant du script, mais elle peut tout de même être transmise à une méthode de mise en page telle que `grid()` au moment même de l'instanciation, en une seule instruction composée.

Jusqu'à présent, nous avons créé des objets divers (par instanciation à partir d'une classe quelconque), en les affectant à chaque fois à des variables. Par exemple, lorsque nous avons écrit :

```
txt1 = Label(fen, text ='Premier champ :')
```

nous avons créé une instance de la classe `Label()`, que nous avons assignée à la variable `txt1`.

La variable `txt1` peut alors être utilisée pour faire référence à cette instance, partout ailleurs dans le script, mais dans les faits nous ne l'utilisons qu'une seule fois pour lui appliquer la méthode `grid()`, le widget dont il est question n'étant rien d'autre qu'une simple étiquette descriptive. Or, créer ainsi une nouvelle variable pour n'y faire référence ensuite qu'une seule fois (et directement après sa création) n'est pas une pratique très recommandable, puisqu'elle consiste à réserver inutilement un certain espace mémoire.

10. D'autres formats d'image sont possibles, mais à la condition de les traiter à l'aide des modules graphiques de la bibliothèque PIL (*Python Imaging Library*), qui est une extension de Python disponible sur : http://www.pythonware.com/products/pil/.

Lorsque ce genre de situation se présente, il est plus judicieux d'utiliser la composition d'instructions. Par exemple, on préférera le plus souvent remplacer les deux instructions :

```
somme = 45 + 72
print(somme)
```

par une seule instruction composée, telle que :

```
print(45 + 72)
```

on fait ainsi l'économie d'une variable.

De la même manière, lorsque l'on met en place des widgets auxquels on ne souhaite plus faire référence par la suite, comme c'est souvent le cas pour les widgets de la classe `Label()`, on peut en général appliquer la méthode de mise en page directement au moment de la création du widget, en une seule instruction composée.

Cela s'applique seulement aux widgets qui ne sont plus référencés après qu'on les ait créés. **Tous les autres doivent impérativement être assignés à des variables, afin que l'on puisse encore interagir avec eux ailleurs dans le script.** Et dans ce cas, il faut obligatoirement utiliser deux instructions distinctes, l'une pour instancier le widget, et l'autre pour lui appliquer ensuite la méthode de mise en page. Vous ne pouvez pas, par exemple, construire une instruction composée telle que :

```
entree = Entry(fen).pack()      # faute de programmation !!!
```

En apparence, cette instruction devrait instancier un nouveau widget et l'assigner à la variable `entree`, la mise en page s'effectuant dans la même opération à l'aide de la méthode `pack()`. Dans la réalité, cette instruction produit bel et bien un nouveau widget de la classe `Entry()`, et la méthode `pack()` effectue bel et bien sa mise en page dans la fenêtre, mais la valeur qui est mémorisée dans la variable `entree` n'est pas la référence du widget ! C'est **la valeur de retour de la méthode** `pack()`, à savoir un objet vide (`None`).

Pour obtenir une vraie référence du widget, vous devez obligatoirement utiliser deux instructions :

```
entree = Entry(fen)    # instanciation du widget
entree.pack()          # application de la mise en page
```

Lorsque vous utilisez la méthode `grid()`, vous pouvez simplifier encore un peu votre code, en omettant l'indication de nombreux numéros de lignes et de colonnes. À partir du moment où c'est la méthode `grid()` qui est utilisée pour positionner les widgets, tkinter considère en effet qu'il existe forcément des lignes et des colonnes [11]. Si un numéro de ligne ou de colonne n'est pas indiqué, le widget correspondant est placé dans la première case vide disponible. Le script ci-dessous intègre les simplifications que nous venons d'expliquer :

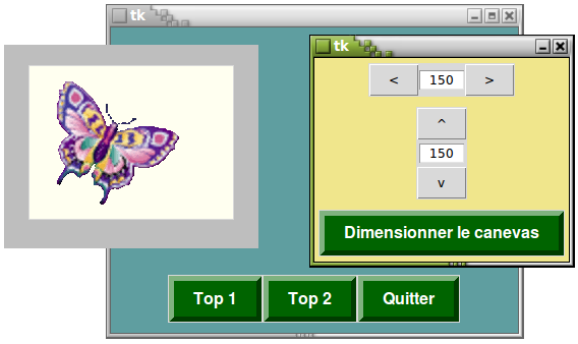
```
from tkinter import *

class Fenetre(Tk):

    def __init__(self):
        Tk.__init__(self)

        # Création et positionnement de widgets 'Label' et 'Entry':
        Label(self, text = 'Premier champ : ').grid(sticky = E)
        Label(self, text = 'Second : ').grid(sticky = E)
        Label(self, text = 'Troisième : ').grid(sticky = E)
        self.entr1, self.entr2, self.entr3 = Entry(self), Entry(self), Entry(self)
        self.entr1.grid(row = 0, column = 1)    # pour pouvoir référencer ces widgets
        self.entr2.grid(row = 1, column = 1)    # plus tard, il faut assigner à chacun
        self.entr3.grid(row = 2, column = 1)    # une variable distincte
        # Création d'un widget 'Canvas' contenant une image bitmap:
        self.can = Canvas(self, width = 160, height = 160, bg = 'white')
        self.photo = PhotoImage(file = 'martin_p.gif')
        self.can.create_image(80, 80, image = self.photo)
        self.can.grid(row = 0, column = 2, rowspan = 4, padx = 10, pady = 5)

# démarrage :
fen = Fenetre()
fen.mainloop()
```

11. Surtout, n'utilisez pas plusieurs méthodes de positionnement différentes dans la même fenêtre ! Les méthodes `gris()`, `pack()` et `place()` sont mutuellement exclusives.

3.3 Application à fenêtres multiples - paramétrage implicite

La classe `Toplevel()` de *tkinter* permet de créer des fenêtres « satellites » de votre application principale. Ces fenêtres sont autonomes, mais elles se refermeront automatiquement lorsque vous fermerez la fenêtre principale. Cette restriction mise à part, elles se traitent de manière habituelle.

La petite application ci-après vous montre quelques-unes de leurs possibilités. Elle est constituée d'une fenêtre principale très ordinaire, contenant simplement trois boutons. Ces boutons sont créés à partir d'une classe dérivée de la classe `Button()` de base, ceci afin de vous montrer encore une fois combien il est facile d'adapter les classes d'objets existantes à vos besoins. Vous pourrez noter au passage quelques options « décoratives » intéressantes.



Le bouton <Top1> fait apparaître une première fenêtre satellite contenant un canevas avec une image. Particulière, cette fenêtre ne possède ni bandeau de titre, ni bordure, et il est impossible de la redimensionner à l'aide de la souris. De plus cette fenêtre est **modale** : on qualifie ainsi une fenêtre qui reste toujours au premier plan, devant toutes les autres fenêtres d'application éventuellement présentes à l'écran.

Le bouton <Top2> fait apparaître une deuxième fenêtre satellite plus classique, qui contient deux exemplaires d'un petit widget composite `SpinBox`. Ce widget se compose de deux boutons et d'un libellé indiquant une valeur numérique. Les boutons permettent d'augmenter ou de diminuer la valeur affichée. En plus de ces deux SpinBoxes, la fenêtre contient un gros bouton décoré. En l'actionnant, l'utilisateur provoque le redimensionnement du canevas dans l'autre fenêtre satellite, en accord avec les valeurs numériques affichées dans les deux SpinBoxes.

```
from tkinter import *

class FunnyButton(Button):
    "Bouton de fantaisie : vert virant au rouge quand on l'actionne"
    def __init__(self, boss, **Arguments):
        Button.__init__(self, boss,  bg ="dark green", fg ="white", bd =5,
                        activebackground ="red", activeforeground ="yellow",
                        font =('Helvetica', 12, 'bold'), **Arguments)

class SpinBox(Frame):
    "widget composite comportant un Label et 2 boutons 'up' & 'down'"
    def __init__(self, boss, largC=5, largB =2, vList=[0], liInd=0, orient =Y):
        Frame.__init__(self, boss)
        self.vList =vList           # liste des valeurs à présenter
        self.liInd =liInd           # index de la valeur à montrer par défaut
        if orient ==Y:
            s, augm, dimi = TOP, "^", "v"      # Orientation 'verticale'
        else:
            s, augm, dimi = RIGHT, ">", "<"    # Orientation 'horizontale'
        Button(self, text =augm, width =largB, command =self.up).pack(side =s)
        self.champ = Label(self, bg ='white', width =largC,
                           text =str(vList[liInd]), relief =SUNKEN)
        self.champ.pack(pady =3, side =s)
        Button(self, text=dimi, width=largB, command =self.down).pack(side =s)

    def up(self):
        if self.liInd < len(self.vList) -1:
            self.liInd += 1
        else:
            self.bell()        # émission d'un bip
        self.champ.configure(text =str(self.vList[self.liInd]))

```

```
33      def down(self):
34          if self.liInd > 0:
35              self.liInd -= 1
36          else:
37              self.bell()         # émission d'un bip
38          self.champ.configure(text =str(self.vList[self.liInd]))
39
40      def get(self):
41          return self.vList[self.liInd]
42
43  class FenDessin(Toplevel):
44      "Fenêtre satellite (modale) contenant un simple canevas"
45      def __init__(self, boss, **Arguments):
46          Toplevel.__init__(self, boss, **Arguments)
47          self.geometry("250x200+100+240")
48          self.overrideredirect(1)            # => fenêtre sans bordure ni bandeau
49          self.transient(self.master)         # => fenêtre 'modale'
50          self.can =Canvas(self, bg="ivory", width =200, height =150)
51          self.img = PhotoImage(file ="papillon.gif")
52          self.can.create_image(90, 80, image =self.img)
53          self.can.pack(padx =20, pady =20)
54
55  class FenControle(Toplevel):
56      "Fenêtre satellite contenant des contrôles de redimensionnement"
57      def __init__(self, boss, **Arguments):
58          Toplevel.__init__(self, boss, **Arguments)
59          self.geometry("250x200+400+230")
60          self.resizable(width =0, height =0)     # => empêche le redimensionnement
61          p =(10, 30, 60, 90, 120, 150, 180, 210, 240, 270, 300)
62          self.spX =SpinBox(self, largC=5,largB =1,vList =p,liInd=5,orient =X)
63          self.spX.pack(pady =5)
64          self.spY =SpinBox(self, largC=5,largB =1,vList =p,liInd=5,orient =Y)
65          self.spY.pack(pady =5)
66          FunnyButton(self, text ="Dimensionner le canevas",
67                      command =boss.redimF1).pack(pady =5)
68
69  class Demo(Tk):
70      def __init__(self):
71          Tk.__init__(self)
72          self.config(bg = 'cadet blue')
73          self.geometry('400x340+200+200')
74          Frame(master=self, width=400, height=300, bg = 'cadet blue').grid(row = 1,
                column = 1, columnspan = 5)
75          FunnyButton(self, text ="Top 1",
76                      command =self.top1).grid(row = 2, column = 2)
77          FunnyButton(self, text ="Top 2",
78                      command =self.top2).grid(row = 2, column = 3)
79          FunnyButton(self, text ="Quitter",
80                      command =self.quit).grid(row = 2, column = 4)
81
82      def top1(self):
83          self.fen1 =FenDessin(self, bg ="grey")
84
85      def top2(self):
86          self.fen2 =FenControle(self, bg ="khaki")
87
88      def redimF1(self):
89          dimX, dimY = self.fen2.spX.get(), self.fen2.spY.get()
90          self.fen1.can.config(width =dimX, height =dimY)
91
92  if __name__ =="__main__":               # --- Programme de test ---
93      demo = Demo()
94      demo.mainloop()
95      demo.destroy()
```

Analysons ce code :

- Lignes 3 à 8 : si vous souhaitez pouvoir disposer du même style de boutons à différents endroits de votre projet, n'hésitez pas à créer une classe dérivée, comme nous l'avons fait ici. Cela vous évitera d'avoir à reprogrammer partout les mêmes options spécifiques.
  Notez bien les deux astérisques qui préfixent le nom du dernier paramètre du constructeur : `**Arguments`. Elles signifient que la variable correspondante sera en fait un *dictionnaire*, capable de réceptionner automatiquement n'importe quel ensemble d'arguments « avec étiquettes ». Ces arguments pourront alors être transmis tels quels au constructeur de la classe parente (à la ligne 8). Cela nous évite d'avoir à reproduire dans notre liste de paramètres toutes les options de paramétrage du bouton de base, qui sont fort nombreuses. Ainsi, vous pourrez instancier ces boutons de fantaisie avec n'importe quelle combinaison d'options, du moment que celles-ci soient disponibles pour les boutons de base. On appelle ce qui précède un **paramétrage implicite**. Vous pouvez utiliser cette forme de paramétrage avec n'importe quelle fonction ou méthode.
- Lignes 10 à 24 : le constructeur de notre widget `SpinBox` ne nécessite guère de commentaires. En fonction de l'orientation souhaitée, la méthode `pack()` disposera les boutons et le libellé de haut en bas ou de gauche à droite (arguments TOP ou RIGHT pour son option `side`).
- Lignes 26 à 38 : Ces deux méthodes ne font rien d'autre que de modifier la valeur affichée dans le libellé. Notez au passage que la classe `Frame()` dispose d'une méthode `bell()` pour provoquer l'émission d'un « bip » sonore.
- Lignes 43 à 53 : La première fenêtre satellite de type `Toplevel` est définie ici. Remarquez à nouveau l'utilisation du **paramétrage implicite** du constructeur, à l'aide de la variable `**Arguments`. C'est lui qui nous permet d'instancier cette fenêtre (à la ligne 83) en spécifiant une couleur de fond (nous pourrions aussi demander une bordure, etc). Les méthodes invoquées aux lignes 47 à 49 définissent quelques propriétés particulières (applicables à n'importe quelle fenêtre). La méthode `geometry()` permet de fixer à la fois les dimensions de la fenêtre (`"250x200"`) et son emplacement à l'écran (l'indication `+100+240` signifie que son coin supérieur gauche doit être décalé de 100 pixels vers la droite et de 240 pixels vers le bas, par rapport au coin supérieur gauche de l'écran).
- Lignes 55 à 67 : Veuillez remarquer le paramètre `boss`, qui est présent dans le constructeur de `FenControle`. Ce paramètre sert à transmettre la référence du widget « maître » à son « esclave ». Il est très généralement nécessaire (à la ligne 67, par exemple, nous nous en servons pour référencer une méthode de l'application principale), mais il n'est pas absolument indispensable. Remarquez à la ligne 60 la méthode permettant d'empêcher le redimensionnement d'une fenêtre (dans le cas d'une fenêtre sans bordure ni bandeau-titre, comme `FenDessin`, cette méthode est inutile).
- Lignes 88 à 90 : Cette méthode récupère les valeurs numériques affichées dans la fenêtre de contrôle (méthode `get()` de ce widget), pour redimensionner le canevas de la fenêtre de dessin. Cet exemple simple vous montre une fois de plus comment peuvent s'établir les communications entre divers composants de votre programme.

3.4 Barres d'outils - expressions lambda

De nombreux programmes comportent une ou plusieurs « barres d'outils » (**toolbar**) constituées de petits boutons sur lesquels sont représentés des pictogrammes. Cette façon de faire permet de proposer à l'utilisateur un grand nombre de commandes spécialisées, sans que celles-ci n'occupent une place excessive à l'écran.

L'application décrite ci-après comporte une barre d'outils et un canevas. Lorsque l'utilisateur clique sur l'un des 8 premiers boutons de la barre, le pictogramme qu'il porte est recopié dans le canevas, à un emplacement choisi au hasard. Lorsqu'il clique sur le dernier bouton, le contenu du canevas est effacé.

```
from tkinter import *
from random import randrange

class ToolBar(Frame):
    "Barre d'outils (petits boutons avec icônes)"
    def __init__(self, boss, images =[], command =None):
        Frame.__init__(self, boss, bd =1)
        # <images> = liste des noms d'icônes à placer sur les boutons
        self.command =command            # commande à exécuter lors du clic
        nBou =len(images)                 # Nombre de boutons à construire
        # Les icônes des boutons doivent être placées dans des variables
        # persistantes. Une liste fera l'affaire :
        self.photoI =[None]*nBou
        for b in range(nBou):
            # Création de l'icône (objet PhotoImage Tkinter) :
            self.photoI[b] =PhotoImage(file = images[b] +'.gif')
            # Création du bouton. On fait appel à une fonction lambda
            # pour pouvoir transmettre un argument à la méthode <action> :
            bou = Button(self, image =self.photoI[b], bd =2, relief =GROOVE,
                         command = lambda arg =b: self.action(arg))
            bou.pack(side =LEFT)

    def action(self, index):
        # Exécuter <command> avec l'indice du bouton comme argument :
        self.command(index)

class Application(Frame):
    def __init__(self):
        Frame.__init__(self)
        # noms des fichiers contenant les icones (format GIF):
        icones =('floppy_2', 'coleo', 'papi2', 'pion_1', 'pion_2', 'pion_3',
                 'pion_4', 'help_4', 'clear')
        # Création de la barre d'outils :
        self.barOut =ToolBar(self, images =icones, command =self.transfert)
        self.barOut.pack(expand =YES, fill =X)
        # Création du canevas destiné à recevoir les images :
        self.ca = Canvas(self, width =400, height =200, bg ='orange')
        self.ca.pack()
        self.pack()

    def transfert(self, b):
        if b ==8:
            self.ca.delete(ALL)        # Effacer tout dans le canevas
        else:
            # Recopier l'icône du bouton b (extraite de la barre) => canevas :
            x, y = randrange(25,375), randrange(25,175)
            self.ca.create_image(x, y, image =self.barOut.photoI[b])

Application().mainloop()
```

L'exécution du script précédent lancera la fenêtre suivante :

Métaprogrammation - expressions lambda

Vous savez qu'en règle générale, on associe à chaque bouton une **commande**, laquelle est la référence d'une méthode ou d'une fonction particulière qui se charge d'effectuer le travail lorsque le bouton est activé. Or, dans l'application présente, tous les boutons doivent faire à peu près la même chose (recopier un dessin dans le canevas), la seule différence entre eux étant le dessin concerné.

Pour simplifier le code, nous voudrions pouvoir associer l'option `command` de tous nos boutons **avec une seule et même méthode** (ce sera la méthode `action()`), mais en lui transmettant à chaque fois la référence du bouton particulier utilisé, de manière à ce que l'action accomplie puisse être différente pour chacun d'eux.

Une difficulté se présente, cependant, parce que l'option **command** du widget Button accepte seulement une **valeur** ou une **expression**, et non une **instruction**. Il s'agit en fait de lui indiquer la référence d'une fonction, mais certainement pas d'invoquer la fonction à cet endroit avec des arguments (l'invocation ne pourra avoir lieu en effet que lorsque l'utilisateur cliquera sur le bouton : c'est alors le réceptionnaire d'événements de *tkinter* qui devra la provoquer). C'est la raison pour laquelle on indique seulement le nom de la fonction, **sans parenthèses**. Cette difficulté peut être résolue à l'aide des **expression lambda**. Rappelons que ce mot réservé Python désigne une **expression** qui renvoie un **objet fonction**, similaire à ceux que vous créez avec l'instruction **def**, mais avec la différence que **lambda** étant une expression et non une instruction, on peut l'utiliser comme interface afin d'invoquer une fonction (avec passage d'arguments) là où ce n'est normalement pas possible. Notez au passage qu'une telle fonction est **anonyme** (elle ne possède pas de nom). Par exemple, l'expression

```
lambda ar1=b, ar2=c : bidule(ar1,ar2)
```

renvoie la référence d'une fonction anonyme créée à la volée, qui pourra elle-même invoquer la fonction `bidule()` en lui transmettant les arguments `b` et `c` , ceux-ci étant utilisés comme valeurs par défaut dans la définition des paramètres `ar1` et `ar2` de la fonction.

Comprenons bien la ligne 20 du programme ci-dessus :

```
command = lambda arg =b: self.action(arg)
```

Dans cette portion d'instruction, la commande associée au bouton se réfère à une fonction anonyme dont le paramètre `arg` possède une valeur par défaut : la valeur de l'argument `b`.
Invoquée sans argument par la commande, cette fonction anonyme peut tout de même utiliser son paramètre `arg` (avec la valeur par défaut) pour faire appel à la méthode cible `self.action()`, et l'on obtient ainsi un véritable transfert d'argument vers cette méthode.

Nous ne détaillerons pas davantage ici la question des expressions lambda, car elle déborde du cadre que nous nous sommes fixés pour ce cours d'initiation. Si vous souhaitez en savoir plus, veuillez donc consulter un ouvrage de référence.

Passage d'une fonction (ou d'une méthode) comme argument

Vous avez déjà rencontré de nombreux widgets comportant une telle option `command`, à laquelle il faut à chaque fois associer le nom d'une fonction (ou d'une méthode). En termes plus généraux, cela signifie donc qu'une fonction avec paramètres peut recevoir la référence d'une autre fonction comme argument, et l'utilité de la chose apparaît clairement ici.

Nous avons d'ailleurs programmé nous-même une fonctionnalité de ce genre dans notre nouvelle classe `ToolBar()`. Vous pouvez constater que nous avons inclus le nom `command` dans la liste de paramètres de son constructeur, à la ligne 6. Ce paramètre attend la référence d'une fonction ou d'une méthode comme argument. La dite référence est alors mémorisée dans une variable d'instance (à la ligne 9), de manière à être accessible depuis les autres méthodes de la classe. Celles-ci peuvent dès lors invoquer la fonction ou la méthode (au besoin en lui transmettant des arguments si nécessaire, suivant la technique expliquée à la rubrique précédente). C'est ce que fait donc notre méthode `action()`, à la ligne 25. En l'occurrence, la méthode ainsi transmise est la méthode `transfert()` de la classe `Application` (cf. ligne 34).

Nous sommes parvenus ainsi à développer une classe d'objets `ToolBar` parfaitement réutilisables dans d'autres contextes. Comme le montre notre petite application, il suffit en effet d'instancier ces objets en indiquant la référence d'une fonction quelconque en argument de l'option `command`. Cette fonction sera alors automatiquement appelée elle-même avec le numéro d'ordre du bouton cliqué par l'utilisateur. Libre à vous d'imaginer à présent ce que la fonction devra effectuer !

Pour en terminer avec cet exemple, remarquons encore un petit détail : chacun de nos boutons apparaît entouré d'un sillon (option `relief =GROOVE`). Vous pouvez aisément obtenir d'autres aspects en choisissant judicieusement les options `relief` et `bd` (bordure) dans l'instruction d'instanciation de ces boutons. En particulier, vous pouvez choisir `relief =FLAT` et `bd =0` pour obtenir des petits boutons « plats », sans aucun relief.

3.5 Fenêtres avec menus

Pour terminer notre petite visite guidée des widgets `tkinter`, nous allons décrire à présent la construction d'une fenêtre d'application dotée de différents types de menus « déroulants », chacun de ces menus pouvant être « détaché » de l'application principale pour devenir lui-même une petite fenêtre indépendante, comme dans l'illustration ci-contre.

Cet exercice un peu plus long nous servira également de révision, et nous le réaliserons par étapes, en appliquant une stratégie de programmation que l'on appelle **développement incrémental**.

Cette méthode consiste à commencer l'écriture d'un programme par une ébauche, qui ne comporte que quelques lignes seulement mais qui est déjà fonctionnelle. On teste alors cette ébauche soigneusement afin d'en éliminer les *bugs* éventuels. Lorsque l'ébauche fonctionne correctement, on y ajoute une fonctionnalité supplémentaire. On teste ce complément jusqu'à ce qu'il donne entière satisfaction, puis on en ajoute un autre, et ainsi de suite...

Cela ne signifie pas que vous pouvez commencer directement à programmer sans avoir au préalable effectué une analyse sérieuse du projet, dont au moins les grandes lignes devront être convenablement décrites dans un **cahier des charges** clairement rédigé.

Il reste également impératif de commenter convenablement le code produit, au fur et à mesure de son élaboration. S'efforcer de rédiger de bons commentaires est en effet nécessaire, non seulement pour que votre code soit facile à lire (et donc à maintenir plus tard, par d'autres ou par vous-même), mais aussi pour que vous soyez forcés d'exprimer ce que vous souhaitez vraiment que la machine fasse.

Cahier des charges de l'exercice

Notre application comportera simplement une barre de menus et un canevas. Les différentes rubriques et options des menus ne serviront qu'à faire apparaître des fragments de texte dans le canevas ou à modifier des détails de décoration, mais ce seront avant tout des exemples variés, destinés à donner un aperçu des nombreuses possibilités offertes par ce type de widget, accessoire indispensable de toute application moderne d'une certaine importance.

Nous souhaitons également que le code produit dans cet exercice soit bien structuré. Pour ce faire, nous ferons usage de deux classes : une classe pour l'application principale, et une autre pour la barre de menus. Nous voulons procéder ainsi afin de bien mettre en évidence la construction d'une application type incorporant plusieurs classes d'objets interactifs.

Première ébauche du programme

Lorsque l'on construit l'ébauche d'un programme, il faut tâcher d'y faire apparaître le plus tôt possible la structure d'ensemble, avec les relations entre les principaux blocs qui constitueront l'application définitive. C'est ce que nous nous sommes efforcés de faire dans l'exemple ci-dessous :

```
from tkinter import *

class MenuBar(Frame):
    """Barre de menus déroulants"""
    def __init__(self, boss=None):
        Frame.__init__(self, borderwidth=2)

        ##### Menu <Fichier> #####
        fileMenu = Menubutton(self, text='Fichier')
         fileMenu.pack(side=LEFT)
         # Partie "déroulante" :
         me1 = Menu(fileMenu)
         me1.add_command(label='Effacer', underline=0,
                         command=boss.effacer)
         me1.add_command(label='Terminer', underline=0,
                         command=boss.quit)
         # Intégration du menu :
         fileMenu.configure(menu=me1)

class Application(Frame):
    """Application principale"""
    def __init__(self, boss=None):
        Frame.__init__(self)
        self.master.title('Fenêtre avec menus')
        mBar = MenuBar(self)
        mBar.pack()
        self.can = Canvas(self, bg='light grey', height=190,
                          width=250, borderwidth=2)
        self.can.pack()
        self.pack()

    def effacer(self):
        self.can.delete(ALL)

if __name__ == '__main__':
    app = Application()
    app.mainloop()
```

Veuillez donc encoder ces lignes et en tester l'exécution. Vous devriez obtenir une fenêtre avec un canevas gris clair surmonté d'une barre de menus. À ce stade, la barre de menus ne comporte encore que la seule rubrique « Fichier ».

Cliquez sur la rubrique « Fichier » pour faire apparaître le menu correspondant : l'option « Effacer » n'est pas encore fonctionnelle (elle servira plus loin à effacer le contenu du canevas), mais l'option « Terminer » devrait déjà vous permettre de fermer proprement l'application.

Comme tous les menus gérés par `tkinter`, le menu que vous avez créé peut être converti en menu « flottant » : il suffit de cliquer sur la ligne pointillée apparaissant en-tête de menu. Vous obtenez ainsi une petite fenêtre satellite, que vous pouvez alors positionner où bon vous semble sur le bureau.

La structure de ce petit programme devrait vous apparaître familière : afin que les classes définies dans ce script puissent éventuellement être (ré)utilisées dans d'autres projets par importation, comme nous l'avons déjà expliqué précédemment, le corps principal du programme (lignes 35 à 37) comporte l'instruction désormais classique :

```
if __name__ == '__main__':
```

Les deux instructions qui suivent consistent seulement à instancier un objet `app` et à faire fonctionner sa méthode `mainloop()`. Comme vous le savez, nous aurions pu également condenser ces deux instructions en une seule.

L'essentiel du programme se trouve cependant dans les définitions de classes qui précèdent.

La classe `MenuBar()` contient la description de la barre de menus. Dans l'état présent du script, elle se résume à une ébauche de constructeur.

- Ligne 5 : Le paramètre `boss` réceptionne la référence de la fenêtre maîtresse du widget au moment de son instanciation. Cette référence va nous permettre d'invoquer les méthodes associées à cette fenêtre maîtresse, aux lignes 14 et 16.
- Ligne 6 : Activation obligatoire du constructeur de la classe parente.
- Ligne 9 : Instanciation d'un widget de la classe `Menubutton()`, défini comme un « esclave » de `self` (c'est-à-dire l'objet composite « barre de menus » dont nous sommes occupés à définir la classe). Comme l'indique son nom, ce type de widget se comporte un peu comme un bouton : une action se produit lorsque l'on clique dessus.
- Ligne 12 : Afin que cette action consiste en l'apparition véritable d'un menu, il reste encore à définir celui-ci : ce sera encore un nouveau widget, de la classe `Menu()` cette fois, défini lui-même comme un « esclave » du widget `Menubutton` instancié à la ligne 9.
- Lignes 13 à 16 : On peut appliquer aux widgets de la classe `Menu()` un certain nombre de méthodes spécifiques, chacune d'elles acceptant de nombreuses options. Nous utilisons ici la méthode `add_command()` pour installer dans le menu les deux items « Effacer » et « Terminer ». Nous y intégrons tout de suite l'option `underline`, qui sert à définir un raccourci clavier : cette option indique en effet lequel des caractères de l'item doit apparaître souligné à l'écran. L'utilisateur sait alors qu'il lui suffit de frapper ce caractère au clavier pour que l'action correspondant à cet item soit activée (comme s'il avait cliqué dessus à l'aide de la souris).

  L'action à déclencher lorsque l'utilisateur sélectionne l'item est désignée par l'option `command`. Dans notre script, les commandes invoquées sont toutes les deux des méthodes de la fenêtre maîtresse, dont la référence aura été transmise au présent widget au moment de son instanciation par l'intermédiaire du paramètre `boss`. La méthode `effacer()`, que nous définissons nous-même plus loin, servira à vider le canevas. La méthode prédéfinie `quit()` provoque la sortie de la boucle `mainloop()` et donc l'arrêt du réceptionnaire d'événements associé à la fenêtre d'application.

- Ligne 18 : Lorsque les items du menu ont été définis, il reste encore à reconfigurer le widget maître `Menubutton` de manière à ce que son option « menu » désigne effectivement le `Menu` que nous venons de construire. En effet, nous ne pouvions pas déjà préciser cette option lors de la définition initiale du widget `Menubutton`, puisqu'à ce stade le `Menu` n'existait pas encore. Nous ne pouvions pas non plus définir le widget `Menu` en premier lieu, puisque celui-ci doit être défini comme un « esclave » du widget `Menubutton`. Il faut donc bien procéder en trois étapes comme nous l'avons fait, en faisant appel à la méthode `configure()`. Cette méthode peut être appliquée à n'importe quel widget préexistant pour en modifier l'une ou l'autre option.

La classe `Application()` contient la description de la fenêtre principale du programme ainsi que les méthodes gestionnaires d'événements qui lui sont associées.

- Ligne 20 : Nous préférons faire dériver notre application de la classe `Frame()`, qui présente de nombreuses options, plutôt que de la classe primordiale `Tk()`. De cette manière, l'application toute entière est encapsulée dans un widget, lequel pourra éventuellement être intégré par la suite dans une application plus importante. Rappelons que, de toute manière, tkinter instanciera automatiquement une fenêtre maîtresse de type `Tk()` pour contenir cette `Frame`.
- Lignes 23-24 : Après l'indispensable activation du constructeur de la classe parente, nous utilisons l'attribut `master` que tkinter associe automatiquement à chaque widget, pour référencer la classe parente (dans le cas présent, l'objet correspondant est la fenêtre principale de l'application) et en redéfinir le bandeau-titre.
- Lignes 25 à 29 : Instanciation de deux widgets esclaves pour notre cadre (`Frame`) principal. La « barre de menus » est évidemment le widget défini dans l'autre classe.
- Ligne 30 : Comme n'importe quel autre widget, notre cadre principal doit être confié à une méthode de mis en place afin d'apparaître véritablement.
- Lignes 32-33 : La méthode servant à effacer le canevas est définie dans la classe présente (puisque l'objet canevas en fait partie), mais elle est invoquée par l'option `command` d'un widget esclave défini dans une autre classe. Comme nous l'avons expliqué plus haut, ce widget esclave reçoit la référence de son widget maître par l'intermédiaire du paramètre `boss`. Toutes ces références sont hiérarchisées à l'aide de la qualification des noms par points.

Ajout de la rubrique Musiciens

Continuez le développement de ce petit programme, en ajoutant les lignes suivantes dans le constructeur de la classe `MenuBar()` (après la ligne 18) :

```
        ##### Menu <Musiciens> #####
        self.musi = Menubutton(self, text ='Musiciens')
        self.musi.pack(side =LEFT, padx ='3')
        # Partie "déroulante" du menu <Musiciens> :
        me1 = Menu(self.musi)
        me1.add_command(label ='17e siècle', underline =1,
                        foreground ='red', background ='yellow',
                        font =('Comic Sans MS', 11),
                        command = boss.showMusi17)
        me1.add_command(label ='18e siècle', underline =1,
                        foreground='royal blue', background ='white',
                        font =('Comic Sans MS', 11, 'bold'),
                        command = boss.showMusi18)
        # Intégration du menu :
        self.musi.configure(menu = me1)
```

... ainsi que les définitions de méthodes suivantes à la classe `Application()` (après la ligne 33) :

```
    def showMusi17(self):
        self.can.create_text(10, 10, anchor =NW, text ='H. Purcell',
                             font=('Times', 20, 'bold'), fill ='yellow')

    def showMusi18(self):
        self.can.create_text(245, 40, anchor =NE, text ="W. A. Mozart",
                             font =('Times', 20, 'italic'), fill ='dark green')
```

Sauvegardez le script et exécutez-le.

Votre barre de menus comporte à présent une rubrique supplémentaire : la rubrique « Musiciens ».

Le menu correspondant propose deux items qui sont affichés avec des couleurs et des polices personnalisées. Vous pourrez vous inspirer de ces techniques décoratives pour vos projets personnels. À utiliser avec modération !

Les commandes que nous avons associées à ces items sont évidemment simplifiées afin de ne pas alourdir l'exercice : elles provoquent l'affichage de petits textes sur le canevas.

Les seules nouveautés introduites dans ces lignes concernent l'utilisation de polices de caractères bien déterminées (option `font`), ainsi que de couleurs pour l'avant-plan (option `foreground`) et le fond (option `background`) des textes affichés.

Veuillez noter encore une fois l'utilisation de l'option `underline` pour désigner les caractères correspondant à des raccourcis claviers (en n'oubliant pas que la numérotation des caractères d'une chaîne commence à partir de zéro), et surtout que l'option `command` de ces widgets accède aux méthodes de l'autre classe, par l'intermédiaire de la référence mémorisée dans l'attribut `boss`.

La méthode `create_text()` du canevas doit être utilisée avec deux arguments numériques, qui sont les coordonnées x et y d'un point dans le canevas. Le texte transmis sera positionné par rapport à ce point, en fonction de la valeur choisie pour l'option `anchor` : celle-ci détermine comment le fragment de texte doit être « ancré » au point choisi dans le canevas, par son centre, par son coin supérieur gauche, etc., en fonction d'une syntaxe qui utilise l'analogie des points cardinaux géographiques (`NW` = angle supérieur gauche, `SE` = angle inférieur droit, `CENTER` = centre, etc.).

Ajout de la rubrique Peintres

Cette nouvelle rubrique est construite d'une manière assez semblable à la précédente, mais nous lui avons ajouté une fonctionnalité supplémentaire : des menus « en cascade ». Veuillez donc ajouter les lignes suivantes dans le constructeur de la classe `MenuBar()` :

```
        ##### Menu <Peintres> #####
        self.pein = Menubutton(self, text ='Peintres')
        self.pein.pack(side =LEFT, padx='3')
        # Partie "déroulante" :
        me1 = Menu(self.pein)
        me1.add_command(label ='classiques', state=DISABLED)
        me1.add_command(label ='romantiques', underline =0,
                        command = boss.showRomanti)
        # Sous-menu pour les peintres impressionistes :
        me2 = Menu(me1)
        me2.add_command(label ='Claude Monet', underline =7,
                        command = boss.tabMonet)
        me2.add_command(label ='Auguste Renoir', underline =8,
                        command = boss.tabRenoir)
        me2.add_command(label ='Edgar Degas', underline =6,
                        command = boss.tabDegas)
        # Intégration du sous-menu :
        me1.add_cascade(label ='impressionistes', underline=0, menu =me2)
        # Intégration du menu :
        self.pein.configure(menu =me1)
```

... et les définitions suivantes dans la classe `Application()` :

```
    def showRomanti(self):
        self.can.create_text(245, 70, anchor =NE, text = "E. Delacroix",
```

```
                font =('Times', 20, 'bold italic'), fill ='blue')

    def tabMonet(self):
        self.can.create_text(10, 100, anchor =NW, text = 'Nymphéas à Giverny',
                font =('Technical', 20), fill ='red')

    def tabRenoir(self):
        self.can.create_text(10, 130, anchor =NW,
                text = 'Le moulin de la galette',
                font =('Dom Casual BT', 20), fill ='maroon')

    def tabDegas(self):
        self.can.create_text(10, 160, anchor =NW, text = 'Danseuses au repos',
                font =('President', 20), fill ='purple')
```

Vous pouvez réaliser aisément des menus en cascade, en enchaînant des sous-menus les uns aux autres jusqu'à un niveau quelconque (il vous est cependant déconseillé d'aller au-delà de 5 niveaux successifs : vos utilisateurs s'y perdraient).

Un sous-menu est défini comme un menu « esclave » du menu de niveau précédent (dans notre exemple, `me2` est défini comme un menu « esclave » de `me1`). L'intégration est assurée ensuite à l'aide de la méthode `add_cascade()`.

L'un des items est désactivé (option `state = DISABLED`). L'exemple suivant vous montrera comment vous pouvez activer ou désactiver à volonté des items, par programme.

Ajout de la rubrique Options

La définition de cette rubrique est un peu plus compliquée, parce que nous allons y intégrer l'utilisation de variables internes à tkinter.

Les fonctionnalités de ce menu sont cependant beaucoup plus élaborées : les options ajoutées permettent en effet d'activer ou de désactiver à volonté les rubriques « Musiciens » et « Peintres », et vous pouvez également modifier à volonté l'aspect de la barre de menus elle-même.

Veuillez donc ajouter les lignes suivantes dans le constructeur de la classe `MenuBar()` :

```
        ##### Menu <Options> #####
        optMenu = Menubutton(self, text ='Options')
        optMenu.pack(side =LEFT, padx ='3')
        # Variables tkinter :
        self.relief = IntVar()
        self.actPein = IntVar()
        self.actMusi = IntVar()
        # Partie "déroulante" du menu :
        self.mo = Menu(optMenu)
        self.mo.add_command(label = 'Activer :', foreground ='blue')
        self.mo.add_checkbutton(label ='musiciens',
                    command = self.choixActifs, variable =self.actMusi)
        self.mo.add_checkbutton(label ='peintres',
                    command = self.choixActifs, variable =self.actPein)
```

```
        self.mo.add_separator()
        self.mo.add_command(label = 'Relief :', foreground ='blue')
        for (v, lab) in [(0,'aucun'), (1,'sorti'), (2,'rentré'),
                         (3,'sillon'), (4,'crête'), (5,'bordure')]:
            self.mo.add_radiobutton(label =lab, variable =self.relief,
                                    value =v, command =self.reliefBarre)
        # Intégration du menu :
        optMenu.configure(menu = self.mo)
```

... ainsi que les définitions de méthodes suivantes (toujours dans la classe **MenuBar()**) :

```
    def reliefBarre(self):
        choix = self.relief.get()
        self.configure(relief =[FLAT,RAISED,SUNKEN,GROOVE,RIDGE,SOLID][choix])

    def choixActifs(self):
        p = self.actPein.get()
        m = self.actMusi.get()
        self.pein.configure(state =[DISABLED, NORMAL][p])
        self.musi.configure(state =[DISABLED, NORMAL][m])
```

Menu avec cases à cocher

Notre nouveau menu déroulant comporte deux parties. Afin de les mettre en évidence, nous avons inséré une ligne de séparation et deux « faux items » (« Activer : » et « Relief : ») qui servent simplement de titres. Nous faisons apparaître ceux-ci en couleur pour que l'utilisateur ne les confonde pas avec de véritables commandes.

Les items de la première partie sont dotées de « cases à cocher ». Lorsque l'utilisateur effectue un clic de souris sur l'un ou l'autre de ces items, les options correspondantes sont activées ou désactivées, et ces états « actif / inactif » sont affichés sous la forme d'une encoche. Les instructions qui servent à mettre en place ce type de rubrique sont assez explicites. Elles présentent en effet ces items comme des widgets de type `chekbutton` :

```
    self.mo.add_checkbutton(label = 'musiciens', command = choixActifs,
                            variable = mbu.me1.music)
```

Il est important de comprendre ici que ce type de widget comporte nécessairement une variable interne, destinée à mémoriser l'état « actif / inactif » du widget. Comme nous l'avons déjà expliqué plus haut, cette variable ne peut pas être une variable Python ordinaire, parce que les classes de la bibliothèque tkinter sont écrites dans un autre langage. Et par conséquent, on ne pourra accéder à une telle variable interne qu'à travers un objet-interface, que nous appellerons variable tkinter pour simplifier.

C'est ainsi que dans notre exemple, nous utilisons la classe tkinter `IntVar()` pour créer des objets équivalents à des variables de type entier.

- Nous instancions donc un de ces objets-variables, que nous mémorisons comme attribut d'instance :

  ```
  self.actMusi =IntVar()
  ```

  Après cette affectation, l'objet référencé dans `self.actMusi` contient désormais l'équivalent d'une variable de type entier, dans un format spécifique à tkinter.
- Il faut ensuite associer l'option `variable` de l'objet `checkbutton` à la variable tkinter ainsi définie :

  ```
  self.mo.add_checkbutton(label = 'musiciens', variable =self.actMusi)
  ```

- Il est nécessaire de procéder ainsi en deux étapes, parce que tkinter ne peut pas directement assigner des valeurs aux variables Python. Pour une raison similaire, il n'est pas possible à Python de lire directement le contenu d'une variable tkinter. Il faut utiliser pour cela les méthodes spécifiques de cette classe d'objets : la méthode `get()` pour lire, et la méthode `set()` pour écrire :

  ```
  m = self.actMusi.get()
  ```

  Dans cette instruction, nous affectons à `m` (variable ordinaire de Python) le contenu de la variable tkinter `self.actMusi` (laquelle est elle-même associée à un widget bien déterminé).

Tout ce qui précède peut vous paraître un peu compliqué. Considérez simplement qu'il s'agit de votre première rencontre avec les problèmes d'interfaçage entre deux langages de programmation différents, utilisés ensemble dans un projet composite.

Menu avec choix exclusifs

La deuxième partie du menu « Options » permet à l'utilisateur de choisir l'aspect que prendra la barre de menus, parmi six possibilités. Il va de soi que l'on ne peut activer qu'une seule de ces possibilités à la fois. Pour mettre en place ce genre de fonctionnalité, on fait classiquement appel à des widgets de type « boutons radio ». La caractéristique essentielle de ces widgets est que plusieurs d'entre eux doivent être associés à une seule et même variable tkinter. À chaque bouton radio correspond alors une valeur particulière, et c'est cette valeur qui est affectée à la variable lorsque l'utilisateur sélectionne le bouton.

Ainsi, l'instruction :

```
    self.mo.add_radiobutton(label ='sillon', variable =self.relief,
                            value =3, command =self.reliefBarre)
```

configure un item du menu « Options » de telle manière qu'il se comporte comme un bouton radio.

Lorsque l'utilisateur sélectionne cet item, la valeur 3 est affectée à la variable tkinter `self.relief` (celle-ci étant désignée à l'aide de l'option `variable` du widget), et un appel est lancé en direction de la méthode `reliefBarre()`. Celle-ci récupère alors la valeur mémorisée dans la variable tkinter pour effectuer son travail.

Dans le contexte particulier de ce menu, nous souhaitons proposer 6 possibilités différentes à l'utilisateur. Il nous faut donc six « boutons radio », pour lesquels nous pourrions encoder six instructions similaires à celle que nous avons reproduite ci-dessus, chacune d'elles ne différant des cinq autres que par ses options `value` et `label`. Dans une situation de ce genre, la bonne pratique de programmation consiste à placer les valeurs de ces options dans une liste, et à parcourir ensuite cette liste à l'aide d'une boucle `for`, afin d'instancier les widgets avec une instruction commune :

```
    for (v, lab) in [(0,'aucun'), (1,'sorti'), (2,'rentré'),
                     (3,'sillon'), (4,'crête'), (5,'bordure')]:
        self.mo.add_radiobutton(label =lab, variable =self.relief,
                                value =v, command =self.reliefBarre)
```

La liste utilisée est une liste de 6 tuples (valeur, libellé). À chacune des 6 itérations de la boucle, un nouvel item `radiobutton` est instancié, dont les options `label` et `value` sont extraites de la liste par l'intermédiaire des variables `lab` et `v`.

Dans vos projets personnels, il vous arrivera fréquemment de constater que vous pouvez ainsi remplacer des suites d'instructions similaires par une structure de programmation plus compacte (en général, la combinaison d'une liste et d'une boucle, comme dans l'exemple ci-dessus).

Vous découvrirez petit à petit encore d'autres techniques pour alléger votre code : nous en fournissons un exemple dans le paragraphe suivant. Tâchez cependant de garder à l'esprit cette règle essentielle : un bon programme doit avant tout rester **très lisible** et **bien commenté**.

Contrôle du flux d'exécution à l'aide d'une liste

Veuillez à présent considérer la définition de la méthode `reliefBarre()`.

À la première ligne, la méthode `get()` nous permet de récupérer l'état d'une variable tkinter qui contient le numéro du choix opéré par l'utilisateur dans le sous-menu « Relief : ».

À la seconde ligne, nous utilisons le contenu de la variable `choix` pour extraire d'une liste de six éléments celui qui nous intéresse. Par exemple, si `choix` contient la valeur 2, c'est l'option `SUNKEN` qui sera utilisée pour reconfigurer le widget.

La variable `choix` est donc utilisée ici comme un `index`, servant à désigner un élément de la liste. En lieu et place de cette construction compacte, nous aurions pu programmer une série de tests conditionnels, comme :

```
if choix ==0:
    self.configure(relief =FLAT)
elif choix ==1:
    self.configure(relief =RAISED)
elif choix ==2:
    self.configure(relief =SUNKEN)
...
etc.
```

D'un point de vue strictement fonctionnel, le résultat serait exactement le même. Vous admettrez cependant que la construction que nous avons choisie est d'autant plus efficace que le nombre de possibilités de choix est élevé. Imaginez par exemple que l'un de vos programmes personnels doive effectuer une sélection dans un très grand nombre d'éléments : avec une construction du type ci-dessus, vous seriez peut-être amené à encoder plusieurs pages de `elif` !

Nous utilisons encore la même technique dans la méthode `choixActifs()`. Ainsi l'instruction :

```
self.pein.configure(state =[DISABLED, NORMAL][p])
```

utilise le contenu de la variable `p` comme index pour désigner lequel des deux états `DISABLED`, `NORMAL` doit être sélectionné pour reconfigurer le menu « Peintres ».

Lorsqu'elle est appelée, la méthode `choixActifs()` reconfigure donc les deux rubriques « Peintres » et « Musiciens » de la barre de menus, pour les faire apparaître « normales » ou « désactivées » en fonction de l'état des variables `m` et `p`, lesquelles sont elles-mêmes le reflet de variables tkinter.

Ces variables intermédiaires `m` et `p` ne servent en fait qu'à clarifier le script. Il serait en effet parfaitement possible de les éliminer, et de rendre le script encore plus compact, en utilisant la composition d'instructions. On pourrait par exemple remplacer les deux instructions :

```
m = self.actMusi.get()
self.musi.configure(state =[DISABLED, NORMAL][m])
```

par une seule, telle que :

```
self.musi.configure(state =[DISABLED, NORMAL][self.actMusi.get()])
```

Notez cependant que ce que l'on gagne en compacité peut se payer d'une certaine perte de lisibilité.

Pré-sélection d'une rubrique

Pour terminer cet exercice, voyons encore comment vous pouvez déterminer à l'avance certaines sélections, ou bien les modifier par programme.

Veuillez donc ajouter l'instruction suivante dans le constructeur de la classe `Application()` (juste avant l'instruction `self.pack()`, par exemple) :

```
mBar.mo.invoke(2)
```

Lorsque vous exécutez le script ainsi modifié, vous constatez qu'au départ la rubrique « Musiciens » de la barre de menus est active, alors que la rubrique « Peintres » ne l'est pas. Programmées comme elles le sont, ces deux rubriques devraient être actives toutes deux par défaut. Et c'est effectivement ce qui se passe si nous supprimons l'instruction `mBar.mo.invoke(2)`.

Nous vous avons suggéré d'ajouter cette instruction au script pour vous montrer comment vous pouvez effectuer par programme la même opération que celle que l'on obtient normalement avec un clic de souris.

L'instruction ci-dessus invoque le widget `mBar.mo` en actionnant la commande associée au deuxième item de ce widget. En consultant le listing, vous pouvez vérifier que ce deuxième item est bien l'objet de type `checkbutton` qui active/désactive le menu « Peintres » (rappelons encore une fois que l'on numérote toujours à partir de zéro).

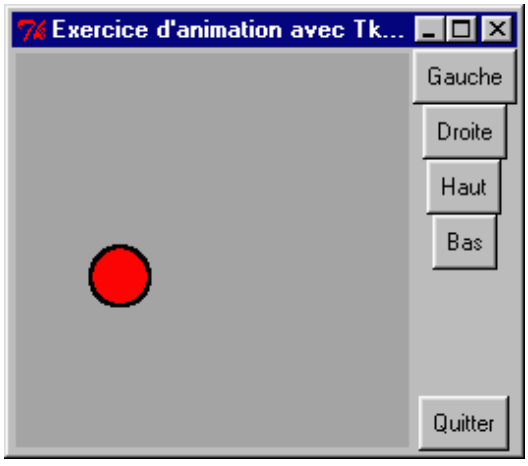
Au démarrage du programme, tout se passe donc comme si l'utilisateur effectuait tout de suite un premier clic sur la rubrique « Peintres » du menu « Options », ce qui a pour effet de désactiver le menu correspondant.

4 Animation d'un objet

À ce stade de votre apprentissage, vous souhaitez probablement pouvoir faire apparaître un petit dessin quelconque dans un canevas, et puis le déplacer à volonté, par exemple à l'aide de boutons.

```
from tkinter import *

class Application(Tk):

    def __init__(self):
        # Création du widget principal
        Tk.__init__(self)
        self.title("Exercice d'animation avec tkinter")

        # Coordonnées initiales de position de la balle:
        self.x, self.y = 10, 10

        # Création des widgets "esclaves":
        self.can = Canvas(self, bg = "dark grey", height=300, width=300)
        self.balle = self.can.create_oval(self.x, self.y, self.x+30, self.y+30,
                                          width=2, fill='red')
        self.can.pack(side=LEFT)
        Button(self,text='Quitter',command=self.destroy).pack(side=BOTTOM)
        Button(self,text='Gauche',
               command=lambda gd=-10, hb=0 : self.avance(gd, hb)).pack()
        Button(self,text='Droite',
               command=lambda gd=10, hb=0 : self.avance(gd, hb)).pack()
        Button(self,text='Haut',
               command=lambda gd=0, hb=-10 : self.avance(gd, hb)).pack()
        Button(self,text='Bas',
               command=lambda gd=0, hb=10 : self.avance(gd, hb)).pack()


    def avance(self, gd, hb):
        "Déplacement de la balle"
        self.x, self.y = self.x+gd, self.y+hb
        self.can.coords(self.balle, self.x, self.y, self.x+30, self.y+30)


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

Application().mainloop()
```

Ce programme reprend de nombreux éléments connus : nous y créons une classe de fenêtres `Application`, dans laquelle nous installons un canevas contenant lui-même un cercle coloré, plus cinq boutons de contrôle.



Vous remarquerez que nous n'instancions pas les widgets boutons dans des variables (c'est inutile, puisque nous n'y faisons plus référence par la suite) : nous devons donc appliquer la méthode `pack()` directement au moment de la création de ces objets.

De plus, le gestionnaire d'événements de chaque bouton de déplacement est une fonction-lambda permettant d'appeler la méthode `avance` avec deux arguments `gd` et `hb` indiquant le sens et la longueur des déplacements horizontaux et verticaux.

La vraie nouveauté de ce programme réside d'ailleurs dans cette méthode `avance()`. Chaque fois qu'elle sera appelée, cette méthode redéfinira les coordonnées de l'objet « cercle coloré » que nous avons installé dans le canevas, ce qui provoquera l'animation de cet objet.

Cette manière de procéder est tout à fait caractéristique de la programmation « orientée objet » : on commence par créer des objets, puis on agit sur ces objets **en modifiant leurs propriétés, par l'intermédiaire de méthodes**.

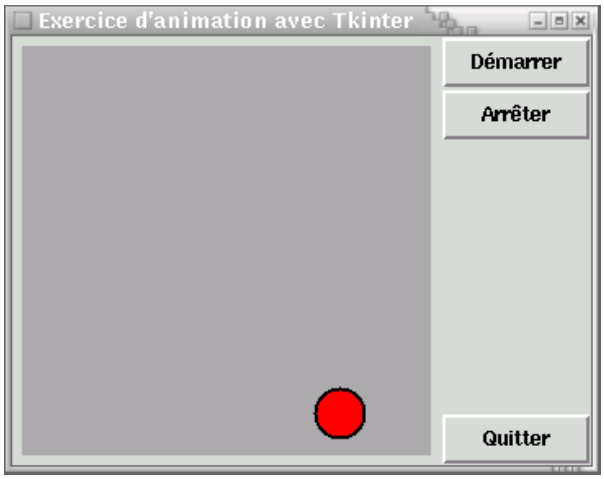
En programmation impérative « à l'ancienne » (c'est-à-dire sans utilisation d'objets), on anime des figures en les effaçant à un endroit pour les redessiner ensuite un petit peu plus loin. En programmation « orientée objet », par contre, ces tâches sont prises en charge automatiquement par les classes dont les objets dérivent, et il ne faut donc pas perdre son temps à les reprogrammer.

Animation automatique d'un objet

Pour conclure cette prise de contact avec l'interface graphique tkinter, voici un dernier exemple d'animation, qui fonctionne cette fois de manière autonome dès qu'on l'a mise en marche.

```
from tkinter import *

class Application(Tk):

    def __init__(self):
        # Création du widget principal
        Tk.__init__(self)
        self.title("Exercice d'animation avec tkinter")

        # Coordonnées initiales de position de la balle:
        self.x, self.y = 10, 10

        # 'Pas' du déplacement de la balle:
        self.dx, self.dy = 15, 0

        # Commutateur:
        self.flag=0

        # Création des widgets "esclaves":
        self.can = Canvas(self, bg = "dark grey", height=250, width=250)
        self.balle = self.can.create_oval(self.x, self.y, self.x+30, self.y+30,
                                          width=2, fill='red')
        self.can.pack(side=LEFT)
        Button(self,text='Quitter',command=self.destroy).pack(side=BOTTOM)
        Button(self,text='Démarrer',
               command=self.start_it).pack()
        Button(self,text='Arrêter',
               command=self.stop_it).pack()

    def move(self):
        "déplacement de la balle"

        self.x, self.y = self.x+self.dx, self.y+self.dy

        if self.x >210:
            self.x, self.dx, self.dy = 210, 0, 15
        if self.y >210:
            self.y, self.dx, self.dy = 210, -15, 0
        if self.x <10:
            self.x, self.dx, self.dy = 10, 0, -15
        if self.y <10:
            self.y, self.dx, self.dy = 10, 15, 0

        self.can.coords(self.balle,self.x,self.y,self.x+30,self.y+30)

        if self.flag >0:
            self.after(50,self.move)        # boucler après 50 millisecondes
```

```
    def stop_it(self):
        "arret de l'animation"

        self.flag =0

    def start_it(self):
        "démarrage de l'animation"

        if self.flag ==0:   # pour éviter que le bouton ne puisse lancer plusieurs
            boucles
            self.flag =1
            self.move()


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

Application().mainloop()
```



La seule nouveauté mise en œuvre dans ce script se trouve tout à la fin de la définition de la méthode `move()` : vous y noterez l'utilisation de la méthode `after()`. Cette méthode peut s'appliquer à un widget quelconque. Elle déclenche l'appel d'une fonction ou d'une méthode **après qu'un certain laps de temps se soit écoulé**. Ainsi par exemple, `window.after(200,qqc)` déclenche pour le widget `window` un appel de la fonction `qqc()` après une pause de 200 millisecondes.

Dans notre script, la fonction qui est appelée par `after()` est la méthode `move()` elle-même. Nous utilisons donc ici un type de **récursivité**. On obtient ainsi une boucle, qui peut se perpétuer indéfiniment si l'on ne prévoit aucun moyen pour l'interrompre.

Voyons comment cela fonctionne dans notre exemple.

La méthode `move()` est invoquée une première fois lorsque l'on clique sur le bouton « Démarrer ». Elle effectue son travail (c'est-à-dire positionner la balle). Ensuite, par l'intermédiaire de la méthode `after()`, elle s'invoque elle-même après une petite pause. Elle repart donc pour un second tour, puis s'invoque elle-même à nouveau, et ainsi de suite indéfiniment...

C'est du moins ce qui se passerait si nous n'avions pas pris la précaution de placer quelque part dans la boucle une instruction de sortie. En l'occurrence, il s'agit d'un simple test conditionnel : à chaque itération de la boucle, nous examinons le contenu de l'attribut `self.flag` à l'aide d'une instruction `if`. Si le contenu de la variable `self.flag` est zéro, alors le bouclage ne s'effectue plus et l'animation s'arrête.

Nous obtenons ainsi un mécanisme simple pour lancer ou arrêter notre animation : un premier clic sur le bouton « Démarrer » assigne une valeur non-nulle à la variable `self.flag`, puis provoque immédiatement un premier appel de la méthode `move()`. Celle-ci s'exécute, puis continue ensuite à s'appeler elle-même toutes les 50 millisecondes, tant que `self.flag` ne revient pas à zéro. Si l'on continue à cliquer sur le bouton « Démarrer », la méthode `move()` ne peut plus être appelée, parce que la valeur de `self.flag` vaut désormais 1. On évite ainsi le démarrage de plusieurs boucles concurrentes.

Le bouton « Arrêter » remet `self.flag` à zéro, et la boucle s'interrompt.

5 Glossaire du chapitre

La liste suivante répertorie dans l'ordre alphabétique les principaux termes introduits dans ce chapitre. La connaissance de leur définition est indispensable à la suite de votre apprentissage :

gestionnaire d'événements : fonction ou méthode appelée lorsqu'un certain type d'événements survient.

GUI : de l'anglais *Graphical User Interface*, un GUI est un ensemble de programme fournissant un cadre de travail simplifiant la manipulation des appareils informatiques à l'aide d'interfaces graphiques. Ces programmes permettent à un utilisateur d'interagir avec un programme à l'aide d'un environnement graphique plutôt qu'en ligne de commandes textuelles.

option : paramètre optionnel d'une fonction ou d'une méthode.

réceptionnaire d'événements : logiciel intégré au système d'exploitation de l'ordinateur scrutant sans cesse les périphériques et réagissant immédiatement lorsqu'un événement y est détecté (déplacement de la souris, appui sur une touche, top d'horloge,...) en envoyant divers messages décrivant l'événement aux programmes qui souhaitent en être avertis.

récursivité : une fonction qui s'appelle elle-même est dite récursive.

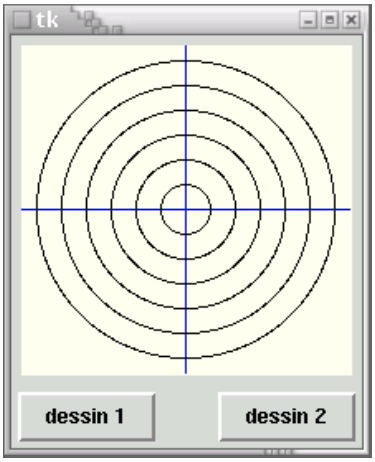
widget : résultat de la contraction de l'expression "*window gadget*", ce terme désigne toute entité susceptible d'être placée dans une fenêtre d'applications (bouton, case à cocher, image,...)

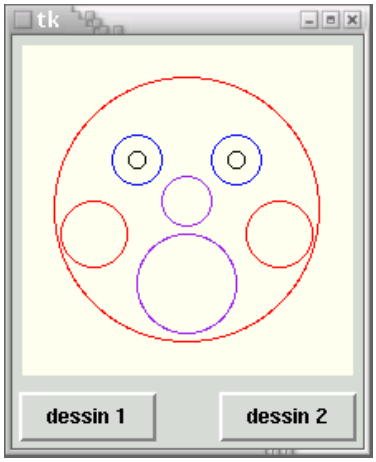
6 Exercices

Exercice 8.1
Modifiez l'exemple 1 (*Tracé de lignes dans un canevas*) de telle sorte que le nouveau programme réponde aux conditions suivantes :

- le canevas a une largeur de 500px sur une hauteur de 650px ;
- les lignes tracées sont uniquement de couleurs "*cyan*", "*maroon*" et "*green*" ;
- les lignes tracées sont horizontales, parallèles et séparées de 10px les unes des autres à partir du bas du canevas ;
- la longueur des lignes est telle que les extrémités de ces dernières se confondent avec les bords du canevas ;
- un quatrième bouton portant l'indication "*Tracer une croix*" permet d'appeler le gestionnaire d'événements "*drawcross*" qui trace deux lignes rouges de largeur 10px en croix au centre du canevas.

Exercice 8.2
Le script suivant permet d'afficher l'un ou l'autre des deux dessins reproduits ci-dessous, en fonction du bouton choisi :



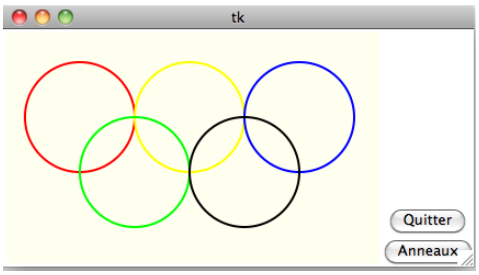



```
from tkinter import *

def cercle(x, y, r, coul ='black'):
    "Tracé d'un cercle de centre (x,y) et de rayon r"
    can.create_oval(x-r, y-r, x+r, y+r, outline=coul)

def figure_1():
    "Dessiner une cible"
    # Effacer d'abord tout dessin préexistant :
    can.delete(ALL)
    # tracer les deux lignes (vert. et horiz.) :
    can.create_line(100, 0, 100, 200, fill ='blue')
    can.create_line(0, 100, 200, 100, fill ='blue')
    # tracer plusieurs cercles concentriques :
    rayon = 15
    while rayon < 100:
        cercle(100, 100, rayon)
        rayon += 15

def figure_2():
    "Dessiner un visage simplifié"
    # Effacer d'abord tout dessin préexistant :
    can.delete(ALL)
    # Les caractéristiques de chaque cercle sont
    # placées dans une liste de listes :
    cc =[[100, 100, 80, 'red'],      # visage
         [70, 70, 15, 'blue'],       # yeux
```

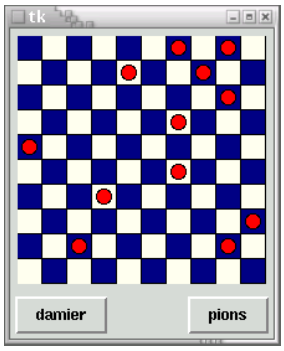
```
            [130, 70, 15, 'blue'],
            [70, 70, 5, 'black'],
            [130, 70, 5, 'black'],
            [44, 115, 20, 'red'],       # joues
            [156, 115, 20, 'red'],
            [100, 95, 15, 'purple'],    # nez
            [100, 145, 30, 'purple']]   # bouche
    # on trace tous les cercles à l'aide d'une boucle :
    i =0
    while i < len(cc):          # parcours de la liste
        el = cc[i]              # chaque élément est lui-même une liste
        cercle(el[0], el[1], el[2], el[3])
        i += 1

##### Programme principal : ############
fen = Tk()
can = Canvas(fen, width =200, height =200, bg ='ivory')
can.pack(side =TOP, padx =5, pady =5)
bou1 = Button(fen, text ='cible', command =figure_1)
bou1.pack(side =LEFT, padx =3, pady =3)
bou2 = Button(fen, text ='visage', command =figure_2)
bou2.pack(side =RIGHT, padx =3, pady =3)
fen.mainloop()
```

Transformez le code de ce script de telle sorte qu'il soit orienté objet. Votre programme comportera la définition de la fonction `cercle` ainsi que la définition d'une classe `Application()` dérivée de la classe `Tk`. Les instances de cette classe seront des fenêtres graphiques dont l'apparence et les fonctionnalités seront identiques à celle illustrée ci-dessus.

Exercice 8.3
Créer un programme permettant de dessiner les 5 anneaux olympiques dans un canevas de couleur "*ivory*", de hauteur 210px pour une largeur de 340px. Chaque anneau aura un rayon de 50px.



Exercice 8.4
Créer un programme faisant apparaître un damier (dessin de cases noires sur fond blanc) lorsque l'utilisateur clique sur le bouton "*damier*" et des pions rouges au hasard sur le damier lorsqu'il clique sur le bouton "*pions*". Chaque pression sur ce bouton fera apparaître un nouveau pion.



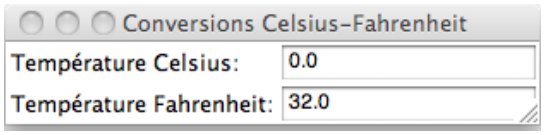
Exercice 8.5
Modifiez le script de l'exemple 2 (*Petit train*) de telle sorte que :

- le bouton "*Hello*" fasse apparaître uniquement 3 personnages dans le wagon 2 et 2 personnages dans le wagon 4 aux fenêtres 1 et 3 ;
- le constructeur de la classe `Wagon()` possède un paramètre `couleur` (avec comme valeur par défaut "*navy*") qui permette de déterminer la couleur de la cabine du wagon construit. Testez ce nouveau paramètre de telle sorte que le wagon 1 apparaisse dans la couleur 'navy', le wagon 2 en rouge, le wagon 3 en rose et finalement le wagon 4 en gris ;
- ajouter à la classe `Wagon()` une méthode `allumer()`, qui servira à changer la couleur des 3 fenêtres d'un wagon (initialement noires) en jaune, afin de simuler l'allumage d'un éclairage intérieur. Ajoutez un bouton à la fenêtre principale, qui permette d'enclencher l'allumage dans les wagon 2 et 4.

Exercice 8.6
En vous inspirant de l'exemple 3 (*Calculatrice minimaliste*), écrivez un programme faisant apparaître une fenêtre avec deux champs de texte, l'un indiquant une température en degrés Celsius et l'autre la même température exprimée en degrés Fahrenheit.

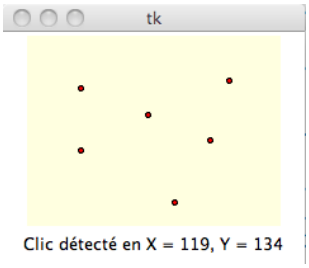
Chaque fois que l'on change une quelconque des deux températures, l'autre est corrigée en conséquence. Rappelons que pour convertir les degrés Fahrenheit en Celsius et vice-versa, on utilise la formule ci-dessous :

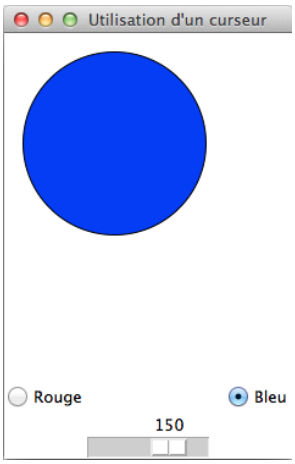
$$T_F = T_C \cdot 1.80 + 32$$



Pour insérer ou effacer une chaîne de caractères dans un objet de type `Entry`, référez-vous à la documentation en ligne de *tkinter* (par exemple, http://www.pythonware.com/library/tkinter/introduction/entry.htm)

Exercice 8.7
Modifiez l'exemple 4 (*Détection et positionnement d'un clic de souris*) de manière à faire apparaître un petit cercle rouge à l'endroit où l'utilisateur a effectué son clic. A ce dessein, remplacez d'abord le widget de type `Frame()` par un widget de type `Canvas()`





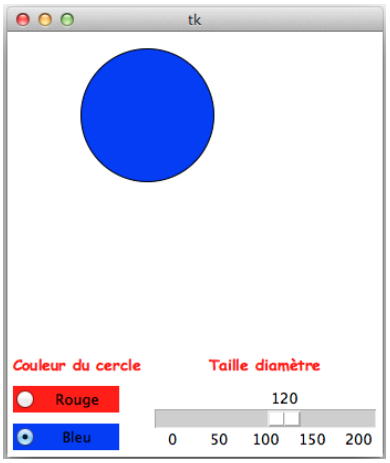
Exercice 8.8
Définissez une classe `DemoCurseur()` qui permet d'instancier des fenêtres contenant un canevas blanc de 220px de large sur 270px de haut dans lequel est situé un cercle initialisé de couleur rouge, de diamètre de 10px et inscrit dans un carré imaginaire dont le point supérieur gauche a les coordonnées (10, 10).

Chaque objet `DemoCurseur()` contiendra deux boutons radio permettant à l'utilisateur de changer la couleur du cercle (entre rouge et bleu) ainsi qu'un curseur pouvant se déplacer sur une plage de réglage allant de 0 à 200 et permettant d'augmenter et diminuer la taille du diamètre du cercle situé dans le canevas.

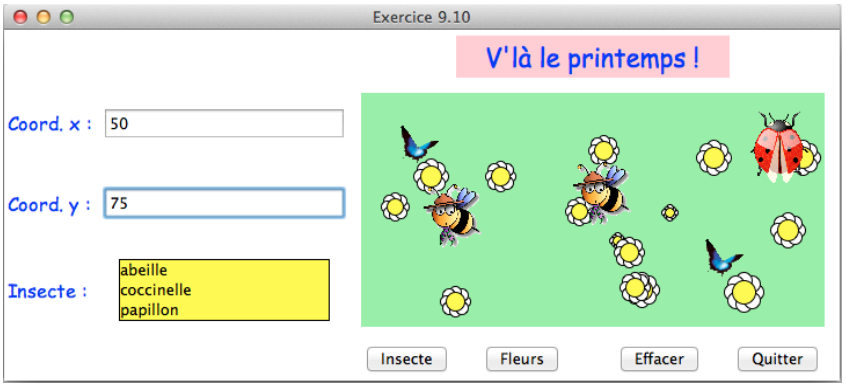
Exercice 8.9
Modifiez le code de l'exercice précédent de telle sorte que la fenêtre se présente comme le modèle ci-contre. En particulier, votre code devra satisfaire les conditions suivantes :

- Chaque widget est encadré d'une légère marge.
- Les objets de la classe `Label` apparaissent en gras avec une police *Comic sans MS* de taille 14 et de couleur rouge.
- Les objets de la classe `Radiobutton` apparaissent les uns sous les autres, sur un fond rouge ou bleu et sont alignés à gauche.
- L'objet `Scale` est centré horizontalement par rapport aux boutons radio et s'étend sur 200 pixels avec une graduation de 50 unités.

Epurez au maximum votre code de telle sorte qu'il ne comporte ni `self` ni affectation inutile.



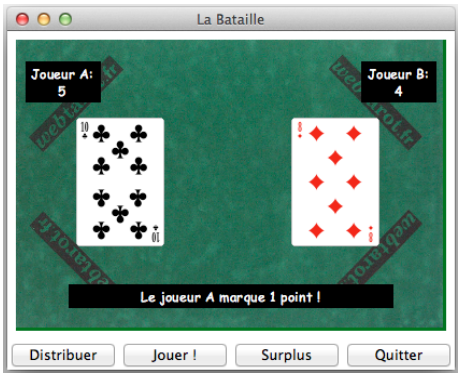
Exercice 8.10
Ecrivez un programme capable de peupler un pré de fleurs, d'abeilles, de coccinelles et de papillons. Votre programme aura une apparence similaire à celle de l'image ci-dessous :



Un canevas de couleur vert clair fera office de pré et trois images (situées sur le site du cours) joueront le rôle d'insectes. La fenêtre possédera 3 boutons principaux ayant les rôles suivants :

- Le bouton **Insecte** doit permettre d'afficher l'insecte sélectionné dans la liste à choix latérale au point de coordonnées précisées dans les zones de texte prévues à cet effet. Si la position de l'insecte n'est pas indiquée, le programme choisira aléatoirement un point dans le canevas pour y placer l'insecte sélectionné ;
- Le bouton **Fleurs** doit permettre d'afficher de manière aléatoire un tapis de 15 fleurs de différents diamètres ;
- Le bouton **Effacer** doit permettre d'effacer l'entier des éléments situés sur le canevas.

Exercice 8.11
Créez une interface graphique simulant le jeu *La Bataille* en important les classes définies dans l'exercice 7.9. Complétez à ce dessein le constructeur de la classe `Carte()` de telle sorte qu'il permette d'initialiser un attribut de type *PhotoImage* référençant le fichier image contenant la représentation de la carte. Les fichiers images nécessaires à ce jeu sont à disposition sur le site du cours. Votre programme aura la disposition ci-dessous :



Commencez par définir une classe `Application()`, dérivant de la classe `Tk()` du module tkinter. Lorsqu'on instanciera un nouvel objet de cette classe, son constructeur disposera différents éléments graphiques dans une fenêtre : un canevas de 400px sur 269px muni d'une image de tapis de jeu en arrière fond et disposé sur 4 lignes et 4 colonnes, une étiquette de texte affichant des messages à chaque opération du jeu et initialisée avec le texte *BIENVENUE A LA BATAILLE !* ainsi que 4 boutons. Ces derniers auront les rôles suivants (rôles gérés par autant de méthodes gestionnaires d'événements) :

- **Distribuer** permettra de lancer une nouvelle partie. A ce dessein, la méthode associée au bouton devra enlever du tapis de jeu les dernières cartes de la partie précédente, instancier un nouveau jeu de cartes, instancier une main à chacun des joueurs A et B, distribuer 18 cartes à chacun des 2 joueurs, initialiser le compteur de points de chacun des joueurs A et B (par exemple, dans un dictionnaire) et initialiser le compte à rebours des échanges entre les deux joueurs à 12. En outre, la méthode créera également deux étiquettes de texte permettant d'afficher en tout temps le résultat actualisé des 2 joueurs. Prenez garde à ne pas permettre de redistribuer les cartes en cours de partie !

- **Jouer !** permettra de lancer le prochain échange entre les joueurs. Ce bouton ne sera fonctionnel qu'après avoir distribué les cartes à chacun des 2 joueurs et uniquement si la partie n'est pas terminée. Si la partie n'est pas terminée, la méthode associée au bouton devra tirer une carte de chacune des mains des 2 joueurs, afficher l'image associée aux cartes tirées sur le tapis de jeu puis comparer les valeurs des cartes tirées. Un point sera ajouté au joueur possédant la carte dont la valeur est la plus élevée. Si les deux valeurs sont égales, aucun joueur ne marque de point. Les étiquettes de texte affichant le résultat de chacun des joueurs seront mises à jour à chaque appel de la méthode.
- **Surplus** permettra d'afficher, en fin de partie uniquement et dans deux fenêtres secondaires, les cartes restantes de chacun des deux joueurs après que ces dernières ont été triées. A ce dessein, la méthode associée au bouton instanciera deux objets de la classe `Secondaire()`.

  La classe `Secondaire()`, dérivant de la classe `Toplevel()` du module *tkinter*, permettra de créer de nouvelles fenêtres affichant le surplus de cartes de chacun des joueurs. Le constructeur de cette classe recevra donc en paramètres une référence à l'application principale ainsi que le nom du joueur dont il s'agit d'afficher le surplus de cartes. A l'aide de ces informations, le constructeur créera un canevas de 400px sur 269px muni d'un image de tapis de jeu en arrière fond et y disposera l'image des cartes restantes du joueur en question, triées, comme l'illustre le schéma ci-dessous :

- **Quitter** permettra de fermer simultanément la fenêtre principale du jeu ainsi que les éventuelles fenêtres satellites encore ouvertes.

Exercice 8.12
En vous inspirant de l'exemple 11 (*Animation manuelle*), écrivez un programme qui fait apparaître une fenêtre avec un canevas. Dans ce canevas, placez un cercle censé représenter une balle. Sous le canevas, placez un bouton. Chaque fois que l'on clique sur le bouton, la balle doit avancer d'une petite distance vers la droite, jusqu'à ce qu'elle atteigne l'extrémité du canevas. Si l'on continue à cliquer, la balle doit alors revenir en arrière jusqu'à l'autre extrémité, et ainsi de suite.

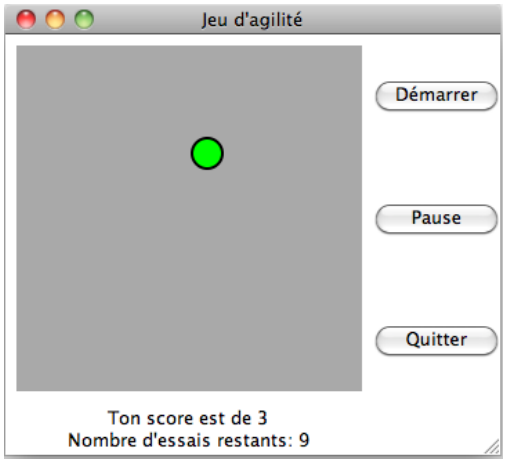
Exercice 8.13
Améliorez le programme précédent pour que la balle décrive cette fois une trajectoire circulaire dans le canevas lorsqu'on clique continuellement sur le bouton de déplacement. En outre, modifiez le programme de telle sorte qu'en se déplaçant, la balle laisse derrière elle une trace de la trajectoire décrite.
Pour obtenir le résultat escompté, vous devrez nécessairement définir un attribut qui représentera l'angle décrit, et utiliser les fonctions sinus et cosinus pour positionner la balle en fonction de cet angle.

Exercice 8.14
Améliorez le programme précédent de telle sorte que la balle se déplace le long d'un cercle en changeant de couleur à chaque tour effectué. L'animation doit être lancée lors d'un clic sur le bouton "Démarrer" et stoppée lors d'un clic sur le bouton "Arrêter". Inspirez-vous du code de l'exemple 12 (*Animation automatique*).

Exercice 8.15
Modifiez le programme précédent de telle sorte que la balle parte dans une direction fixée initialement puis effectue des rebonds contre les « parois » du canevas comme une bille de billard.

Exercice 8.16
Écrivez un programme simulant le mouvement d'une balle qui tombe sous l'effet de la pesanteur et qui rebondit sur le sol. Il s'agit dans cet exercice de mouvements accélérés.

Exercice 8.17
Écrivez le code d'un jeu fonctionnant de la manière suivante : une balle se déplace au hasard sur un canevas, à vitesse faible. Le joueur doit essayer de cliquer dans cette balle à l'aide de la souris. S'il y arrive, il gagne un point, mais la balle se déplace désormais un peu plus vite, change de couleur et son rayon diminue de 10%. A chaque clic, le score atteint est indiqué et le jeu s'arrête une fois que le quota de clics à disposition est épuisé.



Définissez une classe `Application()` dérivant de la classe `Tk()`. Le constructeur de cette classe devra configurer le jeu en initialisant le nombre d'essais possibles, le nombre de points de la partie et le nombre de clics déjà réalisés. En outre, il instanciera le canevas, les boutons, l'étiquette de texte ainsi que la balle animée.

Les méthodes suivantes devront être définies :

- `start_it()` permettra de lancer l'animation de la balle dans le canevas.
- `stop_it()` permettra de stopper temporairement l'animation de la balle dans le canevas.
- `terminate()` permettra de fermer et détruire l'application.
- `pointeur()` permettra de gérer les clics réalisés dans le canevas.

La balle initialisée dans l'objet `Application()` sera une instance de la classe `Balle()`. Lorsqu'on instanciera un nouvel objet de cette classe, on pourra préciser le canevas dans lequel la balle se déplacera, la position initiale de la balle ainsi que les composantes de son vecteur de déplacement, l'intervalle de temps entre deux déplacements ainsi que la couleur et le rayon de la balle créée. Les méthodes suivantes devront alors être définies :

- `move()` permettra d'animer la balle en la faisant rebondir contre les parois du canevas.
- `set_color(newColor)` permettra de changer la couleur de la balle.
- `set_time(newTime)` permettra de changer l'intervalle de temps entre deux déplacements.
- `set_radius(newRadius)` permettra de changer le rayon de la balle.

Exercice 8.18
Modifiez l'exercice précédent de telle sorte qu'avant une partie, l'utilisateur puisse choisir le nombre de balles avec lesquelles il désire jouer. Ce choix se fera à l'aide de trois boutons-radio proposant les valeurs 1, 3 et 5. Une fois le nombre de balles choisi, le jeu pourra commencer et les balles initialisées seront positionnées et déplacées de manière aléatoire dans le canevas. En plus du constructeur qui devra initialiser les boutons-radio ainsi qu'une liste contenant les balles présentes dans le canevas, les méthodes suivantes devront également être modifiées :

- `start_it()` permettra, en début de partie, d'instancier les balles en les plaçant dans une liste.
- `pointeur()` permettra de gérer les points en contrôlant l'impact d'un clic avec toutes les balles présentes dans le canevas.