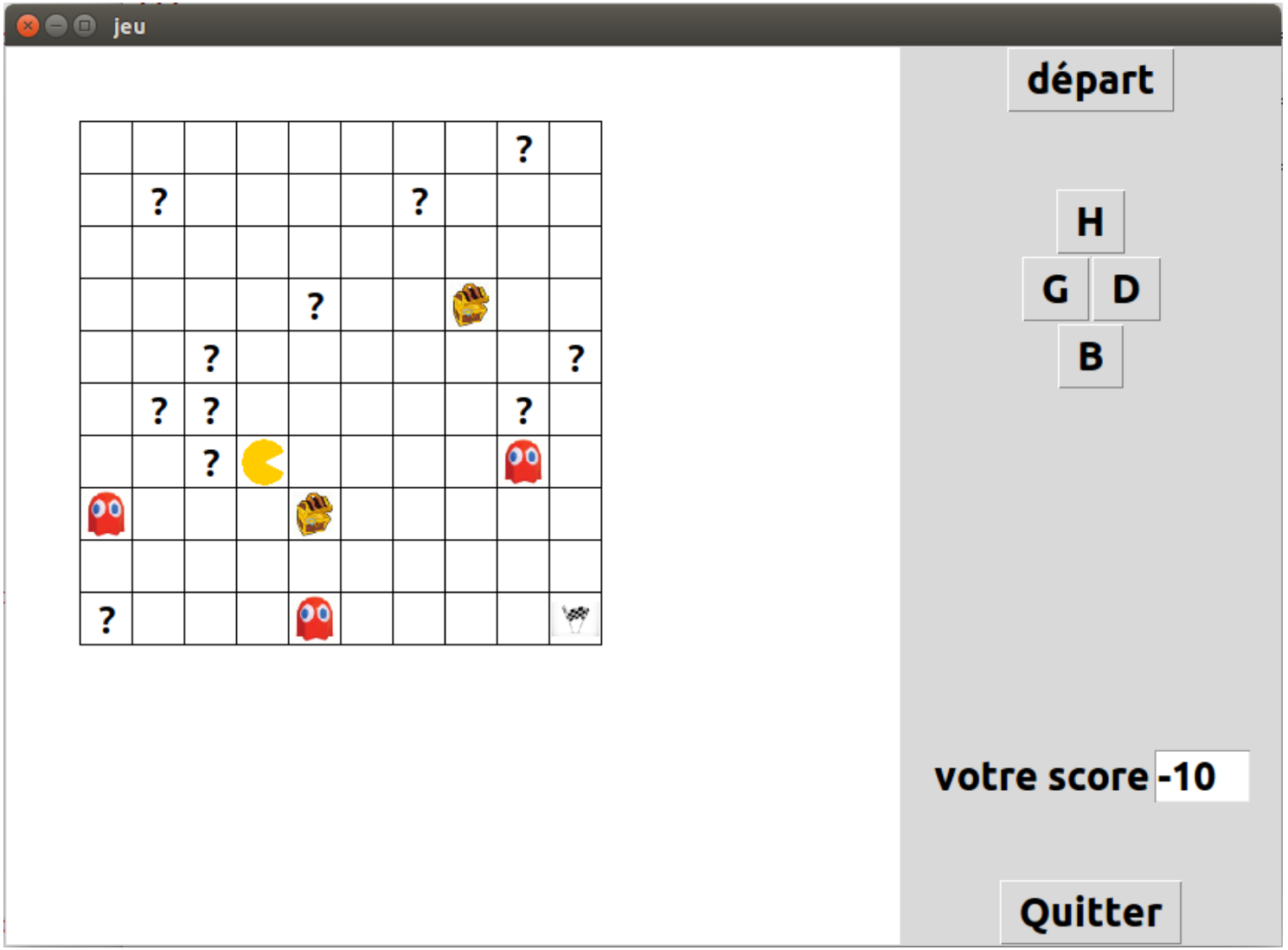
PYTHON: utilisation d'une interface graphique: Tkinter

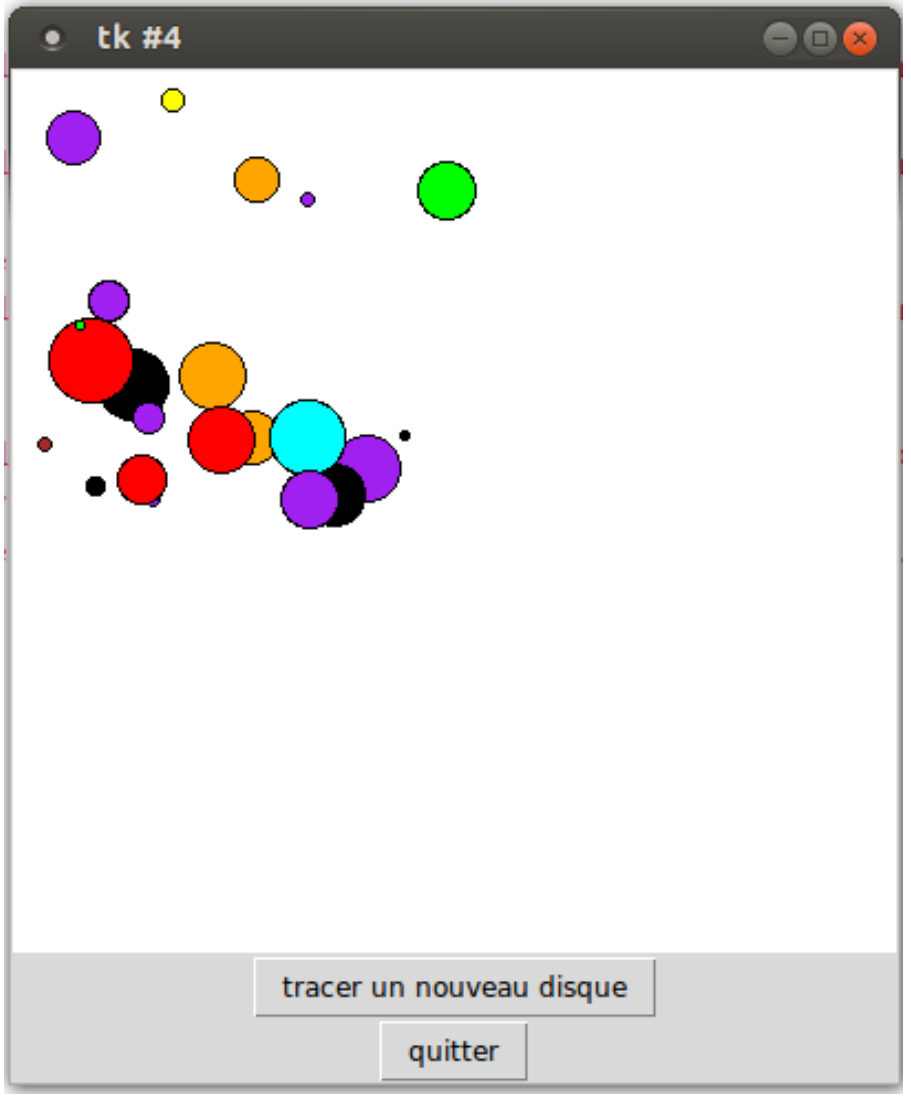
Françoise Lambert

francoise.lambert@unicaen.fr

Exemple d'un jeu réalisé par des étudiants en L1 info



→ **un dessin aléatoire plus simple que nous allons mettre en place rapidement:**



Nous allons utiliser le module **Tkinter** qui permet de faire des interfaces graphiques (GUI) donc d'avoir des fenêtres, des boutons à cliquer, des zones d'affichages, de saisie.....

De ce fait nous allons forcément utiliser :
→ des objets (ici des objets graphiques)
→ des méthodes associées.

Comment cela marche?

C'est de la programmation **évènementielle:**

Il y a une boucle qui tourne en permanence (quand on lance l'application) qui réagit aux clics (ou autres évènements comme taper sur une touche...), cliquer sur un bouton...

Du coup quand le mainloop est lancé on n'a plus accès à python jusqu'à la fin de la boucle.

Nous verrons que les programmes de jeu style de pierre-feuille-ciseau ou jeu du nombre mystérieux n'ont plus la même structure: en particulier ils ne fonctionnent pas avec une boucle while

→ premier exemple très simple:

Il nous faut :
→ une fenêtre
→ une zone où sera écrit un texte (ici sous forme d'un label) qui est ici “Bonjour tout le monde”
→ un bouton à cliquer (button) qui va déclencher une action (qui est ici fermer la fenêtre)

D'où le programme:

```
from tkinter import *

mafen = Tk()

texte = Label(mafen, text='Bonjour tout le monde !', fg='red')

boutonQuitter = Button(mafen, text='Quitter', command =
mafen.quit)

texte.pack()
boutonQuitter.pack()

mafen.mainloop()
```

Explications du programme

mafen=Tk()
création d'une fenêtre qu'on nomme mafen:
on crée une instance de la classe fenêtre; il se passe ici deux choses à la fois :
- la création d'un nouvel objet
- l'affectation d'une variable (ici mafen) qui va désormais servir de référence pour manipuler l'objet.

tex1 = Label(mafen, text='Bonjour tout le monde !', fg='red')
On crée une zone d'écriture ici un label ; on utilise les paramètres :
- où dessiner le bouton (ici dans mafen)
- le texte (ici “bonjour tout le monde”)
- la couleur d'écriture (ici “red”) (par défaut l'écriture est noire)

bou1 = Button(mafen, text='Quitter', command = mafen.quit)

On crée un bouton: les paramètres sont ici:

- où dessiner le bouton (ici dans mafen)
- le texte apparaissant sur le bouton (ici “quitter”)
- la commande à exécuter si on clique sur le bouton: ici il s'agit de la méthode prédéfinie **quit** qui arrête le programme.

tex1.pack()

bou1.pack()

La méthode pack permet de **positionner** les widgets; si on omet une commande de positionnement(pack ou grid cette année), les widgets n'apparaitront pas à l'écran.

mafen.mainloop()

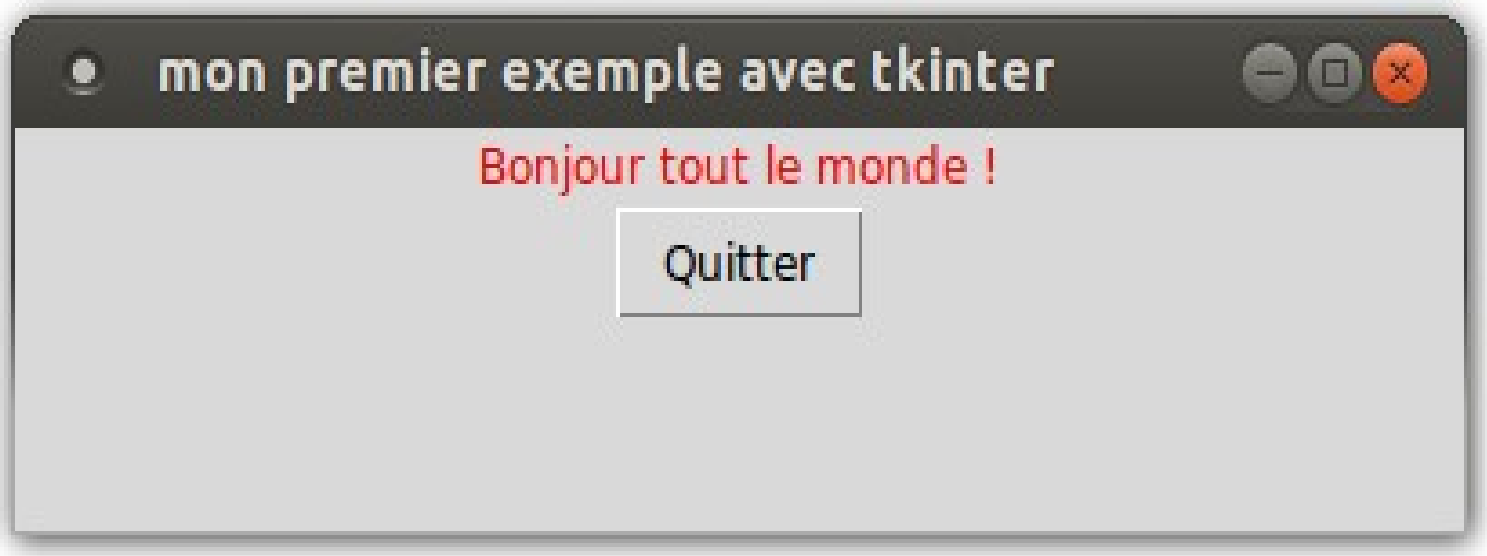
C'est la boucle principale qui lance le script.

Attention il faut aussi avoir prévu la fin de programme!!!

(par exemple avec un bouton quitter et la commande associée)

On pourrait rajouter:

```
from tkinter import *

mafen = Tk()

mafen.title("mon premier exemple avec tkinter")
mafen.geometry("400x100")
## ou mafen.geometry("400x100+150+150")
## pour une position de la fenêtre

texte = Label(mafen, text='Bonjour tout le monde !', fg='red')

boutonQuitter = Button(mafen, text='Quitter', command =
mafen.quit)

texte.pack()
boutonQuitter.pack()

mafen.mainloop()
```

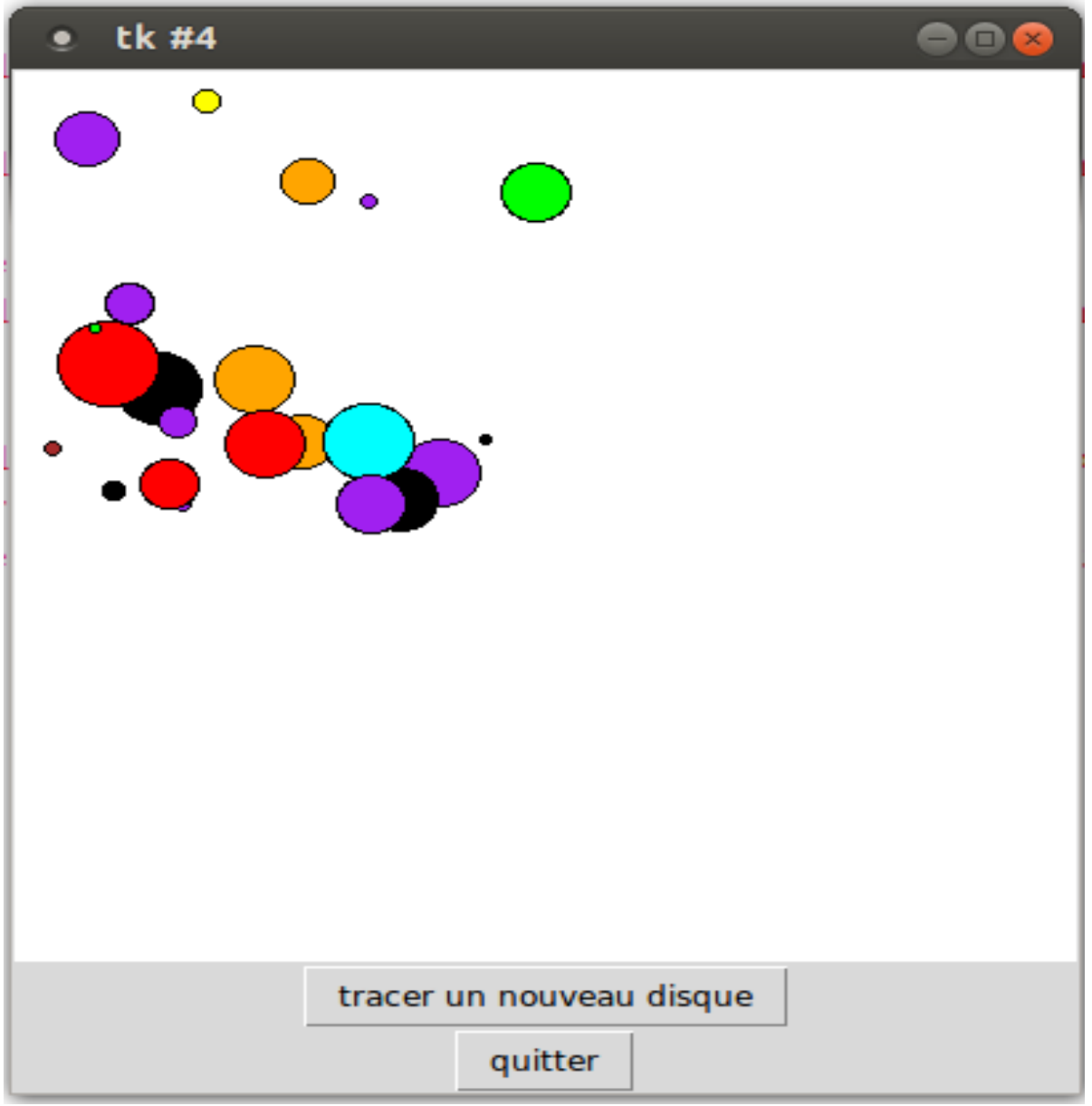


D'où la fenêtre ci-dessus:

title: donnera le titre (dans le cadre foncé)

geometry: fixera les dimensions minimales (sinon la fenêtre s'adapte au contenu) et éventuellement la position

l'exemple du dessin aléatoire :



Il nous faut donc:

- la fenêtre principale
- un bouton “tracer un disque” dont l'action sera de tracer un disque de couleur aléatoire
- un bouton “quitter” dont l'action sera de quitter l'application
- une zone pour dessiner: on va prendre un widget: ***canvas***

Il faudra importer les modules Tkinter et random (pour l'aléatoire)

```
from tkinter import *
from random import *
```

On va définir une variable globale regroupant les valeurs possibles dans ce programme pour la couleur du crayon:

```
mescouleurs=["red","pink","yellow","green","blue","cyan","purple"
,"brown","black","orange"]
```

définition des widgets:

#la fenêtre principale

```
mafen=Tk()

#le canvas
can=Canvas(mafen,width=400,height=400,bg="white")
can.pack()

# les deux boutons: définition  et positionnement
bd=Button(mafen,text="tracer un nouveau disque”,
command=disque)
bd.pack()



bq=Button(mafen,text="quitter",command=mafen.quit)
bq.pack()


# le lancement de l'application
mafen.mainloop()
```

On doit définir la fonction qui est une commande de bouton **avant** les définitions des widgets dans le fichier (sinon erreur)

```
def disque():  # commande de tracage du disque
    x,y,d=randint(1,200),randint(1,200),randint(1,20)
    coul=choice(mescouleurs)
    can.create_oval(x-d,y-d,x+d,y+d,fill=coul)

  # choix aléatoire d'un centre et d'un diamètre
  # choix aléatoire d'une couleur
  # puis dessin d'un disque à l'aide de la méthode create_oval
```

remarque:

On a deux méthodes principales de fin de programme:

- la methode quit permet de quitter l'application sans fermer la fenêtre
- la méthode destroy qui quitte l'application et ferme la fenêtre

Selon les machines parfois quit ou destroy ne quitte pas “correctement” et ne redonne pas la main..... à voir selon sa configuration!!!

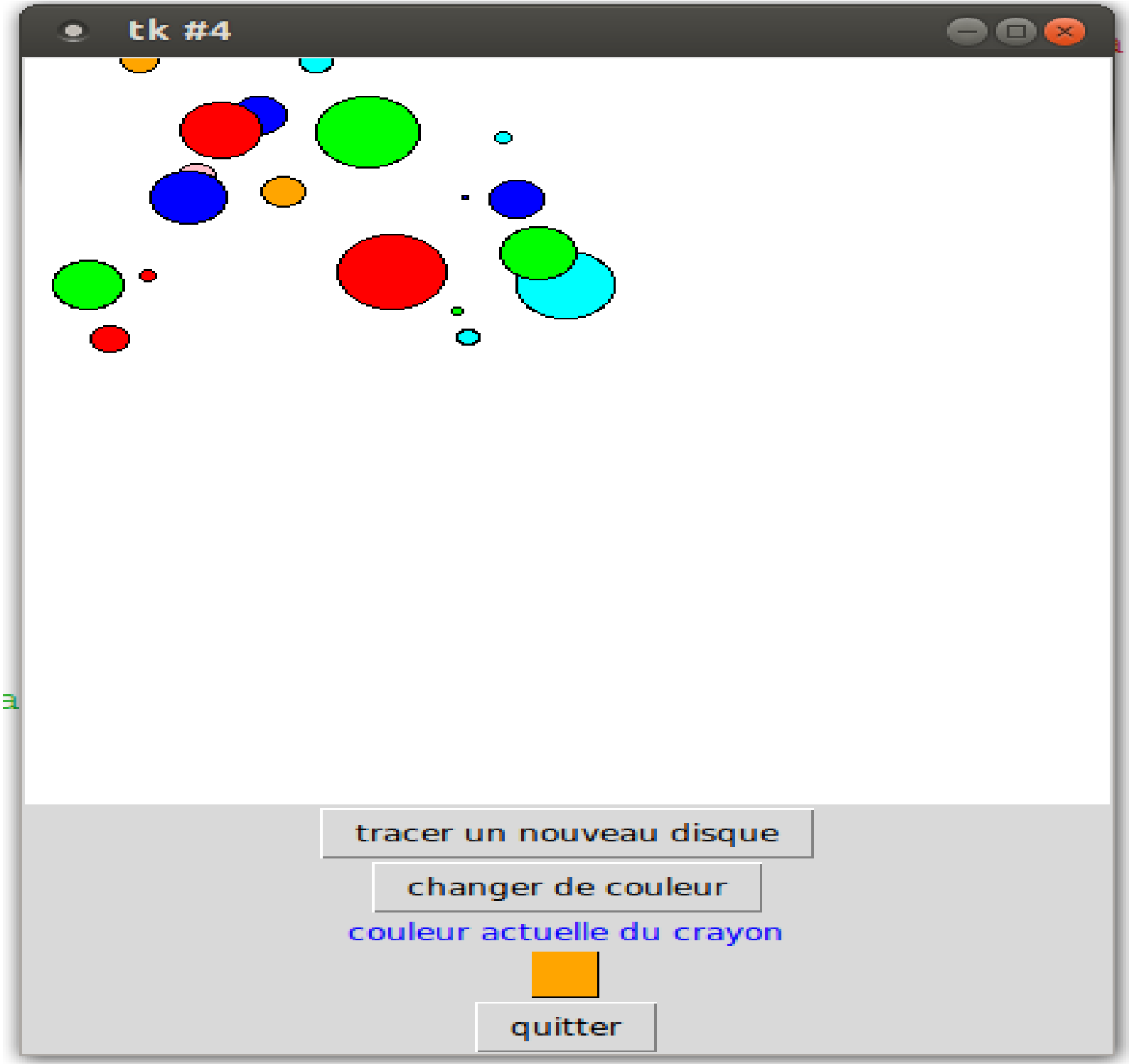
on peut alors éventuellement faire un quit et un destroy dans cet ordre:

```
def monquitter():
  mafen.quit()
  mafen.destroy()

bq=Button(mafen,text="quitter",command=monquitter)
```

On voudrait pouvoir choisir quand on change de couleur et avoir un petit carré témoin de la couleur qui est “en cours”:

on va rajouter:

- un label pour indiquer “couleur actuelle du crayon”
- un canvas où on va dessiner un carré de cette couleur (on aurait aussi pu utiliser un bouton inactif).
- un bouton “changer de couleur”


tk #4
tracer un nouveau disque
changer de couleur
couleur actuelle du crayon
quitter

D'où les changements dans le programme:

```
def changercouleur():
   return "   # fonction à écrire

#définition d'une variable globale pour la couleur
coul=”red”  # initialisation

# ajout d'un label
lab=Label(mafen,text="couleur actuelle du crayon",fg="blue")
lab.pack()


# ajout du petit canvas
temoin=Canvas(mafen,width=25,height=25)
temoin.pack()

# premier dessin de la couleur au debut de l'application:
temoin.create_rectangle(0,0,25,25,fill=coul)
```

Reste à écrire la fonction pour changer de couleur : elle doit modifier la variable globale coul

Problème: comment modifier la valeur d'une variable globale depuis une fonction ?

C'est un problème plus général que les widgets dans Tkinter même si c'est là que l'on utilisera le plus cette possibilité.

modification d'une variable globale

On voudrait qu'une fonction puisse modifier une variable globale

essayons un peu:

```
def augmente (y):
        y= y+1
        print (“on a y =” , y)

>>> y= 4
>>> augmente(y)
on a y = 5
>>> y
4
```

On voit que y n'est pas modifié globalement, même si localement il change de valeur

Si on veut ne pas mettre y en paramètre

```
def augmente ():
        y= y+1
        print (“on a y =” , y)
```

```
>>> y=3
>>> augmente()
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    augmente()
  File "/home/lambert/hjjh.py", line 2, in augmente
    y= y+1
UnboundLocalError: local variable 'y' referenced before assignment
```

Le programme plante car y n'est pas défini avant sa modification (localement)
En fait il faut indiquer qu'il faut prendre la variable y comme variable **“globale”**.

En fait une variable globale ne pourra être modifiée par un programme que si on la déclare comme **variable globale** en début de programme: c'est une sécurité: un programme ne peut pas “sans faire exprès” modifier une variable globale (hors mutations physiques)

```
def  aug():
     global p   #ceci déclare qu'il faut prendre la var globale p
     p=p+1
     print (p)

>>> p=20
>>> aug()
21
>>> p
21
>>> aug()
22
>>> p
22
```

Remarque importante:
Si la variable globale est mutable (comme une liste) et qu'on la modifie physiquement depuis un programme, elle est effectivement modifiée **sans avoir besoin d'utiliser global**

```
def double(t):
    for i in range(len(t)):
        t[i]=2*t[i]
    return t

def change():
    double(t)  “ ne renvoie rien mais fait la modif
```

```
>>> t
[1, 2, 3, 4, 5]
>>> change()
>>> t
[2, 4, 6, 8, 10]
```

Remarque importante: si le programme fabrique une nouvelle liste (au lieu de travailler avec des modifications physiques) , il faudra utiliser global

```
def doublecopie(t):
    return [2*x for x in t]


def change2():
    global t
    t=doublecopie(t)

>>> t
[1, 2, 3, 4, 5]
>>> change2()
>>> t
[2, 4, 6, 8, 10]
```

D'où finalement pour la fonction changer de couleur:

```
def changercouleur():
    global coul    # importation de la variable  pour pouvoir la modifier
    coul=choice (mescouleurs)  # nouvelle valeur
    temoin.create_rectangle(0,0,25,25,fill=coul)  # mise à jour du canvas
```

Structure d'un programme Tkinter

La structure de base d'un programme simple utilisant Tkinter est globalement (et dans cet ordre):

- importation du module tkinter (et autres modules si nécessaire)
- définition des fonctions qui sont les commandes
- définition de la fenêtre principale
- définition des widgets
- positionnement des widgets
- lancement de l'application avec mainloop

Ecriture d'un programme Tkinter

Généralement on s'occupe des lignes de codes dans l'ordre suivant (mais en les mettant dans l'ordre donné précédemment)

- importation du module tkinter
- définition de la fenêtre principale
- lancement de l'application avec mainloop

- définition des widgets
- positionnement des widgets
- écriture des fonctions qui sont les commandes

Ecriture d'un programme Tkinter

remarque: pour pouvoir tester les widgets et leurs positionnements, on peut commencer par définir des fonctions qui ne font “rien”

```
def commande():
        return ''
```

Par contre, si une fonction de commande n'est pas définie, on ne pourra pas “lire” le programme il y aura un message d'erreur lors du RUN.

Nos premiers Widgets

- **Button**: un bouton à cliquer, 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, ou créer des éditeurs graphiques et bien entendu pour mettre des images

- **Entry**: un champ de saisie où on pourra insérer un texte quelconque à partir du clavier.

- **Label**: permet d'afficher un texte quelconque ou une image.

- **Text:** zone d'affichage de texte

Nos premiers Widgets

. sont définis dans le programme

. le premier paramètre est oblogatoirement le widget qui le contient (par exemple la fenêtre définie à l'appel de Tk(), ou un widget parent pour mieux les disposer....)

. Tous les autres paramètres sont optionnels.....

. quelque possibilités:

- une couleur d'écriture, une couleur de fond
- une taille
- une commande associée .....

Toutes ces options ont une valeur par défaut (l'écriture par exemple est noire, la taille s'adapte au contenu...)

quelques options communes :

- width: valeur positive indiquant la largeur du widget en unité de caractère (donc en pixel pour certains et en caractères dans la police donnée par l'option 'font' pour d'autres) .

- height: valeur positive indiquant la hauteur du widget .Doit valoir au moins 1. (pas de height pour Entry qui a forcément une hauteur d'une ligne). Pour une zone de texte c'est en lignes d'écriture (donc ne pas mettre trop grand), pour un canvas c'est en pixels.

- background (bg): couleur de l'arrière plan du widget

- foreground (fg): couleur du premier plan du widget c'est à dire de l'écriture (sauf Canvas)

Important : “agencement”

- Un widget doit être **positionné** pour apparaître. Il ne suffit pas de le définir.... (la définition et l'agencement ne sont pas forcément au même endroit du programme).

- 2 méthodes seront utilisées cette année:

  grid

  pack

  Attention: il ne faut pas mélanger plusieurs méthodes d'agencement dans le même programme (sinon les widgets n'apparaissent pas ou pas tous)

2 méthodes seront utilisées cette année:

grid et **pack**

Par défaut si on met widget_num_i.pack() à tous les widgets alors ils apparaîtront les uns en dessous des autres sans l'ordre où les lignes auront été lues (comme dans l'exemple des disques de couleur).

Idem pour widget_num_i.grid()

- Grid: ce gestionnaire d’agencement crée des affichages en organisant les widgets dans une grille 2D; on donnera donc des indices de lignes et de colonnes (row, column) ces numéros de lignes et de colonnes n'étant pas absolus mais relatifs

- Pack: ce gestionnaire d’agencement permet de positionner les widgets en les empaquetant dans un widget parent. Les widgets sont traités comme des blocs rectangulaires placés dans un cadre. (side= LEFT, BOTTOM, TOP, RIGHT)

grid

Ce gestionnaire positionne les widgets selon lignes et colonnes

(grille 2D) grâce à row et column. Chaque widget doit être

positionné pour apparaître (mais on peut les faire apparaître

dans une commande au cours d'un programme)

grid considère la fenêtre comme une sorte de grille .

On indique dans quelle ligne (row) et dans quelle colonne (column) de ce tableau on souhaite placer les widgets. Ce n'est pas absolu: 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**.

Si on ne fournit aucun numéro pour une ligne ou une colonne, la valeur par défaut sera zéro.

Tkinter détermine automatiquement le nombre de lignes et de colonnes nécessaires.

Les objets sont positionnés dans l'ordre des lignes et des colonnes mais pas de façon absolue (il n'y a pas de “case 1,2) mais quelque chose situé en colonne a est plus à gauche que quelque chose situé en colonne b si a<b (de même pour les lignes).

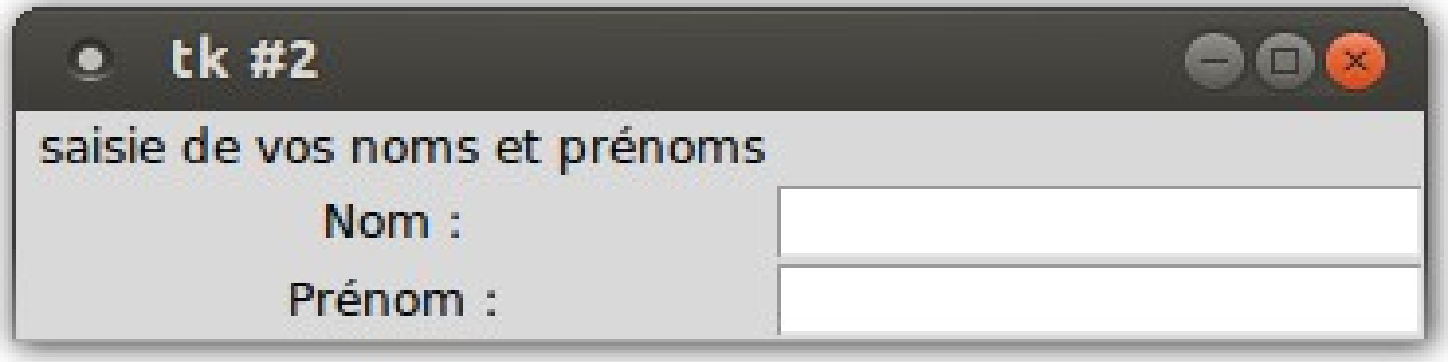
Premier exemple: on veut obtenir cette disposition:

On va positionner les deux labels en colonne 0 et les deux entry en colonne 1 (mais on pourrait prendre 5 et 10 comme numéros de colonne)

```
from tkinter import *
fen1 = Tk()
lab1 = Label(fen1, text = 'Nom :')
lab2 = Label(fen1, text = 'Prénom :')
entr1 = Entry(fen1)
entr2 = Entry(fen1)
lab1.grid(row =0)  # automatiquement column=0
lab2.grid(row =1)  # automatiquement column=0
entr1.grid(row =0, column =1)
entr2.grid(row =1, column =1)
fen1.mainloop()
```

Si on rajoute un “titre” avec un autre label:

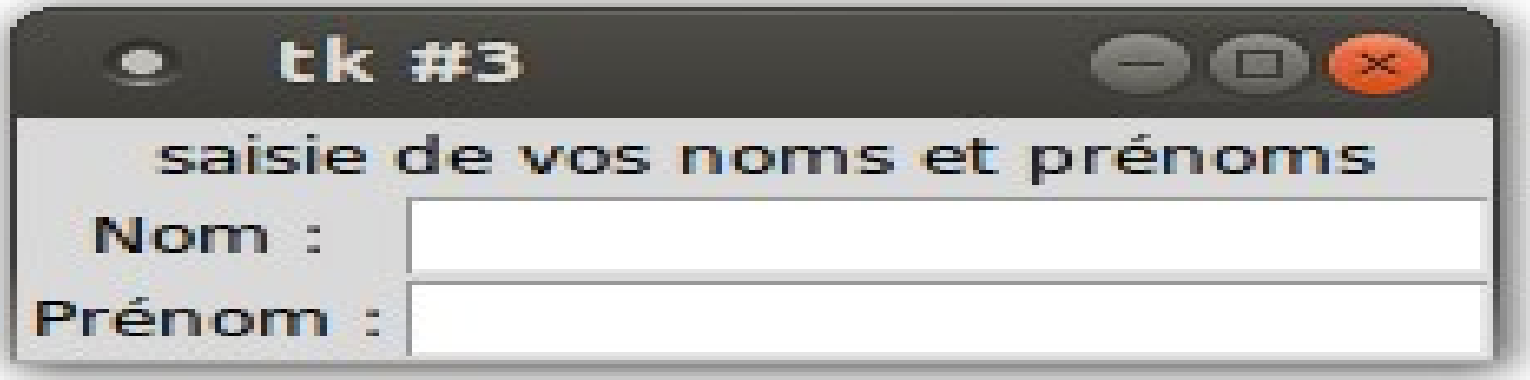
```
from Tkinter import *
fen1 = Tk()
titre=Label(fen1,text= “saisie des noms et prénoms”)
titre.grid(row=0,column=0)
lab1 = Label(fen1, text = 'Premier champ :')
lab2 = Label(fen1, text = 'Second :')
entr1 = Entry(fen1)
entr2 = Entry(fen1)
lab1.grid(row =10)# on modifie les lignes pour mettre “avant”
lab2.grid(row =20)
entr1.grid(row =10, column =10)
entr2.grid(row =20, column =10)
fen1.mainloop()
```



Si on ne fait rien: le titre est trop vers la gauche puisqu'il est dans la même colonne que les deux autres labels prénom et Nom

Il suffit de rajouter une instruction pour faire s'étirer le widget:

```
titre.grid(row=0,column=0, columnspan=15)
```



pack

Ce gestionnaire de “géométrie” permet de positionner les widgets dans le widget parent selon 4 zones: LEFT,RIGHT,TOP et BOTTOM en utilisant side=BOTTOM par exemple comme argument de pack.

side: spécifie le côté contre lequel le widget doit être 'paqué'. Pour paquer les widgets verticalement, utiliser TOP (valeur par défaut). Pour paquer les widgets horizontalement, utiliser LEFT. On peut aussi utiliser BOTTOM et RIGHT

Reprenons l'exemple du dessin aléatoire des cercles avec pack: on avait

can.pack()

bd.pack()

bc.pack()

bq.pack()

lab.pack()

temoin.pack()

Ce qui met tous les widgets les uns en dessous des autres en les centrant d'où

on obtient le dessin:

on peut mettre le bouton quitter en bas et le dessin à gauche:

can.pack(side=LEFT)

bd.pack()

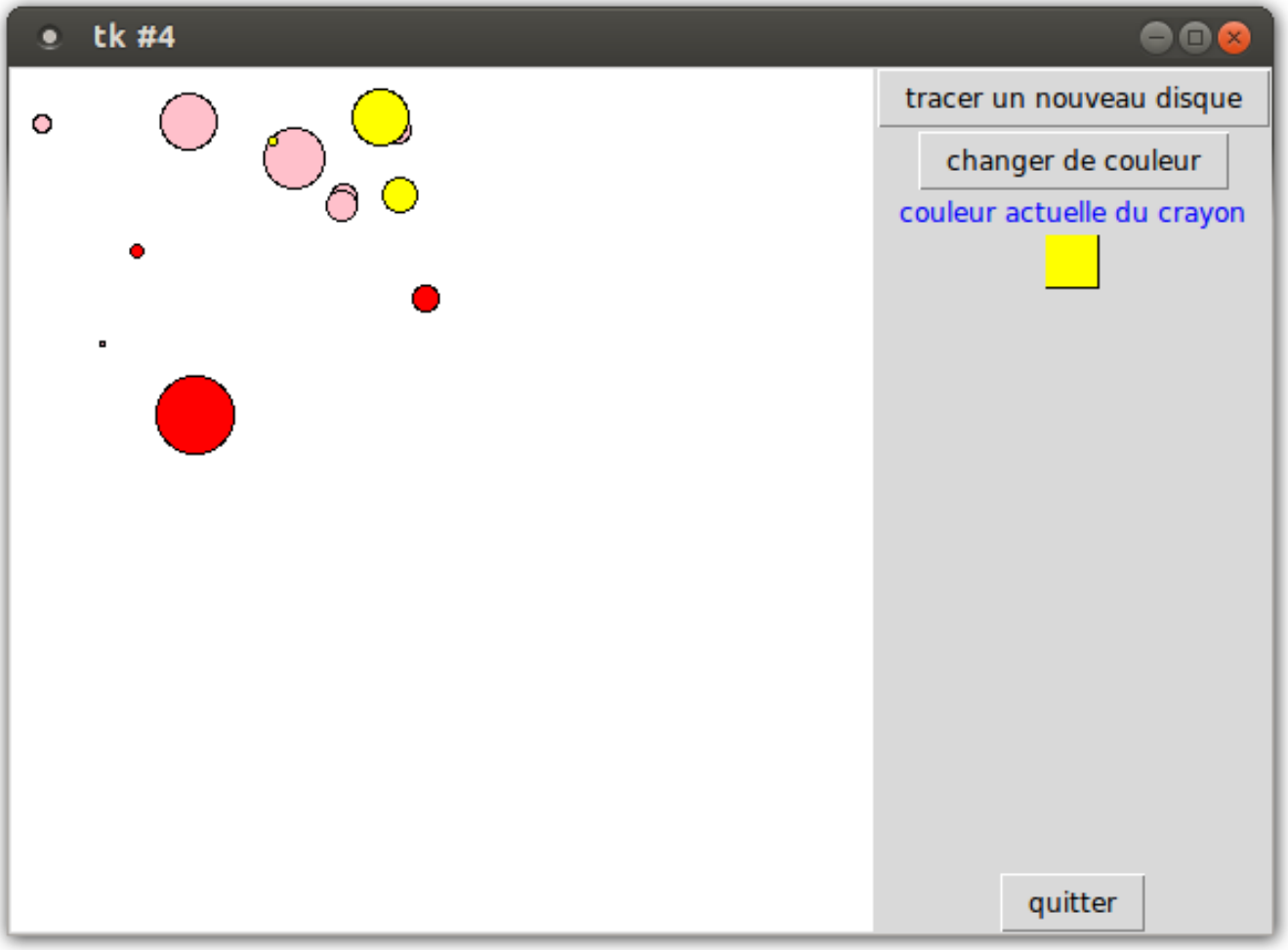
bc.pack()

bq.pack(side=BOTTOM)

lab.pack()

temoin.pack()

on obtient alors le dessin:



Remarque si on met tout à gauche sauf le canvas:

```
can.pack()
bd.pack(side=LEFT)
bc.pack(side=LEFT)
bq.pack(side=LEFT)
lab.pack(side=LEFT)
temoin.pack(side=LEFT)
```

on obtient alors le dessin:

Il y a d'autres possibilités pour grid et pack que nous verrons plus tard ou pas (et voir doc).

Pour les deux instructions: on peut faire “disparaître” un widget en “l'oubliant”:

Truc.pack_forget()

Truc.grid_forget()

Dans les deux cas le widget n'est plus à l'écran puisqu'il n'est plus positionné (on peut le re-positionner dans une autrefonction du programme)

Revenons à nos widgets:

button

Il faut obligatoirement déclarer dans quel widget se place le bouton.

Pour le reste on peut avoir les options (entre autres):

- bd: largeur de la bordure en pixels (2 par défaut)
- bg: couleur du fond du bouton
- command: fonction de commande du bouton
- fg: couleur du texte sur le bouton
- font: paramètres de la police pour le texte
- text: texte sur le bouton
- width, height: largeur et hauteur du bouton

On pourra aussi mettre une image sur un bouton, et il y a encore d'autres options (voir les docs on line).....

button

Ces options peuvent être choisies lors de la définition du bouton comme:

B1= Button(mafen, text=”OUI”)

ou configurées au moment où on le souhaite à l'aide d'instructions:

B1.configure (fg=”blue”, text=”NON”)

canvas

Un canvas possède les élément prédéfinis suivants pour les “dessins”

- line
- oval (pour dessiner cercles ou ellipses)
- rectangle
- polygon
- image ou bitmap (pouyr insérer une image)
- text
- arc (arc, corde, part de "camembert")

les lignes du canvas

On a les méthodes:

- create_line (x0, y0,x1, y1, options...) : crée une ligne joignant les points de coordonnées (x0, y0) et (x1, y1)
- delete (item) : détruit un élément "ligne".
- create_line(x0, y0, x1 ,y1, ..., xn, yn, options): permet de dessiner plusieurs lignes à la fois (mais elles auront alors forcément les mêmes options de dessin)
- itemconfigure (item, options...) : modifie les options d'une ou de plusieurs éléments "lignes".

et quelques options : (il y en a d'autres.....)

- width: Largeur de la ligne (par défaut 1 pixel)
- fill: Couleur de la ligne (par défaut noire)

l'"oval" du canvas: pour tracer des ellipses

On a les méthodes:

- create_oval (x0, y0, x1, y1, options...) : crée un élément "oval" c'est à dire une ellipse qui s'inscrit dans le rectangle dont le coin en haut à gauche est (x0,y0) et le coin en bas à droite est (x1,y1). On peut donc obtenir un cercle si le dessin est fait dans un carré

- delete (item) : détruit un élément "oval".

- coords (item, x0, y0) : déplace un ou plusieurs éléments "oval".

- itemconfigure (item, options...) : modifie les options d'un ou de plusieurs éléments de type "oval".

Options pour les tracés:

On a les options: (qui seront les mêmes pour les “ovals”, les rectangles et les polygones):

- fill: couleur de l'intérieur de l'objet (transparent par défaut). Si on entre une chaîne de caractères vide, l'intérieur n'est pas colorié

- outline: couleur du contour de l'objet (noir par défaut) ; il n'y a pas de dessin du contour si outine =””

- width: largeur du contour (1 pixel par défaut)

tracer des rectangles dans un canvas:

On a les méthodes:

- create_ rectangle(x0, y0, x1, y1, options...) : crée un élément rectangle dont le coin en haut à gauche est (x0,y0) et le coin en bas à droite est (x1,y1). On peut donc obtenir un carré
- delete (item) : détruit un élément rectangle
- coords (item, x0, y0, x1, y1) : modifie les coordonnées d'un ou de plusieurs éléments rectangle. L'argument item peut désigner un ou plusieurs éléments "rectangle" ou tout autre élément ayant les mêmes quatre coordonnées.

- itemconfigure (item, options...) : modifie les options d'un ou de plusieurs élément "rectangle".

(avec les options vues précédemment)

tracer des polygones dans un canvas:

On a les méthodes:

- create_polygon (xy, options...) ou create_polygon (x0, y0, x1, y1, x2, y2, x3,y3,..., xn, yn, options...) : créent un élément polygone. On doit préciser au moins 3 couples de coordonnées pour créer un nouveau polygone. On peut soit entrer directement 2n coordonnéees ou mettre en paramètre une liste de coordonnées.

- delete (item) : détruit un élément polygone.

- coords (item, x0, y0, x1, y1, x2, y2,..., xn, yn) : modifie les coordonnées d'un ou de plusieurs éléments polygone. Notez que les coordonnées doivent être données séparément ; on ne peut pas utiliser une liste comme pour la méthode create_polygon.

- itemconfigure (item, options...) : modifie les options d'un ou de plusieurs élément "polygone".

(toujours avec les mêmes options)

les images dans un canvas:

On a les méthodes:

- delete, coords, itemconfigure comme précédemment

- create_image (x0, y0, options...) : crée un élément "image" et le place à la position donnée.

Attention: Il faut que l'image ait été “importée“ en Python: pour cela on utilise PhotoImage (voir exemple)

remarque: il vaut mieux que les images soient au format .gif (certains formats ne passent pas)

Exemple d'images dans un canvas:

Pour dessiner une image dans un canvas:

- on la définit (“importation”):

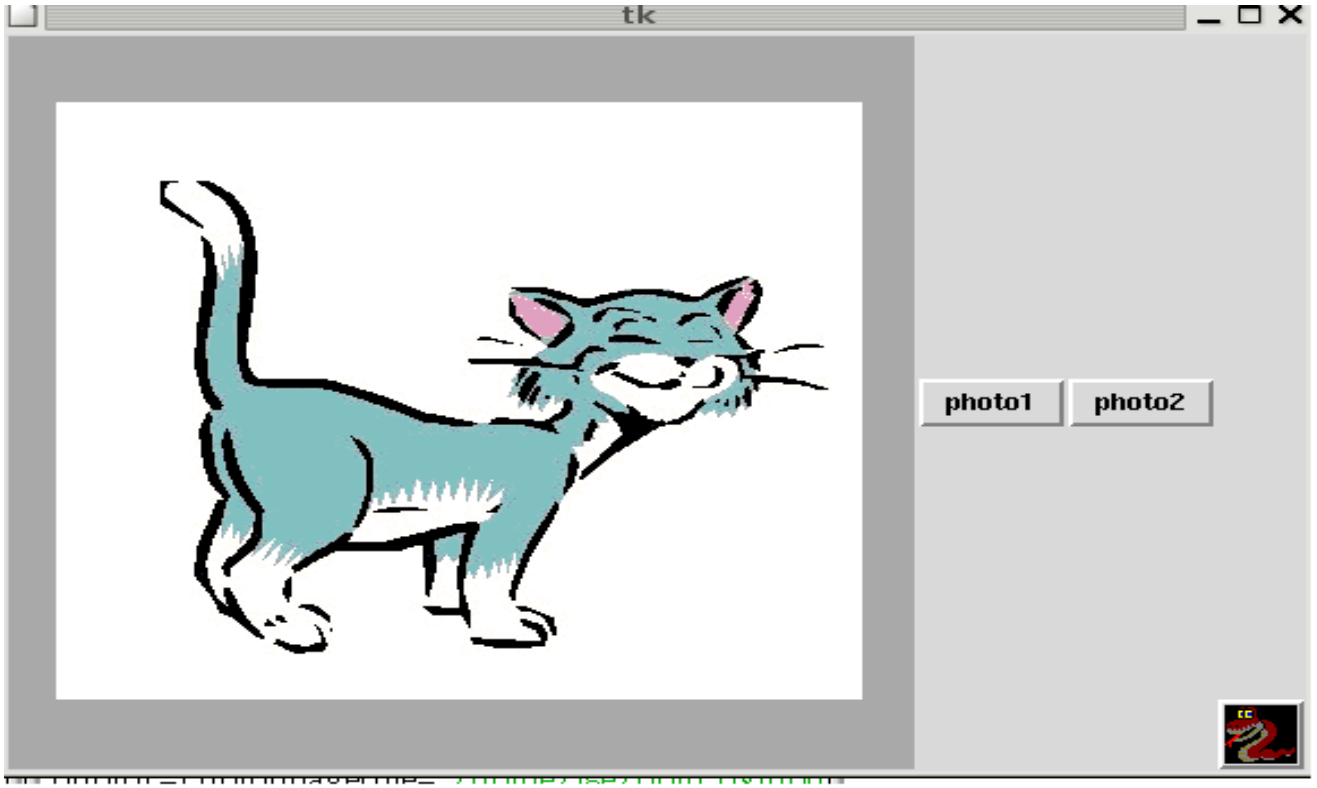
maphoto1=PhotoImage(file=”imagespython\cadeau.gif”)

- Si un canvas appelé mondessin a été défini,on aura ensuite:

mondessin.create_image(50,50, image=maphoto1)

(le paramètre 50,50 permettant de positionner l'image dans le canvas).

exemple:



On voudrait produire la fenêtre ci-dessus: si on clique sur photo1 s'affiche la photo1, si on clique sur photo2, s'affiche la photo2, et on veut une icone python sur le bouton qui permet de quitter l'application (puisqu'on retourne à Python!)

```
from Tkinter import *

def affiche1():  # affichage de la photo1
    can.delete(ALL)
    c=can.create_image(200,200,image=photo1)

def affiche2():# affichage de la photo2
    can.delete(ALL)
    c=can.create_image(200,200,image=photo2)

dessin=Tk()

# “importation” des images en python
photo1=PhotoImage(file="chat1.gif")
photo2=PhotoImage(file="chat2.gif")
photo3=PhotoImage(file="pyt.gif")
```

```
can= Canvas(dessin,height=400,width=400,bg="dark grey")
can.pack(side=LEFT)
b2=Button(dessin,text="photo1",command=affiche1)
b2.pack(side=LEFT)
b3=Button(dessin,text="photo2",command=affiche2)
b3.pack(side=LEFT)
b1=Button(dessin,image=photo3,command=dessin.destroy)
b1.pack(side=BOTTOM)

dessin.mainloop()
```

le widget Text

Il nous servira essentiellement à afficher du texte

exemple: montexto=Text(lieu, width=400,height=3) # 3 lignes

Des méthodes pour utiliser les index dans le “text”:

- insert (index,text) : insère du texte à l'endroit donné (index)

  attention l'index correspond à ligne ET colonne donc onn a des paramètres du style “0.0”

- delete(index), delete(start,stop) : supprime le caractère de position "index" ou l'ensemble du texte situé entre start et stop.

On peut par exemple effacer tout le contenu de Text en utilisant:

montexto.delete(“0.0”,END)

- get(index), get(start,stop) : retourne le caractère de position "index" ou l'ensemble du texte situé entre start et stop.

le widget Entry

Ce widget permet d'afficher, mais surtout de *saisir* une ligne de texte qu'on pourra "récupérer" facilement ; rappel: ce widget possède une seule ligne

On a les méthodes:

- insert (index,text) : insère du texte au niveau de l'index spécifié.

On utilisera insert(index,text) pour insérer du texte au niveau du curseur et insert(END,text) pour ajouter en fin de texte

Ici l'index ne contient que la place dans la colonne donc du style 0.

- delete (index), delete (from,to) : supprime le caractère situé à la position de l'index indiquée ou à l'intérieur d'un domaine donné.
- On utilisera **delete(0,END)** pour effacer l'intégralité du texte du widget.
- get ( ) : récupère le contenu du champ de saisie.

Attention: ces fonctions sont des **méthodes** donc toujours à utiliser avec le nom du widget sur lequel on veut que cela s'applique

le widget Label

On a déjà vu son utilisation dans les exemples:

lab=Label(lieu, text=”bonjour”, fg =”red”)

Si on veut modifier le texte écrit on peut utiliser configure

lab.configure(text=”au revoir”)

lab.configure(text=””): permettra d'effacer ce qui a été écrit

Avec lab.configure() on peut aussi modifier n'importe quel autre paramètre du label.

remarque générale pour toutes les écritures:

On peut configurer la police, la taille, gras ou non .....

Pour cela il y a le paramètre “font” composé de 3 données:

nom de la police, taille, aspect

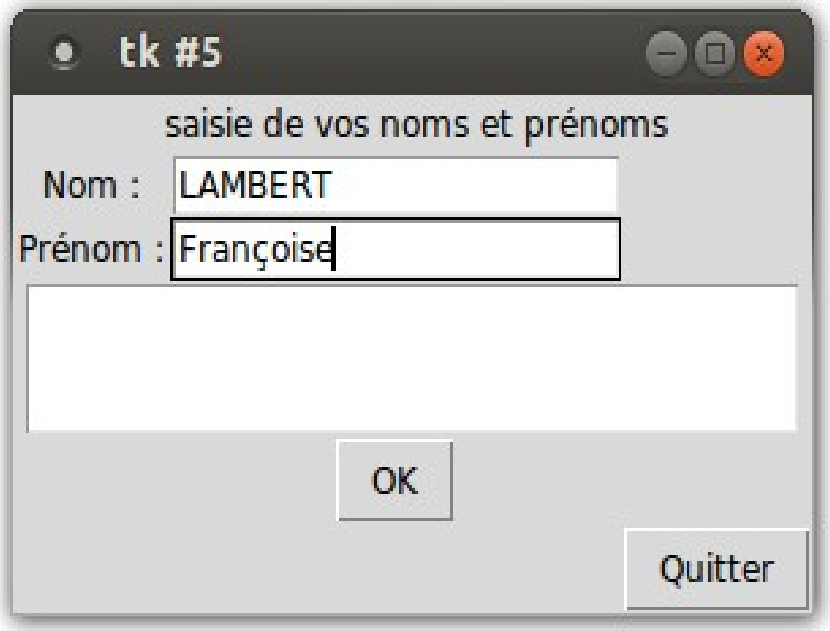
exemples:

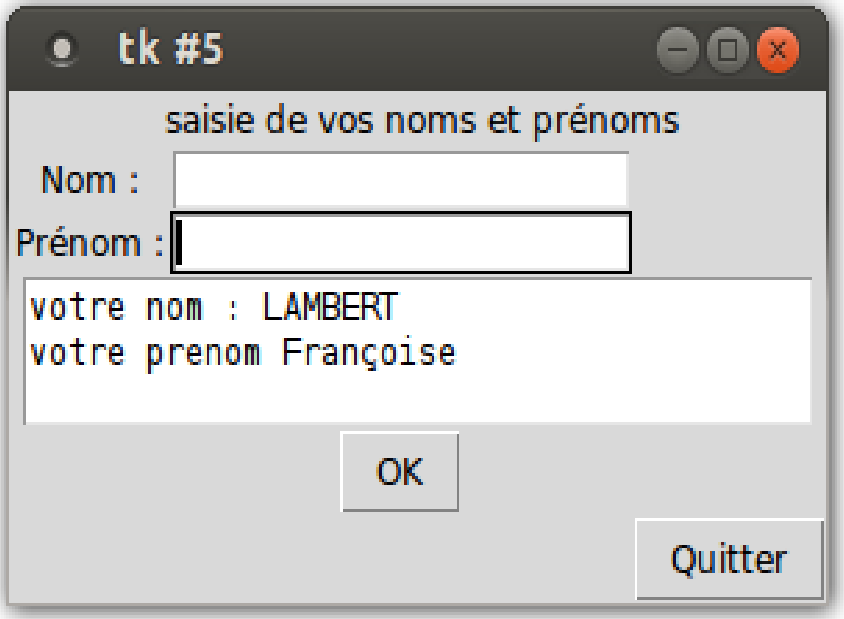
```
bq=Button(fen,text="Q",font=("courier",12,"bold"),
          command=fen.destroy)
```

on peut aussi se définir un paramètre de font pour s'en resservir plusieurs fois:

```
Helv12=("Helvetica",12,"bold")
etiq=Label(fen,text=" truc ",font=Helv12)
```

On veut réaliser la petite interface suivante:





Que faut-il pour notre exemple?

- Une zone texte
- Deux boutons et les commandes associées
- La commande du bouton OK devra
  - récupérer ce qui est dans les zones de saisie
  - effacer les zones de saisie
  - effacer la zone texte
  - écrire dans la zone texte

```
fen1 = Tk()
titre=Label(fen1,text= " saisie de vos noms et prénoms")
titre.grid(row=0,column=0,columnspan=3)
lab1 = Label(fen1, text = 'Nom :')
lab2 = Label(fen1, text = 'Prénom :')
entr1 = Entry(fen1)
entr2 = Entry(fen1)
lab1.grid(row =10)
lab2.grid(row =20)
entr1.grid(row =10, column =1)
entr2.grid(row =20, column =1)

texto=Text(fen1, width=40,height=3)
texto.grid(row=30, columnspan=3)

boutonOK = Button(fen1, text='OK', command = affiche)
boutonOK.grid(row=35, column=1)

boutonQuitter = Button(fen1, text='Quitter', command = monquitter)
boutonQuitter.grid(row=40,column=2)
```

Avec la fonction affiche:

```
def affiche():
    n=entr1.get()   # on récupère
    p=entr2.get()
    entr1.delete(0,END)   # on efface l'entry
    entr2.delete(0,END)
    texto.delete("0.0",END)  # on efface la zone texte
    texto.insert(END, "votre nom : "+n + "\n"+ "votre prenom "+ p)
```

Les frames

Les frames sont des widgets particuliers qui permettent de regrouper d'autres widgets pour un meilleur positionnement (que ce soit avec grid ou pack).

Si on ne met pas un fond de couleur on ne les voit pas apparaître

Le principe est de positionner un cadre (très souvent transparent), puis de positionner d'autres widgets à l'intérieur. Les positions des widgets internes seront calculées dans ce cadre (et non pas par rapport à la fenêtre). Ceci permet notamment de regrouper des widgets.....

Exemple

On veut faire un pavé de déplacement avec 4 boutons pour aller en haut à gauche à droite et en bas

Le plus simple est de mettre les deux boutons du milieu dans un cadre, puis de positionner ensuite.

D'où:

```
fen1 = Tk()
cadre=Frame(fen1)
boutonG = Button(cadre, text='G')
boutonD = Button(cadre, text='D')
boutonH = Button(fen1, text='H')
boutonB = Button(fen1, text='B')
boutonG.pack(side=LEFT)  # position des boutons dans le cadre
boutonD.pack(side=LEFT)  # l'un à côté de l'autre
boutonH.pack()
cadre.pack()
boutonB.pack()

boutonQuitter = Button(fen1, text='Quitter', command = fen1.quit)
boutonQuitter.pack()
fen1.mainloop()
```

Mais si on veut positionner ce “pavé” dans une fenêtre plus complète , le mieux est de le mettre dans un cadre qu'on pourra positionner ensuite.

Exemple:

```
fen1 = Tk()
cadre1=Frame(fen1)  # cadre1 contient le pavé
cadre=Frame(cadre1)
boutonG = Button(cadre, text='G')
boutonD = Button(cadre, text='D')
boutonH = Button(cadre1, text='H')
boutonB = Button(cadre1, text='B')
boutonG.pack(side=LEFT)  # là tout est positionné dans cadre1
boutonD.pack(side=LEFT)    # on peut faire ce que l'on veut de ce cadre
boutonH.pack()  # qui pour l'instant n'apparait pas
cadre.pack()
boutonB.pack()

can=Canvas(fen1, width=400, height=200, bg="white")
can.pack(side=LEFT)
can.create_oval(50,50, 60, 60, fill="cyan")

cadre1.pack()

boutonQuitter = Button(fen1, text='Quitter', command = fen1.quit)
boutonQuitter.pack(side=BOTTOM)
fen1.mainloop()
```

remarque:

Les programmes vus jusqu'à présent démarrait lors de la sortie du fichier mais on peut tout à fait faire des fonctions.

Par exemple si on veut simuler la fonction alert du javascript:
Il s'agit d'écrire une fonction qui ouvre une fenêtre, afficher un message dedans et se ferme quand l'utilisateur tape sur OK (et là un éventuel programme pourrait continuer à tourner).

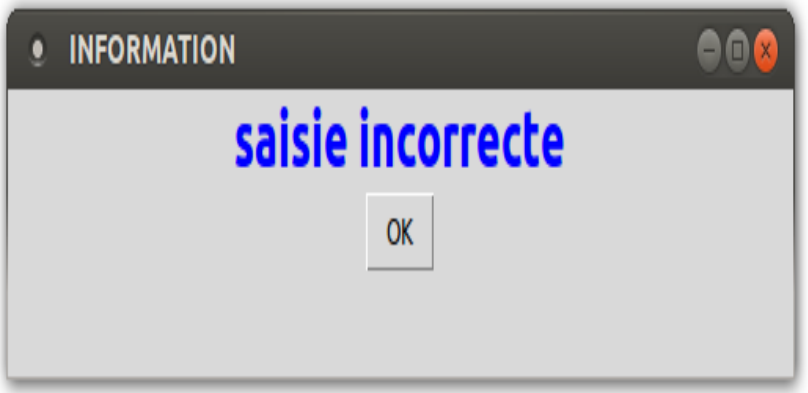
```
def alert(message):
    def monquitter():
        mafen.quit()
        mafen.destroy()

    mafen = Tk()
    mafen.title("INFORMATION")
    mafen.geometry("500x100")
    tex1 = Label(mafen, text=message, fg='blue', font=
                                ("Ubuntu",20, "bold"))
    bou1 = Button(mafen, text='OK', command = monquitter)
    tex1.pack()
    bou1.pack()
    mafen.mainloop()
```

```
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>>
>>> alert (" attention ")
```

On peut alors imaginer une alerte lors d'une saisie pour des réponses incorrectes

```
def saisienb():
    n=input("Entrez un entier compris entre 0 et 20 ou -1 pour
arrêter" )
    if n in ["-1"]+[str(i) for i in range(21)]:
        return(int(n))
    else:
        alert("saisie incorrecte")
        return saisienb()

def saisie():
    l=[ ]
    n=saisienb()
    while n!=-1:
        l+=[n]
        n=saisienb()
    return l
    mafen.mainloop()
```

```
>>>
>>>
>>>
>>>
>>> saisie()
Entrez un entier compris entre 0 et 20 ou -1 pour arrêter5
Entrez un entier compris entre 0 et 20 ou -1 pour arrêter6
Entrez un entier compris entre 0 et 20 ou -1 pour arrêter9
Entrez un entier compris entre 0 et 20 ou -1 pour arrêter15
Entrez un entier compris entre 0 et 20 ou -1 pour arrêter22
```



On a là une première base (première semaine Tkinter pour les L1 infos), on va maintenant compléter....

quels compléments?

→ gestion des évènements: on veut pouvoir déclencher une commande sans forcément cliquer sur un bouton (par exemple click de souris, touche return....)

→ de nouveaux widgets: la Listbox , un ascenseur si on le souhaite, des boutons “radios”

→ faire apparaître et disparaître des widgets à volonté ou les désactiver

→ travailler avec une grille et d'autres exemples

→ quelques fonctions ou fonctionnalités utiles....

gestion du click “souris”

On veut récupérer les coordonnées d'un click de souris dans un canvas

On a besoin:

- de la fenêtre principale
- d'un canvas où on va cliquer
- d'une zone texte où s'afficheront les coordonnées du click
- d'un bouton pour quitter

les commandes seront:

- quitter l'application
- afficher les coordonnées lors d'un click

d'où le programme:

```
from tkinter import *    # a jouter def monquitter():  comme d'habitude

def donne_position(event):
    texto.delete("0.0",END)                    # on efface l'écriture précédente
    texto.insert(END,"clic detecte en x="+str(event.x)
        + "   et y = " + str(event.y))
# event.x et event.y donnent les coordonnées du click

fen = Tk()
cadre=Canvas(fen,width=400, height=200, bg="light blue")
cadre.grid()

cadre.bind("<Button-1>", donne_position)

texto=Text (fen,height=1)    # zone de réponse
texto.grid()
b=Button(fen,text="quit",command=monquitter)
b.grid()
fen.mainloop()
```

Les nouveautés:

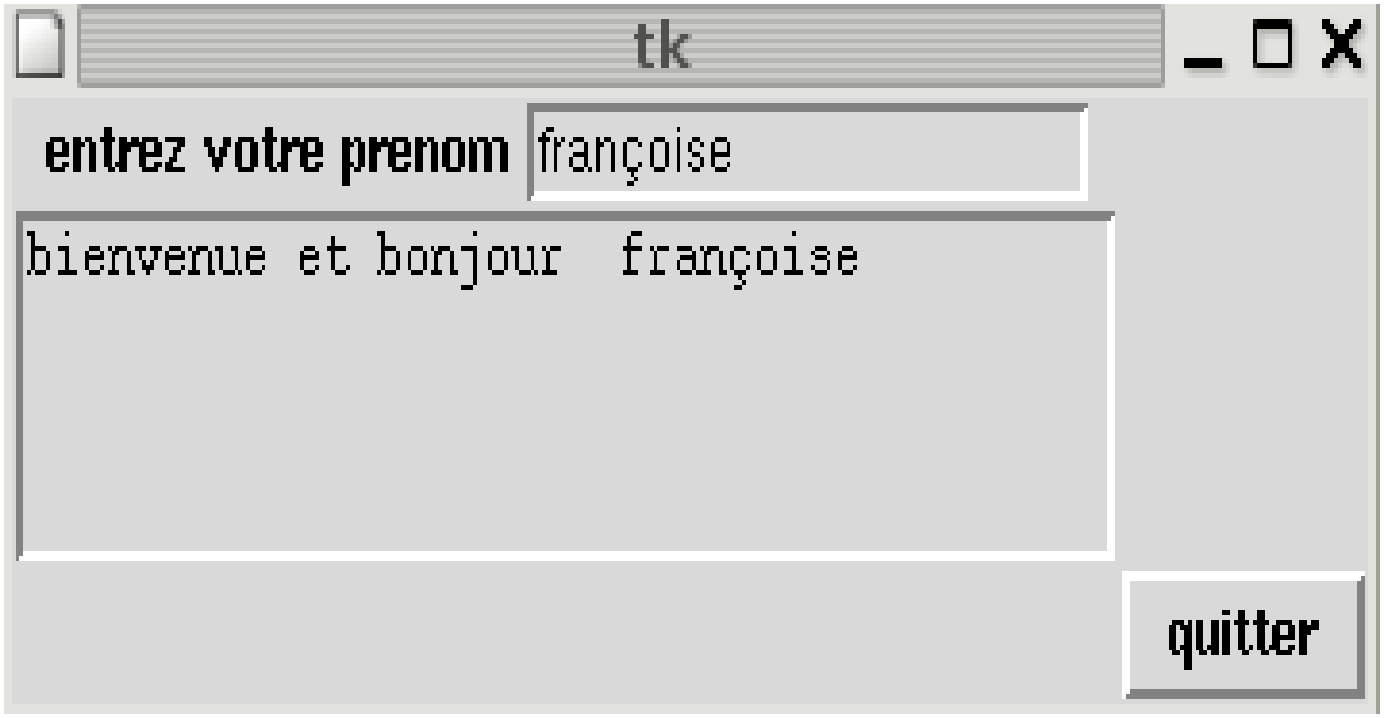
cadre.bind("<Button-1>", donne_position)

associe la commande donne_position à l'évènement “cliquer sur le bouton de la souris dans le widget cadre”

Il y a toujours un paramètre (**event**) à cette fonction même si on ne s'en sert pas. Ici on s'en sert pour récupérer les coordonnées du click, coordonnées qui sont données par event.x et event.y

```
def donne_position(event):
    texto.delete("0.0",END)  # effacer ce qui est écrit
    texto.insert(END,"clic detecte en x="+str(event.x)+ "   et y
    = " + str(event.y))
```

gestion d'un return dans un widget entry



On veut qu'un retour chariot après l'écriture du prénom, déclenche l'écriture “bienvenue et bonjour prénom” dans la zone de dialogue

Ici, on va utiliser un frame pour regrouper les 2 widgets du haut.

on a une fenêtre

```
fen = Tk()
```

dans laquelle on définit un cadre

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

et dans le cadre on met un label et une entry

```
lab_ent=Label(cadre,text="entrez votre prenom")
lab_ent.grid (row=0,column=0)

entree=Entry(cadre)
entree.grid(row=0,column=1)

# row et column donnent ici les positions dans le cadre
```

On complète les widgets:

```
txt=Text(fen, height=5,width=40)
txt.grid(row=1,column= 0)

b2=Button(fen,text="quitter",command=monquitter)
b2.grid(row=2,column =1)

# ici row et column donnent les positions dans la fenêtre puisqu'il s'agit du bouton qui est dans la fenêtre
```

Remarque: ici, on pourrait utiliser columnspan vu au cours précédent pour étirer la zone du dessous sur les deux widgets du haut à la place du frame

Et on complète avec les fonctions , le lancement et les “liaisons”:

```
def monquitter():
    fen.quit()
    fen.destroy()

def affiche(event):
    p=entree.get() # pour récupérer de qui a été saisi (une string)
    txt.insert(END,"bienvenue et bonjour  “+p)

# la liaison
entree.bind("<Return>", affiche)


fen.mainloop()
```

Les nouveautés:

entree.bind("<Return>",affiche)
Fait le lien entre la commande affiche et le retour chariot dans le widget Entry nommé entree

La fonction affiche a obligatoirement un paramètre event même si on ne s'en sert pas, comme ici.

```
def affiche(event):
    p=entree.get()
    txt.insert(END,"bienvenue et bonjour  " +p)
```

On peut aussi utiliser le %s pour gérer les insertions de variables dans les écritures.

Application: jeu du nombre mystérieux

Il était un peu pénible d'appuyer sur OK à chaque proposition....

On peut :

- soit supprimer le bouton OK

- soit le garder et de pouvoir à la fois appuyer sur return et sur le bouton OK

Dans les deux cas il suffit de relier dans la zone de saisie une fonction de jeu et return

entree.bind(“<Return>”, jouer2)

Attention: on ne peut pas directement utiliser la même fonction jouer pour le bouton et pour le return dans la zone de saisie.

Pourquoi?

Parce qu'une commande de bouton n'a pas de paramètre alors qu'une commande associée à return en a un.....

Donc on duplique.....et on rajoute un paramètre qui ne sert pas dans la fonction qui sera reliée à return....

On aura donc
→ la fonction jouer() qui est la commande du bouton inchangée

```
def jouer():
    ......
```

→ la fonction jouer2(event) qui a exactement le même corps donc on peut dupliquer ou écrire

```
def jouer2(event):
    return jouer()
```

→ et surtout on n'oublie pas la liaison:

```
entree.bind("<Return>", jouer2)
```

Quelques autres évènements possibles:

<KeyPress> : détecte que l'utilisateur a tapé sur une touche

<KeyPress-a> : détecte que l'utilisateur a tapé sur la touche a (minuscule)

<KeyPress-A> : détecte que l'utilisateur a tapé sur la touche a (majuscule)

<Escape>: détecte la touche d'échappement

<Enter> : détecte l'entrée de la souris dans un widget

<Leave> : détecte que la souris quitte un widget

<Double-Button-1>: détecte un double click sur le bouton1 de la souris

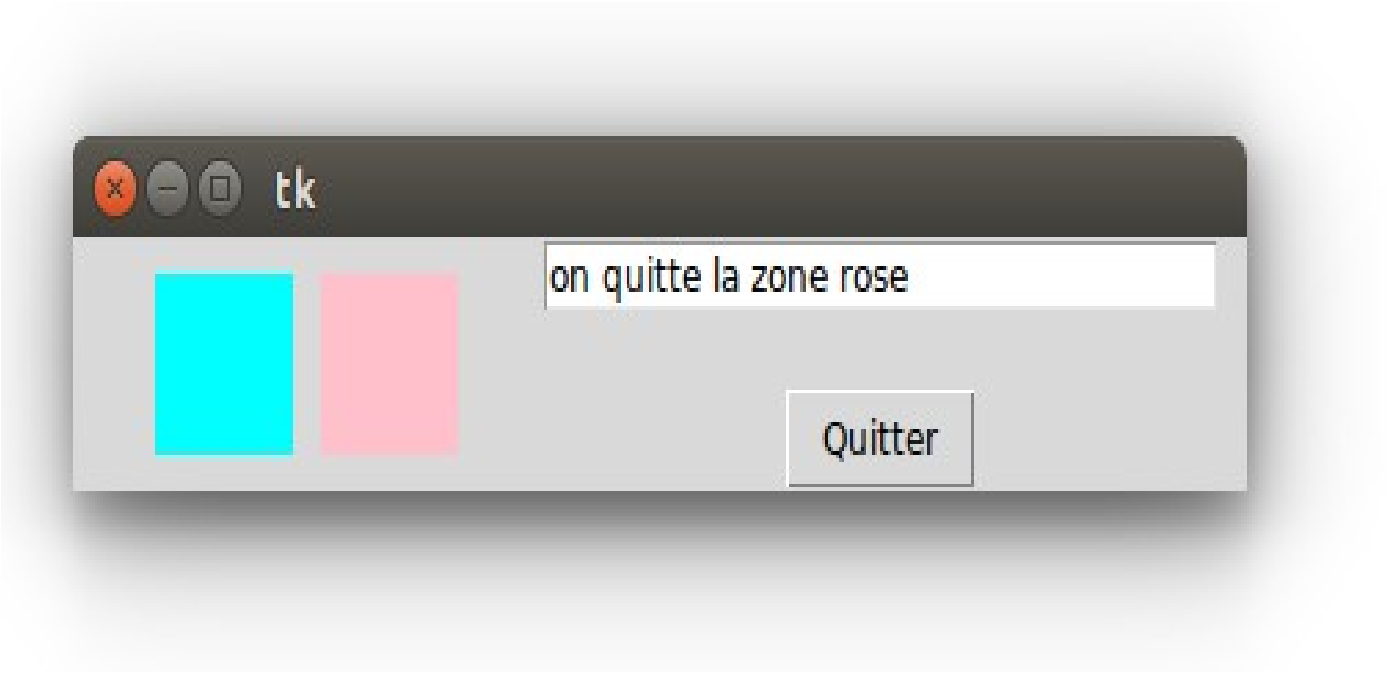
<B1-Motion>: détecte que le bouton reste enfoncé (pour le drag and drop)

Ce qui va nous permettre de nouveaux exemples:

→ signaler une entrée dans une zone ou une sortie

→ bouger un carré avec des touches

→ bouger un carré avec du drag and drop



L'entrée et la sortie de la souris dans les zones roses et bleues sont signalées

```
fen1 = Tk()


can=Canvas(fen1)
can.pack(side=LEFT, padx=20, pady=10)
zone1=Frame(can, width=50, height=50,bg="cyan")
zone2=Frame(can, width=50, height=50,bg="pink")
zone1.pack(side=LEFT, padx=10)
zone2.pack(side=LEFT)

texto=Entry(fen1, width=30)
texto.pack(padx=10)



boutonQuitter = Button(fen1, text='Quitter', command =
monquitter)
boutonQuitter.pack(side=BOTTOM)
```

```
def entree1(event):
    texto.delete(0,END)
    texto.insert(END,"survol de la zone bleue")
def sortie1(event):
    texto.delete(0,END)
    texto.insert(END,"on quitte la zone bleue")

def entree2(event):
    texto.delete(0,END)
    texto.insert(END,"survol de la zone rose")
def sortie2(event):
    texto.delete(0,END)
    texto.insert(END,"on quitte la zone rose")



zone1.bind("<Enter>",entree1)
zone1.bind("<Leave>",sortie1)
zone2.bind("<Enter>",entree2)
zone2.bind("<Leave>",sortie2)
```

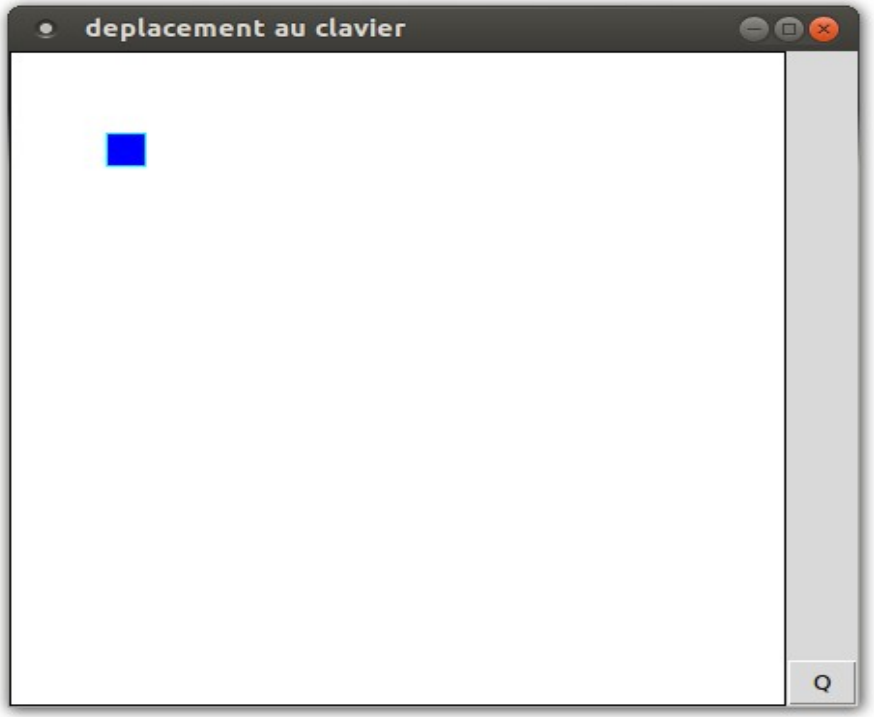
Bouger un carré avec un pavé de touches

Par exemple ici

Z

Q S

W



```
from tkinter import *

X,Y=20,20   # position de départ du carré

def clavier():  # à écrire plus tard
    return ''

fen=Tk()
fen.title("deplacement au clavier")

can=Canvas(fen,width=400, height=400,bg="white")
can.pack(side=LEFT)
carre=can.create_rectangle(X-10,Y-10,X+10,Y+10, fill="blue", outline='cyan')

can.bind('<KeyPress>', clavier)  # la liaison
can.focus_set()  # pour que cela se passe dans le canvas

bquitter=Button(fen, text='Q', command=monquitter)
bquitter.pack(side=BOTTOM)
fen.mainloop()
```

```
def clavier(event):
    global X,Y
    touche=event.keysym   # pour récupérer la touche
   # print(touche)   si vous voulez connaître le code d'une touche
    if touche=='z':
        Y-=10
    elif touche=='q':
        Y+=10
    elif touche == 'w':
        X+= 10
    elif touche == 's':
        X-=10
    can.coords(carre, X-10,Y-10,X+10,Y+10)  # modifie la position du
                    # carré en modifiant les coordonnées
```

Si on veut utiliser les touches de direction il suffit de remplacer le nom des touches:

```
def clavier(event):
    global X,Y
    touche=event.keysym
    if touche=='Up':
        Y-=10
    elif touche=='Down':
        Y+=10
    elif touche == 'Right':
        X+= 10
    elif touche == 'Left':
        X-=10
    can.coords(carre, X-10,Y-10,X+10,Y+10)
```

Un peu de drag and drop:

On prend la même interface mais on veut déplacer le carré en “drag and drop”

On a exactement les mêmes widgets

Première version très simple:

```
def drag(event):
    X,Y=event.x, event.y
    can.coords(carre, X-10, Y-10, X+10, Y+10)
# on modifie les coordonnées du carré en fonction de la position
X,Y=20,20
fen=Tk()
fen.title("drag and drop")

can=Canvas(fen,width=400, height=400,bg="white")
can.pack(side=LEFT)

carre=can.create_rectangle(X-10, Y-10, X+10, Y+10, fill="blue", outline='cyan')

can.bind('<B1-Motion>', drag)  # liaison

bquitter=Button(fen, text='Q', command=monquitter)
bquitter.pack(side=BOTTOM)
fen.mainloop()
```

Inconvénients:

→ le carré peut sortir de l'écran
→ le déplacement se déclenche même si on n'est pas sur le carré (qui rejoint alors la souris)
(voir démo)

Pour y remédier on va faire une deuxième liaison qui gèrera une variable globale;
Cette variable ne sera à True que si on commence par cliquer sur le carré

```
detect=False

def drag(event):
    X,Y=event.x, event.y
    if detect:
        can.coords(carre, X-10, Y-10, X+10, Y+10)

def clic(event):  # si on commence par cliquer dans le carré
    global detect
    X,Y=event.x, event.y
    [xmin,ymin,xmax,ymax]=can.coords(carre)
    if xmin<=X<=xmax and ymin<=Y<=ymax :
        detect=True
    else:
        detect=False

can.bind('<Button-1>', clic)
can.bind('<B1-Motion>', drag)
```

Si on veut éviter que le carré sorte on rajoute des tests:

```
def drag(event):
    X,Y=event.x, event.y
    if detect:
        if X<0:
            X=0
        if X>400:
            X=400
        if Y<0:
            Y=0
        if Y>400:
          Y=400
        can.coords(carre, X-10, Y-10, X+10, Y+10)
```

Comment empêcher de jouer “n'importe quand”

Par exemple dans le jeu du nombre mystérieux, on voudrait que quand le joueur a trouvé (ou quand il a demandé la solution) il ne puisse plus continuer à jouer sans faire une nouvelle partie. Comment faire?

De nombreuses solutions sont possibles:

→ supprimer le bouton OK (voire carrément toute la ligne)

→ rendre le bouton OK (ou toute la ligne) inactif (il apparaîtra à l'écran mais grisé)

→ faire un test dans la fonction de jeu pour qu'elle ne fasse plus rien

→ changer la commande de bouton pour que OK fasse une remise à zéro

→ supprimer le bouton OK (voire carrément toute la ligne)

Il s'agit simplement d'utiliser truc.grid_forget()
En n'oubliant pas de refaire les grid avec les bons paramètres dans la fonction de remise à zéro

→ rendre le bouton OK (ou toute la ligne) inactif (il apparaîtra à l'écran mais grisé)

On re-configure le bouton en changeant son statut

boutonok.configure(state=DISABLED)

Et dans la fonction nouvelle partie (ou remise à 0)

boutonok.configure(state=ACTIVE)

→faire un test dans la fonction de jeu pour qu'elle ne fasse plus rien: on rajoute une variable globale

```
fin=0

def jouer():
   if not fin:
            global cp
            cp=cp+1
            ch=entree.get()
            entree.delete(0,END)
            nb=int(ch)
             if nb==x:
                  global fin
                  fin=1
                  larep.insert(END, "%s bravo vous gagnez en %s coups \n" %
(nb,cp))
              elif nb<x:
                   larep.insert(END, "%s :  trop petit \n" %nb)
            else:
                   larep.insert(END, "%s  :  trop grand\n" %nb)
```

o

Il ne faut pas oublier de ré-initialiser fin dans la fonction de mise à niveau. Et il faut aussi mettre fin à 1 si on a demandé la réponse

```
def paresseux():
    global fin
    fin=1
    larep.insert(END, " paresseux!!! il fallait trouver” + str(x)+ “ \n")
```

→ changer la commande de bouton pour que OK fasse une remise à zéro

Si on a écrit la fonction remise à zéro : new

On peut au moment où on gagne reconfigurer:

Boutonok.configure(command=new)

Mais on peut aussi mettre une fonction qui ne fait rien

```
def rien():
    return ''
```

Et dans reset il faut remettre la configuration originale:

Boutonok.configure(command=ok)

Remarque: une simplification de certaines écritures

Pour alléger un peu un programme Tkinter, on peut, pour les widgets dont on n'a pas besoin du nom dans le programme, ne pas leur donner de nom et adopter une écriture en une seule ligne du style:

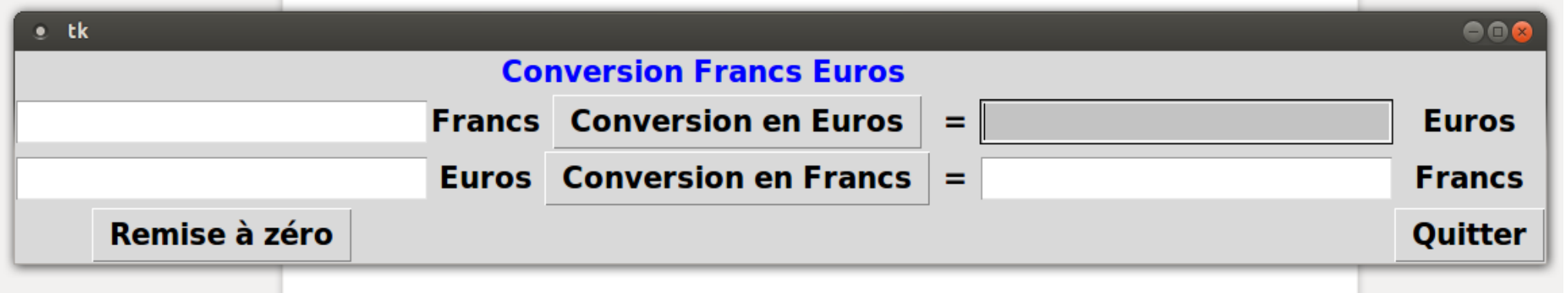
création du widget. positionnement

au lieu des deux lignes

machin= creation du widget

positionnement de machin

Par exemple dans l'exercice de la conversion francs/euros: il y a beaucoup de labels “fixes”



Les labels “conversion francs euros”, “Francs”, “Euros” ne changent jamais

Les noms des boutons de conversion, de remise à zéros et quitter ne servent pas

On a par contre besoin du nom des zones de saisie pour récupérer la valeur du contenu ou du nom des zones textes pour écrire le résultat.

Reprenons le programme: (uniquement la partie de définition des widgets et de positionnement):

```
#Definition des widget et affichage des widgets

titre=Label(fen,text="Conversion Francs Euros")

titre.grid(column=0,row=0,columnspan=5)
```

Comme le widget titre n'est pas ré-utilisé ou modifié, on peut écrire ces deux lignes en une seule (mais du coup on ne pourra plus le re-configurer pendant l'exécution du programme):

```
Label(fen,text="Conversion Francs Euros").grid (column=0,
row=0,columnspan=5)
#écriture en une seule ligne
```

On ne peut pas faire la même chose pour les zones de saisies puisque l'on aura besoin qu'elles aient un nom pour récupérer le contenu!!!

```
# ligne 1 conversion des F vers les Euros
entF=Entry(fen,font=ecriture)
entF.grid(column=0,row=1)

def ConvEuro() :
   ch=entF.get()  # utilisation de entF
   Franc=eval(ch)
   Euro=Franc/taux
   resEuro.insert(END,str(Euro))
```

Dans cet exemple , on peut simplifier l'écriture pour tous les labels et les boutons mais ni pour les Entry (saisie et effacement) ni pour les zones textes (pour les effacer, ou écrire)

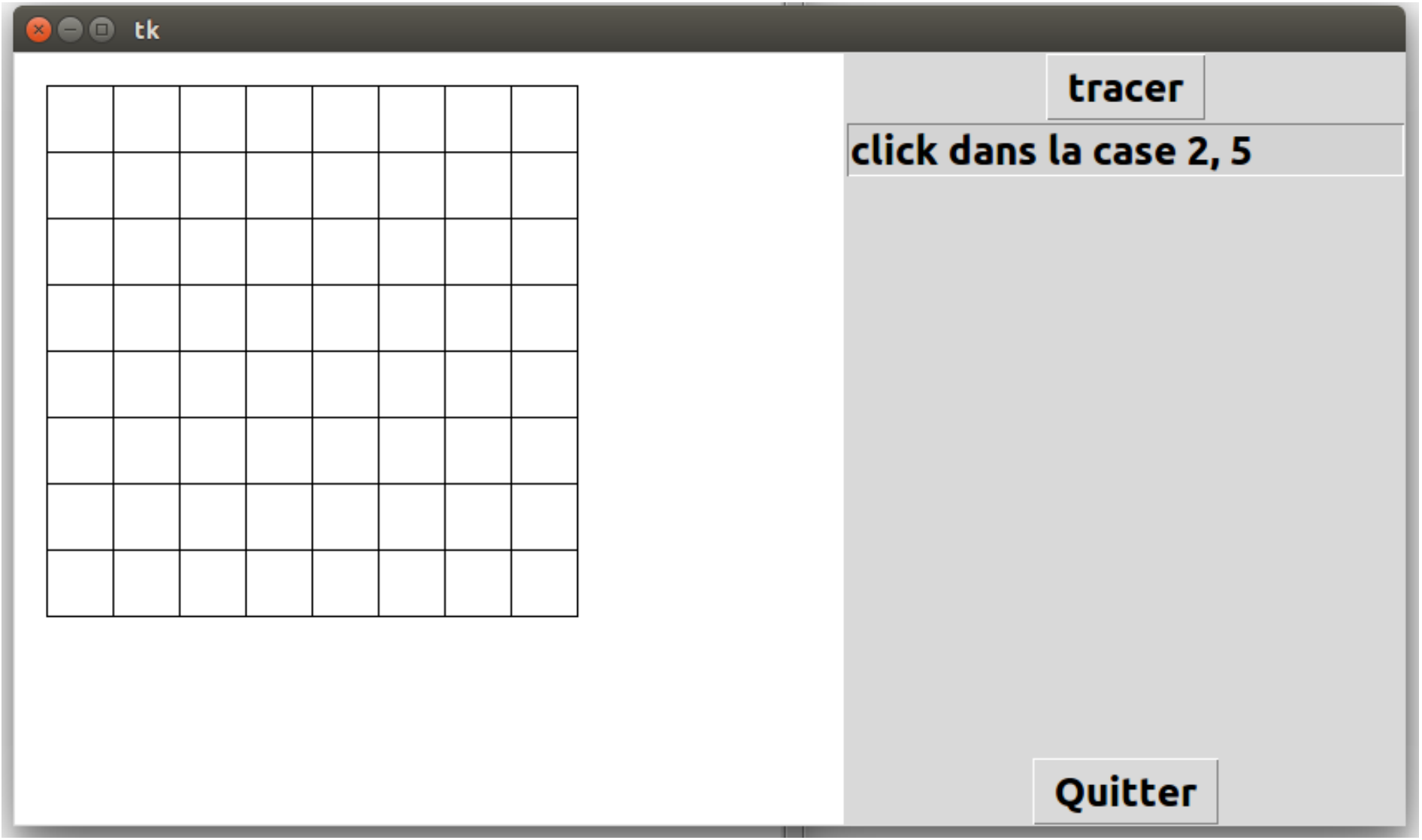
Attention aux écritures suivantes

entF=Entry(fen,font=ecriture).grid(column=0,row=1)

C'est faux car entF prend la valeur None donc le programmme ne pourra pas marcher

Important: travailler avec une grille

On veut pouvoir dessiner une grille, cliquer dans une case et savoir de quelle case il s'agit.... c'est très utile dans de nombreux jeux ou applications.....



Il nous faut des widgets: un canvas, 2 boutons, une zone d'écriture

```
from tkinter import *
dessin=Tk()


can= Canvas(dessin,height=500,width=500,bg="white")
can.pack(side=LEFT)

b2=Button(dessin,text="tracer",command=grille)
b2.pack(side=TOP)
b1=Button(dessin,text="Quitter",command=monquitter)
b1.pack(side=BOTTOM)



t=Text(dessin ,height=3,width=30,bg="light grey")
t.pack(side=TOP)


dessin.mainloop()
```

Il faut pouvoir dessiner une grille. Nous allons le faire de façon générale pour une grille carrée avec un nombre quelconque de cases (donc avec des paramètres à ajuster pour les différentes utilisations)

Ces paramètres seront: **nb, c, x0,y0**
- le nombre de cases par ligne ou colonne (ici nb= 9)
- la taille d'une case carrée (ici c=40)
- les coordonnées du point en haut à gauche de la grille (ici x0,y0=20,20)

```
def grille():
    for i in range(nb+1):
        can.create_line(x0+c*i, y0,x0+c*i,y0 + nb*c)
        can.create_line(x0, y0+c*i,x0+nb*c ,y0+c*i)
```

Il faut pouvoir faire correspondre le click et la case:

```
def correspond(x,y):
    return (y-y0)//c,(x-x0)//c  # on se sert de la division entière ici
```

Attention à ne pas inverser i et j ou x et y !!

On peut alors écrire la fonction déclenchée par l'évènement cliquer dans la grille:

```
def ecrire(event):
    t.delete("0.0",END)
    (i,j)=correspond(event.x,event.y)
    t.insert(END,"click dans la case " + str(i) + ", " + str(j))
```

et la liaison:

```
can.bind("<Button-1>", ecrire)
```

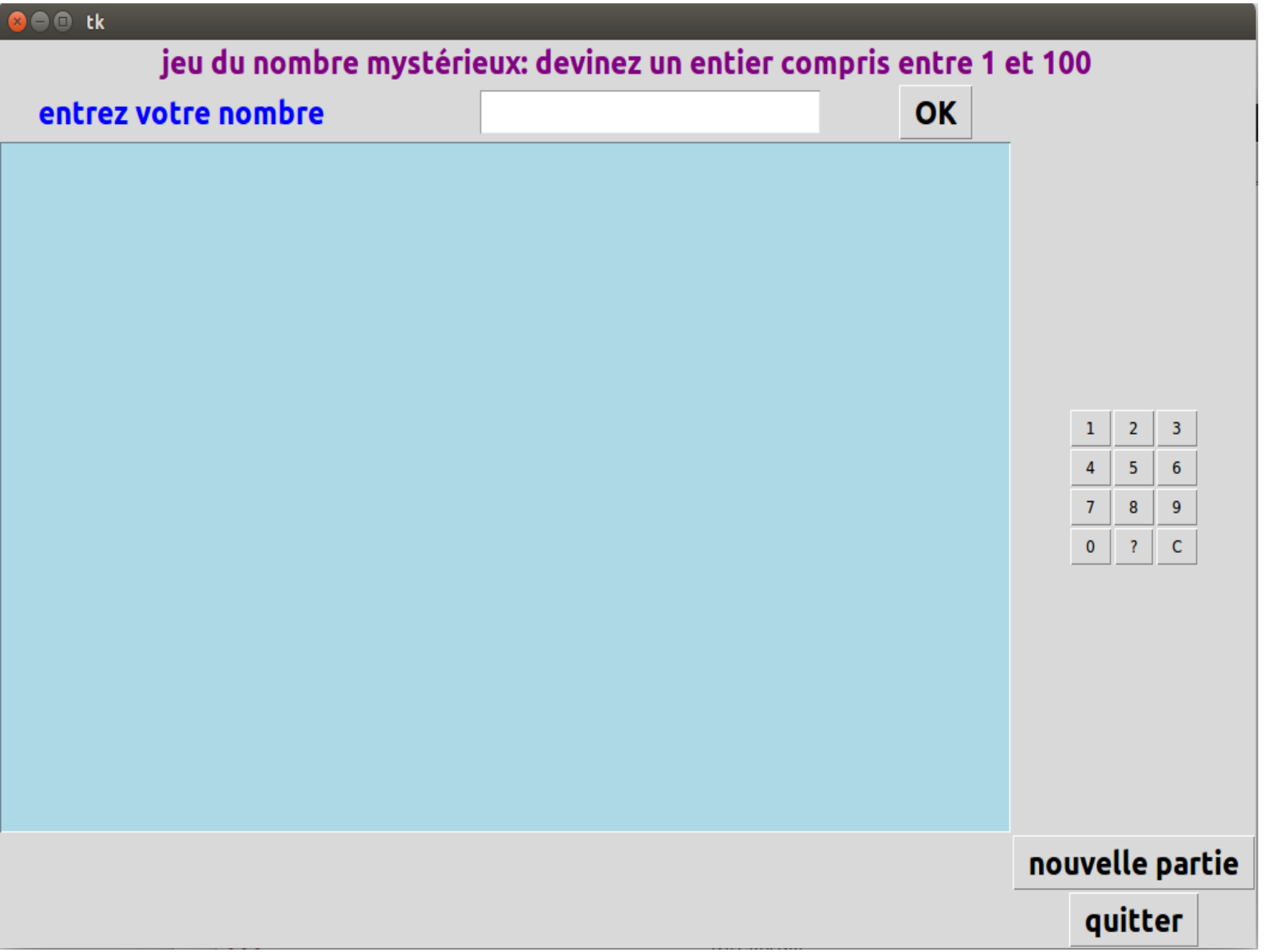
et évidemment il ne faut pas oublier la fonction monquitter

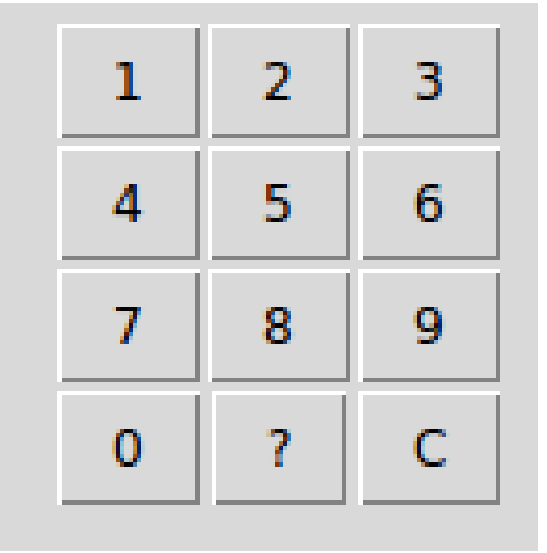
Dans cette version il n'y avait pas de tests , on pouvait cliquer en dehors de la grille et avoir donc des numéros de cases qui n'existent pas vraiment (ce qui finira par causer des erreurs);
il suffit de rajouter un test pour y remédier:

```
def ecrire(event):
    t.delete("0.0",END)
    (i,j)=correspond(event.x,event.y)
    if i in range(nb) and j in range (nb):
        t.insert(END,"click dans la case " + str(i) + ", " + str(j))
    else:
        t.insert(END, "click en dehors de la grille")
```

Le “else” et ce qui suit n'étant pas obligatoire

Créer un pavé numérique du jeu du nombre mystérieux





Si on décide de mettre 12 boutons bien positionnés dans 4 lignes d'un cadre

```
cadre=Frame(fen)
cadre.grid(column=7, row=2)
```

```
B7=Button(cadre, text="7")
B7.grid(column=1, row=3)
B8=Button(cadre, text="8")
B8.grid(column=2, row=3)
B9=Button(cadre, text="9")
B9.grid(column=3, row=3)
B4=Button(cadre, text="4")
B4.grid(column=1, row=2)
B5=Button(cadre, text="5")
B5.grid(column=2, row=2)
B6=Button(cadre, text="9")
B6.grid(column=3, row=2)
B1=Button(cadre, text="1")
B1.grid(column=1, row=1)
B2=Button(cadre, text="2")
B2.grid(column=2, row=1)
B3=Button(cadre, text="3")
B3.grid(column=3, row=1)
B0=Button(cadre, text="0")
B0.grid(column=1, row=4)
B0=Button(cadre, text="?",command=paresseux)
B0.grid(column=2, row=4)
B0=Button(cadre, text="C",command=efface)
B0.grid(column=3, row=4)
```

C'est lourd à écrire !

Problème: il reste les commandes de bouton à écrire.
Pour les 10 chiffres cette commande est globalement la même: écrire le chiffre correspondant dans la zone....

On peut bien entendu écrire 10 fonctions avec du copier/coller style:

```
def commandeb1():
        entree.insert(END, str(1))
```

mais on peut faire autrement que de dupliquer cette fonction.

On peut éventuellement utiliser les “lambdas “

De façon générale, lambda permet de définir des fonctions sans utiliser def

Exemples:

```
>>> lambda x: x*x
<function <lambda> at 0x2c03e20>
```

Une fonction est bien définie mais on ne peut pas s'en resservir .
Si on lui donne un nom:

```
>>> g=lambda x: x*x
>>> g
<function <lambda> at 0x2bc9d98>
>>> g(4)
16
```

Par contre si on écrit une fonction du style

```
def applique(f,l):
    return[f(x) for x in l]
```

On peut définir f lors de l'appel avec une lambda

```
>>> applique(lambda x:x*x, [6,7,2])
[36, 49, 4]
```

Dans notre exemple on peut écrire pour chaque bouton:

```
B7=Button(cadre1, text="7", command=lambda:
entree.insert(END, str(7))
```

Ou alors définir une fonction:

```
def f(i):
   return  lambda: entree.insert(END, str(i) )
```

À appeler pour chaque bouton

```
B7=Button(cadre1, text="7", command=f(7))
```

Mais encore mieux, comme on n'a pas besoin des noms des boutons on peut faire une boucle pour créer les boutons et leur attribuer une commande:

```
for i in range(0,12):
    k=i//3
    j=i%3
    i=i+1
    if i==10:  # 3 cas particuliers
        Button(cadre, text=str(0), command=lambda :
entree.insert(END,str(0))).grid(column=j, row=k )
    elif i==11:
        Button(cadre, text="?",
command=paresseux).grid(column=j, row=k )
    elif i==12:
        Button(cadre, text="C",
command=efface).grid(column=j, row=k )
```

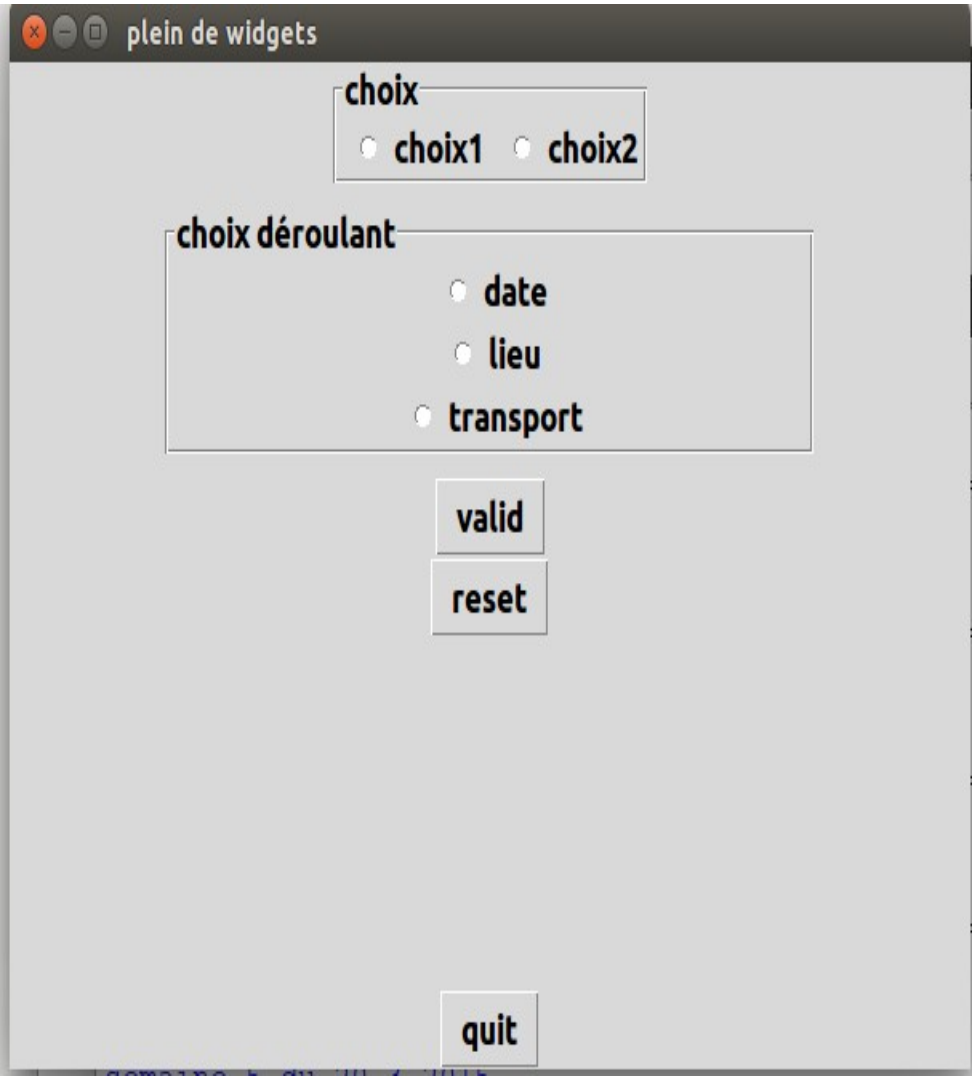
Et le cas général:

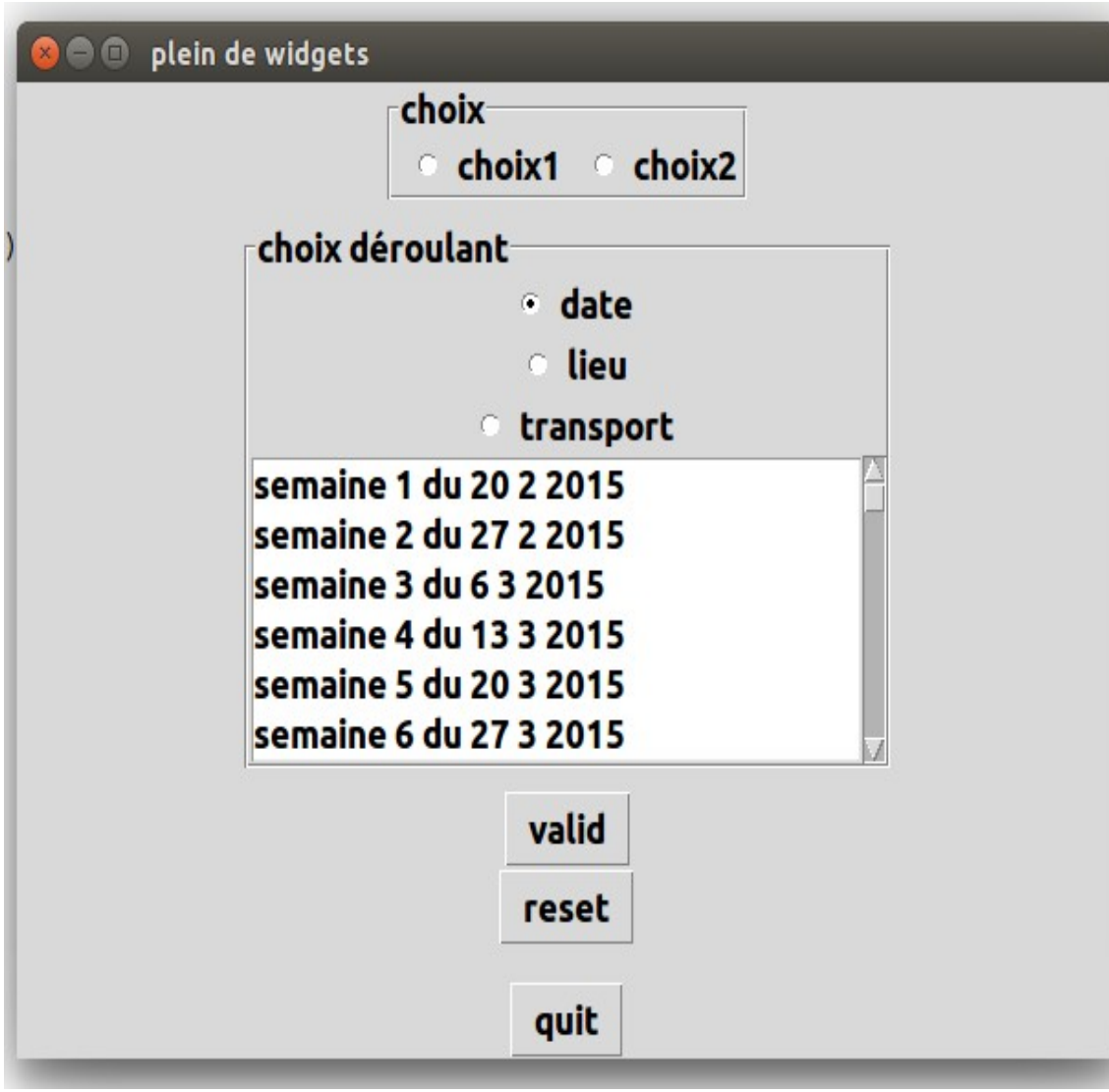
```
   else:
      Button(cadre, text=str(i), command=f(i)).grid(column=j,
row=k )
```

On peut bien entendu remplacer command=f(i) par:

```
command=lambda: entree.insert(END, str(i))
```

De nouveaux widgets pour une saisie





Encore de nouveaux widgets:

→ un cadre-avec-label

→ des boutons radios

→ une liste déroulante avec un ascenseur

Un widget Radiobutton dans un “cadre avec titre”:

```
# labelFrame permet de définir un cadre avec titre
C1=LabelFrame(fen, text="choix",  width=200)
C1.pack(padx=5)
#padx=5  permet un espacement


# on définit une variable dans laquelle sera mise la valeur
de
# ce que l'on va cocher

val1=IntVar()
bc1=Radiobutton(C1, text="choix1", variable=val1,value=1)
bc2=Radiobutton(C1,text="choix2",variable=val1, value=2)
bc1.pack(side=LEFT)
bc2.pack(side=LEFT)
```

```
# choix déroulant
C2=LabelFrame(fen, text="choix déroulant",width=200)
C2.pack(pady=10)


# un cadre pour les trois boutons
cadre=Frame(C2)


# les 3 boutons à cocher et la variable associée
val2=IntVar()
bc21=Radiobutton(cadre, text="date",
variable=val2,value=1)
bc22=Radiobutton(cadre,text="lieu",variable=val2 ,value=2)
bc23=Radiobutton(cadre,text="transport",variable=val2 ,val
ue=3)

bc21.pack()
bc22.pack()
```

On prévoit un cadre où seront affichées les listes déroulantes

```
cadrelistes=Frame(C2, width=400)
cadrelistes.pack(side=LEFT)
```

Il faut maintenant prévoir les listes déroulantes (qui ne s'afficheront que lorsqu'on clique sur un des boutons radios).

Il y en a 2 qui sont fixes et une qui dépend de la date.

Les deux listes déroulantes “simples”

```
# la liste pour les lieux
# on définit une liste correspondant à l'affichage
l2=["Rome", "Barcelone",
"Prague","Lisbonne","Amsterdam"]

# puis le widget listbox
liste2=Listbox(cadrelistes,width=15,height=len(l2))

# on met les éléments de la liste dans la listbox
for element in l2:
  liste2.insert(END, element)
```

Il faudra positionner la listebox quand on clique sur le bouton correspondant
Et il faut qu'un choix d'une ligne déclenche une action

```
liste2.bind('<ButtonRelease-1>',fonction2)
```

Avec comme fonction

```
def fonction2(event):
    global index
    index=int(liste2.curselection()[0])
```

On aura une variable globale index qui donnera la ligne sélectionnée

```
# variable index pour récuperer quelle ligne a été
# selectionnée
index=0
```

lbox.curselection()[0]: va renvoyer la string correspondant au numéro de la ligne sélectionnée (lignes numérotées à partir de 0) (dont on met int pour récupérer la valeur)

Il faut aussi une liaison entre le bouton radio et une fonction:

```
bc22.bind('<Button-1>', radio2)
```

```
def radio2(event):
    cadrelistes.pack(side=LEFT)
    liste2.pack(side=RIGHT)
    liste1.pack_forget()
    s1.pack_forget()  # ascenseur
    liste3.pack_forget()
```

La fonction consiste en gros à positionner la listebox correspondant au choix et à enlever les autres listbox (et l'ascenseur)

Pour la liste avec les semaines: il faut d'abord fabriquer cette liste à partir de la date du jour

```
debut=datetime.now() # depart avec la date du jour
fin=debut+timedelta( days=365, hours=0, minutes=0, seconds=0) # date de fin 1 an après

def extraire(date):
    return str(date.day)+" "+ str(date.month) + " "+ str(date.year)

def fabrique_liste():
    res, dat=[ ],debut
    i=1
    while dat<fin:
        res+= ["semaine "+ str(i) + " du " + extraire(dat)]
        dat=dat+timedelta(days=7, hours=0, minutes=0, seconds=0)
        i+=1
    return res
```

Ensuite on définit les widgets

```
# la liste déroulante pour les semaines avec scrollbar

l1=fabrique_liste()   #fabrication automatique de la liste
liste1=Listbox(cadrelistes,width=30, height=6)

#positionnements des éléments de la liste
for element in l1:
  liste1.insert(END, element)

# définition de l'ascenseur
s1 = Scrollbar(cadrelistes)
s1.config(command = liste1.yview)
liste1.config(yscrollcommand = s1.set)
```

```
# la liaison clic dans la liste /fonction
liste1.bind('<ButtonRelease-1>',truc1)

def truc1(event):
    global index
    index=int(liste1.curselection()[0])

# la liaison bouton/liste : cocher le bouton doit afficher la zone et enlever les autres affichages

bc41.bind('<Button-1>',radio1)

def radio1(event):
    cadrelistes.pack(side=LEFT)
    liste1.pack(side = LEFT, fill = Y)
    s1.pack(side = RIGHT, fill = Y)
    liste2.pack_forget()
    liste3.pack_forget()
```

On a rajouté des boutons valid et reset pour récupérer les valeurs choisies et pour ré-initialiser.

```
# le bouton de validation qui lancera les calculs puis les affichages
bok=Button(fen,text="valid", command=ok)
bok.pack()

# et un bouton de remise à 0
binit=Button(fen,text="reset", command=reinit)
binit.pack()
```

```
def ok():
    if val1.get()==1:
        print("choix1")
    else:
        print("choix2")
    if val2.get()==1:
       print(l1[index])
    elif  val2.get()==2:
        print(l2[index])
    else:
        print(l3[index])
```

La fonctiondu bouton valide: là on fait des print pour montrer comment récupérer les valeurs
Sinon c'est cette fonction qui lancerait le traitement des données.

La fonctiondu bouton reset : il faut tout remettre à zéro

```
def reinit():
    bc1.deselect()
    bc2.deselect()
    bc21.deselect()
    bc22.deselect()
    bc23.deselect()
    liste1.pack_forget()
    liste2.pack_forget()
    liste3.pack_forget()
    s1.pack_forget()
```

Remarque: dans l'exemple de la capture il a été rajouté pour toutes les écritures, une fonte (voir code sous moodle)

Fin pour ce cours .....

mais Tkinter a de nombreuses possibilités (voir les doc et tutoriels.....).

Rien que pour nos simples boutons on pourrait:
- leur donner une taille (par défaut elle s'adapte au contenu: image ou zone texte par exemple)
- positionner l'écriture (ou l'image) sur le bouton
- lui donner des effets (style “enfoncé”, bord coloré, changement de couleur si la souris est dessus.....)
- déclarer un espace autour du bouton (avec des commandes style padx, pady...)

Pour les Entry on peut se mettre en mode mot de passe (écriture d'* par exemple) on peut empêcher l'écriture....

On peut aussi compléter Tkinter avec des modules

tkMessageBox pour des fenêtres de dialogue

tkSimpleDialog , tkFileDialog

tkchooserColor (pour des couleurs en rvb)