La GUI tkinter

Christian Nguyen

Département d'informatique
Université de Toulon et du Var

Tkinter

Tkinter est le module Python spécifiques aux interfaces graphiques (GUI) fourni par défaut.

Il dérive de Tk (ToolKit) une extension graphique du langage de script Tcl (Tool Command Language).

Autres bibliothèques : wxPython, pyQT, pyGTK, etc. On peut aussi utiliser les widgets Java ou les MFC de Windows.

Les composants graphiques (ou contrôles ou widgets [1]) correspondent à des classes d'objets dont il faudra étudier les attributs et les méthodes.

1. window gadget

Tkinter

La fonction `Tk()` produit à l'écran une petite fenêtre graphique vide, la *root window*. Exemple :

```
import tkinter  # Python 2.x : import Tkinter
tkinter.Tk()
tkinter.Button(text="Bienvenue !", command=exit).pack()
```

produit à l'écran la fenêtre suivante :

Un premier client X (l'aspect dépend du WM)

Tkinter

Syntaxe Python. Paramètres très nombreux mais valeur par défaut.

```
canv.create_line(x1, y1, x2, y2, fill=col, width=ep)
```

Cette méthode comporte de nombreuses options dont : `activewidth`, `arrow`, `capstyle`, `dash`, `disabledwidth`, `joinstyle`, `smooth`, `state`, `tags`, ...

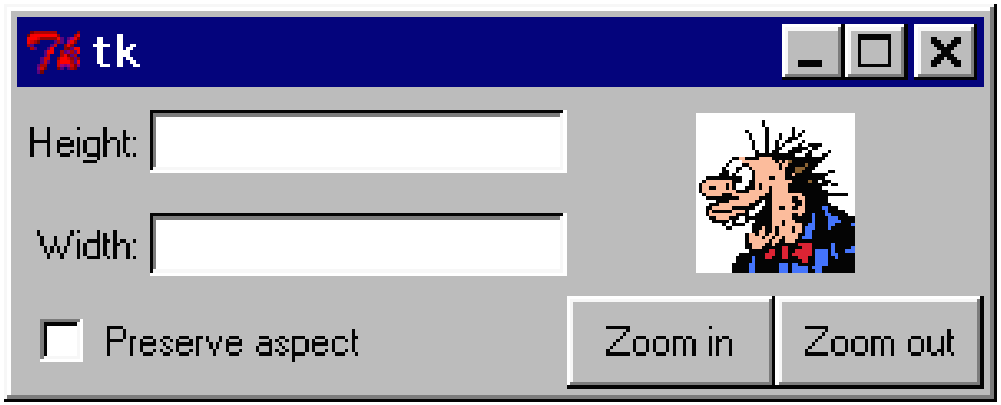
Instances de widgets organisées hiérarchiquement. Tout widget est fils de la fenêtre principale (*root window*).

```
root = Tk()  # fenetre principale
win1 = Toplevel(root)  # fenetre fille
canv1 = Canvas(win1, ...)  # widget graphique
```

Tkinter

Fenêtres principale (*root*) et indépendantes (*top-level*) gérée par le gestionnaire de fenêtres (*window manager* ou WM).

Le gestionnaire de widgets (*geometry manager*) prend en charge la taille, la position, la priorité et l'affichage des widgets (essentiellement *grid* ou *pack*).



Distinction entre fonctions d'initialisation et gestionnaires d'évènements (*binding*, *callback*, *listener*, ...).

Quatre grands groupes fonctionnels

- création (méthode correspondant à la classe) et destruction de widgets (`destroy`),
- visualisation des widgets grâce à un gestionnaire (`pack`, `place`, `grid` ou de la classe `Canvas`),
- communication avec un widget (`configure`, `cget`),
- inter-connexion : interaction (*binding*), partage du dispositif d'affichage (*display*), coopération (*scrollbar*).

  Cinq autres formes d'inter-connexion :
  - la selection (reprend le mécanisme fourni par X11),
  - la mire (*input focus*),
  - le gestionnaire de fenêtres (*window manager*),
  - les fenêtres modales (*grab*),
  - un mécanisme de communication inter-applications.

Organisation d'un programme

Structure d'un programme tkinter

```
# inclusion des modules
import tkinter as tk

# definition des fonctions internes
# definition des fonctions interactives (cf. binding)

# initialisation du système graphique
root = tk.Tk()

# initialisation des widgets
# binding
# placement et affichage des widgets

# boucle de scrutation
tk.mainloop() # root.mainloop() : associée à la fenêtre
principale

# liberation des ressources
exit(0)
```

Méthodes communes

Les widgets ont des comportements communs.

Configuration

- `w.config(option=value)` : modification d'un attribut,
- `value = w.cget("option")` : récupération d'un attribut,
- `k = w.keys()` : dictionnaire des attributs.

Gestion des évènements

- `w.wait_variable(var)` : attente *locale* de modification,
- `w.wait_visibility(window)` : attente *locale* d'apparition,
- `w.wait_window(window)` : attente *locale* de destruction,
- `w.update()` : traitement de *tous* les évènements en attente,
- `w.update_idletasks()` : traitement des tâches inactives, du gestionnaire géométrique et mise à jour des widgets (canvas).

Méthodes communes

wait, update

wait (variable, visibility, window)

```
tns = tk.Toplevel(root)  # fenetre
...
# en deux temps, indispensable
root.wait_visibility(tns)  # attente active
tns.grab_set()  # fenetre bloquante
```

update_idletasks

```
def refresh():
    """ refraichit l'ecran (force le dessin) """
    # update_idletasks prend en compte les animations
    # en cours ce qui n'est pas le cas du simple update
    canv.update_idletasks()
```

Méthodes communes

Callbacks (bindings)

- `w.bind(event, callback)` : ajout d'un binding,
- `w.unbind(event)` : suppression du binding,
- `w.bindtags(tags=None)` : set/get ordre des bindings.

Callbacks non évènementiels (alarmes)

- `id = w.after(time, callback)` : *alarm callback* appelé après `time` ms,
- `id = w.after_idle(callback)` : callback quand le système est en attente (appelé après tous les évènements traités par la mainloop),
- `w.after_cancel(id)` : annulation du callback `id`.

Méthodes communes

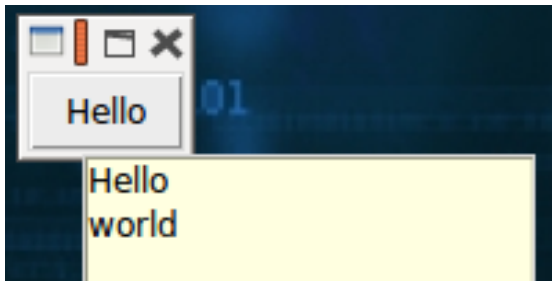
binding

```
# binding sur la roue centrale de la souris
# Windows
mem.bind("<MouseWheel>", mouse_wheel)
# Linux
mem.bind("<Button-4>", mouse_wheel)
mem.bind("<Button-5>", mouse_wheel)

def mouse_wheel(event):
  # Linux ou Windows "wheel event"
  if event.num == 5 or event.delta == -120:
      ...
  if event.num == 4 or event.delta == 120:
      ...
```

Méthodes communes

callback

```
button.bind("<Enter>", enter)
button.bind("<Leave>", leave)

def enter(event=None):
  global id
  id = button.after(1500, showtip)

def leave(event=None):
  if id:
    self.button.after_cancel(id)
```



Méthodes communes

Gestionnaire des fenêtres / widgets

- `w.lift()` : au sommet de la pile ou au dessus d'une fenêtre,
- `w.lower()` : en bas de la pile ou en dessous d'une fenêtre.

Informations relatives aux fenêtres / widgets

- `w.winfo_width()`, `w.winfo_height()` : largeur et hauteur en pixels,
- `w.winfo_reqwidth()`, `w.winfo_reqheight()` : largeur et hauteur requises (minimales),
- `w.winfo_id()` : identifiant.

Base de données des options

- `w.option_add(pattern, value)` : fixer une option,
- `w.option_get(name, class)` : récupérer la valeur d'une option.

Méthodes communes

option_add

```
root = Tk()

# fonts for all widgets
root.option_add("*Font", "courier")

# font to use for label widgets
root.option_add("*Label.Font", "helvetica 20 bold")

# make all widgets light blue
root.option_add("*Background", "light blue")
# root window already created, have to update it
root.config(background="light blue")

# use gold/black for selections
root.option_add("*selectBackground", "gold")
root.option_add("*selectForeground", "black")
```

Les fenêtres

window manager (WM) et *toplevel*

```
from tkinter import *

root = Tk()

# configuration de la fenetre via le WM
root.geometry("500x375+10+10") # dimension et position par defaut
root.title("Une fenetre") # titre de la fenetre
root.minsize(400, 300) # taille minimum de la fenetre
root.maxsize(1024,768) # taille maximum de la fenetre
root.positionfrom("user") # placement manuel de la fenetre
root.sizefrom("user") # dimensionnement manuel de la fenetre
root.protocol("WM_DELETE_WINDOW", root.destroy) # evenement WM

# creation d'une fenetre toplevel de nom ftopl
tl1 = Toplevel(root) # une toplevel depend de la root window
```

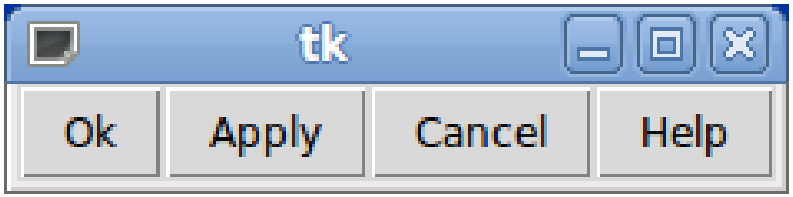
Les cadres (*frames*)

```
root = Tk()
# on cree 5 cadres d'aspect different
fr = {}
for relief in ("raised", "sunken", "flat", "groove", "ridge"):
    # creation d'une frame fille de la fenetre principale root
    fr[relief] = Frame(root, width="15m", height="10m", \
                       relief=relief, borderwidth=4)
    # chaque nouvelle frame est placee a droite de la precedente
    fr[relief].pack(side="left", padx="2m", pady="2m")
fr["flat"].configure(background="black")
```

Des *frames* différentes par leur aspect

Les boutons (*buttons*)

```
for mesg in ("Ok", "Apply", "Cancel", "Help"):
    # chaque bouton est place a gauche (!) dans la zone
    # non occupee par le bouton precedent (a sa droite)
    Button(text=mesg).pack(side="left")
```



Les *buttons* obligatoires

Paramètres courants : `text`, `command`, `image`, `state`, `textvariable`.

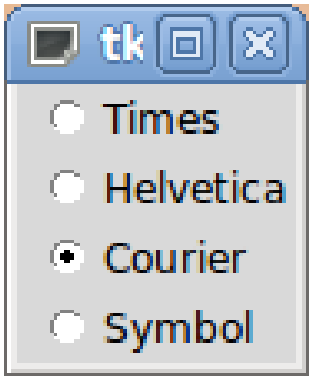
Les boîtes à cocher (*checkbutton*)

```
vcb = {}
for txt in ("gras", "italique", "souligne"):
    vcb[txt] = False
    Checkbutton(text=txt.capitalize(), variable=vcb[txt], \
                anchor="w").pack(side="top", fill="x")
```

Options courantes associées : `variable`, `indicatoron`.

Les boutons radio (*radiobutton*)

```
police = StringVar()

for txt in ("times", "helvetica", "courier", "symbol"):
    Radiobutton(text=txt.capitalize(), variable=police,
      value=txt, anchor="w").pack(side="top", fill="x")

police.set("courier")
```



Options courantes associées : `variable`.

Les classes `variable`

4 types : BooleanVar, DoubleVar, IntVar, StringVar.

Le mécanisme de *tracing* permet de changer un contenu de widget quand une variable est modifiée.

Tkinter met à disposition des *variable wrappers* afin de pouvoir tracer des variables.

Méthodes associées :

- `get()` : retourne la valeur,
- `set(string)` : instancie la variable et notifie tous les observateurs,
- `trace(mode, callback)` : avec mode = {'r', 'w', 'u'[2]}

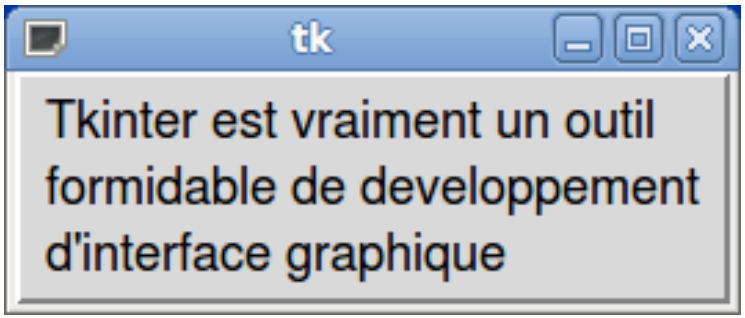
2. undefined, état d'une variable désallouée

Bouton menu (*menubuttons*) et menu (*menus*)

```
# menu bouton : lettre T soulignee et raccourci clavier Alt-t
mbtxt=Menubutton(root, text="Texte", underline=0)

# menu associe
menu1=Menu(mbtxt, tearoff=False)
m1cb = {}
for txt in ("gras", "italique", "souligne"):
    m1cb[txt] = BooleanVar()
    menu1.add_checkbutton(label=txt.capitalize(), variable=m1cb[txt])
menu1.add_separator()
police=StringVar()
menu1.add_radiobutton(label="Times", variable=police, value="times")
menu1.add_radiobutton(label="Symbol", variable=police, value="symbol")
menu1.add_separator()
menu1.add_command(label="Marges et tabulations", command=Page)

mbtxt["menu"]=menu1 # options accessibles via un dico
mbtxt.pack()
```

Labels (*label*) et messages (*message*)

```
Message(width="8c", justify="left", relief="raised", bd=2, \
        font="-Adobe-Helvetica-Medium-R-Normal--*-180-*", \
        text="Tkinter est vraiment un outil formidable de \
        developpement d'interface graphique").pack()
```

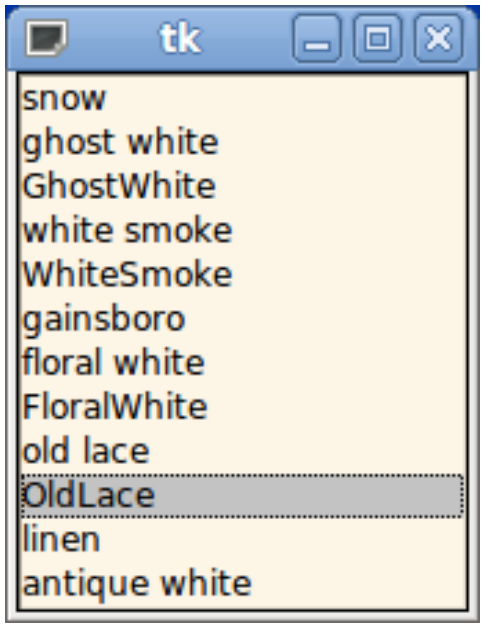


Options courantes associées : `aspect`, `justify`, `text`, `textvariable`.

Les listes (*listboxes*)

```
lc=Listbox(height=6)
lc.pack()
fd=open("/etc/X11/rgb.txt", 'r')
li=fd.readline() # on saute la 1ere ligne
li=fd.readline()
while li!='':
    lc.insert(END,li.split('\t')[2].strip(" \n"))
    li=fd.readline()
fd.close()

def lc_bg_color(event=None):
    selec=lc.curselection()
    lc.configure(background=lc.get(selec[0]))

lc.bind('<Double-Button-1>', lc_bg_color)
```

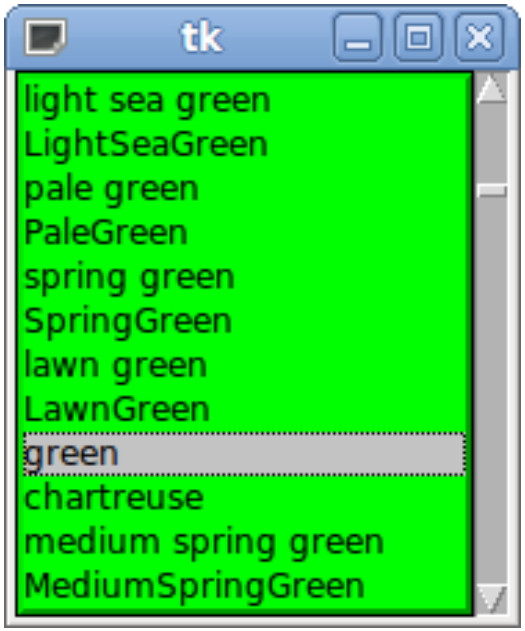
Les listes (*listboxes*)



Options courantes associées : `height`, `selectmode` ('browse', 'single', 'multiple', 'extended').

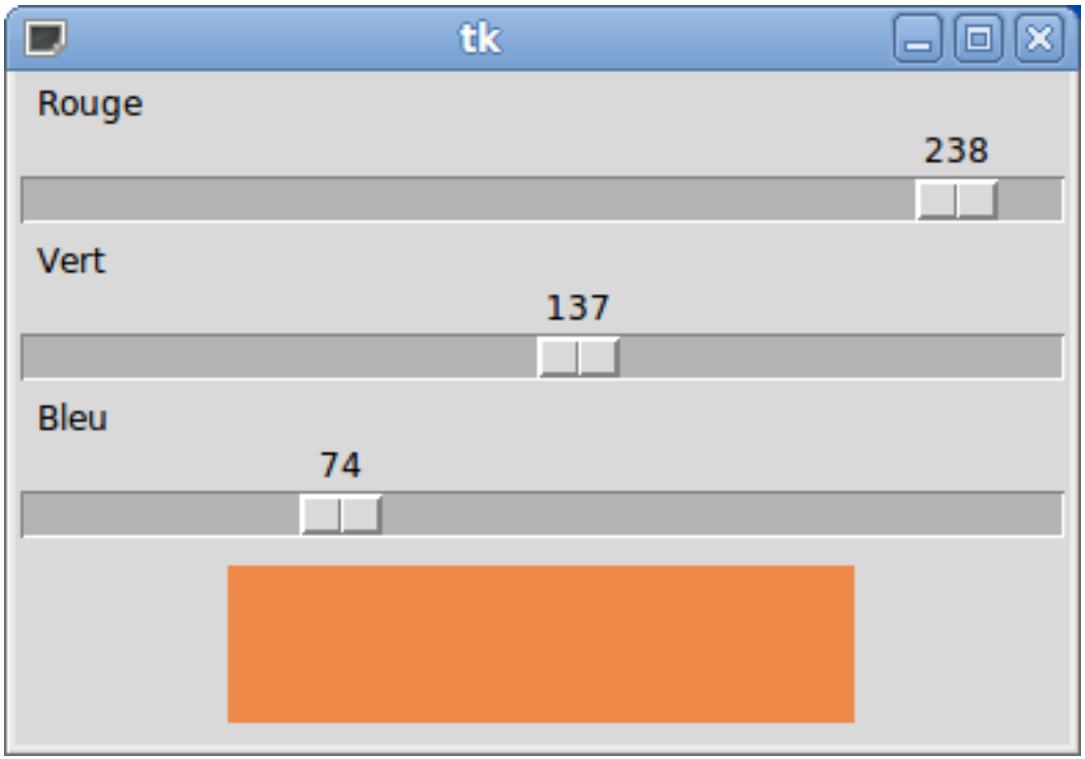
Les barres de défilement (*scrollbars*)

```
sc=Scrollbar(command=lc.yview)
sc.pack(side="right", fill="y")

lc.configure(yscrollcommand=sc.set)
```



Options courantes associées : `command`, `orient`.

Les tirettes (*scales*)



Options courantes associées : `command`, `from_`, `label`, `length`, `orient`, `resolution`, `showvalue`, `state`, `to`, `variable`, `width`.

Les tirettes (*scales*)

```
def Couleur(pval):
    coul="#{0:02x}{1:02x}{2:02x}".format(dcoul["rouge"].get(),\
         dcoul["vert"].get(), dcoul["bleu"].get())
    fvue.configure(background=coul)

dcoul={}
for coul in ("rouge", "vert", "bleu"):
    dcoul[coul]=Scale(label=coul.capitalize(), from_=0, to=255,\
              length="10c", orient="horizontal", command=Couleur)
    dcoul[coul].pack(side="top")

fvue=Frame(height="1.5c", width="6c")
fvue.pack(side="bottom", pady="2m")
```

Les entrées *entries*

```
Label(text="Nom du fichier : ").pack(side="left", padx="1m",\
      pady="2m")
nomfic=StringVar()
Entry(width=20, relief="sunken", bd=2, textvariable=nomfic).\
     pack(side="left", padx="1m", pady="2m")
```

Options courantes associées : `justify`, `state`, `textvariable`, `width`

Le widget text

tk

```
from tkinter import *

def Couleur(pval):
    coul="#{0:02x}{1:02x}{2:02x}".format(dcoul["rouge"].get(), dcoul[
"vert"].get(), dcoul["bleu"].get())
    fvue.configure(background=coul)

fenp=Tk()

dcoul={}
for coul in ("rouge", "vert", "bleu"):
    dcoul[coul]=Scale(label=coul.capitalize(), from_=0, to=255, length="10c", or
ient="horizontal", command=Couleur)
    dcoul[coul].pack(side="top")

fvue=Frame(height="1.5c", width="6c")
fvue.pack(side="bottom", pady="2m")

fenp.mainloop()
```

Le widget text

```
txt=Text()
txt.pack(side="top", expand=True, fill="both")

fd=open("scale.py", 'r')
li=fd.readline()
while li!='':
    txt.insert(END, li)
    li=fd.readline()
fd.close()

# configuration des styles de polices
txt.tag_config("pnom", font="Courier 12")
txt.tag_config("pbold", font="Courier 12 bold")

txt.tag_add("pbold", 4.9, 4.33)
```

Options les plus utilisées : `height`, `width`, `state`, `wrap`.

Le widget text

Les marques :

- définition : `txt.mark_set("debut", 0.0)`,
- utilisation : `txt.insert("debut", "Il etait une fois,")`
- les marques prédéfinies : `insert`, `end`, `current`,
- modificateur de position : `<position> linestart`, `<position> lineend`, `<position> ±<nb> chars`, `<position> ±<nb> lines`

Les *tags*, qui permettent de différencier les zones de texte :

- définition : `<widget>.tag_config(<ident>, <param>)` avec `<param>` {`font`, `justify`, `foreground`},
- utilisation : `<widget>.tag_add(<ident>, <pos1>, <pos2>)`

Le widget canvas

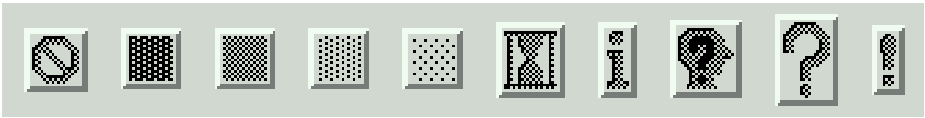
```
# creation
canv=Canvas(width=320, height=240, bg="white")
canv.pack()

# creation d'un rectangle rouge, defini par deux sommets
# la commande create retourne l'indice du rectangle dans le canvas
lobj=[]
lobj.append(canv.create_rectangle(10, 10, 200, 50, fill="red"))

# le rectangle devient bleu
canv.itemconfig(lobj[-1], fill="blue")
```

Commandes les plus utilisées (tag ou id) : `delete`, `coords`, `create_<type>`, `itemcget`, `itemconfigure`, `lower`, `raise`, `move`, `scale`

Le widget canvas

Les objets graphiques :

- `arc` : circonscrit par un rectangle, défini par 2 angles,
- `bitmap` : image en N&B (error, hourglass, info, questhead, etc.),

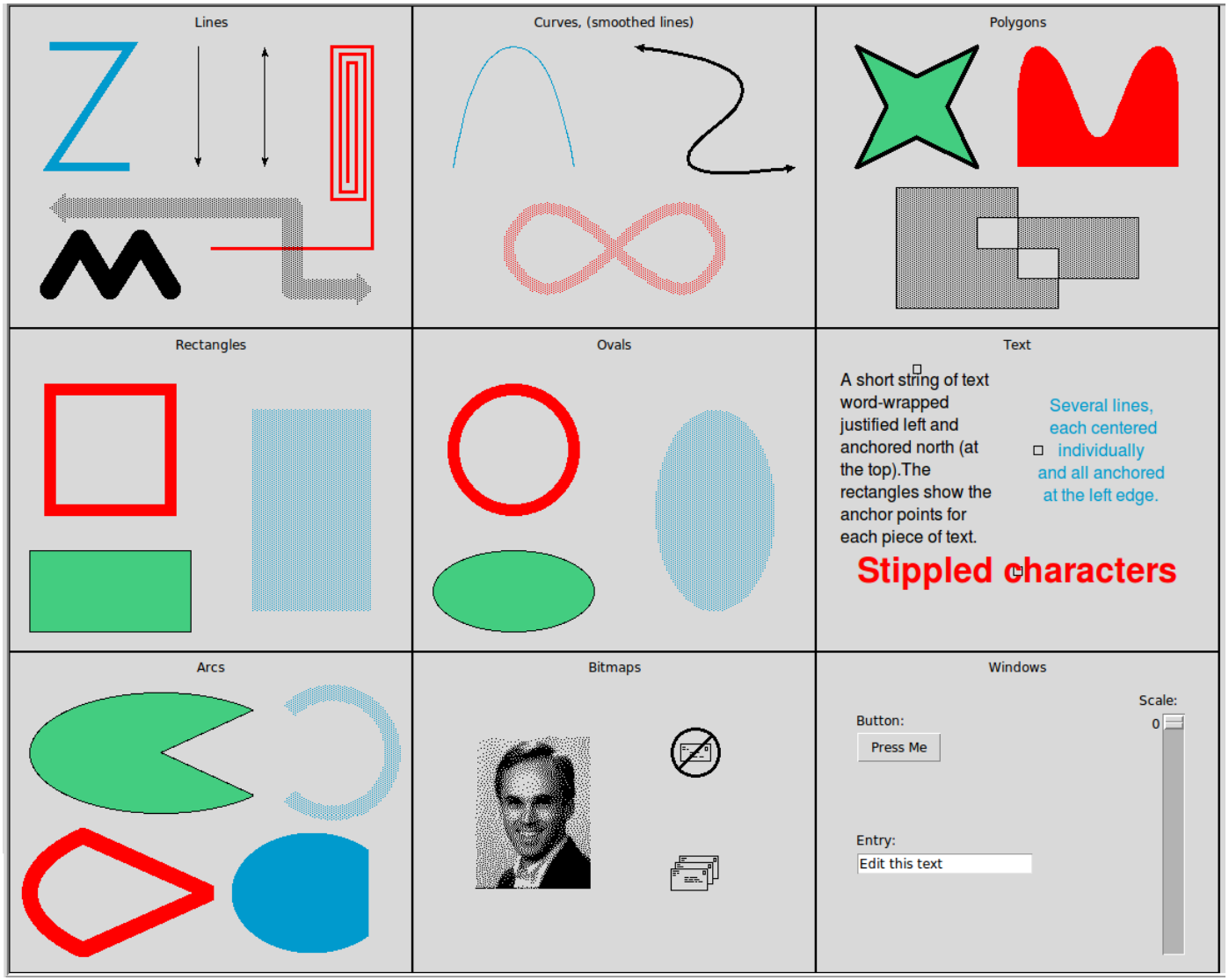


- `image` (avec PIL, Image et ImageTk) :

```
im = Image.open("fichier_image")
imp = ImageTk.PhotoImage(im)
canv.create_image(x, y, anchor=NW, image=imp, tags="truc")
```

- `line` : ligne brisée ou courbe de Bézier,
- `oval` : circonscrit par un rectangle,
- `polygon` : fermé (automatiquement),
- `rectangle` : définit par 2 sommets opposés,
- `text` : police, point de référence,
- `window` : conteneur de widgets.

Le widget canvas



Le widget canvas

id vs tag :

- chaque objet d'un canvas a un identificateur unique, son id,
- on peut associer des "marques" personnelles à chaque objet, les tags (sous la forme de tuples) ; un tag interne particulier : `current`.

Méthodes associées (du canvas) :

- coords : modification des coordonnées des objets,
- find : recherche d'objet (`withtag`, `closest`),
- gettags : liste des tags,
- move : translation relative,
- scale : homothétie (dilatation).

Le widget canvas

id, tag, coords, find

```
# dessin des carres de couleur
canv.create_rectangle(i, j, i+10, j+10, fill=coul, tags=("couleur", nom))

def affiche_couleur(event=None):
  nonlocal choixcoul
  id = canv.find_withtag("current")
  choixcoul.set(canv.gettags(id)[1])

canv.tag_bind("couleur", "<1>", affiche_couleur)
```

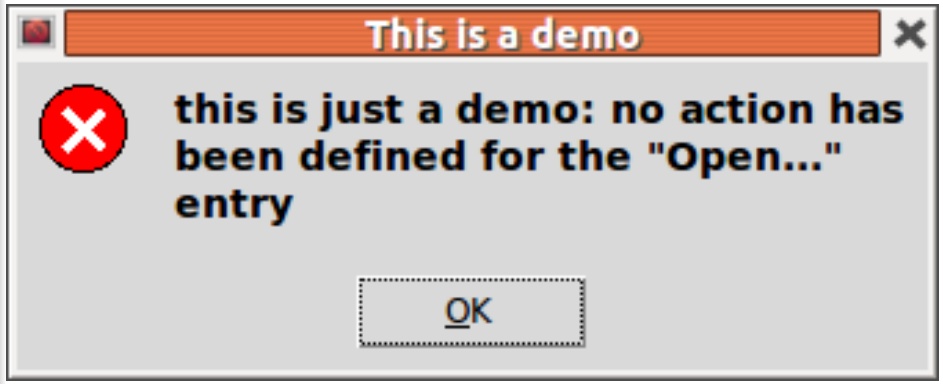
```
def obj_get_centre(pid):
    """centre geometrique de l'objet pid en coordonnees entieres"""
    lcoord = canv.coords(pid)
    # calcul de la somme des abscisses et des ordonnees
    ...
    n = len(lcoord)/2  # nombre de sommets
    return int(x/n), int(y/n)
```

Boîtes de dialogue prédéfinies

Le module tkMessageBox

- `askokcancel(title=None, message=None, **options)` : True si ok,
- `askquestion`,
- `askretrycancel` : recommencer, True si ok,
- `askyesno` : question, True si ok,
- `askyesnocancel` : question, None si annuler,
- `showerror`, `showinfo`, `showwarning`.

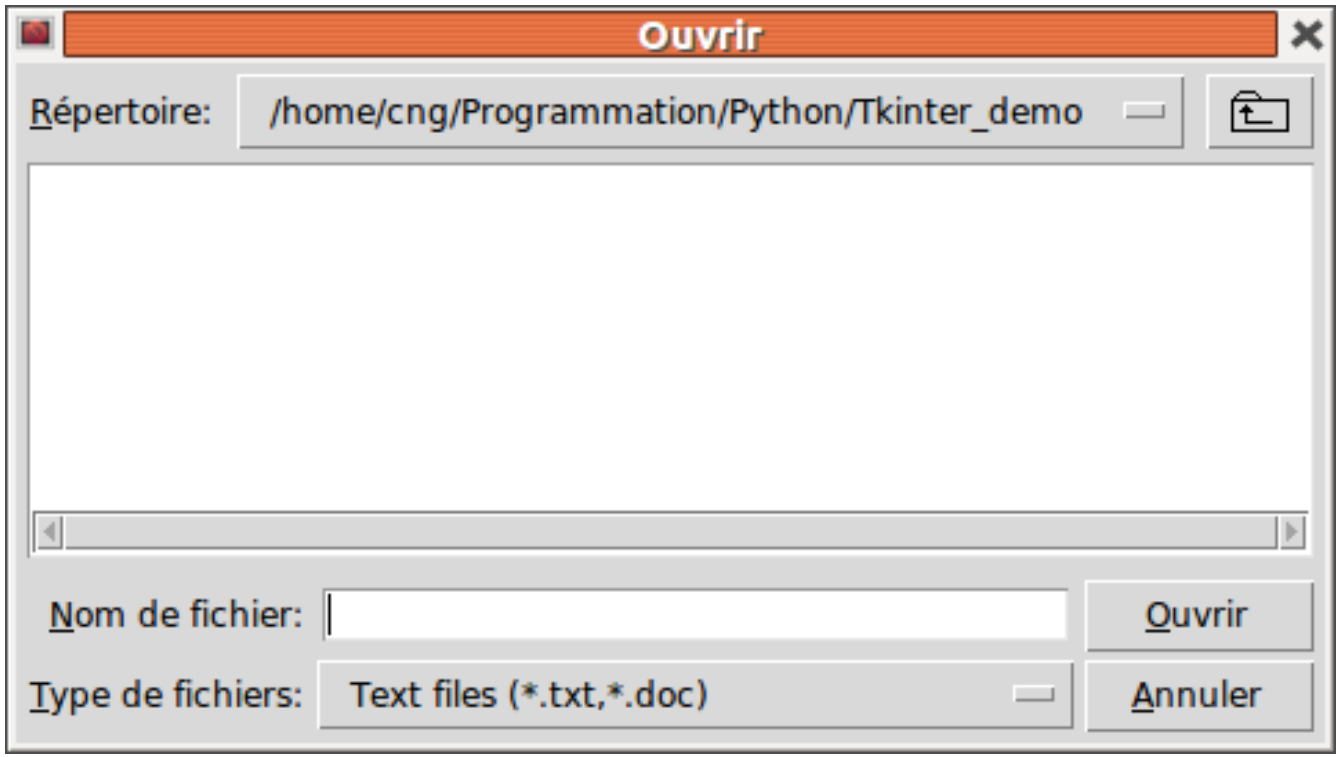
```
if sys.version_info >= (3,):
  from tkinter import *
  from tkinter import messagebox
else:
  from Tkinter import *
  import tkMessageBox

messagebox.showerror(...)
```



Boîtes de dialogue prédéfinies

Le module tkFileDialog

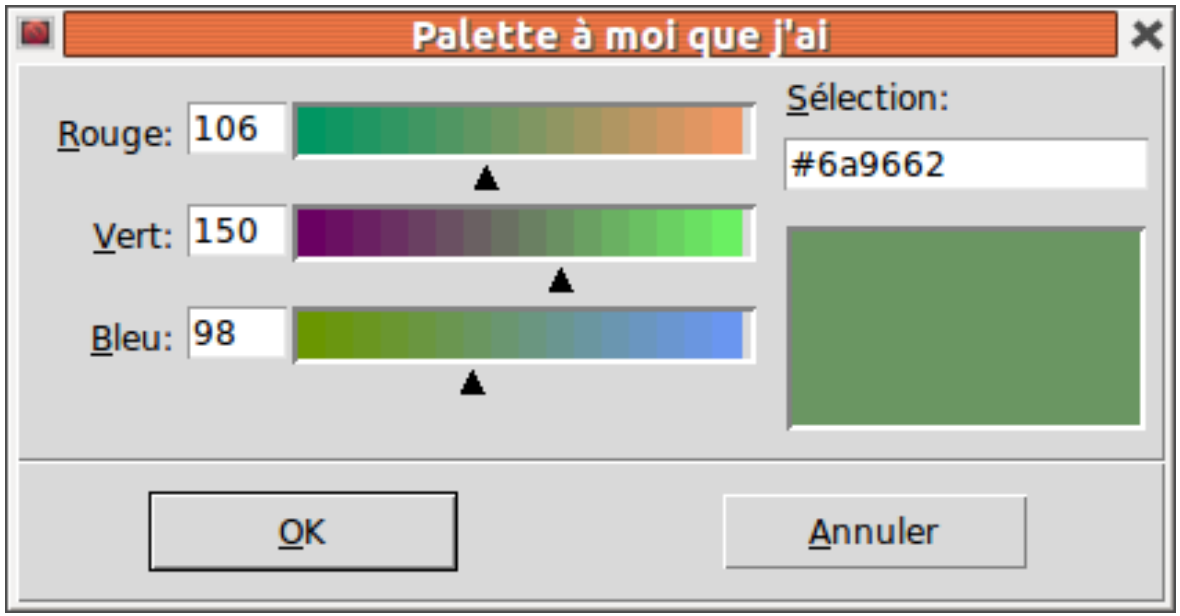
```
from tkinter import filedialog
filedialog.askopenfilename()
```



Boîtes de dialogue prédéfinies

Le module tkColorChooser

```
from tkinter import colorchooser
colorchooser.askcolor(color="#6A9662",
                      title = "Palette a moi que j'ai")
```



ttk (Tk themed widget)

Séparation entre comportement et aspect.

Mise en œuvre

```
from tkinter import *
from tkinter.ttk import *
```

Remplace : `Button`, `Checkbutton`, `Entry`, `Frame`, `Label`, `LabelFrame`, `Menubutton`, `PanedWindow`, `Radiobutton`, `Scale` et `Scrollbar`.

Meilleur « look & feel » mais certaines options ne sont plus compatibles (fg, bg, ...). Voir pour cela la classe `ttk.Style`.

```
style = ttk.Style()
style.configure("BW.TLabel", foreground="black", background="white")

l1 = ttk.Label(text="Test", style="BW.TLabel")
```

Les gestionnaires géométriques

Le *placer* et le *grider* (placements statiques) ou le *packer* (placement dynamique).

Leur rôle est de contrôler l'organisation de l'interface graphique en proposant différentes approches pour placer les widgets qui la composent.

Attention : ne *jamais* utiliser plus d'un gestionnaire géométrique par fenêtre, ils sont mutuellement exclusifs.

Les gestionnaires géométriques

Le *placer* (placement statique)

C'est un gestionnaire qui ne sert que rarement au concepteur d'interfaces. Il est plutôt réserver aux concepteurs de widgets mais il peut néanmoins être utile pour organiser certains dialogues.

Il se base sur des paramètres positionnels en absolu (`x`, `y`) ou en relatif (`relx`, `rely`).

Les dimensions peuvent être données en absolu (`width`, `height`) ou en relatif (`relwidth`, `relheight`) également.

Le paramètre `anchor` permet de changer le repère local.

Les gestionnaires géométriques

Le *grider* (placement statique)

La méthode `grid` s'appuie sur une décomposition implicite du conteneur (une fenêtre par exemple) en lignes et en colonnes, une grille imaginaire.

Le placement des widgets dans ces lignes et colonnes se fait sans préciser la taille requise (calculée implicitement).

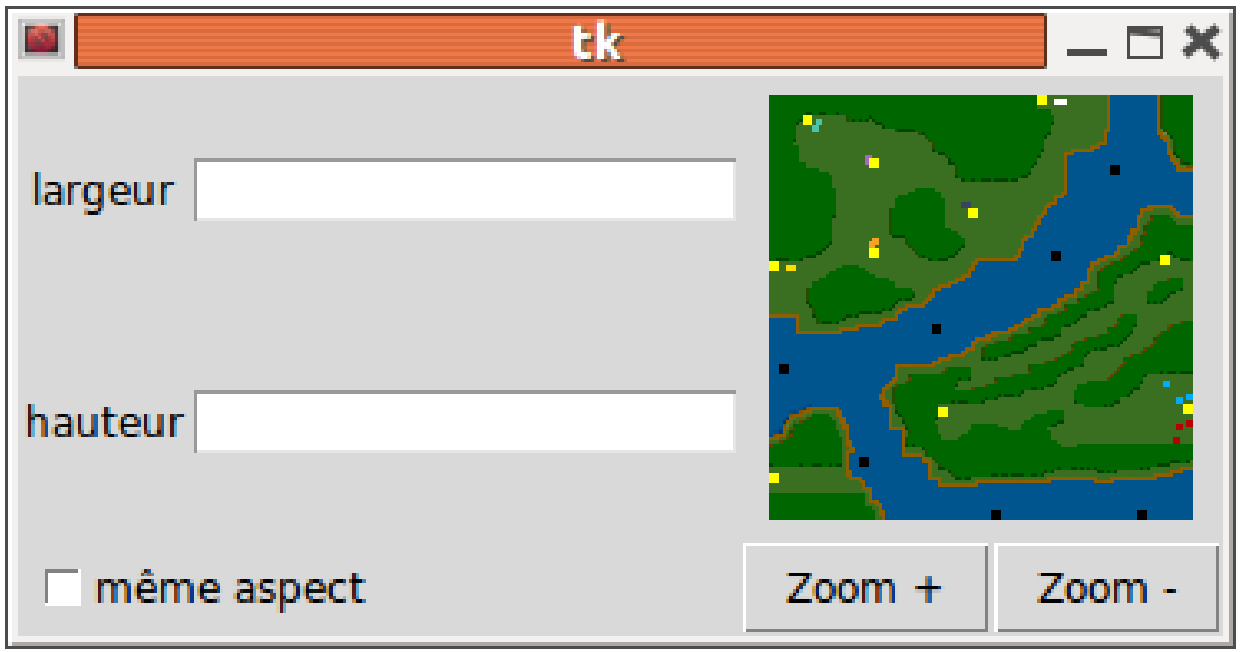
Le repère local est au centre de chaque widget par défaut. Il peut être précisé grâce au champ `sticky` dont les valeurs sont dans l'ensemble `N, S, E, W`.

Un widget peut occuper plusieurs lignes (resp. colonnes) grâce au champ `rowspan` (resp. `columnspan`).

Les gestionnaires géométriques

Le *grider* (placement statique)

<label 1>	<entry 2>	<image>
<label 1>	<entry 2>	
<checkbutton>	<button 1>	<button 2>



Les gestionnaires géométriques

Le *grider* (placement statique)

```
Label(master, text="largeur").grid(row=0)
Label(master, text="hauteur").grid(row=1)

Entry(master).grid(row=0, column=1)
Entry(master).grid(row=1, column=1)

Checkbutton(text="même aspect").grid(columnspan=2, sticky=W)

im = PhotoImage(file="battle07.gif")
Label(image=im).grid(row=0, column=2, columnspan=2, rowspan=2,
                     sticky=W+E+N+S, padx=5, pady=5)

Button(text="Zoom +").grid(row=2, column=2)
Button(text="Zoom -").grid(row=2, column=3)
```

Les gestionnaires géométriques

Le *packer* (placement dynamique)

La méthode `pack` positionne les widgets les uns par rapport aux autres, en s'appuyant sur la frontière du widget conteneur.

- nécessité de l'utilisation fréquente de *frames*.

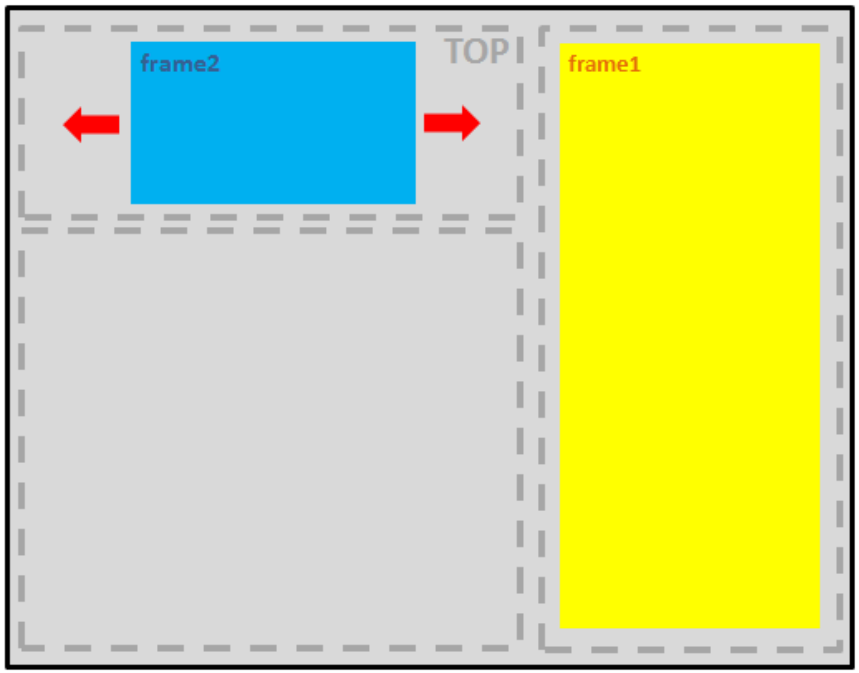
Options les plus couramment utilisées :

- `after=<widget>` et `before=<widget>`
- `expand=<bool>`
- `fill=<style>` (avec `<style>` = X, Y ou BOTH)
- `padx=<la>` (resp. `ipadx=<la>`) et `pady=<ha>` (resp. `ipady=<ha>`)
- `side=<cote>` (avec `<cote>` = LEFT, RIGHT, TOP ou BOTTOM)

Les gestionnaires géométriques

Le *packer*

```
f1 = Frame(root, bg="yellow",width=300,height=300,padx=10,pady=10)
f1.pack(side=RIGHT,fill=Y)

f2 = Frame(root, bg="blue",width=300,height=150)
f2.pack(side=TOP, fill=X)
```

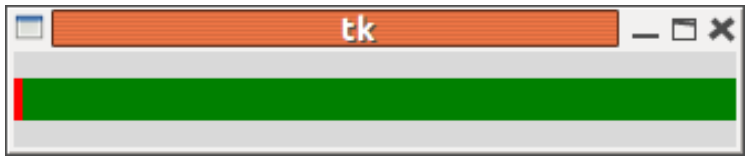


Les gestionnaires géométriques

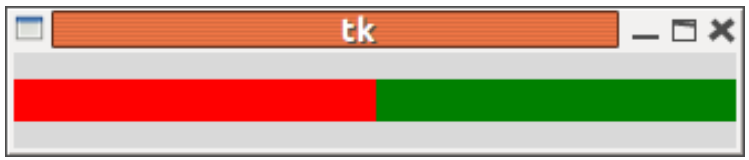
Le *packer*

Conséquence de l'utilisation ou non de l'option `expand`

```
root = tk.Tk()
lab1 = tk.Label(root, bg='red')
lab2 = tk.Label(root, bg='green')
lab1.pack(side=tk.LEFT, fill=tk.X)
lab2.pack(side=tk.RIGHT, expand=True, fill=tk.X)
```




```
lab1.pack(side=tk.LEFT, expand=True, fill=tk.X)
```



Événements et protocoles

Lier un événement à une commande : *binding*

```
# la combinaison Ctrl-c permet de quitter l'application
def Quitter(event):
    root.destroy()

root.bind_all("<Control-c>", Quitter)
```

Interaction entre l'application et le WM : *protocol handlers*[3]

```
from tkinter import messagebox

# callback
def ConfirmerQuitter():
    if messagebox.askokcancel("Quitter", "Voulez-vous vraiment quitter ?"):
        root.destroy()

root.protocol("WM_DELETE_WINDOW", ConfirmerQuitter)
```

3. cf. Inter-Client Communication Conventions Manual - ICCCM

L'association événement-commande

Types d'événements :

`ButtonPress`	`ButtonRelease`	`KeyPress`
`KeyRelease`	`FocusIn`	`FocusOut`
`Enter`	`Leave`	`Motion`

Modificateurs :

`Control`	`Shift`	`Alt`	`Button[1-5]`
`Mod[1-5]`	`Lock`	`Double`	`Triple`

Champs spéciaux des *callbacks* (via le paramètre `event`) :

`widget`	`x, y`	`x_root, y_root`
`char`	`keysym`	`keycode`
`num`	`width, height`	`type`

Interaction - gérer les évènements

Récupération des données associées à l'utilisation d'un périphérique.

- le paramètre `event` des fonctions associées au binding.

Exemple, gestion des coordonnées du pointeur souris

```
def PositionPointeur(event):
  print(event.x, event.y)

canv = tk.Canvas(root, ...)
canv.bind('<Motion>', PositionPointeur)
```

Interaction - gérer les évènements

Communication avec les widgets : transmission des données liées aux composants graphiques d'interaction.

- la méthode `configure` : modification des valeurs des options associées à un widget.
- la méthode `cget` : récupération des valeurs des options associées à un widget.

Exemple d'une double interaction

```
entry = tk.Entry(root)
label = tk.Label(root)
...
def evaluer(event):
  label.configure(text = 'Saisie : ' + str(eval(entry.get())))
...
entry.bind('<Return>', evaluer)
```

Interaction - émission d'évènements

La méthode `event_generate`.

Exemple d'une double interaction

```
canv = tk.Canvas(root, ...)
entry = tk.Entry(root)
...
def affiche(event):
  x=int(entry.get())
  canv.create_rectangle(x, x, x+10, x+10, fill='blue')
...
def setValue(event):
  canv.event_generate('<Control-Z>')
...
root.bind('<Control-Z>', affiche)
entry.bind('<Return>', setValue)
```

L'association événement-commande

Passage d'arguments à un callback

Une erreur classique :

```
def callback(n):
    print("bouton", n)

Button(text="un",   command=callback(1)).pack()
Button(text="deux", command=callback(2)).pack()
```

Dans ce cas, Python fait appel à la fonction *avant* de créer chaque bouton et retourne le *résultat* de l'appel à tkinter.

Solution (par les fonctions lambda) :

```
Button(text="un",   command=lambda: callback(1)).pack()
Button(text="deux", command=lambda: callback(2)).pack()
```

Le widget canvas

binding des objets graphiques

Les objets graphiques du canvas peuvent réagir aux actions de l'utilisateur.

```
canv=Canvas(width=320, height=240, bg="white")
canv.pack()
canv.create_rectangle(50, 50, 200, 200, fill="red", tags="clic")
# rem : canv.addtag_withtag("clic", 1)

def Couleur(event):
    id = canv.find_withtag("current")
    canv.itemconfig(id, fill="blue")

canv.tag_bind("clic", "<1>", Couleur)
```

Dans cet exemple, un clic gauche sur un rectangle ayant le tag *clic* provoque son changement de couleur (il passe du rouge au bleu).