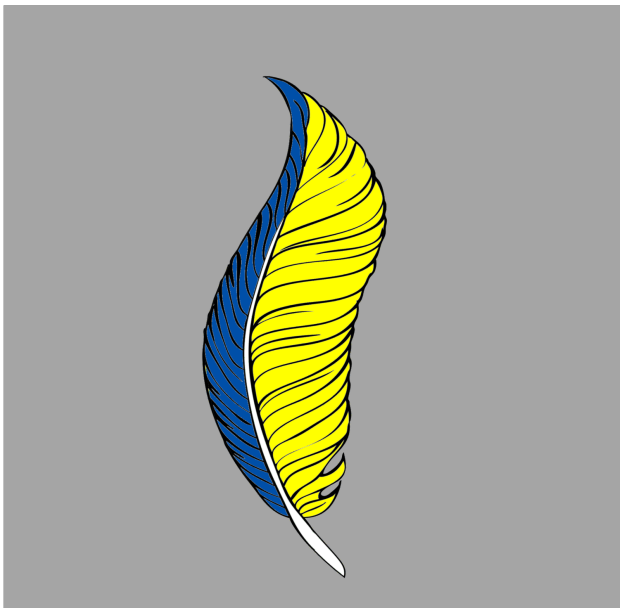
Pascal ORTIZ



Documentation Tkinter

Version du 12 février 2022

Table des matières

Chapitre I

Informations générales et transversales

1 L'écosystème Tkinter

Documentation de Tkinter

La documentation (au sens large) de Tkinter est très inégale.

Documentations de référence

La documentation réalisant le meilleur compromis entre fiabilité, lisibilité et exhaustivité est celle de Fredrik Lundh qui est un contributeur historique du module Tkinter. Depuis mai 2020, cette documentation n'est plus disponible sur le site de l'auteur. Dans ce fil de discussion, il été demandé à l'auteur s'il pouvait donner une date de remise en service mais, à ce jour, sans réponse de sa part. Toutefois le site a été archivé sur web archive. La documentation de effbot n'est pas mise à jour mais comme Tkinter évolue très peu, ce n'est pas vraiment gênant.

Sinon, l'unique document de référence et très complet qui existait jusqu'en 2019 a été archivé : Tkinter 8.5 reference: a GUI for Python (Shipman, 2013). Un autre lien, plus simplement accessible, est sur tkdocs ou encore LÀ. L'ancien lien vers l'université de New-Mexico n'est définitivement plus valide, comme expliqué ICI. On peut télécharger une version au format pdf ICI ou encore LÀ et très agréablement présentée (en Latex). Il traite aussi l'extension Ttk de Tkinter. Noter cependant que le document n'a pas été modifié depuis 2013 et il n'est pas parfaitement exhaustif. Par exemple, il ne décrit pas la classe Tk (et donc aucune de ses méthodes qui sont parfois indispensables), ne décrit pas la méthode `pack` (très courante d'emploi) et ne décrit que très peu la classe PhotoImage.

Il existe une traduction en français, soignée et très bien présentée : Tkinter pour ISN. L'auteur semble être Étienne Florent. La documentation ne traite pas du module Ttk et ne propose pas de version au format pdf.

Il n'existe pas de documentation officielle. La documentation du module Tkinter dans le document officiel est très incomplète et sommaire. Elle renvoie à d'autres documentations. Le code source officiel contient en revanche quelques codes de démonstration. L'incomplétude de cette documentation a été signalée à diverses occasions, par exemple ICI.

Pour une compréhension en profondeur de Tkinter, il faut se plonger dans le code-source Python du module Tkinter puis dans la documentation de Tcl-Tk, en particulier les deux liens suivants :

— Tutorial Tk
— Commandes Tk

Les livres spécifiques

Il y a peu de livres spécifiques à Tkinter. En voici trois.

— Bhaskar Chaudhary, Tkinter GUI Application Development Blueprints, (Packt, 2e édition, 2018) : livre très complet et bien organisé. Bien qu'il parte de zéro concernant la programmation graphique, la progression est assez rapide et le contenu dense voire très dense dans certains chapitres. Il contient la réalisation de nombreux projets, très variés, non triviaux et présentés avec progressivité : une boîte à rythmes, un jeu d'échecs, une application type Paint, etc. Les codes-source sont téléchargeables sur Github et sont fonctionnels quel que soit le système d'exploitation. Les applications sont accompagnées de copies d'écran et de schémas explicatifs. Les applications présentées ne sont pas minimalistes mais en jouant avec leur code source on peut beaucoup apprendre. En contre-partie, le livre traitant de beaucoup de méthodes et de widgets, il n'entre pas toujours dans les détails. Et il vaut mieux assez bien connaître Python, en particulier la programmation objets qu'il utilise à partir du chapitre 3. Un chapitre est consacré au traitement de la boucle événementielle, qui est un concept essentiel en pogrammation graphique. Il y a deux chapitre décrivant respectivement les widgets Canvas et Text.

— Mark Roseman (2020) : Modern Tkinter. L'auteur est un bon connaisseur du language Tcl/Tk dont Tkinter dérive, il maintient le site TkDocs. La 3e édition est beaucoup plus fournie que les précédentes et décrit tous les aspects de Tkinter. Elle a été mise-à-jour pour Python 3.9. Le code source des exemples du livre est accessible sur ce dépôt. Dans un chapitre en début d'ouvrage, l'auteur expose, sur un exemple de programme typique, la problématique d'une interface graphique qu'il reprend dans le chapitre suivant. Il passe ensuite en revue, chapitre par chapitre, les différentes composantes de Tkinter. Il manque un traitement des threads sous Tkinter. Il y a un chapitre original montrant quelles personnalisations il serait possible d'apporter à l'interface graphique classique (IDLE) qui est écrite en Tkinter. Je recommande cet ouvrage à tout lecteur ayant une certaine maturité et qui cherche des informations fiables sur Tkinter.

— Jason Briggs : Python for kids (no starch press, 2012). Livre très pédagogique et progressif. La 1re moitié du livre utilise Turtle et la seconde utilise Tkinter. L'ouvrage fournit des explications très méticuleuses avec commentaire de code, numérotation et surlignage des lignes de code. L'objectif de la partie sur Tkinter est de créer un petit jeu de plateforme avec des sprites. Le code source est téléchargeable sur le site de l'éditeur. Hélas, le livre souffre de quelques défauts. Le code n'est pas très pythonique, on dirait que l'auteur a traduit en Python des procédés venant du C ou de Java, il y a des complications ou des anomalies de codage dans des simples instructions `if` et la POO est assez factice. Mais surtout, comme expliqué dans un message sur StackOverflow, l'auteur utilise un procédé inadapté à Tkinter pour gérer la boucle de jeu : il crée sa propre mainloop qu'il temporise de quelques ms avec un appel à sleep et met à jour avec les méthodes `update` et `update_idletasks`.

On pourra aussi consulter Tkinter GUI Programming by Example édité chez Pack (2018) dont l'exposé est essentiellement basé sur la réalisation de projets.

Il existe également un mémento d'Alejandro Rodas de Paz, Tkinter GUI Application Development Cookbook datant de 2018. Des codes-types sont donnés et suivis d'explications. Toutefois, le livre est loin d'être exhaustif et ne parle parfois que des situations les plus simples. Noter un intéressant chapitre sur l'utilisation de threads sous Tkinter.

Tutoriels ou FAQ sur Internet

— Zetcode: Tkinter tutorial. Propose un tutoriel gratuit de présentation de Tkinter et qui culmine avec le codage d'un snake. Il existe aussi une version plus complète mais payante. Le code source et les images du snake sont à récupérer sur une page Github. Fait le tour des possibilités de Tkinter ce qui permet de se donner un aperçu. Les exemples de widgets sont décontextualisés donc on ne voit pas forcément comme en agréger plusieurs. Le jeu snake proposé est fluide et jouable et le code montre bien comment utiliser les méthodes du canevas pour gérer les collisions ; toutefois la qualité du code du snake pourrait être meilleure (confusion entre boucle for et boucle while, absence de else, non-utilisation de booléens, utilisation non nécessaire de la bibliothèque tierce Pillow) et l'auteur ne semble pas connaître l'astuce pour faire un snake (inutile de décaler chaque bloc, il suffit de placer en tête de snake le dernier bloc en tenant compte de la direction de déplacement fournie par le joueur).

— Python Tkinter By Example. Cet ouvrage qui n'est pas destiné à des débutants propose le codage de plusieurs applications assez avancées (par exemple une todo-list avec barres de défilement et base de données sqlite, un éditeur de texte avec coloration syntaxique et complétion, une application de traduction utilisant l'API Translation de Google).

— Xavier Dupré. Exposé assez rapide avec quelques exemples et quelques concepts.

— FAQ de developpez.com Tkinter pour ISN : contient quelques conseils, basés sur des exemples, à respecter par le public ISN qui veut débuter dans la création d'une interface graphique sous Tkinter.

Les livres généralistes

Comme Tkinter est le toolkit graphique par défaut de Python, de nombreux ouvrages généralistes le décrivent, avec plus ou moins de détails. Voici une sélection.

— Gérard Swinnen, Apprendre à programmer avec Python 3 : le plus grand classique sur Python en langue française et sans équivalent en anglais. Livre au format pdf, téléchargeable sur le site de l'auteur (il existe une version imprimée chez Eyrolles). De nombreux programmes Tkinter sont proposés dont certains assez sophistiqués. Code de qualité. Le livre date de 2012.

— Liang, Introduction to Programming Using Python 3 est assez bien fait du point de vue pédagogique et contient beaucoup d'exemples qui aident à débuter. Attention, le code n'est pas vraiment pythonnique, certaines pratiques ne sont pas à conseiller (usage récurrent de la fonction `eval`) et les animations ne sont pas codées de façon canonique (boucle infinie + méthode `update` au lieu d'utiliser la méthode `after`).

— Lutz, Programming Python (OReilly, 2011) : exposé très long (presque 400 pages). Assez complet et d'un niveau plutôt élevé.

— Summerfield : Programming in Python 3, Prentice Hall (2011) : contient deux exemples complets mais le code est assez dense et ne s'adresse pas à des débutants.

— Summerfield : Python in practice, Prentice Hall (2014) : utilise l'extension ttk, le code est très dense, il y a un chapitre assez court proposant un jeu complet.

Forums

— OpenClassrooms, Python : Tkinter fait l'objet de nombreuses questions.

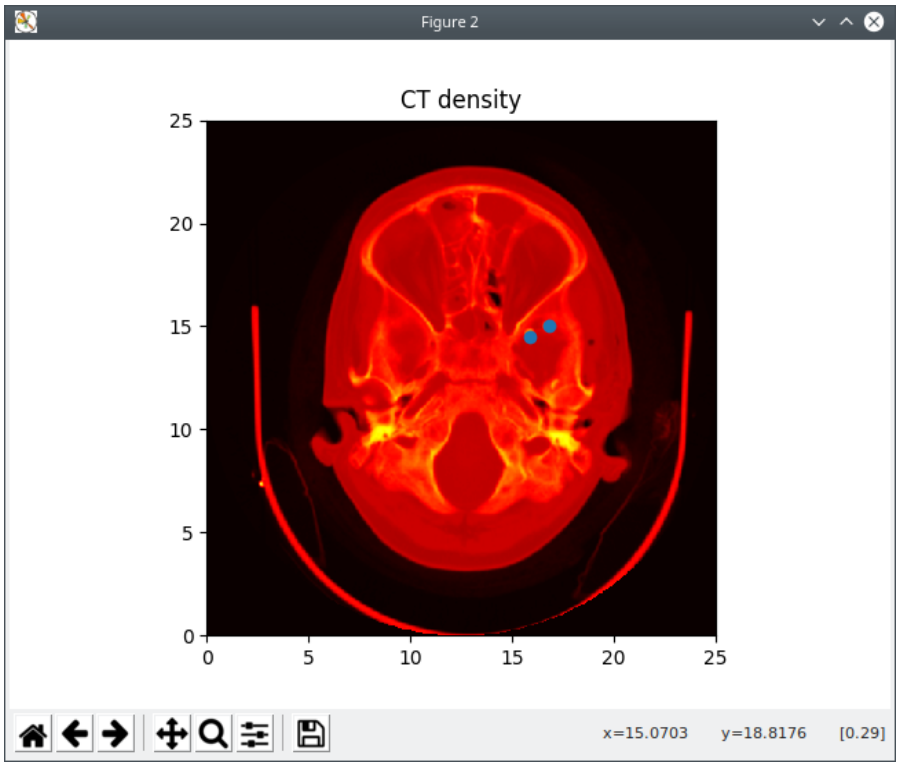
— Stackoverflow : une mine de réponses, en particulier pour les questions délicates. Prêter une attention particulière aux interventions du membre nommé Bryan Oakley. Voir aussi celles d' A. Rodas qui a reviewé plusieurs livres sur Tkinter.

Logiciels écrits en Tkinter

L'écosystème de Tkinter semble très réduit. En février 2019, la page Wikipedia recensait encore cinq interfaces graphiques écrites avec cette bibliothèque mais désormais (vérifié en novembre 2021), plus aucune n'est mentionnée.

Le logiciel écrit en Tkinter le plus utilisé est probablement IDLE qui est l'IDE proposé par la distribution officielle de Python.

Une autre source d'utilisation courante de Tkinter est via Matplotlib dont les graphiques peuvent être générés dans un canevas Tkinter :



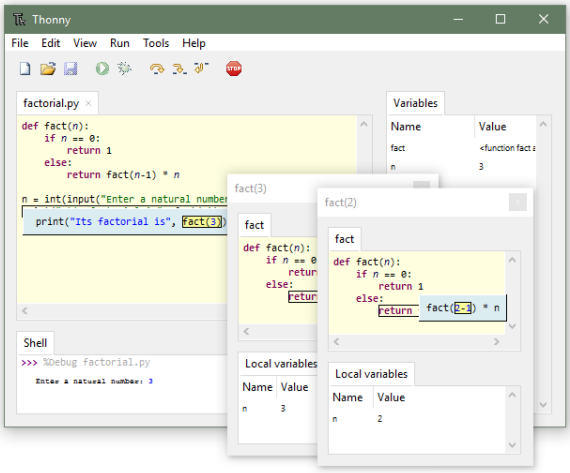
Il semblerait qu'il soit utilisé comme backend dans les installations par défaut sous Windows et Linux. Pour le savoir, lancer le code suivant :

```
from matplotlib import get_backend

print(get_backend())
```

et qui devrait afficher :

```
TkAgg
```

Il existe un éditeur de code Python, fort réussi, du nom de Thonny



qui a fait l'objet de quelques reportages et d'une discussion sur Reddit.

A lire des messages sur stackoverflow : tkinter applications, il semble toutefois que Tkinter soit aussi utilisé dans des projets « locaux » pour créer des interfaces graphiques.

Sur Github, on peut trouver une interface graphique pour un client FTP.

Il semble surtout que Tkinter soit utilisé dans un **cadre d'apprentissage** de Python ou de la programmation graphique.

Sur Reddit, un jeu 3D écrit en Tkinter, réalisé dans le cadre d'un projet de lycée, a été annoncé mais le code source n'est pas disponible.

La bibliothèque TCL/Tk écrite en langage C et qui sert de base à Tkinter est aussi utilisée pour gérer le graphisme en Ocaml.

Origine de Tkinter

La bibliothèque Tkinter trouve son origine dans le langage de programmation TCL. Ce langage dispose d'une extension nommée Tk et permettant de réaliser des interfaces graphiques. Tkinter est un binding de l'extension Tk. D'ailleurs, Tkinter signifie *Tk interface*. Python n'est pas le seul langage ayant réalisé un binding de Tk ; c'est aussi le cas des langages OCaml, Perl et Ruby.

Tk est lié à CPython par une extension écrite en C, dans le fichier `_tkinter.c`. La documentation de Python explique l'architecture de liaison de Tk. Le livre Fluent Python analyse la hiérarchie de classes définissant le module Tkinter (chapitre 12).

L'identité du créateur de Tkinter n'est pas claire. Selon Shipman, 2013, pdf, page 3, repris par Wikipedia, c'est Fredrik Lundh (qui a aussi écrit par exemple le moteur du module d'expressions régulières). La documentation et le code source officiels indiquent que les auteurs sont Steen Lumholt and Guido van Rossum ; Fredrik Lundh aurait adapté l'interface à Tk 4.2.

Selon le code source officiel, Steen Lumholt est l'auteur du code de `_tkinter.c`.

D'ailleurs, dans le forum comp.lang.python en 2001, Fredrik Lundh dit lui-même : *I'm pretty sure Steen Lumholt wrote it. Guido has been hacking on it, and I've been hacking my way around it, but Steen did the original work.*

2 Programmation par événements

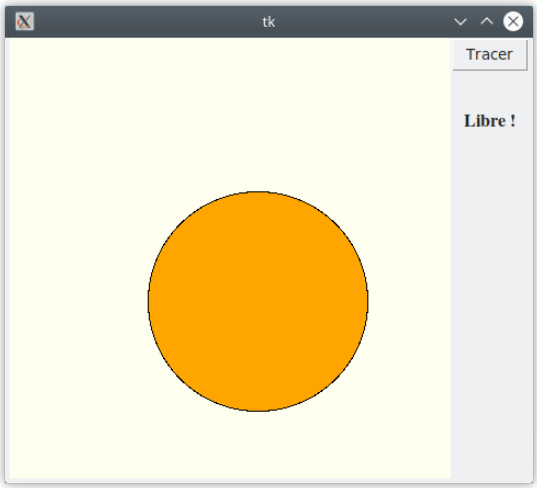
La notion de programmation événementielle

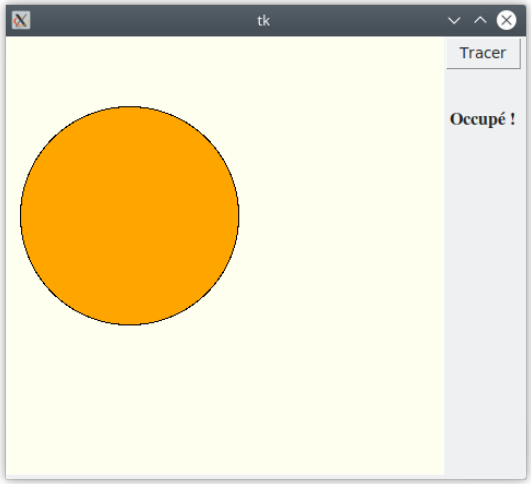
Quand vous utilisez une interface graphique (par exemple, un traitement de texte), la plupart du temps il ne se passe rien : le programme attend que vous interveniez en écrivant quelque chose ou en cliquant sur un bouton. Vos interventions qui font réagir le programme sont ce qu'on appelle des **événements**. La *surveillance* des évenements (on dit parfois l' *écoute*) est réalisée par un programme qu'on appelle une *boucle d'événements* car c'est une boucle infinie qui scanne les événements et réagit à ceux-ci. Ce type de programmation est dite *programmation événementielle* ou *asynchrone*. Ce n'est pas propre aux interfaces graphiques, par exemple un serveur http sous Nodejs fonctionne ainsi.

En Tkinter, la boucle des événements est gérée par la méthode `mainloop`. Voici un exemple typique :

```
from tkinter import *
from random import randrange

def tracer():
    cnv.delete("all")
    u, v = randrange(SIDE), randrange(SIDE)
    R=SIDE//4
    cnv.create_oval(u-R,v-R,u+R, v+R, fill='orange')

SIDE=400

root=Tk()
cnv=Canvas(root, width=SIDE, height=SIDE, background="ivory")
cnv.pack(side=LEFT)


bouton=Button(root, text="Tracer", command=tracer)
bouton.pack()

root.mainloop()
```

Typiquement, le programme ci-dessous agit en fonction des événements qu'il capture (des clics de souris) :





La plupart du temps, le programme ne fait rien, il est en attente d'événements provoqués par l'utilisateur. La méthode `mainloop` (ligne 20) surveille l'action de l'utilisateur. Si le bouton défini à la ligne 17 est cliqué, le programme est conçu pour réagir : la fonction `tracer` (lignes 4-8) est appelée (cf. ligne 17) et dessine un disque orange. Une fois le disque dessiné et affiché, le programme se remet en attente.

Particularités de la programmation événementielle

Ce qui suit peut être réservé à une seconde lecture.

Un programme événementiel a les deux particularités suivantes :

— pendant l'exécution de la boucle d'événements, l'exécution du programme est confinée dans la boucle (donc du code qui viendrait **après** ne s'exécuterait pas, cf. ligne 11 ci-dessous) ;

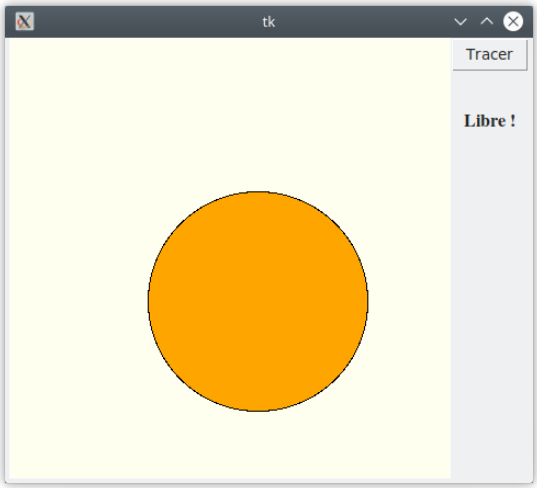
— la boucle principale ne doit pas être bloquée.

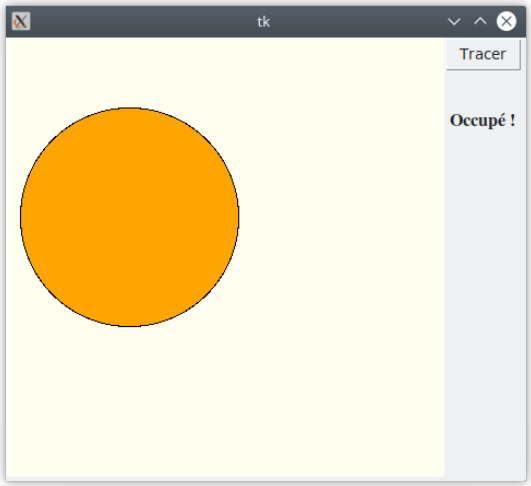
Voici un code qui illustre le premier point :

```
1 from tkinter import *
2
3 SIDE=400
4
5 root=Tk()
6 cnv=Canvas(root, width=SIDE, height=SIDE, background="ivory")
7 cnv.pack(side=LEFT)
8
9 root.mainloop()
10
11 print("Bonjour !")
```

Tant que la fenêtre créé en ligne 5 n'est pas détruite (en cliquant sur la croix), le message de la ligne 11 n'apparaîtra pas car le programme est confiné à la ligne 9.

Concernant le **blocage de la boucle événementielle** : cette dernière doit d'abord surveiller la réalisation d'événements. Si le programme réalise une opération qui mobilise ses ressources (un calcul intensif par exemple), le programme ne peut plus écouter et il ne va donc plus réagir. Soit par exemple le programme suivant :





Le code du programme (visible ci-dessous) est écrit pour que, au bout de 5 secondes après son ouverture (cf. ligne 30), il effectue un gros calcul (cf. lignes 13-14) qui va **empêcher l'interface graphique de réagir** et si on clique sur le bouton rien ne se passe. Juste avant le lancement du calcul (10 ms), un petit message annonce que l'interface va être bloquée (lignes 29 et 10-11). Une fois le calcul terminé, le programme redevient réactif, un message que le calcul est terminé (ligne 15) et le clic sur le bouton montre à nouveau le disque orange.

Ci-dessous, le code du programme :

```
from tkinter import *
from random import randrange

def tracer():
    cnv.delete("all")
    u, v = randrange(SIDE), randrange(SIDE)
    R=SIDE//4
    cnv.create_oval(u-R,v-R,u+R, v+R, fill='orange')

def msg():
    lbl["text"]="Occupé !"

def calcul():
    z=10**6000000
    lbl["text"]="Libre !"

SIDE=400

root=Tk()
cnv=Canvas(root, width=SIDE, height=SIDE, background="ivory")
cnv.pack(side=LEFT)

bouton=Button(root, text="Tracer", command=tracer)
bouton.pack()
```

```
lbl=Label(root, text="Libre !", font="Times 12 bold")
lbl.pack(pady=30)

root.after(4990, msg)
root.after(5000, calcul)

root.mainloop()
```

En particulier, sauf cas exceptionnel ou programme à visée pédagogique, une interface graphique en Tkinter n'utilise **jamais** la fonction `sleep` car elle **bloque** la boucle événementielle et empêche donc l'écoute des événements comme c'est expliqué dans de nombreux messages sur StackOverflow :

— Using Python Tk as a front-end for threaded code (B. Oakley)
— Python: Tkinter - One animation stops when the other begins (abarnert)
— Python Tkinter enter event (B. Oakley)

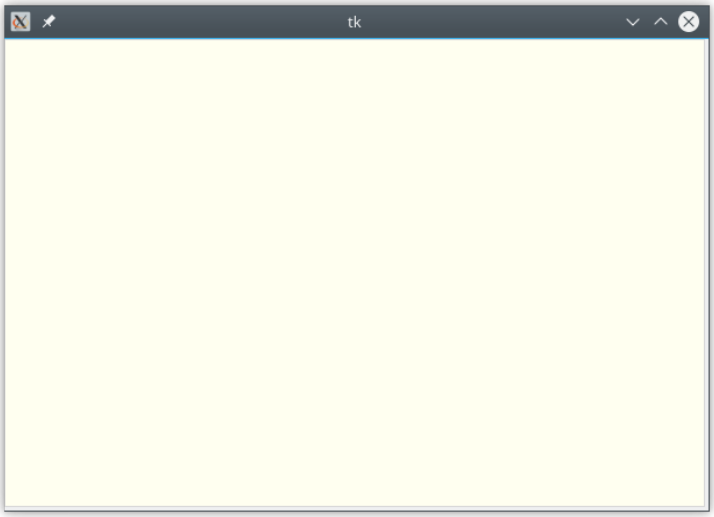
Création d'une fenêtre et d'un widget

Une application graphique de bureau s'exécute dans une fenêtre qui contient des widgets. Un widget est juste un composant graphique comme un menu ou un bouton. Voici un code "Hello Tkinter !" qui crée une fenêtre et un widget sous Tkinter :

fenetre_widget.py

```
1 from tkinter import *
2
3 WIDTH=600
4 HEIGHT=400
5
6 my_root=Tk()
7 cnv=Canvas(my_root, width=WIDTH, height=HEIGHT, background='ivory')
8 cnv.pack()
9
10 my_root.mainloop()
```

ce qui ouvre l'« application » suivante



— Ligne 1 : on importe toutes les fonctionnalités de Tkinter même si on n'a pas besoin de toutes celles-ci. C'est pratique pour de petits programmes. On procède légèrement autrement dans des usages plus avancés.
— Ligne 6 : on créé une fenêtre Tkinter en appelant le constructeur `Tk`, dite fenêtre « maîtresse » (*master*).
— Ligne 7 : on crée un « canevas » dans la fenêtre : c'est un widget permettant d'effectuer du graphisme, des animations, etc.
— Ligne 8 : on indique avec `pack` comment le widget doit être intégré dans le widget-maître (ici la fenêtre `my_root`).
— Ligne 10 : on lance la « boucle principale » (*mainloop*) de l'application. C'est typique de la programmation événementielle.

Important

Pour créer un widget (ligne 7), on utilise le constructeur approprié, par exemple ici `Canvas`. Ensuite, on passe des arguments à ce constructeur :

— le **premier argument** est le widget-maître qui va contenir le widget qu'on va construire. Par exemple, pour le canevas (ligne 7), le premier argument est la fenêtre `my_root`.
— les **autres arguments** sont les **options** de construction du widget. Ces options sont nommées, c'est-à-dire de la forme `option=truc`, où option est, par exemple, une dimension comme `width`, une option de couleur telle `background`, etc.

Remarques sur l'écriture et l'exécution d'une interface graphique

— Les interfaces graphiques sont construites par **emboîtement de widgets** comme des boutons, des entrées, des menus, etc. Le conteneur qui n'est contenu dans aucun autre conteneur est souvent référencé (ligne 6) par une variable appelée `root`, `master`, `window`, etc.
— Il est courant de placer en début de programme des variables écrites en majuscules (lignes 3 et 4) et qui représentent des données invariables du programme, comme certaines dimensions, le chemin vers certaines resources, etc.

— Si on retire la dernière ligne du programme (ligne 10), la fenêtre ne sera pas créée : l'application graphique tourne forcément dans la `mainloop`.
— Si on écrit du code après la ligne contenant le lancement de la mainloop (ligne 10), il ne sera pas exécuté, à moins de tuer la fenêtre de l'application graphique.
— Beaucoup d'éléments d'une application graphique sont des objets au sens de la programmation objet. Voilà pourquoi on utilise des notations pointées comme `cnv.pack()` (ligne 8 ou 10) pour faire des appels de méthodes.
— La ligne 6 est indispensable : on est obligé d'indiquer explicitement que l'on construit une fenêtre racine.
— La ligne 8 (ou un équivalent) est aussi indispensable : il faut indiquer au toolkit comment un widget est placé dans son contenant. Ici, le placement est effectué avec le gestionnaire `pack`.
— Notre programme ici est en attente d'événements. Mais, comme il est minimal, le seul événement qu'il peut enregistrer est un clic de fermeture sur la croix de la fenêtre.
— Programmer une interface graphique n'est pas incompatible avec la lecture de message dans la console produit avec la fonction `print` (ça permet de déboguer un minimum).

Comprendre le fonctionnement de la boucle principale

Ce qui suit sera réservé à une 2e lecture. Il contient essentiellement des références provenant toutes de StackOverFlow.

La mainloop de Tkinter est en fait une gigantesque boucle `while` infinie comme expliqué dans le message Python3, Understanding the GUI loop. Il est donc inutile d'utiliser sa propre boucle infinie appelant périodiquement la méthode `update`, comme expliqué dans ce message Tkinter understanding mainloop 1.

Si on utilise la fonction `sleep` dans une interface graphique Tkinter, **l'interface ne répond plus**. On n'utilise donc quasiment jamais `sleep` dans une application Tkinter. Les messages suivants détaillent ce point :

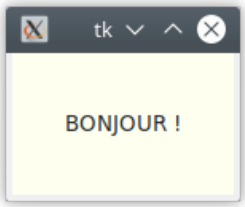
— Python Tkinter enter event
— Python: Tkinter - One animation stops when the other begins
— Python Tkinter enter event
— Using Python Tk as a front-end for threaded code

La méthode `mainloop` étant bloquante, dans certaines circonstances, il peut être envisagé de ne pas l'utiliser comme illustré dans Use TkInter without mainloop.

3 Les fenêtres

Le constructeur `Tk`

`Tk` est une classe définie dans le module `tkinter` et qui permet la création de la fenêtre-maîtresse de l'application :




```
1 from tkinter import Tk, Label
2
3 racine=Tk()
4 annonce=Label(height=5, width= 15, text="BONJOUR !", bg='ivory')
5 annonce.pack()
6 racine.mainloop()
```

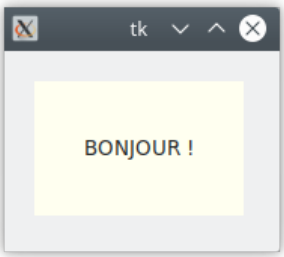
— Ligne 3 : construction de la fenêtre « racine » de l'application.

— Ligne 4 : création d'un widget, ici un widget Label (un court texte).

— Ligne 5 : placement du widget dans la fenêtre racine.

— Ligne 6 : lancement de l'application graphique.

On dit que c'est un constructeur car un appel à Tk construit véritablement une fenêtre-maîtresse. En principe, on ne donne aucun argument à Tk. Les éléments de positionnement, comme le bord d'un widget par rapport au bord de la fenêtre principale sont définis par la méthode qui placera le widget dans la fenêtre-maîtresse et non cette dernière. Par exemple, le code ci-dessous place entre la fenêtre-maîtresse et le widget un bord de 20 pixels :

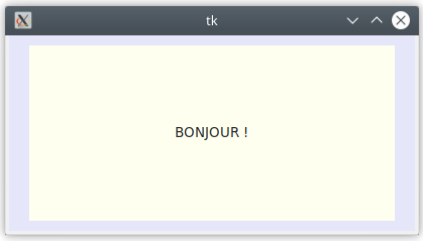


Le code correspondant est :

```
1 from tkinter import Tk, Label
2
3 racine=Tk()
4 annonce=Label(height=5, width= 15, text="BONJOUR !", bg='ivory')
5 annonce.pack(padx=20, pady=20)
6 racine.mainloop()
```

— Ligne 5 : le widget lui-même (non la fenêtre principale définie en ligne 3) gère le placement d'un widget dans la fenêtre qui le contient.

Bien qu'une fenêtre Tk ne puisse prendre aucun argument, on peut toutefois lui donner une couleur de fond (mais qui pourrait être caché par un widget) :




```
1 from tkinter import Tk, Label
2
3 racine=Tk()
4
5 racine['bg']="lavender"
6
7 annonce=Label(height=20, width= 20, text="BONJOUR !", font= "Times 20", bg='ivory')
8 annonce.pack(padx=10, pady=10)
9
10 racine.mainloop()
```

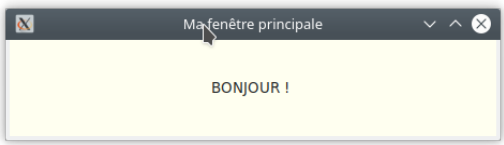
— Ligne 5 : racine donne accès à un dictionnaire dont un des éléments est la chaîne bg (pour background) et reçoit une couleur Tkinter (ici couleur lavande).

— Ligne 8 : les options de padding du label laissent découverte la fenêtre ce qui permet d'apercevoir sa couleur.

Titre de fenêtre

Par défaut, le bandeau d'une fenêtre porte le titre de tk. On peut changer ce titre par défaut en appelant la méthode title de Tk :

```
1 from tkinter import Tk, Label
2
3 racine=Tk()
4
5 racine.title("Ma fenêtre principale")
6
7 annonce=Label(height=5, width= 50, text="BONJOUR ! ", bg='ivory')
8 annonce.pack()
9 racine.mainloop()
```

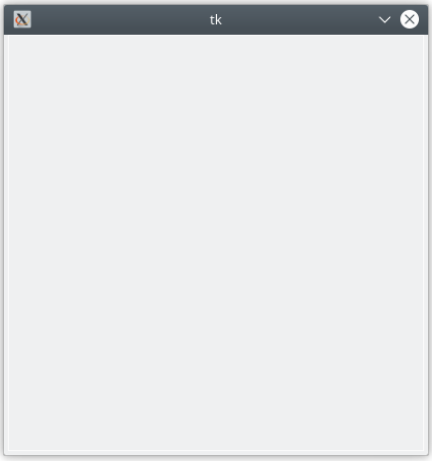


Fenêtre non redimensionnable

Dans de nombreuses situations, redimensionner la fenêtre n'a pas de sens car le contenu de la fenêtre n'est pas redessiné pour s'adapter aux nouvelles dimensions. Dans ce cas, il vaut mieux bloquer le redimensionnement, ce qui peut se faire comme ci-dessous :

```
1 from tkinter import Tk, Canvas
2
3 root = Tk()
4 cnv = Canvas(root, width=400, height=400)
5 cnv.pack()
6
7 root.resizable(False, False)
8
9 root.mainloop()
```

— Ligne 7 : les dimensions de la fenêtre sont bloquées.



On observera qu' **il n'y a plus d'icône** pour modifier la taille de la fenêtre.

En réalité, le redimensionnement n'est pas absolu : la modification d'un widget interne à la fenêtre a une influence sur la taille de la fenêtre qui le contient.

La position de la fenêtre principale dans le bureau

Lorsqu'une application utilise une seule fenêtre, la position de cette fenêtre dans le bureau est codée dans une chaîne de caractères appelée la *chaîne de géométrie* de la fenêtre et à laquelle on

accède par la méthode `geometry` (ligne 8 ci-dessous) :

```
1 from tkinter import *
2
3 root = Tk()
4 cnv = Canvas(root, width=600, height=600, bg='ivory')
5 cnv.pack()
6
7 root.update_idletasks()
8 print(root.geometry())
9 root.mainloop()
```

```
10 300x200-5+40
```

Cette chaîne donne les informations suivantes concernant la fenêtre générée :

— sa largeur est de 300 px,

— sa hauteur est de 200 px

— le bord droit de la fenêtre est à 5 px du bord droit du bureau,

— le bord supérieur de la fenêtre est à 40 px du haut du bureau

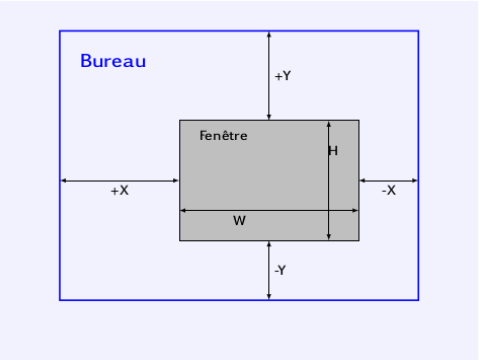
La ligne `root.update_idletasks()` est parfois nécessaire pour initialiser certaines tâches (sinon, dans notre cas, le message est `1x1+0+0`).
D'une manière générale, une chaîne de géométrie de fenêtre a la syntaxe générale suivante :

$$\texttt{WxH} \pm \texttt{X} \pm \texttt{Y}$$

où `W`, `H`, `X`, `Y` désignent des mesures en pixels avec la signification suivante :

Lettre	**Signification**
W	Largeur de la fenêtre
H	Hauteur de la fenêtre
X	Distance à l'un des deux bords verticaux du bureau. Bord gauche du bureau si signe +. Bord droit du bureau si signe -.
Y	Distance à l'un des deux bords horizontaux du bureau. Bord supérieur du bureau si signe +. Bord inférieur du bureau si signe -.

Le schéma ci-dessous explique les mesures :



En particulier, on comprend que le couple `(+X,+Y)` représente les **coordonnées du coin supérieur gauche de la fenêtre** dans le repère habituel du bureau (origine en haut à gauche, abscisses = bord supérieur orienté vers la droite, ordonnées = bord gauche orienté vers le bas).

On peut modifier la position initiale de la fenêtre principale en changeant les paramètres que porte cette chaîne.

Par exemple, pour placer la fenêtre à 500 pixels des bords gauche et supérieur :

```
from tkinter import Tk, Canvas

root = Tk()
cnv = Canvas(root, width=400, height=400)
cnv.pack()

root.geometry("+500+500")

root.mainloop()
```

Pour centrer une fenêtre dont les dimensions sont données à l'avance, on peut utiliser le code suivant provenant d'un message sur StackOverFlow :

```
import tkinter as tk


root = tk.Tk() # create a Tk root window

w = 800 # width for the Tk root
h = 650 # height for the Tk root

# get screen width and height
ws = root.winfo_screenwidth() # width of the screen
hs = root.winfo_screenheight() # height of the screen

# calculate x and y coordinates for the Tk root window
x = (ws/2) - (w/2)
y = (hs/2) - (h/2)

# set the dimensions of the screen
```

```
# and where it is placed
root.geometry('%dx%d+%d+%d' % (w, h, x, y))

root.mainloop() # starts the mainloop
```

Modifier dynamiquement la géométrie de la fenêtre

La géométrie de la fenêtre peut changer en cours d'exécution :

```
1  from tkinter import Tk, Canvas
2
3  root = Tk()
4  cnv = Canvas(root, width=400, height=400)
5  cnv.pack()
6
7  def printgeom():
8      root.geometry("600x600+200+50")
9      print(root.geometry())
10
11 print(root.geometry())
12 cnv.after(2500, printgeom)
13
14 root.mainloop()
```

— Ligne 12 : au bout de 2,5 s, la géométrie de la fenêtre est modifiée

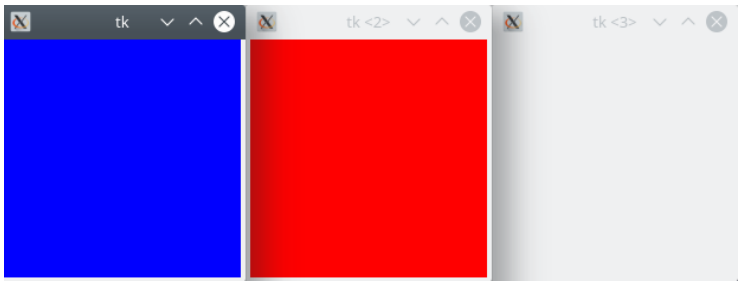
— Ligne 8 : modification de la géométrie qui est affichée.

Ouvrir plusieurs fenêtres

Le code suivant

```
1  from tkinter import *
2
3  root = Tk()
4  a = Toplevel(root, bg='red')
5  b = Toplevel(root, bg='blue')
6
7  root.mainloop()
```

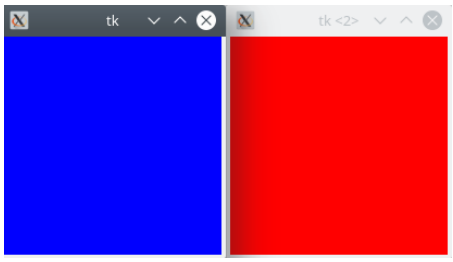
va ouvrir 3 fenêtres : deux fenêtres et leur fenêtre maîtresse comme on le voit ci-dessous :



Pour cacher la fenêtre maîtresse, utiliser `withdraw` :

```
from tkinter import *

root = Tk()
a = Toplevel(root, bg='red')
b = Toplevel(root, bg='blue')
root.withdraw()

root.mainloop()
```

ce qui produit :



Toutefois, comme la fenêtre est cachée, il n'y a plus moyen de fermer définitivement l'application puisque la croix de fermeture de l'application est invisible. On peut y remédier de la manière suivante :

```
from tkinter import *

def quit_a():
    a.destroy()
    if closed[1]:
        root.destroy()
    else:
        closed[0]=True

def quit_b():
    b.destroy()
    if closed[0]:
        root.destroy()
    else:
```

```
        closed[1]=True

root = Tk()
a = Toplevel(root, bg='red')
b = Toplevel(root, bg='blue')

a.protocol("WM_DELETE_WINDOW", quit_a)
b.protocol("WM_DELETE_WINDOW", quit_b)

closed=[False, False]

root.withdraw()

root.mainloop()
```

Résolution de l'écran

Il peut être utile parfois de coder une application graphique en fonction de la résolution de l'écran de l'hôte. Tkinter permet de récupérer ces informations sur la résolution :

```
1 from tkinter import Tk, Canvas
2
3 root = Tk()
4 cnv = Canvas(root, width=400, height=400)
5 cnv.pack()
6
7 print(root.winfo_screenwidth(), root.winfo_screenheight())
8
9 root.mainloop()
```

```
10 1920 1080
```

— Ligne 7 : les deux dimensions de la résolution.

Le mode plein écran

On peut placer une fenêtre en mode plein écran avec l'option `"fullscreen"` qu'on place à `True`. La sortie du mode plein écran n'est pas prévue par défaut donc il faut l'écrire soit même en liant par exemple la touche Echap au retour de l'écran à sa position normale :

```
1 from tkinter import *
2
3 SIDE = 1000
4
5
6 def normalscreen(event):
7     master.attributes("-fullscreen", False)
8
9
```

```
master = Tk()
master.attributes("-fullscreen", True)

cnv = Canvas(master, width=SIDE, height=1.3 * SIDE, bg='ivory')
cnv.pack()

master.bind("<Escape>", normalscreen)

master.mainloop()
```

Appeler plusieurs fois Tk

On lit parfois dans du code de débutants plusieurs appels successifs au constructeur Tk afin de créer plusieurs fenêtres. Dans l'immense majorité des cas c'est inutile, comme expliqué dans ce message Why are multiple instances of Tk discouraged? (B. Oakley). Dans une discussion du forum OpenClassroom, le type d'inaccessibilité que provoque l'usage de plusieurs appels à Tk est illustré par un exemple. Voir aussi cette discussion où le même problème a été rencontré.

Si l'objectif est juste de créer plusieurs fenêtres, le faire en appelant le widget TopLevel :

```
from tkinter import *

root = Tk()
a = Toplevel(root, bg='red')
b = Toplevel(root, bg='blue')

root.mainloop()
```

4 Géométrie des widgets

Gestionnaires de géométrie

Une fois une fenêtre créée, il faut placer ses widgets à l'intérieur. Le placement de tout widget se fait **obligatoirement** par l'intermédiaire d'un gestionnaire de géométrie. Tkinter dispose de trois gestionnaires de géométrie : pack, grid et place.

Voici un résumé des principales caractéristiques de ces trois gestionnaires :

Nom du gestionnaire	Commentaire	Documentation
pack	Empilements verticaux ou horizontaux. Le plus simple si interface simple. Complexe, peu robuste si interface complexe avec beaucoup de contraintes d'alignement. Nombreuses options pour affiner le placement.	Lundh : pack
grid	Place selon une grille 2D. Assez simple. Adapté si interface non triviale ou nécessitant des alignements et des regroupements. Nombreuses options pour affiner le placement. Seul gestionnaire expliqué dans la doc non officielle.	Lundh : grid Tkinter ISN
place	Place à une position définie en pixels ou en valeur relative. Usage spécialisé seulement.	Lundh : place

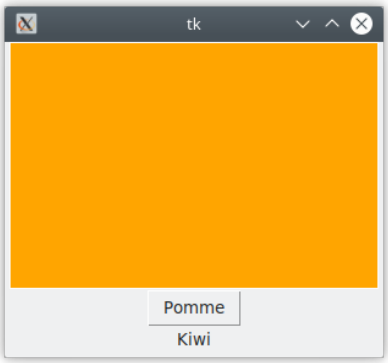
Bien noter que dans une fenêtre conteneur donnée, on **ne peut pas** utiliser simultanément les gestionnaires `pack` et `grid` (une exception sera levée).

En pratique, tout placement de widget, disons `w`, dans une fenêtre, disons `root`, nécessite un appel de la forme `root.geometrie(w)` où `geometrie` est le nom de l'un des trois gestionnaires ci-dessus.

Voici un exemple représentatif :

```
1 from tkinter import *
2
3 root=Tk()
4
5 cnv=Canvas(root, width=300, height=200, bg="orange")
6 cnv.pack()
7
8 bouton=Button(root, text="Pomme")
9 bouton.pack()
10
```

```
11 entree=Label(root, width='10', text="Kiwi")
12 entree.pack()
13
14 root.mainloop()
```



Les widgets sont incorporés dans la fenêtre `root` (ligne 3, la fenêtre initiale) à la ligne 9 et à la ligne 12 avec le gestionnaire `pack`.

Organiser des widgets, de façon transparente pour le programmeur, de manière souple et visuellement satisfaisante est une tâche complexe, remplie de calculs géométriques. L'intérêt des deux premiers gestionnaires est qu'ils nous en dispensent tout en assurant un placement satisfaisant.

Pour une description des gestionnaires et des conseils d'utilisation, consulter les messages suivants sur le forum StackOverflow :

— Bryan Oakley : description
— Bryan Oakley : conseils pour designer son interface graphique
— Bryan Oakley : éviter le gestionnaire place
— Bryan Oakley : inconvénients de place
— Bryan Oakley : cas d'usage du gestionnaire place

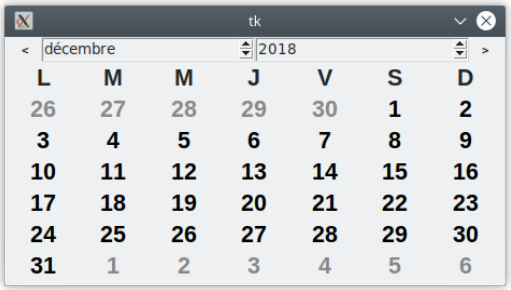
Dans ce document, le gestionnaire `place` n'est pas décrit : les deux autres gestionnaires répondent, et répondent *bien*, à l'immense majorité des usages.

Attention que les gestionnaires `grid` et `pack` ne sont pas à *fenêtre fixe* : si un widget change de taille (par exemple, on ajoute ou on retranche du texte quelque part) alors toute la fenêtre contenante change de taille ce qui est souvent souhaité mais parfois aussi non souhaité.

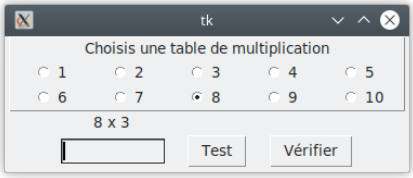
Enfin, comme indiqué par Bryan Oakley, que ce soit la géométrie pack ou grid, par défaut les widgets contenants ou même n'importe quel widget va adapter ses dimensions pour faire tenir tout ce qu'il contient (et c'est le comportement souhaité dans `99,9 %` des cas).

Le gestionnaire grid

Le gestionnaire grid permet d'organiser ses widgets selon une grille 2D. C'est un gestionnaire assez simple à utiliser et très efficace. Un cas typique d'utilisation est la réalisation d'un calendrier car il y a beaucoup d'alignements à gérer :



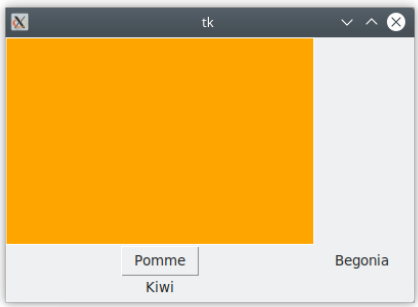
Un autre cas est la réalisation d'un quizz car il y a plusieurs widgets et qui en outre se placent naturellement en grille :



Voici un exemple représentatif de code utilisant le gestionnaire `grid` :

```
from tkinter import *

root=Tk()

cnv=Canvas(root, width=300, height=200, bg="orange")
cnv.grid(row=0, column=0)

pomme=Button(root, text="Pomme")
pomme.grid(row=1, column=0)

kiwi=Label(root, width='10', text="Kiwi")
kiwi.grid(row=2, column=0)

begonia=Label(root, width='10', text="Begonia")
begonia.grid(row=1, column=1)

root.mainloop()
```

qui produit



La grille est implicite (il n'y a pas de traits séparateurs visibles). Et on ne définit pas la grille à l'avance : on donne aux widgets un placement dans la grille en utilisant les arguments nommés `row` (pour l'indice de ligne) et `column` (l'indice de colonne) et Tkinter se charge de comprendre combien de lignes et de colonnes il y aura. Les indices des cellules commencent à 0.

Les options du gestionnaire grid

On peut affiner le placement des widgets avec le gestionnaire `grid`. Les principales options sont les suivantes :

Option	Caractéristiques	Valeurs possibles	Valeur par défaut
sticky	Étirement du widget dans certaines directions nord, sud, etc pour remplir place restante dans la cellule	Un ancre : N, E, W, etc	CENTER
columnspan	Répartir sur plusieurs colonnes consécutives	Entier positif	1
rowspan	Répartir sur plusieurs lignes consécutives	Entier positif	1
padx	Marge horizontale (en pixels)	Entier positif	0
pady	Marge verticale (en pixels)	Entier positif	0

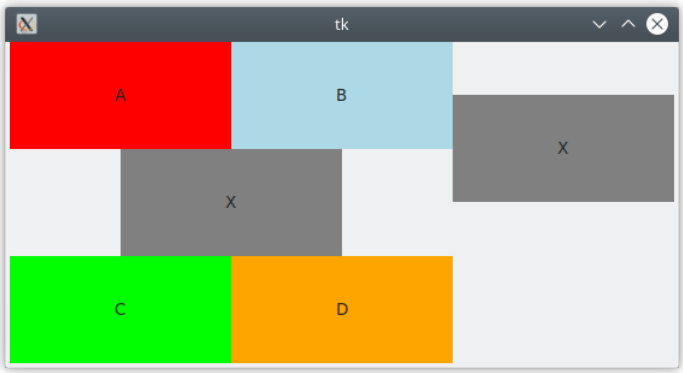
Gestionnaire grid : l'option span

Le gestionnaire `grid` permet d'étendre un widget sur plusieurs lignes ou colonnes consécutives :

```
1 from tkinter import *
2
3 root=Tk()
4
5 a=Label(root, text="A", bg='red', width=20, height=5)
6 a.grid(row=0, column=0)
7
```

```
8  b=Label(root, text="B", bg='lightblue', width=20, height=5)
9  b.grid(row=0, column=1)
10
11 x=Label(root, text="X", bg='gray', width=20, height=5)
12 x.grid(row=1, column=0,columnspan=2)
13
14 y=Label(root, text="X", bg='gray', width=20, height=5)
15 y.grid(row=0, column=2,rowspan=2)
16
17 c=Label(root, text="C", bg='lime', width=20, height=5)
18 c.grid(row=2, column=0)
19
20 d=Label(root, text="D", bg='orange', width=20, height=5)
21 d.grid(row=2, column=1)
22
23 root.mainloop()
```

qui produit

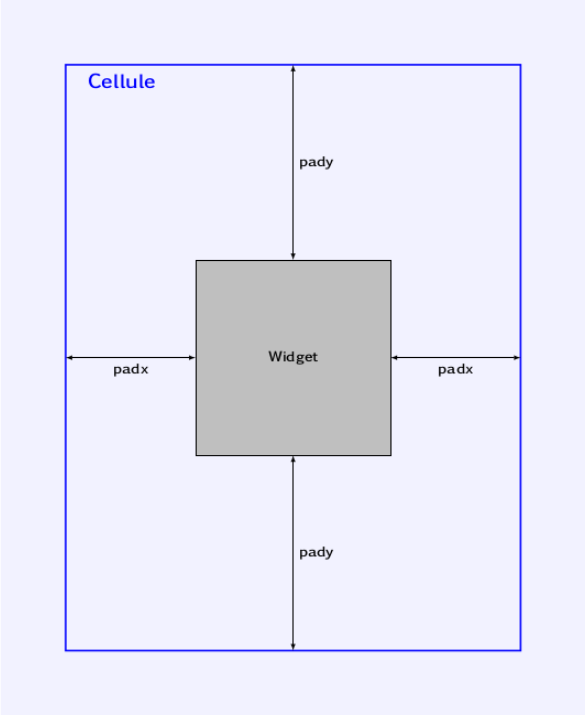


Par exemple, ligne 11, on définit un label gris, qui affiche le texte X. Ce label s'étend sur 2 colonnes (à cause de l'option `columnspan=2`) et commence ligne 1 et colonne 0.

Gestionnaire grid : les options padx et pady

Les options padx et pady ajoutent de l'espace autour du widget.

Le dessin ci-dessous illustre le positionnement des marges padx et pady :



Voici un exemple d'utilisation :

```
1  from tkinter import *
2
3  root=Tk()
4
5  a=Label(root, text="A", bg='red', width=20, height=5)
6  a.grid(row=0, column=0, padx=20)
7
8  b=Label(root, text="B", bg='lightblue', width=20, height=5)
9  b.grid(row=0, column=1, padx=50, pady=100)
10
11 c=Label(root, text="C", bg='lime', width=20, height=5)
12 c.grid(row=1, column=0)
13
14 d=Label(root, text="D", bg='orange', width=20, height=5)
15 d.grid(row=1, column=1)
16
17 root.mainloop()
```

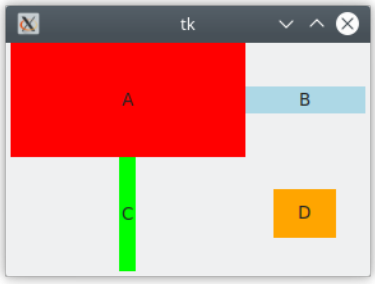
La distance padx est comptée sur l'axe des x (horizontal) et pady sur l'axe vertical. la marge padx semble affecter toute la colonne où cette option est placée : ci-dessus, par exemple, la widget a est affecté d'un padx=20 tandis que dans la même colonne, le widget c ne reçoit aucun padx déclaré et pourtant le widget est autant décalé que a.

Gestionnaire grid : l'option sticky

Partons de l'interface suivante :

```
from tkinter import *

root=Tk()

a=Label(root, text="A", bg='red', width=20, height=5)
a.grid(row=0, column=0)

b=Label(root, text="B", bg='lightblue', width=10)
b.grid(row=0, column=1)


c=Label(root, text="C", bg='lime',  height=5)
c.grid(row=1, column=0)

d=Label(root, text="D", bg='orange', width=5, height=2)
d.grid(row=1, column=1)

root.mainloop()
```

qui produit



et modifions-la pour observer l'effet de l'option `sticky` :

```
from tkinter import *

root=Tk()

a=Label(root, text="A", bg='red', width=20, height=5)
a.grid(row=0, column=0)

b=Label(root, text="B", bg='lightblue', width=10)
b.grid(row=0, column=1, sticky=N)


c=Label(root, text="C", bg='lime',  height=5)
c.grid(row=1, column=0, sticky=E+W)

d=Label(root, text="D", bg='orange', width=5, height=2)
d.grid(row=1, column=1, sticky=NE)

root.mainloop()
```

qui produit

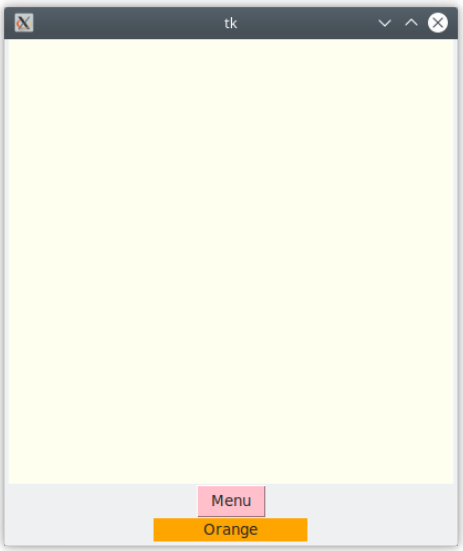
Dans l'option `sticky`, on indique des directions nord, sud, etc mais écrites en anglais et abrégées en une seule lettre capitale, par exemple `W` pour la direction ouest. On peut aussi utiliser des combinaisons comme NE (pour nord est) ou encore `N+E`. L'effet de sticky est de **coller** le widget dans la direction indiquée, avec un éventuel effet de **remplissage par étirement** de la zone libre de la cellule. L'option `N+E+S+W` est de remplir toute la cellule.

Gestionnaire pack

Le gestionnaire pack est très direct d'utilisation et est très employé pour les interfaces très simples comportant peu de widgets et dont l'organisation n'est pas cruciale :

```
from tkinter import *

root=Tk()
cnv=Canvas(root, width=400, height=400, background="ivory")
btn = Button(root, text="Menu", bg="pink")
lbl=Label(root, text="Orange", bg="orange", width=15)

cnv.pack()
btn.pack()
lbl.pack()

root.mainloop()
```

ce qui produit



et qui a pour effet d'empiler (*pack*) les widgets les uns sur les autres, bord à bord. Par défaut, on voit que les widgets sont alignés de haut en bas.

S'il s'agit d'obtenir des **alignements précis**, le gestionnaire pack est **peu approprié** et les placements ne sont pas toujours faciles à réaliser.

La méthode pack dispose en outre de plusieurs options pour affiner le placement des widgets. Les principales options sont les suivantes :

Option	Caractéristiques	Valeurs possibles	Valeur par défaut
side	Le côté de la **zone restante** sur lequel le widget va s'appuyer	TOP, BOTTOM, RIGHT, LEFT	TOP
fill	Remplissage vertical ou horizontal	X, Y, BOTH, NONE	NONE
expand	Expansion verticale ou horizontale dans la **fenêtre maîtresse**	True, False	False
anchor	Ancrage du widget dans l'espace qui lui reste	9 ancres possibles : CENTER, N, NE, E, etc	CENTER
padx	Marge (en pixels) horizontale à l'extérieur et de part et d'autre du widget	entier positif	O
pady	Marge (en pixels) verticale à l'extérieur et de part et d'autre du widget	entier positif	O

Gestionnaire pack : l'option side

Au fur et à mesure que des widgets sont placés avec le gestionnaire `pack`, une zone à remplir est redéfinie.

Quand on place l'option `side` à un widget, par exemple `side=BOTTOM`, cela signifie que le widget va être comme collé à la partie inférieure de l'espace restant. En particulier, une fois le widget placé, la partie inférieure du widget ne fera plus partie de l'espace restant.

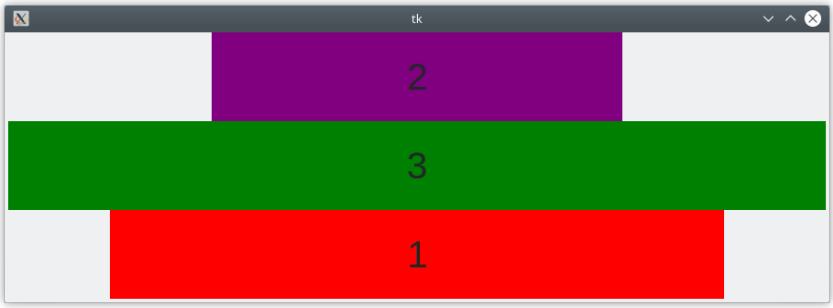
Par défaut, un widget est placé avec l'option `side=TOP` et cela explique pourquoi, par défaut, les widgets sont empilés de haut en bas.

Voici une succession de trois placements (les numéros sont les numéros d'ordre d'apparition dans le code source) :

```
from tkinter import *

root=Tk()

Label(bg="red", width=30, height=2, text="1",
      font="arial 30").pack(side=BOTTOM)
Label(bg="purple", width=20, height=2, text="2",
      font="arial 30").pack()
```

```
Label(bg="green", width=40, height=2, text="3",
      font="arial 30").pack()
root.mainloop()
```

ce qui produit :



— Le widget rouge est collé en bas (`side=BOTTOM`).
— Le widget mauve a un placement par défaut, donc il va se placer en haut (`TOP`) de la zone restante sonc au-dessus du widget rouge.
— Le widget vert est lui aussi placé avec l'option par défaut `side=TOP` donc dans la partie supérieure de la zone restante. Le widget mauve étant collé tout en haut de la fenêtre, le vert ne peut pas s'y placer. La zone restante est coincée au-dessus du widget rouge et sous le widget mauve, ce qui explique le placement.

Faisons trois nouveaux placements :

```
1  from tkinter import *
2
3  root=Tk()
4
5  Label(bg="red", width=30, height=2, text="1", font="arial 30").pack(side=BOTTOM)
6  Label(bg="purple", width=20, height=2, text="2", font="arial 30").pack()
7  Label(bg="green", width=40, height=2, text="3", font="arial 30").pack()
8  Label(bg="pink", width=15, height=2, text="4", font="arial 30").pack(side=RIGHT)
9  Label(bg="orange", width=25, height=2, text="5", font="arial 30").pack(side=LEFT)
10 Label(bg="lightblue", width=25, height=2, text="6", font="arial 30").pack()
11 Label(bg="lavender", width=5, height=2, text="7", font="arial 30").pack(side=RIGHT)
12 root.mainloop()
```

ce qui donne :

— Ligne 8 : le widget rose n°4 est placé à droite donc collé à droite de la fenêtre et sous les widgets n°2 et 3
— Ligne 9 : le widget orange n°5 est placé à gauche donc collé à gauche de la fenêtre et sous les widgets n°2 et 3
— Ligne 10 : le widget bleu clair n°6 est placé en haut de la zone restante donc collé sous le widget n°3
— Ligne 11 : le widget lavande n°7 est placé à droite donc collé à gauche du widget n°4 (qui lui était collé à droite de la fenêtre).

On remarque que la progression se fait de l'extérieur vers l'intérieur.

Pour placer côte à côte de la gauche vers la droite, utiliser l'option `side=LEFT` : chaque widget sera placé à gauche du précédent. Exemple :

```
from tkinter import *

root=Tk()

Label(bg="red", width=5, height=2, text="1", font="arial 30").pack(side=LEFT)
Label(bg="purple", width=5, height=2, text="2", font="arial 30").pack(side=LEFT)
Label(bg="green", width=5, height=2, text="3", font="arial 30").pack(side=LEFT)

root.mainloop()
```

ce qui donne :

Gestionnaire pack : l'option fill

L'option fill permet d'étendre le widget dans sa zone de placement. On donne l'option sous la fome `fill=v` ou `v` vaut

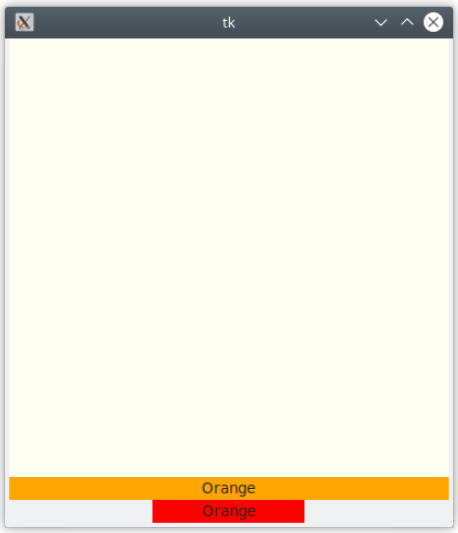
— `X` pour une expansion dans le sens horizontal,
— `Y` dans le sens vertical
— `NONE` pour une absence d'expansion (valeur par défaut)
— `BOTH` pour une expansion dans toutes les directions

Par exemple, le code suivant

```
1 from tkinter import *
2
3 root=Tk()
4 cnv=Canvas(root, width=400, height=400, background="ivory")
```

```
lbl1=Label(root, text="Orange", bg="orange", width=15)
lbl2=Label(root, text="Orange", bg="red", width=15)

cnv.pack()
lbl1.pack(fill=X)
lbl2.pack()

root.mainloop()
```

produit :



Le label orange a été élargi à sa gauche et à sa droite, le label rouge lui n'est pas étendu.

Gestionnaire pack : l'option expand

L'option expand permet d'étendre le widget par rapport à la fenêtre maîtresse. On donne l'option sous la fome expand=v ou v vaut `True` ou `False`. Par défaut, l'option est placée à `False`. S'il ne reste pas de place dans la fenêtre maîtresse, l'option est sans action.
Voici un exemple :

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

Label(root, text='Label 1', bg='green', height=10).pack(side='left')
Label(root, text='Label 2', bg='lightblue').pack(fill=Y, expand=False)

root.mainloop()
```

produit :

Malgré l'option `fill=Y` (ligne 5), l'espace disponible dans la fenêtre (et fourni à cause de la hauteur du label vert ligne 4) n'est pas utilisé.

Si on place l'option expand à `True` :

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

Label(root, text='Label 1', bg='green', height=10).pack(side='left')
Label(root, text='Label 2', bg='lightblue').pack(fill=Y, expand=True)

root.mainloop()
```

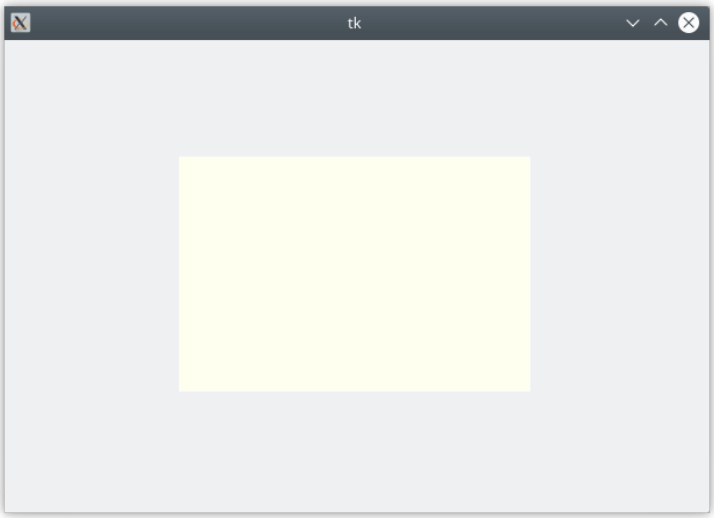
le résultat est différent, l'espace disponible dans la fenêtre est utilisé :

Gestionnaire pack : les options padx et pady

Quand on place un widget W dans un conteneur C grâce au gestionnaire pack, il y a la possibilité de placer une « marge » autour de W dans C : cela s'obtient avec les arguments nommés padx et pady :

```
1 from tkinter import Tk, Canvas
2
3 root=Tk()
4 cnv=Canvas(root, width= 300, height=200, bg="ivory")
5 cnv.pack(padx=150, pady=100)
```

```
6 root.mainloop()
```



`padx=150` désigne une marge horizontale de 150 pixels à gauche et à droite du canevas dessiné. Et `pady=100` désigne une marge verticale de 100 pixels en haut et en bas du canevas

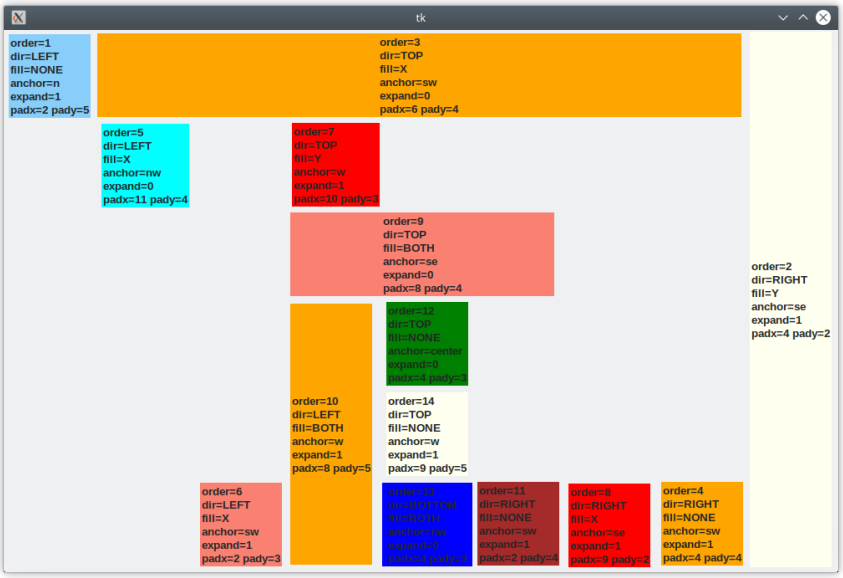
L'algorithme de placement `pack`

L'algorithe complet de placement de pack est assez complexe lorsqu'on prend en compte toutes les options possibles. Dans les documentations, les descriptions de l'algorithme sont assez rares.

Une description très minutieuse en est faite dans la documentation officielle de la bibliothèque Tk : the packer algorithm.

Une description plus lisible mais moins détaillée est fournie par Fredrik Lundh qui suggère d'imaginer un domaine fermé entouré par une frontière élastique et que l'on va remplir par des widget en s'appuyant sur un bord du domaine élastique.

Pour mieux observer les placements opérés par le gestionnaire pack, voici le résultat d'une interface graphique générée automatiquement et aléatoirement et qui utilise les différentes options possibles :



Le code source correspondant est :

```
from tkinter import *

fen=Tk()

Label(fen,
      text='order=1\ndir=LEFT\nfill=NONE\nanchor=n\nexpand=1\npadx=2 pady=5',
      font='Arial 10 bold', bg='light sky blue', justify=LEFT).pack(
          side=LEFT, padx=2, pady=5, fill=NONE, anchor='n', expand=1)
Label(fen,
      text='order=2\ndir=RIGHT\nfill=Y\nanchor=se\nexpand=1\npadx=4 pady=2',
      font='Arial 10 bold', bg='ivory', justify=LEFT).pack(
          side=RIGHT, padx=4, pady=2, fill=Y, anchor='se', expand=1)
Label(fen,
      text='order=3\ndir=TOP\nfill=X\nanchor=sw\nexpand=0\npadx=6 pady=4',
      font='Arial 10 bold', bg='orange', justify=LEFT).pack(
          side=TOP, padx=6, pady=4, fill=X, anchor='sw', expand=0)
Label(fen,
      text='order=4\ndir=RIGHT\nfill=NONE\nanchor=sw\nexpand=1\npadx=4 pady=4',
      font='Arial 10 bold', bg='orange', justify=LEFT).pack(
          side=RIGHT, padx=4, pady=4, fill=NONE, anchor='sw', expand=1)
Label(fen,
      text='order=5\ndir=LEFT\nfill=X\nanchor=nw\nexpand=0\npadx=11 pady=4',
      font='Arial 10 bold', bg='cyan', justify=LEFT).pack(
          side=LEFT, padx=11, pady=4, fill=X, anchor='nw', expand=0)
Label(fen,
      text='order=6\ndir=LEFT\nfill=X\nanchor=sw\nexpand=1\npadx=2 pady=3',
      font='Arial 10 bold', bg='salmon', justify=LEFT).pack(
          side=LEFT, padx=2, pady=3, fill=X, anchor='sw', expand=1)
```

```
29 Label(fen,
30       text='order=7\ndir=TOP\nfill=Y\nanchor=w\nexpand=1\npadx=10 pady=3',
31       font='Arial 10 bold', bg='red', justify=LEFT).pack(
32           side=TOP, padx=10, pady=3, fill=Y, anchor='w', expand=1)
33 Label(fen,
34       text='order=8\ndir=RIGHT\nfill=X\nanchor=se\nexpand=1\npadx=9 pady=2',
35       font='Arial 10 bold', bg='red', justify=LEFT).pack(
36           side=RIGHT, padx=9, pady=2, fill=X, anchor='se', expand=1)
37 Label(fen,
38       text='order=9\ndir=TOP\nfill=BOTH\nanchor=se\nexpand=0\npadx=8 pady=4',
39       font='Arial 10 bold', bg='salmon', justify=LEFT).pack(
40           side=TOP, padx=8, pady=4, fill=BOTH, anchor='se', expand=0)
41 Label(fen,
42       text='order=10\ndir=LEFT\nfill=BOTH\nanchor=w\nexpand=1\npadx=8 pady=5',
43       font='Arial 10 bold', bg='orange', justify=LEFT).pack(
44           side=LEFT, padx=8, pady=5, fill=BOTH, anchor='w', expand=1)
45 Label(fen,
46       text='order=11\ndir=RIGHT\nfill=NONE\nanchor=sw\nexpand=1\npadx=2 pady=4',
47       font='Arial 10 bold', bg='brown', justify=LEFT).pack(
48           side=RIGHT, padx=2, pady=4, fill=NONE, anchor='sw', expand=1)
49 Label(fen,
50       text='order=12\ndir=TOP\nfill=NONE\nanchor=center\nexpand=0\npadx=4 pady=3',
51       font='Arial 10 bold', bg='green', justify=LEFT).pack(
52           side=TOP, padx=4, pady=3, fill=NONE, anchor='center', expand=0)
53 Label(fen,
54       text='order=13\ndir=BOTTOM\nfill=BOTH\nanchor=nw\nexpand=0\npadx=4 pady=3',
55       font='Arial 10 bold', bg='blue', justify=LEFT).pack(
56           side=BOTTOM, padx=4, pady=3, fill=BOTH, anchor='nw', expand=0)
57 Label(fen,
58       text='order=14\ndir=TOP\nfill=NONE\nanchor=w\nexpand=1\npadx=9 pady=5',
59       font='Arial 10 bold', bg='ivory', justify=LEFT).pack(
60           side=TOP, padx=9, pady=5, fill=NONE, anchor='w', expand=1)
61
62 fen.mainloop()
```

Ce code crée une suite de labels, par exemple, lignes 29-32, le label n° 7. Des options aléatoires sont fournies à la méthode `pack`, par exemple, pour le label n°7, regarder ligne 32. Le label rend visibles ces options, cf. ligne 30 pour le label n°7.

Bien comprendre que le remplissage se fait de manière successive et va de l'extérieur vers l'intérieur. Ainsi, `side=RIGHT` va placer le widget courant dans la partie droite de l'intérieur de la zone restante. Par ailleurs, l'ajout d'un widget peut modifier le placement relatif des précédents.

Un fichier Python a servi à générer le code ci-dessus dont voici le code :

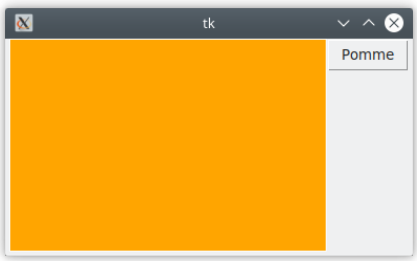
```
from random import randrange

COLORS=["red", "blue", "gray", "orange", "brown",
        "magenta", "green", "salmon", "ivory",
        "cyan", 'sky blue', 'light sky blue']
DIR=["RIGHT", "LEFT", "TOP", "BOTTOM"]
```

```
cnv = """\
Label(fen,
      text='%s',
      font='Arial 10 bold',
      bg='%s', justify=LEFT).pack(
      side=%s, padx=%s, pady=%s, fill=%s, anchor='%s', expand=%s)"""

FILL=["NONE", "NONE", "BOTH", "X", "Y"]
ANCHOR="NW N NE E SE S SW W CENTER".lower().split()
JUSTIFY="LEFT CENTER RIGHT".split()
EXPAND="0 1".split()

L=[]

for  i in range(30):
    b=COLORS[randrange(len(COLORS))]
    c= DIR[randrange(4)]
    d=FILL[randrange(len(FILL))]
    e=ANCHOR[randrange(len(ANCHOR))]
    f="%s" %JUSTIFY[randrange(len(JUSTIFY))]
    x=EXPAND[randrange(2)]

    padx=randrange(2,12)
    pady=randrange(2,6)
    options=(r"order=%s\ndir=%s\nfill=%s\n"\
             r"anchor=%s\nexpand=%s\npadx=%s pady=%s")
    a= options%(i+1, c, d, e, x, padx, pady)
    L.append(cnv %(a,b,c,padx, pady, d,e, x))

code=["from tkinter import *"]
for i in range(1,15):
    code.append("# Fenêtre n°%s" %i)
    code.append("fen=Tk()\n")
    code.append("\n".join(L[:i]))
    code.append("#-----------------------------------\n\n")
code.append("fen.mainloop() ")

open("demo_pack.py", "w").write('\n'.join(code))
```

Ce code génère un fichier de démonstration `demo_pack.py`. L'exécution de ce fichier crée un certain de nombre de fenêtres (ici 14) chacune utilisant la méthode `pack`. La fenêtre visible ci-dessus est l'une d'entre.

Centrer un widget et méthode pack

Soit l'interface suivante :



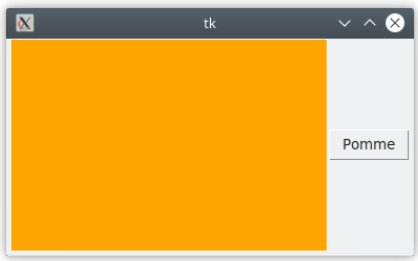
où le widget de droite n'est pas verticalement centré. Le code est :

```
1 from tkinter import *
2
3 root=Tk()
4
5 cnv=Canvas(root, width=300, height=200, bg="orange")
6 cnv.pack(side='left')
7
8 bouton=Button(root, text="Pomme")
9 bouton.pack()
10
11 root.mainloop()
```

L'option `side=left` de la ligne 6 assure que les widgets sont côte à côte. Le 2e widget est placé par défaut avec l'option `side=TOP`. Cela explique qu'il ne soit pas centré. Pour obtenir un centrage, il suffit de placer l'option `side="left"` ou `side="right"` :

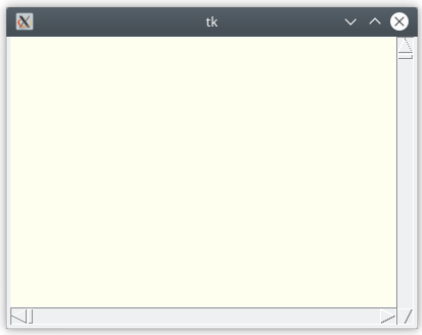
```
1 from tkinter import *
2
3 root=Tk()
4
5 cnv=Canvas(root, width=300, height=200, bg="orange")
6 cnv.pack(side='left')
7
8 bouton=Button(root, text="Pomme")
9 bouton.pack(side="left")
10
11 root.mainloop()
```

ce qui produit :



Placer en grille avec la méthode pack

La méthode pack n'est pas adaptée à présenter une interface dont les widgets sont placés en grille.
Typiquement, soit à réaliser le motif suivant



où un canevas est entouré, en bas et à droite de barres de défilement.
Ce motif se réalise très facilement avec la méthode `grid` :

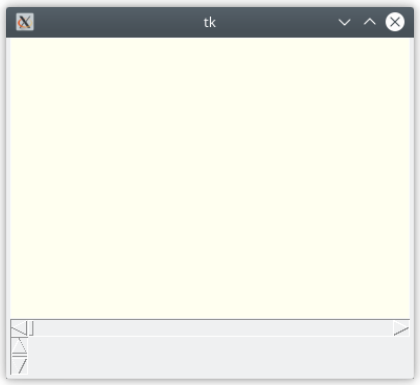
```
from tkinter import *

root=Tk()

Canvas(root, bg="ivory").grid(row=0, column=0)
Scrollbar(root, orient="vertical").grid(row=0, column=1, rowspan=2, sticky=NS)
Scrollbar(root, orient="horizontal").grid(row=1, column=0, sticky=EW)

root.mainloop()
```

Avec la méthode pack, c'est moins immédiat. On peut se dire qu'on empile le canevas sur la barre de défilement horizontale puis on place à droite ou gauche la barre de défilement vertical :

```
from tkinter import *

root=Tk()
```

```
Canvas(root, bg="ivory").pack()
Scrollbar(root, orient="horizontal").pack(fill=X)
Scrollbar(root, orient="vertical").pack(side=LEFT, fill=Y)

root.mainloop()
```

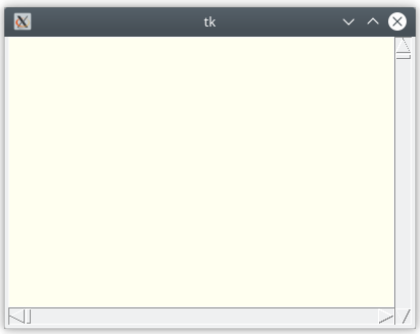
mais voilà ce que ça donne :



et ça ne marche pas non plus si `side=RIGHT`.
Une méthode qui fonctionne est la suivante :

```
from tkinter import *

root=Tk()

Scrollbar(root, orient="vertical", command=None).pack(side=RIGHT, fill=Y)
Canvas(root, bg="ivory").pack()
Scrollbar(root, orient="horizontal", command=None).pack(fill=X)

root.mainloop()
```

qui produit :



5 Quelques techniques générales

Version de Tkinter utilisée

Pour connaître la version de Tk que vous utilisez, tapez dans une console le code suivant :

```
>>> from tkinter import *
>>> TkVersion
8.6
>>>
```

Le code ci-dessus est obtenu pour les versions 3.4 à 3.7 de Python.

Tk a évolué assez sensiblement à sa version 8.5 (au moins 2003) : thèmes natifs sous Windows et Mac Os X, création de nouveaux widgets, antialising sous Linux pour le fenêtrage X-11, cf. Tcl/Tk 8.5

La version 8.6 de Tk (à partir de 2008) se distingue de la version 8.5 en ce qu'elle supporte désormais le format d'image png, cf. Tcl/Tk 8.6. En 2019, la dernière version de Tk est la version 8.6.9 et il existe une version alpha 8.7.

Certaines installation sous Mac Os X ont parfois des retards d'installation des dernières versions de Tk, cf. IDLE and tkinter with Tcl/Tk on macOS. À partir de la version 3.7 de Python, il semblerait que la version de Tcl/Tk qui sert de base à Tkinter soit 8.6, laquelle prend en charge le format png, cf. par exemple le bug tracker.

Importer tkinter

D'abord, le nom du module sous Python 3 est `tkinter` en minuscule. Le nom `Tkinter` s'applique à Python 2 et ne sera pas valide sous Python 3 (rappel : Python 2 n'est plus maintenu en 2020).

En Python, d'une manière générale, il n'est pas recommandé d'importer un module avec la syntaxe suivante :

```
from this_module import *
```

car cela peut créer des collision de noms. Concernant l'importation de Tkinter par :

```
from tkinter import *
```

elle est souvent déconseillée. Elle ajoute à l'espace de noms de Python environ 140 noms dont certains relativement communs comme

N	TOP	WORD	BOTTOM
E	YES	enum	Button
S	sys	Entry	CENTER
W	CHAR	FALSE	INSIDE
X	Grid	Image	NORMAL
Y	LEFT	Label	Widget
NO	Menu	PAGES	Message
ON	Misc	Place	VERTICAL
re	NONE	RIGHT	mainloop

```
    ALL     |      Pack     |      ROUND     |      constants
    END     |      TRUE     |      SOLID
    OFF     |      Text     |      UNITS
```

et que l'on risque d'écraser accidentellement. Par ailleurs, certains considèrent que l'usage libre des noms de la bibliothèque nuit à la lisibilité du code (car on ne sait pas quel nom provient de Tkinter et quel nom provient d'un autre module).

Les programmes Tkinter peu élaborés importent peu de noms, donc ce genre de programme pourrait se contenter d'une importation du type

```
from tkinter import Tk, Canvas, Button
```

si par exemple vous avez juste besoin d'une fenêtre `Tk`, des widgets `Canvas` et `Button`.

Toutefois, des besoins plus importants nécessiteraient parfois d'importer beaucoup plus de noms.

Certains utilisateurs et connaisseurs de Tkinter proposent d'écrire plutôt :

```
import tkinter
```

ce qui oblige à systématiquement préfixer, par exemple `tkinter.Canvas` et impose des noms relativement long ; d'où la simplification suivante

```
import tkinter as tk
```

ce qui permet d'accéder au Canvas avec la syntaxe assez légère `tk.Canvas`. C'est ainsi que procède le module turtle dans CPython.

Enfin, il faut relativiser ces mises-en-garde : beaucoup de code, y compris du code-source disponible dans **CPython** (l'implémentation officielle et largement majoritaire de Python), utilise, à des degrés divers, une importation de type `from tkinter import *`. Par exemple, c'est le cas de la plupart des modules de IDLE, comme le fichier editor.py. Pour du code destiné à l'apprentissage, du code de petits jeux à caractère pédagogique, ou pour des essais de fonctionnalités, ce type d'importation présente peu d'inconvénients et présente même des avantages pour le débutant.

Le codage des couleurs sous Tkinter

On est amené à utiliser des couleurs sous Tkinter dans de nombreuses situations, par exemple, donner une couleur de fond à un bouton ou à un canevas. Il existe deux codages des couleurs sous Tkinter :

- **nom de couleur** : les couleurs standard du html, peuvent être appelées par leur nom, typiquement des noms courants *red* ou d'autres comme *ivory* ;
- **codage hexadécimal** : on fournit une chaîne hexadécimale RGB commençant par le caractère `#` ; il existe plusieurs formes, la plus simple étant un code à 3 chiffres hexadécimaux, du type `"#5fc"` où chacune des trois composantes R, G et B est représentée par un seul chiffre hexadécimal parmi les caractères `"0"`, `"1"`, etc jusqu'à `"F"` (on peut aussi utiliser des lettres minuscules pour les lettres hexadécimales). Ce codage permet de représenter $16^3 = 4096$ couleurs.

Ainsi, typiquement, on pourra rencontrer des codes comme :

```
annonce=Label(height=5, width= 15, text="BONJOUR !", bg='ivory')
cnv=Canvas(root, width=200, height=200, bg="#5fc")
```

Le codage hexadécimal permet aussi d'utiliser deux chiffres hexadécimaux par couleur ; par exemple, la couleur turquoise sera codée par `"#40e0d0"`. On peut même utiliser trois chiffres hexadécimaux par composante RGB.

On trouvera un très pratique nuancier par nom de couleurs HTML dans le message Colour chart for Tkinter and Tix Using Python sur StackOverFlow qui produit le résultat suivant :

Nuancier Tkinter (cliquer pour voir les noms de couleurs).

Le code Tkinter produisant ce nuancier est disponible dans le message.

A toutes fins utiles, voici une fonction renvoie une chaîne de couleur Tkinter depuis un triplet `(r, g, b)` :

```
def rgb_10to16(r,g,b):
    return "#%.02x" %r + "%.02x" %g + "%.02x" %b

# Royal Fuschia
print(rgb_10to16(202, 44, 146))
```

```
#ca2c92
```

Pour convertir depuis ou vers du format HSV, utiliser le module standard colorsys.

Donner un canal alpha à un objet

Le canvas de Tkinter n'accepte pas de canal alpha, cf. ce message sur stackoverflow.

Les polices

Dans un label, un bouton, ou encore pour écrire du texte dans un canevas, on dispose de l'option `font` permattant de définir une fonte. Elle admet deux syntaxes :

```
font="Times 12 bold"
font=("Times", 12, "bold")
```

La 2e est utile si le nom de la police contient des espaces.

Voici un exemple d'utilisation (parmi d'autres) :

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

Label(root, text="ABCDEFGHIJKLMNOPQRSTUVWXYZ",
      font=('Arial', 12)).pack(side=TOP, anchor="w")
Label(root, text="ABCDEFGHIJKLMNOPQRSTUVWXYZ",
      font=("Times New roman", 12)).pack(side=TOP, anchor="w")
Label(root, text="ABCDEFGHIJKLMNOPQRSTUVWXYZ",
      font=("Times New roman", 12)).pack(side=TOP, anchor="w")
Label(root, text="ABCDEFGHIJKLMNOPQRSTUVWXYZ",
      font=("Courier", 18)).pack(side=TOP, anchor="w")
Label(root, text="012345678901234567890123456789",
      font=("Courier", 18)).pack(side=TOP, anchor="w")

root.mainloop()
```

qui affiche

Exemples de polices

Catégorie de polices

Catégorie	Remarque	Exemples	Remarques
Monospace	Chaque lettre a même largeur. Écrire du code. Alignements verticaux.	Courier, Deja Vu Sans Mono, Monaco, Consolas, Fixedsys.	Courier : laide dit-on Monaco : vient de mac, moins laide que courier, pas toujours mono.
Sans empâtement	Moins ornée. Adaptée aux écrans.	Arial, Deja Vu Sans, Helvetica, Ubuntu, Verdana	Arial et Helvetica : très proches. Helvetica : commune. Verdana : Microsoft.
À empâtement		Times New Roman, Palatino,	Très courant

Pour obtenir la liste des polices utilisables :

```
1 from tkinter import font, Tk
2 Tk()
3 print(*font.families(), sep='\n')
```

Fontes disponibles sous Ubuntu

Ci-dessous, quelques fontes disponibles sous une Ubuntu de base :

```
Century Schoolbook L
Ubuntu
DejaVu Sans Mono
Dingbats
DejaVu Sans
DejaVu Serif
Courier 10 Pitch
```

Pour utiliser une autre police :

```
from tkinter import font as tkfont, Tk, Canvas

root = Tk()
canvas = Canvas(root, width=500, height=500, bg="ivory")
canvas.pack()


my_font=tkfont.Font(family="cmsy10")

text=canvas.create_text(420,100,fill="black", font = my_font,
                        text="Portez ce vieux whisky au juge blond qui fume")
root.mainloop()
```

Les ancres

Les ancres permettent d'ancrer un widget ou du texte au nord, au sud, etc d'une certaine zone.
Il y a 9 ancres possibles :

```
NW   N    NE
W  CENTER  E
SW   S    SE
```

Elle s'utilisent en tant qu'option dans différents contextes, pour ajuster :

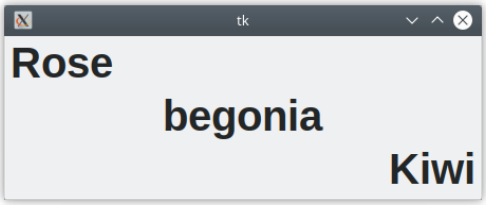
— un widget avec le gestionnaire pack

— placer du texte dans certains widgets (Canvas, Button, etc)

L'utilisation d'une ancre (anchor) permet d'ajuster un objet dans un contexte, le placer plus haut, plus bas, etc.
Voici un exemple d'utilisation :

```
1  from tkinter import *
2
3  root = Tk()
4
5  lbl1 = Label(root, text="Rose", font='Arial 30 bold', anchor=NW, width=20)
6  lbl2 = Label(root, text="begonia", font='Arial 30 bold', width=20)
7  lbl3 = Label(root, text="Kiwi", font='Arial 30 bold', anchor='se', width=20)
8  lbl1.pack()
9  lbl2.pack()
10 lbl3.pack()
11 root.mainloop()
```

qui produit :



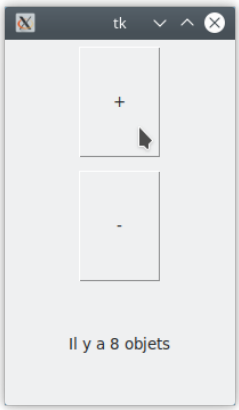
Noter aux lignes 5 et 7 les deux syntaxes possibles pour déclarer l'ancre. Si on utilise une chaîne (ligne 7), il faut l'écrire en minuscule. Dans l'autre cas, pas de guillemets, il s'agit de noms que l'on doit importer de Tkinter.

Modifier dynamiquement une variable de contrôle

La notion de variable de contrôle n'est absolument pas indispensable quand on débute en Tkinter. On peut écrire de nombreux programmes variés sans avoir besoin d'en utiliser.

Tkinter possède un mécanisme original pour mettre à jour automatiquement des variables définies comme options de certains widgets (comme des boutons radio, des entrées ou des labels).

Dans l'interface ci-dessous,



si on clique sur l'un des boutons `+` ou `-`, un compteur représentant des objets est mis-à-jour **automatiquement** dans un label sous les deux boutons.

Le nombre total d'objets est défini de la manière suivante :

```
total=IntVar()
```

Ici, `total` n'est pas exactement un entier mais ce qu'on appelle une Variable Tkinter, ou encore une variable dynamique ou encore une variable de contrôle. Cette variable stocke un entier que l'on peut récupérer par un appel `total.get()`. On peut aussi modifier la valeur interne avec la méthode `set`, par exemple `total.set(42)`.

Le code correspondant de l'application ci-dessus est :

```
1  from tkinter import *
2
3  root=Tk()
4  total=IntVar()
5
6  def incr():
7      total.set(total.get()+1)
8
9  def decr():
10     total.set(total.get()-1)
11
12
13 btn_plus=Button(root, height=5, width=5, text="+", command=incr)
14 btn_plus.pack(padx=5, pady=5)
15 btn_moins=Button(root, height=5, width=5, text="-", command=decr)
16 btn_moins.pack(padx=5, pady=5)
17
18 msg=Label(root, height=5, width=20, textvariable=total)
19 msg.pack(padx=5, pady=5)
```

```
20
21 root.mainloop()
```

— Ligne 4 : une variable de contrôle de type `IntVar` est définie. Elle représente une variable entière qui compte un certain nombre d'objets (les objets comptés sont ici imaginaires, c'est juste pour l'exemple). Par défaut, la valeur interne de `total` est 0.

— Lignes 13-26 : on définit deux boutons + et -. Quand on clique sur le bouton +, on appelle la fonction `incr` qui se charge d'incrémenter la variable de contrôle `total` (lignes 6-7). De même pour le bouton - et la fonction `decr`.

— Lignes 7 et 11 : noter que le contenu de la variable de contrôle est capturé avec la méthode `get` de la variable.

— Lignes 7 et 10 : pour mettre à jour (ou parfois initialiser) une variable de contrôle, on utilise sa méthode `set` et on lui indique la nouvelle valeur.

— La mise à jour est automatique : le clic sur + par exemple incrémente (ligne 7) la variable de contrôle `total` avec la méthode `set` ce qui change automatiquement le contenu du label.

En pratique, une variable `IntVar` est toujours inialisée à 0. Une autre façon d'initialiser (à 42 par exemple) aurait été d'écrire :

```
total = Intvar(value=42)
```

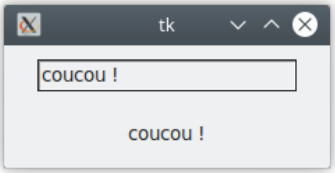
L'intérêt d'une variable de contrôle est de pouvoir mettre à jour **automatiquement** certains paramètres de différents widgets. On trouvera des détails dans la documentation de effbot.

On reconnait la nécessité d'utiliser des variables de contrôle Tkinter lorsqu'un widget a une option qui s'appelle `variable` ou `textvariable`.

Dans de beaucoup de codes, on observe une sur-utilisation des variables Tkinter. Dans de nombreuses situations statiques, elles ne sont pas utiles, comme le confirme Bryan Oakley sur StackOverflow. Elle ne servent vraiment que lorsque des objets variables définis par des événements non contrôlables sont partagées dynamiquement par plusieurs widgets. Je donne un cas typique est dans Exemple d'utilisation de StringVar. Elles sont parfois utiles aussi pour mettre à jour automatiquement un grand nombre d'objets. Certains widgets, pour être convenablement contrôlés, doivent absolument utiliser une variable de contrôle, typiquement le widget entrée, le widget bouton-radio ou le widget case à cocher.

Exemple d'utilisation de StringVar

`StringVar` est une classe propre à Tkinter et qui encapsule une variable option d'un widget et permettant une mise à jour automatique du widget. Voici un exemple avec un widget de type `Label` :



produit par le code :

```
1 from tkinter import *
2
3 root=Tk()
4 msg=StringVar()
5 entree=Entry(root, textvariable=msg)
6 entree.pack( padx=20, pady=10)
7
8 lbl=Label(root, textvariable=msg)
9 lbl.pack(padx=20, pady=10)
10
11 root.mainloop()
```

— Ligne 4 : la variable `msg` est de type `StringVar`.

— Lignes 5 et 8 : chaque fois que le champ de l'entrée est modifié, le texte entré est capturé dans `msg` (ligne 5) et il est automatiquement mis à jour dans le label (ligne 8).

Curseur : placer le code dans des fonctions

Ci-dessous, on explique comment placer du code utilisant des widgets dans des fonctions et les conséquences que cela peut avoir.

Reprenons le code de démo du slider :

```
1 from tkinter import *
2
3 root = Tk()
4 cnv = Canvas(root, width=400, height=400)
5 cnv.pack()
6
7 old=None
8
9 def rayon(r):
10     global old
11     r=int(r)
12     cnv.delete(old)
13     old=cnv.create_oval(200-r,200-r,200+r, 200+r)
14
15 curseur = Scale(root, orient = "horizontal",
16                  command=rayon, from_=0, to=200)
17 curseur.pack()
18
19 root.mainloop()
```

Le code est hors fonction sauf la fonction `rayon` qui elle est obligatoire puisque le slider a besoin d'une **fonction** pour agir. L'avantage d'avoir du code hors fonction est que la fonction `rayon` a accès à des données comme le canevas que, de toute façon, on ne peut pas transmettre à la fonction `rayon` car son paramètre `r` est imposé par Tkinter lui-même. Et pour la même raison, l'identifiant `old` du rayon du précédent cercle ne peut être transmis à la fonction via un paramètre. Noter que le déclarateur `global` sert uniquement à la modification de la variable globale `old`.

Essayons d'écrire le code dans des fonctions :

```
1  from tkinter import *
2
3
4  def rayon(r):
5      global old
6      r=int(r)
7      cnv.delete(old)
8      old=cnv.create_oval(200-r,200-r,200+r, 200+r)
9
10
11 def demo():
12     root = Tk()
13     cnv = Canvas(root, width=400, height=400)
14     cnv.pack()
15     curseur = Scale(root, orient = "horizontal",
16                     command=rayon, from_=0, to=200)
17     curseur.pack()
18
19     root.mainloop()
20
21     old=None
22 demo()
```

Ce code ne va pas fonctionner quand on déplace le curseur car `cnv` (ligne 13) est inconnu de la fonction `rayon` (ligne 5). Par ailleurs, le code ci-dessus ne permet pas de modifier la largeur du canevas (ligne 13) et cette largeur est en outre nécessaire à la fonction `rayon` pour placer le centre du cercle (ligne 8). Comme la fonction `rayon` ne peut prendre qu'un seul paramètre qui nous est imposé (la valeur lue sur le curseur), il est obligatoire de passer par une déclaration globale (lignes 5 et 21). Toutefois, on peut la minimiser, de la manière suivante par exemple :

```
1  from tkinter import Tk, Canvas, Scale
2
3  def rayon(r):
4      cnv, side, old=data
5      r=int(r)
6      m=side/2
7      cnv.delete(old)
8      data[2]=cnv.create_oval(m-r,m-r,m+r, m+r)
9
10 def demo(side):
11     global data
12     root = Tk()
13     cnv = Canvas(root, width=side, height=side)
14     cnv.pack()
15     old=None
16     curseur = Scale(root, orient = "horizontal",
17                     command=rayon, from_=0, to=200)
18     curseur.pack()
```

```
19      data=[cnv, side, old]
20      root.mainloop()
21
22  side=400
23  demo(side)
```

Lorsque `demo` est lancée, une variable globale `data` est créée. C'est une liste (cf. ligne 19) et on y place toutes les données qui seront nécessaires à la fonction `rayon` et qu'on ne peut lui donner en paramètres.

Lorsqu'un nouveau cercle est créé, ne pas oublier d'enregistrer son id dans `data`, à l'indice 2, cf. lignes 8 et 19.

Noter (ligne 1) qu'a juste été importé le strict nécessaire de Tkinter.

Au total, le code ainsi obtenu est beaucoup plus lisible que le code initial.

Eviter `global` avec une clôture

Certains pourront regretter que le code précédent utilise le déclarateur `global`. Si on veut l'éviter, il faut que la fonction `rayon`, qui admet un unique argument, ait accès à `data`. Pour cela, il suffit de créer une « clôture » ainsi que le montre le code suivant :

```
1  def make_move(data):
2      cnv, side, old=data
3      def rayon(r):
4          r=int(r)
5          m=side/2
6          cnv.delete(data[2])
7          data[2]=cnv.create_oval(m-r,m-r,m+r, m+r)
8      return rayon
```

Un appel `make_move(data)` renvoie une fonction `rayon` prenant un unique paramètre mais cette fonction à accès à `data`. Bien noter qu'il faut écrire (ligne 6) `cnv.delete(data[2])` et non `cnv.delete(old)`. D'où le code suivant, qui évite d'utiliser `global` :

```
1  from tkinter import Tk, Canvas, Scale
2
3  def make_move(data):
4      cnv, side, old=data
5      def rayon(r):
6          r=int(r)
7          m=side/2
8          cnv.delete(data[2])
9          data[2]=cnv.create_oval(m-r,m-r,m+r, m+r)
10     return rayon
11
12 def demo(side):
13     root = Tk()
14     cnv = Canvas(root, width=side, height=side)
15     cnv.pack()
16     data=[cnv, side, None]
17     rayon=make_move(data)
```

```
18     curseur = Scale(root, orient = "horizontal",
19                     command=rayon, from_=0, to=200)
20     curseur.pack()
21     root.mainloop()
22 
23 def main():
24     side=400
25     demo(side)
26 
27 main()
```

Curseur : placer le code dans une classe

Ci-dessous, on va montrer l'avantage à utiliser des classes pour gérer une interface graphique Tkinter. Reprenons pour cela le code découpé en fonctions de la démo du slider :

```
1  from tkinter import Tk, Canvas, Scale
2  
3  def rayon(r):
4      cnv, side, old=data
5      r=int(r)
6      m=side/2
7      cnv.delete(old)
8      data[2]=cnv.create_oval(m-r,m-r,m+r, m+r)
9  
10 def demo(side):
11     global data
12     root = Tk()
13     cnv = Canvas(root, width=side, height=side)
14     cnv.pack()
15     old=None
16     curseur = Scale(root, orient = "horizontal",
17                     command=rayon, from_=0, to=200)
18     curseur.pack()
19     data=[cnv, side, old]
20     root.mainloop()
21 
22 side=400
23 demo(side)
```

Une façon d'éviter les variables globales (ligne 11) et de coder de manière plus répandue est d'utiliser une classe (ligne 3 ci-dessous) : ainsi, chaque méthode de la classe aura accès aux attributs de la classe. Ainsi, si on transforme la fonction `rayon` initiale en une méthode, on n'aura **plus besoin** de variable globale.

D'où le code suivant :

```
1 from tkinter import Tk, Canvas, Scale
2 
3 class ScaleDemo:
```

```
4
5       def __init__(self, side):
6           self.side=side
7           self.m=side/2
8           root = Tk()
9           self.cnv = Canvas(root, width=side, height=side)
10          self.cnv.pack()
11          self.old=None
12          curseur = Scale(root, orient = "horizontal",
13                          command=self.rayon, from_=0, to=200)
14          curseur.pack()
15          root.mainloop()
16
17      def rayon(self, r):
18          r=int(r)
19          m=self.m
20          self.cnv.delete(self.old)
21          self.old=self.cnv.create_oval(m-r,m-r,m+r, m+r)
22
23  side=400
24  ScaleDemo(side)
```

Noter que la mainloop reste dans la fonction `__init__` lancée par l'appel `ScaleDemo(side)`.

Le curseur peut appeler la fonction `rayon` (lignes 17-21) en appelant la méthode `rayon` qui est référencée en utilisant `self` (ligne 13). Quand le curseur est modifié par l'utilisateur, Tkinter lance un appel `rayon(r)` et comme `rayon` est une méthode de ScaleDemo, l'interpréteur Python rajoute automatiquement comme premier argument à rayon une instance de la classe (`self`). Cela explique pourquoi `r` est le 2e paramètre dans la définition ligne 17.

Par ailleurs, les variables globales ont disparu. Par exemple, à l'initialisation, le canevas est déclaré comme attribut d'instance (avec `self` ligne 9) si bien que lorsque la méthode `rayon` est appelée, elle a accès, via `self.cnv` au canevas.

Noter qu'on n'a pas eu besoin de donner au curseur le statut d'attribut d'instance (puisque `rayon` n'en a pas explicitement besoin).

La ligne 19 est juste un raccourci pour éviter d'écrire 4 fois `self.m` ligne 21.

Installer Pygame sous Windows

Pygame est une bibliothèque de jeux. Dans ce qui suit, on va expliquer comment installer Pygame sous Windows 10 et sous Linux.

Installation sous Windows 10

Je suppose que vous avez installé Python depuis le site python.org. Probablement que ce qui suit ne s'applique pas si vous avez installé Python via Anaconda auquel cas il faut effectuer ce qui va suivre depuis la ligne de commande propre à Anaconda. Je suppose aussi que vous n'utilisez pas Python sous un environnement virtuel (virtualenv).

Le principe d'installation est assez simple : on tape une ligne de commande dans un shell et cela installe automatiquement Pygame.

Ce qui suit suppose que vous travaillez avec une version 3 de Python.

Pygame serait-il déjà installé ?

Pour le savoir, ouvrir IDLE, l'éditeur par défaut de Python (Menu Démarrer > Python > IDLE). Devant le prompt Python avec les trois chevrons écrire :

```
import pygame
```

Si vous avez un message d'erreur, c'est que Pygame n'est pas installé ou pas accessible.

Le programme `pip` est-il déjà installé ?

Tout ce qui suit suppose que vous disposez du programme `pip`. Vous devez donc vérifier que `pip` est installé sur votre système.

`pip` est un programme d'installation de paquets Python. Son principe est d'aller télécharger sur un dépôt Internet un programme Python et d'installer ce programme sur votre système.

Ouvrir une ligne de commande Windows ; pour cela, taper `cmd` dans Cortana, ce qui devrait montrer une icône portant le nom d'« invite de commandes », cliquer sur l'icône et un terminal noir, avec un prompt devrait s'ouvrir. Il est encore plus simple de taper `command.exe` dans la barre de recherche de Cortana. Le terminal obtenu est un terminal système et pas un terminal propre à Python. Dans ce terminal, écrire

```
pip
```

puis appuyer sur la touche `Entrée`. Si la réponse est :

```
'pip' n'est pas reconnu en tant que commande interne
ou externe, un programme exécutable ou un fichier de commandes.
```

c'est que `pip` n'est pas installé (voir plus loin pour savoir comment y remédier).

La méthode la plus simple

Ouvrir une ligne de commande Windows ; pour cela, taper `cmd` dans Cortana, ce qui devrait montrer une icône portant le nom d'« invite de commandes », cliquer sur l'icône et un terminal noir, avec un prompt devrait s'ouvrir. Ce terminal est un terminal système et pas un terminal propre à Python. Dans ce terminal, écrire

```
pip install pygame
```

puis appuyer sur la touche `Entrée`.

`pip` est un programme d'installation de paquets Python. Son principe est d'aller télécharger sur un dépôt Internet un programme Python et d'installer ce programme sur votre système.

Si le programme `pip` est reconnu par votre système, le déroulé de l'installation devrait être retranscrit dans le terminal et en conclusion vous devriez lire la version de Pygame qui a été installée, par exemple :

```
Successfully installed pygame-1.9.3
```

Pour vérifier que Pygame est bien installé, ré-ouvrir IDLE et devant les 3 chevron du prompt Python écrire, en minuscule :

```
import pygame
```

et aucun message d'erreur ne devrait apparaître. Vous pouvez même déterminer depuis le prompt

Python la version de Pygame qui est installée, en tapant :

```
pygame.version.ver
```

ce qui devrait afficher la version, par exemple

```
'1.9.3'
```

Installation via un fichier wheel

Il se peut que la méthode précédente échoue. Vous pouvez alors télécharger sur Internet un binaire d'extension `whl` (ce qui signifie "wheel") contenant Pygame et l'installer comme ci-dessus avec `pip`. L'ensemble se fait en trois étapes :

Étape 1 : Déterminer l'adressage de votre version de Python

Quand vous téléchargez Python depuis le site `python.org`, il est disponible en version 32 bits ou 64 bits. Il ne faut pas confondre la version 32 ou 64 bits de Python avec l'adressage de votre version de Windows 10 qui elle est très probablement sous 64 bits.

Par défaut, c'est la version 32 bits qui est installée. Mais, pour connaître le mode d'adressage de votre version de Python, ouvrir IDLE et regarder tout en haut de la fenêtre, à la fin de la ligne, ce qui est indiqué, soit "32 bit (Intel)" soit "64 bit (AMD64)" ce qui vous précise votre version. Vous devriez aussi lire votre version de Python, par exemple, Python 3.6.

Étape 2 : télécharger le fichier wheel

Dans Google, taper "unofficial binaries Pygame" et cliquer sur le premier lien transmis par Google. On arrive sur le site Unofficial Windows Binaries for Python Extension Packages. Une fois sur le site, chercher la section Pygame qui contient une suite de fichiers "wheel" d'extension `whl` pour différentes versions de Pygame.

Puis télécharger le fichier correspondant à votre système. Pour illustrer la suite, je vais supposer que vous êtes sous Python 3.5 en 32 bits,et donc que vous téléchargez le fichier nommé `pygame-1.9.4-cp35-cp35m-win32.whl`, le code `35` devant être compris comme se référant à Python 3.5 et le code `win32` se référant à 32 bits ; ce fichier est un binaire contenant la version 1.9.4 de Pygame.

Étape 3 : installer le fichier wheel

- Une fois le fichier téléchargé, ouvrir le répertoire de téléchargement et débrouillez-vous pour obtenir le nom complet du dossier où se trouve le fichier. Le plus simple pour cela est de cliquer sur la barre d'adresse du dossier et de copier le nom complet du dossier, typiquement `C:\Users\Moi\Downloads`.
- Ouvrir une invite de commandes, par exemple en passant par Cortana et y tapant `cmd`. Taper `cd` dans la ligne de commandes (ce qui signifie "change directory") puis après un espace, coller dans la ligne de commandes le nom de dossier que vous aviez copié, puis taper sur la touche `Entrée` : la ligne de commande est alors placée dans le dossier où se trouve le fichier `whl` à installer.
- appeler le programme `pip` sur le fichier que vous avez téléchargé. Pour cela taper dans la ligne de commande `pip` suivi d'un espace puis indiquer le nom du fichier, dans notre exemple, c'était `pygame-1.9.4-cp35-cp35m-win32.whl`. Inutile de tout taper lettre par lettre, il suffit de taper les 4 ou 5 premières lettres et ensuite d'appuyer sur la touche TAB et l'invite de commande complètera avec le bon nom de fichier.
- Valider en appyant sur la touche `Entrée` et sauf anaomalie, cela devrait installer Pygame sur votre système. Vérifier en ouvrant IDLE et en tapant `import pygame`.

Que faire si `pip` n'est pas reconnu sur votre système ?

En principe, depuis la version 3.4 de Python, le programme `pip` est disponible sur votre système car installé en même temps que Python. **Toutefois**, si lors de l'installation de Python, vous avez oublié de cocher la case autorisant le placement des répertoires des binaires Python dans le PATH du système, `pip` ne sera pas reconnu.

Vous pouvez d'ailleurs vérifier en allant voir si le fichier `pip.exe` est présent dans le répértoire des scripts Python. On trouve ce répertoire à une adresse du type

```
C:\Users\MonNom\AppData\Local\Programs\Python\Python37-32\Scripts
```

Si `pip` n'est pas reconnu sur votre système, le plus probable est que le PATH soit incomplet. Deux solutions :

— soit vous désinstallez Python et vous le réinstallez en prenant soin de cocher dans le panneau d'installation, tout au début du processus d'installation, la case d'inclusion de Python au PATH,

— soit vous complétez vous-même le PATH en vous aidant par exemple de Add PIP to the Windows Environment Variables

Installer Pygame sous Linux

Il s'installe aidément en utilisant l'installeur pip3 :

```
pip3 install pygame
```

ce qui affiche

```
$ pip3 install pygame
Collecting pygame
  Downloading https://files.pythonhosted.org/
  packages/b3/5e/fb7c85304ad1fd52008fd25fce97
  a7f59e6147ae97378afc86cf0f5d9146/
  pygame-1.9.4-cp36-cp36m-manylinux1_x86_64.whl
  (12.1MB)
    100% |
Installing collected packages: pygame
Successfully installed pygame-1.9.4
```

Audio sous Tkinter avec Pygame

Nativement, Tkinter ne prend pas en charge la diffusion de flux audio (fichiers mp3, wav, etc). Il faut faire appel à une bibliothèque tierce pour réaliser l'incorporation d'un flux audio dans un programme Tkinter.

La bibliothèque de jeux Pygame prend en charge l'audio que ce soit sous Windows, Linux ou OsX. On peut donc l'associer à Tkinter pour faire émettre du son. C'est ce qu'on va faire ci-dessous. Toutefois, d'autres choix seraient possibles comme Pyglet.

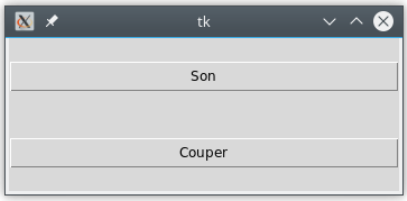
Pygame prend en charge les fichiers de format `wav`. Pour l'utilisation du format `mp3`, voir tout à la fin de cette unité consacrée à l'audio avec Pygame sous Tkinter. Le programme `audio.py`

ci-dessous dépend d'un fichier `clap.wav` placé à côté du fichier `audio.py`. Le fichier wav est téléchargeable ICI. C'est un programme minimal qui associe Tkinter et un flux audio géré par Pygame :

audio.py

```
1  from tkinter import *
2  import pygame
3
4  pygame.mixer.init()
5
6  mon_audio=pygame.mixer.Sound("clap.wav")
7
8  def lancer():
9      mon_audio.play(-1)
10
11 def couper():
12     mon_audio.stop()
13
14 fen = Tk()
15
16 Button(fen,text="Son",command=lancer, width=40).pack(pady=20)
17 Button(fen,text="Couper",command=couper, width=40).pack(pady=20)
18
19 fen.mainloop()
```

L'interface graphique a l'allure suivante :



Le bouton du haut active un son et ce son tourne en boucle, le bouton du bas interrompt le flux audio.

Commentaire de code

- Ligne 2 : on importe Pygame
- Ligne 4 : on initialise le module mixer de Pygame (le module qui gère le son)
- Ligne 6 : on charge le fichier dont on veut écouter le flux audio en indiquant l'adresse du fichier. Cela crée un objet qui permet d'accéder au flux vidéo.
- Ligne 1 : on importe Tkinter.
- Les éléments graphiques :
  - ligne 14 : la fenêtre

  - lignes 16-17 : deux boutons
  - lignes 9 et 13 : les fonctions associées aux boutons
- Ligne 9 : le son est joué (fonction `play`). L'argument `-1` a pour effet que le flux audio est joué en boucle, indéfiniment si on ne l'interrompt pas. Le volume par défaut est maximal.
- Ligne 12 : interruption du flux audio.

Volume du son

Il est possible de définir un volume sonore avec la méthode `set_volume`, par exemple placer entre les lignes 8 et 9 l'instruction :

```
mon_audio.set_volume(0.5)
```

La méthode `set_volume` accepte en un argument un nombre flottant entre 0 (aucun son) et 1 (son d'intensité maximale), donc `0.5` correspond à une intensité médiane.

Ne pas jouer en boucle

On souhaite parfois qu'un son soit joué non pas en boucle mais juste une seule fois. Pour cela il suffit de passer un argument autre que `-1` à la fonction `pygame.mixer.music.play`. Par exemple, `pygame.mixer.music.play()` va jouer le fichier audio une seule fois. Ou encore `pygame.mixer.music.play(5)` va le jouer 6 fois (et non pas 5, au moins sous Linux en tous cas).

Pré-initialisation

Parfois, le lancement du flux audio se lance avec un certain retard ou encore le flux audio est ralenti. Pour y remédier, essayer de placer avant l'appel à `pygame.mixer.init` un appel à la fonction `pre_init`, par exemple :

```
pygame.mixer.pre_init(44100, -16, 1, 512)
```

Les valeurs ci-dessous ont été retranscrises d'une réponse sur StackOverFlow.

Documentation

La bibliothèque Pygame permet une prise de l'audio. La documentation officielle de Pygame du package gérant le son est disponible sur pygame.mixer.music. Il pourra aussi être utile de consulter les fichiers d'exemples proposés dans le code source de Pygame.

Sortie correcte de Pygame

Si on reprend le programme `audio.py`, qu'on clique sur le bouton **Son** alors on entend le flux audio. Et si on interrompt alors le programme en cliquant sur la croix de la fenêtre alors le flux audio sera toujours audible. Pourquoi ? Parce que la croix ferme uniquement ce qui est de la responsabilité de Tkinter, ce qui n'est pas le cas de la gestion du son.

De la même façon qu'on a initialisé l'audio dans Pygame avec pygame.mixer.init, il faut quitter proprement Pygame. Pour cela, Pygame propose la méthode pygame.quit. Ci-dessous, le programme `audio.py` a été modifié pour que la sortie de Pygame soit correcte :

`audio_sortie.py`

```
1 from tkinter import *
2 import pygame
3
4 pygame.mixer.init()
5
```

```
6  mon_audio=pygame.mixer.Sound("clap.wav")
7
8
9  def lancer():
10     mon_audio.set_volume(1)
11     mon_audio.play(-1)
12
13 def couper():
14     mon_audio.stop()
15
16 def quitter():
17     pygame.quit()
18     fen.destroy()
19
20 fen = Tk()
21
22 Button(fen,text="Son",command=lancer, width=40).pack(pady=20)
23 Button(fen,text="Couper",command=couper, width=40).pack(pady=20)
24 fen.protocol("WM_DELETE_WINDOW", quitter)
25
26 fen.mainloop()
```

— Ligne 17 : on quittera Pygame avec la fonction dédiée `quit`.

— Ligne 24 : Cette ligne détecte une tentative de fermeture de la fenêtre en cliquant sur la croix. Fermer la fenêtre est l'événement `"WM_DELETE_WINDOW"`. Lorsque cet événement est détecté par Tkinter, la fonction `quitter`, définie lignes 16-18, sera automatiquement appelée.

— Lignes 16-18 : fonction appelée pour fermer proprement l'interface graphique et Pygame. `pygame.quit` ferme Pygame et libère les ressources que Pygame utilisait. De même, `fen.destroy()` fait disparaître le fenêtre `fen`.

Le cas du format mp3

Pour les flux audio mp3, la documentation de Pygame précise que la prise en charge est limitée. Le conseil qui est souvent donné est de convertir ses fichiers mp3 au format wav ou encore au format ogg (qui, comme le format mp3, est compressé, à la différence du format wav) qui eux sont pleinement pris en charge par Pygame.

Il semble toutefois que l'on puisse jouer des fichiers mp3 sous Tkinter en utilisant Pygame, que ce soit sous Windows comme sous Linux. Les exemples qui suivent utiliseront le fichier `clap.mp3` téléchargeable ICI et qu'on placera à côté du code source Python.

D'abord, hors de Tkinter, on peut écouter un fichier mp3 sous Python avec Pygame en suivant l'exemple suivant :

```
1  import pygame
2
3  pygame.init()
4
5  pygame.mixer.music.load("clap.mp3")
6  pygame.mixer.music.play()
7
```

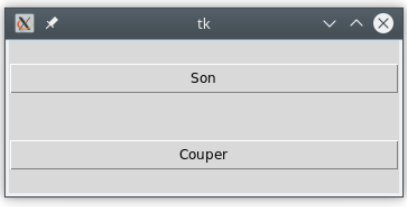
```
8 while pygame.mixer.music.get_busy():
9     pass
10
11 pygame.quit()
```

Cet exemple suit la démo donnée dans le code-source de Pygame. On notera que la méthode Sound applicable au format wav ne semble pas s'appliquer au format mp3. On prendra aussi garde que certains fichiers audio au format mp3, parfaitement audibles dans un lecteur audio, ne seront pas décodés par Pygame, comme rappellé dans cette discussion Pygame fails to play some mp3 files but not others.

La boucle aux lignes 8-9 doit être présente sinon le son n'est pas émis et le programme s'interrompt.

Pour finir, je reprends ci-dessous l'application utilisée sous Tkinter pour décrire l'usage d'un fichier wav, sans commentaire supplémentaire puisque les codes sont assez proches :

```
from tkinter import *
import pygame

pygame.mixer.init()

pygame.mixer.music.load("clap.mp3")

def lancer():
    pygame.mixer.music.play(-1)

def couper():
    pygame.mixer.music.stop()

fen = Tk()

Button(fen,text="Son",command=lancer, width=40).pack(pady=20)
Button(fen,text="Couper",command=couper, width=40).pack(pady=20)

fen.mainloop()
```



Il semblerait que le module mixer s'exécute dans un thread distinct du thread principal dans lequel s'éxecute Tkinter.

Audio sous Tkinter avec winsound

Sous Windows, il est possible de jouer des fichiers audio avec le module winsound. L'intérêt essentiel par rapport à des solutions utilisant Pygame ou Pyglet est qu'il n'y a rien à installer puisque winsound est un module standard de Python. Cette possibilité est offerte uniquement sous Windows ce qui rend le code non portable mais ça peut dépanner. La documentation de winsound est disponible dans la documentation officielle. Bien que ce ne soit pas indiqué, seul le format wav est pris en charge, en particulier le format mp3 n'est pas utilisable avec winsound, comme c'est indiqué dans ce message.

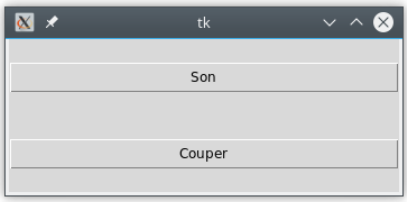
A des fins de démontration, voici un programme permettant de tester winsound sous Tkinter. Le fichier `clap.wav` (téléchargeable ICI) doit être placé à côté du code-source Python :

```
1  from tkinter import *
2  import winsound
3
4  def lancer():
5      winsound.PlaySound('clap.wav', winsound.SND_LOOP | winsound.SND_ASYNC)
6
7  def couper():
8      winsound.PlaySound(None, winsound.SND_PURGE)
9
10 def close_sound():
11     couper()
12     fen.destroy()
13
14 fen = Tk()
15
16 fen.protocol("WM_DELETE_WINDOW", close_sound)
17
18 Button(fen,text="Son",command=lancer, width=40).pack(pady=20)
19 Button(fen,text="Couper",command=couper, width=40).pack(pady=20)
20 fen.mainloop()
```

— ligne 5 : pour jouer un fichier de format `wav`, on utilise la fonction `PlaySound` à qui on transmet le nom du fichier et en 2e argument, on place des flags, ici un flag pour jouer en boucle et un flag pour que le flux audio ne bloque pas le programme principal.

— ligne 8 : pour couper (provisoirement) le son, on utilise encore la fonction `Playsound` mais avec un argument valant `None`.

— Lignes 10-12 et 16 : si on quitte le programme en fermant la fenêtre avec la croix et alors qu'un flux audio est émis, le flux lui n'est pas interrompu, ce qu'il faut corriger manuellement. Il faut donc intercepter le signal de fermeture de la fenêtre (ligne 16) pour appeler une fonction `close_sound` (lignes 10-12) qui avant de fermer l'application graphique (ligne 12) va couper le son (ligne 11).



Pour une solution portable Windows, MacOs et Linux, et prenant en charge le format mp3, il aurait pu être envisageable d'utiliser le module non standard playsound mais il ne semble plus maintenu depuis juin 2017.

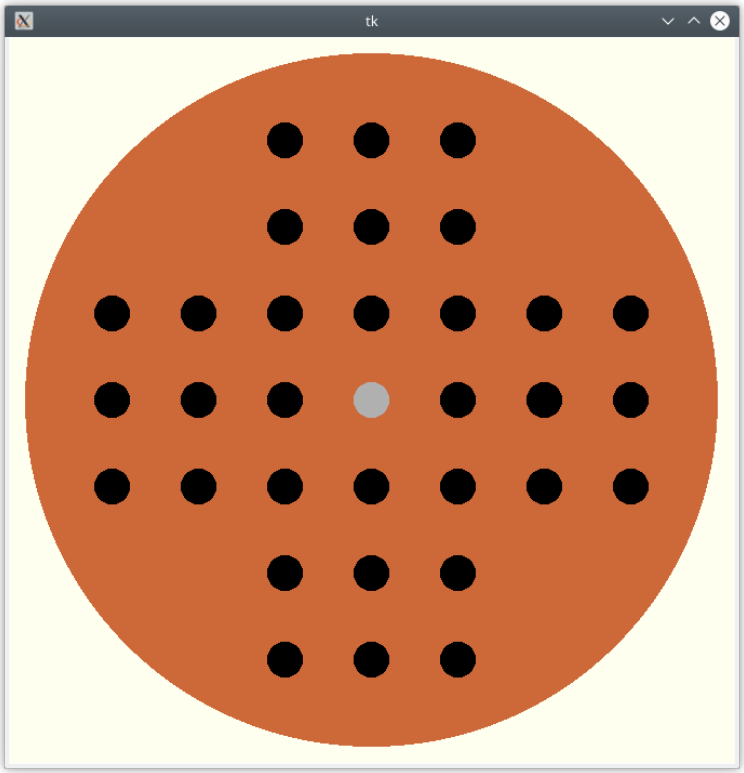
Conseils généraux pour écrire de petites applications Tkinter

Vous devez écrire un petit jeu Tkinter, avec un plateau par exemple. Voici quelques conseils très élémentaires et qui s'adressent à des débutants en Python.

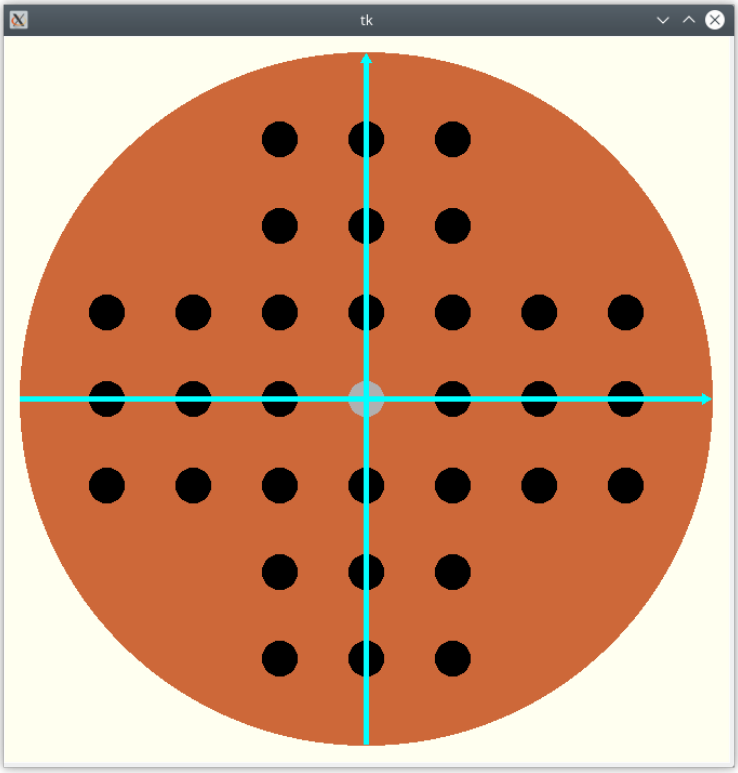
— Encapsuler toutes les données et les distances fixes dans des variables bien nommées, autrement dit éviter les *constantes magiques* dans le code ; ces variables qui seront globales à votre programme sont en fait des constantes et seront écrites en capitales. A terme, limiter toutes ces variables globales et prétendues inamovibles et qui à l'usage ont besoin d'évoluer.
— Découper votre code en fonctions simples et réutilisables et avec une interface adéquate
— Créer une classe qui encapsule des données et des méthodes interdépendantes.
— Séparer la partie vue de l'application et la partie calcul (ce point est essentiel dans de nombreuses situations)
— Faire un croquis, de préférence sur une feuille quadrillée en indiquant les coordonnées
— Ne pas abuser du multi-fenêtrage
— Si l'interface est formée de nombreux widgets, répartir ces widgets selon différents frames.

Changement de repère

Beaucoup de dessins sont naturellement construits avec un repérage défini par le repère mathématique habituel. Par exemple, considérons le dessin d'un jeu de solitaire :



A cause des symétries, il est assez naturel de se placer dans le repère ci-dessous, avec une origine au centre du dessin et les axes mathématiques habituels :



Or, le repère naturel du canevas est différent :

— l'origine est en haut à gauche de la fenêtre

— l'axe vertical est orienté vers le bas et non vers le haut.

Donc une solution envisageable est de faire ses calculs dans le repère mathématique habituel et de convertir toutes les coordonnées dans le repère naturel du canevas. On pourra alors écrire un code tel que celui-ci :

```
1  def chgt(X,Y, center):
2      """(X, Y): coordonnées dans le repère mathématique habituel
3      center=(x0,y0) coordonnées dans le repère du
4      canevas du centre du repère mathématique
5      habituel renvoie coordonnées (x, y) du canevas"""
6
7      return (X+center[0], -Y+center[1])
8
9  center=(SIDE//2, SIDE//2)
10
11 X=-200
12 Y=40
13 x,y=chgt(X,Y, center)
14 cnv.create_text(x, y, text ="Coucou !")
```

— Lignes 1-7 : la fonction de changement de coordonnées. Les explications sont données dans la docstring. On donne les coordonnées mathématiques et les coordonnées du centre de notre repère (par rapport au repère du canevas) et on récupère des coordonnées pour le canevas (utiles pour utiliser Tkinter)

— Ligne 9 : typiquement, le centre du repère mathématique est le centre de la fenêtre

— Ligne 13 : on effectue le changement de coordonnées ; pour cela, on récupère les coordonnées `(x,y)` dans le canevas du point de coordonnées `(X, Y)=(-200, 40)` dans le repère d'origine le point `centre`.

— ligne 14 : on donne ces coordonnées à une méthode du canevas.

Importer des vecteurs

Les vecteurs (des maths ou de la physique) sont souvent utiles pour faire des calculs avec des dessins. Cependant, Tkinter ne semble pas disposer d'une classe `Vector` qui permettrait de générer des vecteurs et de faire des opérations vectorielles. Heureusement, le module standard Turtle dispose d'une classe `Vec2D` qui représente des vecteurs du plan. On peut faire des opérations usuelles avec des vecteurs :

— somme, différence

— produit par un nombre

— produit scalaire habituel

— calcul de la norme (de la longueur du vecteur) avec `abs` (et non `len`)

— opposé d'un vecteur

— accès aux coordonnées aves les indices 0 et 1.

On peut aussi effectuer une rotation vectorielle.
Voici une illustration

```
from turtle import Vec2D

v=Vec2D(3,4)
w=Vec2D(-8,6)

print(v)
print(v[0], v[1])
print(-v)
print(abs(v))
print(10*v)
print(v+w)
print(v*w)
```

```
(3.00,4.00)
3 4
(-3.00,-4.00)
5.0
(30.00,40.00)
(-5.00,10.00)
0
```

Attention toutefois que certaines opérations n'existent pas :

— on peut écrire `u=Vec2D((1,3), (2,5))` et même `2*u` mais cela n'aura pas le sens attendu pour des vecteurs (et qui serait ici de créer le vecteur d'origine `(1,3)` et d'extrémité `(2,5)`);
— si `v` est un vecteur, on n'accède pas à sa 1re coordonnéee par `v.x`;
— un vecteur est immuable (il dérive de la classe `tuple`)
— on ne peut déclarer sans initialiser `u=Vec2D()` (on se serait attendu à obtenir le vecteur nul)
— une opération comme `Vec2D(3,4)/2` de division d'un vecteur par un scalaire n'est pas reconnue.

Si `u=(2, 3)` on ne peut pas écrire `Vec2D(u)` ; il faut extraire les coordonnées avec l'opérateur splat comme ceci : `Vec2D(*u)`. Voici, par exemple, comment on calcule la distance entre deux points A et B :

```
1 from turtle import Vec2D as vect
2
3 A=(2,3)
4 B=(4,1)
5
6 # Accès habituel aux coordonnées
7 AB=vect(B[0], B[1])-vect(A[0], A[1])
8 print(abs(AB))
9
10 # Accès par décompression aux coordonnées
11 AB=vect(*B)-vect(*A)
12 print(abs(AB))
```

```
2.8284271247461903
2.8284271247461903
```

Donnons un autre exemple de calcul. On se donne une sommet `A` et un point `C` et on cherche à construire un point `B` tel que A, B et C soient alignés dans cet ordre et que `AB=L` où `L` est une longueur donnée. L'idée de base est que si v est un vecteur et si N est la norme du vecteur alors v/N est un vecteur de norme 1 et donc que kv/N est de longueur k. Le code ci-dessous calcule le point `A` :

```
from turtle import Vec2D

A=(5,2)
C=(-1, 5)

A=Vec2D(*A)
C=Vec2D(*C)
AC=C-A
u=1/abs(AC)*AC
print(u)

lg=3
B=A+lg*u

print(B)
```

```
(-0.89,0.45)
(2.32,3.34)
```

Rotation de vecteur

On peut avoir besoin d'effectuer une rotation d'un item placé sur un canevas Tkinter. Pour des items géométriques du type `line`, `rectangle`, etc, il suffit de savoir coder une fonction de rotation et de l'appliquer aux sommets de l'item. Pour d'autres types d'items, voir plus bas.

On peut coder soi-même une fonction de rotation. Toutefois, le module Turtle en fournit une comme méthode de sa classe Vec2D (`Vec2D` n'oblige nullement à coder en Turtle).

Voici un exemple d'utilisation. Calculons les coordonnées dans le repère mathématique habituel de l'image `N` du point `M` de coordonnées `(150, 100)` par la rotation de centre le point `C=(100, 100)` et d'angle `60°` :

```
from turtle import Vec2D

C=(100, 100)
M=(150, 100)


CM=Vec2D(*M)-Vec2D(*C)
v=CM.rotate(60)
N=Vec2D(*C)+v
print(N)
```

```
(125.00,143.30)
```

Comme prévu, l'abscisse du point cherché est $\frac{\texttt{C[0]+M[0]}}{2}$ puisque le cosinus de 60° est 1/2. Mathématiquement, `CM` désigne le vecteur $\overrightarrow{CM} = \overrightarrow{OM} - \overrightarrow{OC}$. Ci-dessus, `v` désigne l'image de $\overrightarrow{CM}$ par la rotation *vectorielle* d'angle 60° (et le centre est à l'origine des axes, comme pour toute rotation *vectorielle*) . L'affectation `N=Vec2D(*C)+v` est une conséquence de $\overrightarrow{CN} = \overrightarrow{ON} - \overrightarrow{OC} = \overrightarrow{v}$.

Rreste le cas d'item non géométriques. Pour du texte, cela doit être réalisable car un item texte possède une option angle. Reste le cas des images (format PhotoImage) pour lequel il faudra utiliser le module (non standard) PIL.

Définir une map avec une chaîne triple

Un jeu nécessite parfois de définir une map. Il est fréquent que cette map soit représentée par une liste de listes d'entiers, par exemple :

```
map2D=[
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 3, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1],
    [1, 0, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1],
    [1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1],
    [1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1],
    [1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1],
    [1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1],
    [1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1],
    [1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1],
    [1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1],
    [1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1],
    [1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1],
    [1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 0, 1],
    [1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 4, 1],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]
```

Il sera toutefois plus lisible et plus maintenable de définir ses map sous forme de chaînes triples :

```
map_str="""\
111111111111111
130222222222221
101212121212121
122222222222221
121212121212121
122222222222221
121212121212121
122222222222221
121212121212121
122222222222221
121212121212121
122222222222221
121212121212101
122222222222041
```

```
111111111111111"""
```

Mais, on perd l'accès aux éléments par ligne et par colonne ainsi que les dimension de la map ; donc, derrière ça on peut construire la map sous forme de liste 2D :

```
ma_map="""\
111111111111111
130222222222221
101212121212121
122222222222221
121212121212121
122222222222221
121212121212121
122222222222221
121212121212121
122222222222221
121212121212121
122222222222221
121212121212101
122222222222041
111111111111111"""

map2D=[list(map(int, L)) for L in ma_map.split()]

print(*map2D, sep='\n')
```

```
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[1, 3, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1]
[1, 0, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1]
[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1]
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1]
[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1]
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1]
[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1]
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1]
[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1]
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1]
[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1]
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 2, 1, 0, 1]
[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 4, 1]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

`map2D` a été construite avec une liste en compréhension. Attention que dans l'avant dernière ligne de code, le nom `map` représente une fonction standard de Python, à ne pas confondre avec une map (au sens de carte) dans un jeu ...

Chapitre II

Quelques widgets

Le widget label

Un label est un widget constitué d'un court message textuel (en français, on devrait dire *étiquette*). Par exemple :


```
from tkinter import *

root = Tk()
lbl=Label(root, text="Coucou !", font='Arial 30 bold')
lbl.pack(padx=15, pady=15)

root.mainloop()
```

Le label est créé avec le constructeur `Label`. Le texte est transmis avec l'option `text` et on peut aussi changer la police avec l'option `font`.

Un label est un des widgets les plus simples et est souvent utilisé. Il est soit statique (une description par exemple) soit dynamique (comme un compteur ou une durée qui évoluent).

Voici un exemple de labels qui indiquent le nombre de carrés générés aléatoirement sur un canevas

Ici, il y a deux labels :

— un label statique, à gauche, qui porte la mention *Nombre de carrés*,

— un label dynamique, à droite, qui indique la valeur du nombre de carrés présents sur le canevas.

Le compteur change toutes les demi-secondes.

Le code correspondant est :

```
from tkinter import *
from random import randrange

WIDTH=200
HEIGHT=200

COTE=20

root = Tk()
cnv = Canvas(root, width=WIDTH, height=HEIGHT, background="ivory")
cnv.pack()

def dessiner():
    global cpt
    k=randrange(3)
    cpt+=k
    for i in range(k):
        color="#%s%s%s" %(randrange(10),randrange(10),
                          randrange(10))
        x=randrange(WIDTH)
        y=randrange(HEIGHT)
        cnv.create_rectangle(x, y, x+COTE, y+COTE, fill=color,
                             outline='')

descr = Label(root, text="Nombre de\ncarrés", font='Arial 10 bold')
descr.pack(side='left')

```

```
28 compteur = Label(root, text="0", font='Arial 15 bold')
29 compteur.pack(side='right')
30
31 def animer():
32     dessiner()
33     compteur['text']=str(cpt)
34     cnv.after(500, animer)
35
36 cpt=0
37 animer()
38
39 root.mainloop()
```

— Ligne 25 : création d'un label `descr` portant le texte `"Nombre de carrés"` et placé en bas à gauche de l'interface.

— Ligne 28 : création d'un label `compteur` placé en bas à droite et indiquant le nombre de carrés.

— lignes 25 et 28 : un label est créé avec le constructeur `Label` qui est un widget Tkinter. Un label peut utiliser une police que l'on définit avec l'option `font`.

— Ligne 34 : toutes les demi-secondes, le texte du label est mis à jour en indiquant le nombre de carrés aléatoires (cf. fonction `dessiner` lignes 13-23) présents sur le canevas.

— Ligne 33 : instruction pour mettre à jour le label `compteur`.

Créer et intégrer un widget en une seule instruction

L'interface graphique suivante :

montre un court morceau de texte et qui est placé dans un widget de type label. Le code correspondant est :

```
1 from tkinter import *
2
3 root = Tk()
4 lbl=Label(root, text="Coucou !", font='Arial 30 bold')
5 lbl.pack(padx=15, pady=15)
6
7 root.mainloop()
```

La création du widget (ligne 4) et l'intégration du widget dans la fenêtre en utilisant un gestionnaire de géométrie (`pack` ici, ligne 5) sont effectuées dans deux instructions différentes. Toutefois, la variable `lbl` n'est pas réutilisé ailleurs dans le code. Dans ce cas, il est possible de faire

les deux opérations en une seule ce qui dispense de définir une variable désignant le widget. D'où le code plus simple suivant :

```
1 from tkinter import *
2
3 root = Tk()
4 Label(root, text="Coucou !", font='Arial 30 bold').pack(padx=15, pady=15)
5
6 root.mainloop()
```

— Ligne 4 : création et placement du widget en une seule instruction.

Attention, bugs en vue !

Cette méthode n'est pas toujours bien comprise et peut être source de bugs, en particulier chez les débutants. Il est donc préférable de l'éviter, surtout que son bénéfice est réduit. Dès qu'une interface a un peu de complexité, il est rare qu'un widget qui a été crée ne soit pas référencé dans le code et donc l'astuce ci-dessus devient une nuisance. Certains pensent alors utiliser un code comme celui-ci :

```
1 from tkinter import *
2
3 root = Tk()
4 lbl=Label(root, text="Coucou !", font='Arial 30 bold').pack(padx=15, pady=15)
5
6 print(lbl["text"])
7
8 root.mainloop()
```

L'intention ici est d'afficher dans la console (cf. ligne 6) le contenu du label ("coucou", ligne 4). Certes, une variable `lbl` a été créée sauf qu'elle ne référence pas le label mais le retour de l'appel `Label(...).pack()` qui lui vaut `None`. Donc la ligne 6 va planter le programme :

```
    print(lbl["text"])
TypeError: 'NoneType' object is not subscriptable
```

Pour s'en sortir, il faut découpler la création du widget et son placement avec la méthode `pack`. D'où le code correct suivant :

```
from tkinter import *

root = Tk()
lbl=Label(root, text="Coucou !", font='Arial 30 bold')
lbl.pack(padx=15, pady=15)

print(lbl["text"])

root.mainloop()
```

qui affiche :

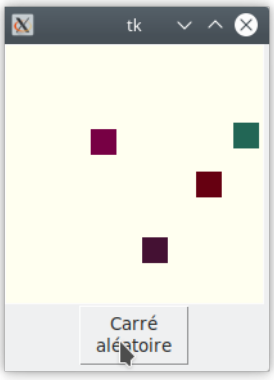
```
Coucou !
```

Le widget bouton

Tkinter dispose d'un widget bouton avec la classe `Button`. On peut donc créer un bouton qui réagira de façon appropriée à chaque clic du bouton.
Exemple :

```
1  from tkinter import *
2  from random import randrange
3
4  WIDTH=200
5  HEIGHT=200
6
7  COTE=20
8
9  root = Tk()
10 cnv = Canvas(root, width=WIDTH, height=HEIGHT, background="ivory")
11 cnv.pack()
12
13 def dessiner():
14     color="#%s%s%s" %(randrange(10),randrange(10),randrange(10))
15     x=randrange(WIDTH)
16     y=randrange(HEIGHT)
17     cnv.create_rectangle(x, y, x+COTE, y+COTE, fill=color, outline='')
18
19 btn = Button(root, text="Carré\naléatoire", command=dessiner)
20 btn.pack()
21
22 root.mainloop()
```

qui produit :



Lorsque l'utilisateur clique sur le bouton, un carré coloré aléatoire est placé sur le canevas.
Décrivons le code en rapport avec le bouton :

— Ligne 19 : un bouton est créé. Il permet l'affichage du texte avec l'option `text`.

— Ligne 19 : on peut donner une option `command` au bouton : cette option doit pointer vers la fonction qui sera exécutée lorsque l'utilisateur appuie sur le bouton. Cette fonction doit être définie avant la définition du bouton (ici ligne 13, la fonction `dessiner`). Ce type de fonction de commande ne doit recevoir aucun argument.

— lignes 13-17 : lorsqu'on clique sur le bouton, la fonction `dessiner` est appelée. Cette fonction dessine avec une couleur aléatoire (ligne 14) à une position aléatoire (lignes 15-16) un carré sur le canevas (ligne 17).

La longueur variable de texte dans un bouton peut modifier la taille du bouton et donc de la fenêtre parente. Pour éviter cela, on peut imposer des dimensions au bouton :

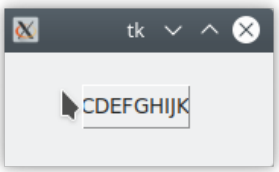
— height = nombre de lignes

— width = nombre de caractères

Si plus de caractères que la capacité permise par la largeur sont donnés dans le texte, des caractères seront invisibles :

```
from tkinter import *

root = Tk()

def play():
    b['text']='ABCDEFGHIJKLM'

b=Button(root, text="Play", command=play, width=5)
b.pack(padx=50, pady=20)
root.mainloop()
```

qui affiche après un clic sur le bouton :



On peut afficher un bouton coloré avec l'option `bg` (qui signifie *background*) :

```
1 from tkinter import *
2
3 root = Tk()
4
5 b=Button(root, text="Bouton\nrouge", width=5, bg="red")
6 b.pack(padx=50, pady=20)
7 root.mainloop()
```

Un bouton pour montrer une image

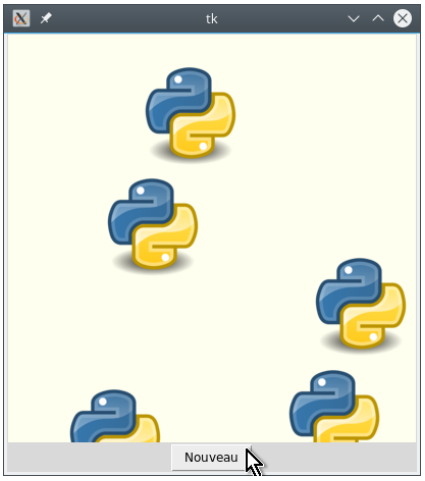
Un bouton, de même que le canevas est un widget. Le code ci-dessous place un bouton à côté d'un canevas :

bouton_image.py

```
1 from tkinter import *
2
3 SIDE=400
4 root = Tk()
5 cnv = Canvas(root, width=SIDE, height=SIDE, bg='ivory')
6 cnv.pack()
7
8 btn=Button(root, text="Nouveau")
9 btn.pack()
10
11 root.mainloop()
```

— Ligne 8 : un bouton est construit avec le constructeur `Button` (c'est une classe).
— Ligne 9 : comme pour tout widget, il faut l'inclure dans son environnement avec une méthode particulière, ici la méthode `pack`.
— Ligne 8 : le texte passé dans l'option `text` est affiché sur le bouton.
— Si on clique sur le bouton, rien ne se passe de visible. Pour lier une action à un bouton, il faudrait lui passer une option `command`.

Donnons une possibilité d'action au bouton : chaque fois qu'on clique le bouton, un logo 80x80 est dessiné sur le canevas :



Voici le code :

bouton_image1.py

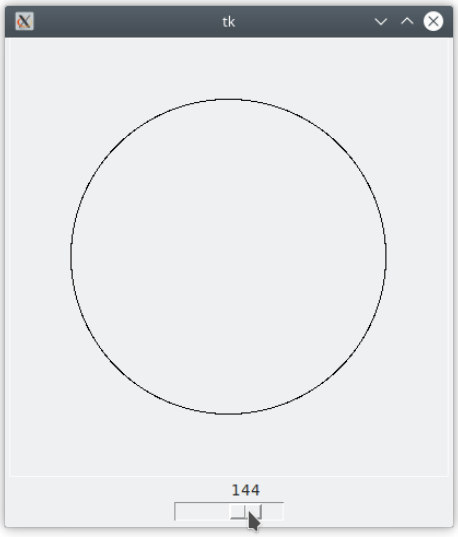
```
1  from tkinter import *
2  from random import randrange
3
4  SIDE=400
5  root = Tk()
6  cnv = Canvas(root, width=SIDE, height=SIDE, bg='ivory')
7  cnv.pack()
8
9  logo = PhotoImage(file="python80.png")
10
11 def show():
12     center= (randrange(SIDE),randrange(SIDE))
13     cnv.create_image(center, image=logo)
14
15 btn=Button(root, text="Nouveau", command=show)
16 btn.pack()
17
18 root.mainloop()
```

— Ligne 15 : une option `command` a été donnée au constructeur `Button` : `command` référence une fonction sans paramètre, ici la fonction `show`, qui est exécutée à chaque pression sur le bouton.

— Lignes 11-13 : la fonction `show` ne peut prendre aucun paramètre ; elle dessine un logo Python aléatoire sur le canevas.

Slider basique

Un slider (en français, un *curseur*) est un widget permettant de modifier une variable ou un état en faisant glisser un curseur sur un axe :



Dans l'exemple ci-dessus, le curseur est mobile et indique le rayon du cercle qui est dessiné sur le canevas.

Le code correspondant est :

```
1  from tkinter import *
2
3  root = Tk()
4  cnv = Canvas(root, width=400, height=400)
5  cnv.pack()
6
7  old=None
8
9  def rayon(r):
10     global old
11     r=int(r)
12     cnv.delete(old)
13     old=cnv.create_oval(200-r,200-r,200+r, 200+r)
14
15 curseur = Scale(root, orient = "horizontal", command=rayon, from_=0, to=200)
16 curseur.pack()
17
18 root.mainloop()
```

— Ligne 15 : création d'un curseur.

— Ligne 16 : placement du curseur.

— Ligne 15 : quelques options du curseur :

  — `orient` : l'orientation horizontale ou verticale

  — `command` : la fonction qui est appelée quand on déplace le curseur (ici, la fonction `rayon`)

  — `from_` : la valeur (numérique) en début du curseur (par défaut à gauche pour une orientation horizontale) transmise à la fonction de commande ; le blanc souligné qui termine

`from_` est utilisé car on ne peut pas utiliser la variable `from` puisque c'est un mot-clé du langage Python.

- `to` : la valeur (numérique) en fin de curseur (par défaut à droite pour une orientation horizontale) transmise à la fonction de commande ;

— Lignes 9-18 : la fonction de commande appelée lorsque le curseur est modifié. Il semblerait que, par défaut, cette fonction reçoive en argument une chaîne de caractères représentant un nombre entier et qui correspond à la position du curseur dans sa graduation.

On peut aussi changer la longueur du curseur avec l'option `length`. Par exemple,

```
curseur = Scale(root, command=tirer, from_=0, to=180, length=200)
```

va créer un curseur de 200 pixels de longueur.

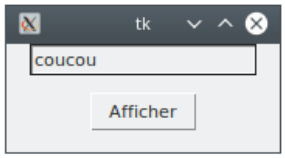
Le widget entrée

Le widget Entry (*entrée* en français) est un widget analogue aux champs des formulaires html : l'utilisateur communique avec le programme en lui transmettant dans une zone de texte des données écrites au clavier (ou par copier-coller).

Exemple simple

Dans l'exemple ci-dessous, on montre comment une entrée transmet son contenu à une autre partie du programme. L'utilisateur remplit l'entrée avec du texte et s'il clique sur un bouton, cela affiche le contenu de l'entrée dans la console :

```
coucou
```



Voici le code commenté correspondant :

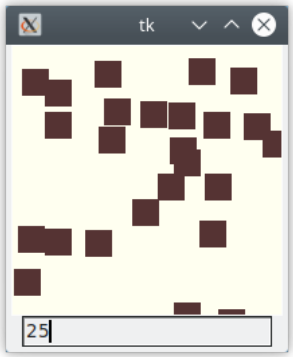
```
1 from tkinter import *
2
3 root = Tk()
4
5 my_entry = Entry(root)
6 my_entry.pack()
7
8 def afficher():
9     print(my_entry.get())
10
11 bouton=Button(root, text="Afficher", command=afficher)
12 bouton.pack(padx=50, pady=10)
13
```

```
14 root.mainloop()
```

— Lignes 5-6 : définition et placement de l'entrée `my_entry`
— Ligne 9 : une entrée Tkinter a une la méthode `get` qui permet de récupérer le contenu de l'entrée. Ici, ce contenu est affiché dans la console avec la fonction `print`.
— Ligne 10-11 : définition et placement du bouton qui, lorsqu'il est cliqué, appelle la fonction `afficher`.

Autre exemple

Voici un autre exemple, du même ordre mais un peu moins simple :



L'utilisateur entre au clavier un entier dans le champ en bas de la fenêtre et il valide cette entrée par appui sur la touche ESPACE. Cela provoque immédiatement l'affichage sur le canvas de carrés aléatoirement placés et en nombre égal à la valeur tapée au clavier.

Le code correspondant est :

```
1  from tkinter import *
2  from random import randrange
3
4  WIDTH=200
5  HEIGHT=200
6
7  COTE=20
8
9  root = Tk()
10 cnv = Canvas(root, width=WIDTH, height=HEIGHT, background="ivory")
11 cnv.pack()
12
13 def dessiner(event):
14     n=int(my_entry.get())
15     color="#%s%s%s" %(randrange(10),randrange(10),randrange(10))
16     for i in range(n):
17         x=randrange(WIDTH)
18         y=randrange(HEIGHT)
19         cnv.create_rectangle(x, y, x+COTE, y+COTE, fill=color, outline='')
20
```

```
21 my_entry = Entry(root)
22 my_entry.pack()
23 my_entry.bind('<space>', dessiner)
24
25 root.mainloop()
```

— Ligne 21 : création d'une entrée.

— Ligne 22 : placement de l'entrée dans sa fenêtre.

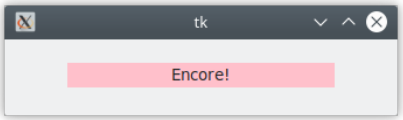
— Ligne 23 : association de l'entrée `my_entry` avec une touche du clavier : quand l'entrée a le focus, si on appuie sur une certaine touche indiquée comme premier argument de `bind` (dans notre exemple la touche ENTRÉE référencée par `<Return>`), une fonction de commande est exécutée (ici la fonction `dessiner`)

— Lignes 13-19 : la fonction appelée lorsque l'entrée est validée. Cette fonction reçoit un événement `event`. Cet événement correspond à l'appui sur la touche ENTRÉE mais n'est pas véritablement utilisé par la fonction `dessiner`. L'objet `my_entry` dispose d'une méthode `get` permettant de capturer le contenu textuel de l'entrée, ici un entier. Cet entier est lu sous forme de chaîne de caractères, donc il faut le convertir avec la fonction `int` (ligne 14). Le reste du code crée un carré aléatoire et le dessine sur le canevas.

Noter que le widget Entry ne gère pas automatiquement la validation du contenu de l'entrée par appui sur une touche (cf. ligne 23).

Options d'un widget : lecture, écriture

Un widget (par exemple un canevas, un bouton, etc) est initialisé avec son constructeur. Par exemple, ci-dessous




```
1 from tkinter import *
2
3 root = Tk()
4 L=Label(root, text="Encore!", bg='pink', width=25)
5 L.pack(padx=50, pady=20)
6
7 root.mainloop()
```

à la ligne 4, le constructeur `Label` définit le texte du label, la couleur de fond et sa longueur.

Une fois le widget construit, on peut souhaiter consulter la valeur d'une option mais aussi la modifier. C'est possible de deux façons :

— soit en utilisant le widget comme on utiliserait un dictionnaire,

— soit en utilisant la méthode `configure` et un ou plusieurs arguments nommés pour accéder aux options.

Le code ci-dessous illustre ces deux possibilités :

```
1 from tkinter import *
2
3 root = Tk()
4 lbl=Label(root, text="Encore!", bg='pink', width=25)
5 lbl.pack(padx=50, pady=20)
6
7 print(lbl['text'])
8 print(lbl.configure('text'))
9 print(lbl.cget('text'))
10
11 root.mainloop()
```

```
12 Encore!
13 ('text', 'text', 'Text', '', 'Encore!')
14 Encore!
```

En lecture, on peut accéder, par exemple, à la valeur de l'option `text` du label en utilisant :

— le label comme dictionnaire (ligne 7)

— la méthode `configure` (ligne 8)

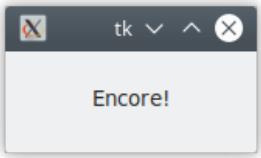
— la méthode `cget` (ligne 9) dont le résultat est moins facilement interprétable.

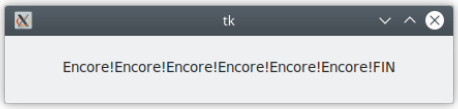
Concernant les modifications (« écriture »), soit le code suivant :

```
1 from tkinter import *
2
3 root = Tk()
4
5 def animer(i):
6     if i<5:
7         print(i)
8         L['text']+="Encore!"
9         root.after(500, animer, i+1)
10    if i==5:
11        L.configure(text=L['text']+'FIN')
12
13
14 L=Label(root, text="Encore!")
15 L.pack(padx=50, pady=20)
16
17 animer(0)
18
19 root.mainloop()
```

Le programme crée un label contenant le texte `Encore!` et modifie le texte du label en le répétant 5 fois et en terminant en ajoutant le mot `FIN`.

Ci-dessous l'évolution du widget :





Deux méthodes sont ici utilisées :

— Ligne 8 : on accède à l'option du label `L` en utilisant `L` comme un dictionnaire avec pour clé le nom de l'option, donc ici, on modofoe `L['text']` ;
— Ligne 11 : on accède à l'option du label `L` en utilisant la méthode `configure` du label et l'argument nommé correspondant à l'option que l'on veut changer, ici `text`.

Image sur un bouton

Sur un bouton, il est usuel de placer du texte. Mais on peut placer une image à la place de texte (ci-dessous, deux images sont placées) :

On suppose qu'on a placé une image `python.gif` à la racine du fichier source. Le code correspondant est :

```
1 from tkinter import *
2
3 root = Tk()
4 logo = PhotoImage(file="python.gif")
5 btnA=Button(root, image=logo)
```

```
6 btnA.pack(padx=10, pady=10)
7 btnB=Button(root, width=50, height=50, image=logo)
8 btnB.pack(padx=10, pady=10)
9
10 root.mainloop()
```

— Ligne 5 : on crée un bouton repéré par une image ; aucune dimension n'est donné donc le bouton prend la taille de l'image. Noter que l'image doit être converti au format PhotoImage (ligne 4).

— Ligne 7 : une autre image est créée. On lui impose des dimensions. Les dimensions sont mesurées sur l'image cible à partir de son coin supérieur gauche.

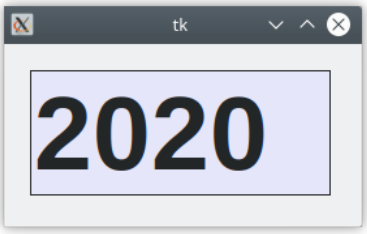
— Lignes 5 et 7 : pour simplifier l'illustration, le clic sur les boutons n'a aucune action visible.

Options d'un widget entrée

Le code ci-dessous montree quelques options d'une entrée :

```
1 from tkinter import *
2 root = Tk()
3
4 my_entry = Entry(root,
5                   font='Arial 60 bold',
6                   width='5',
7                   bg='lavender',
8                   insertofftime=500,
9                   relief=FLAT)
10 my_entry.pack(padx=20, pady=20)
11 my_entry.focus_set()
12
13 root.mainloop()
```

qui produit :



Par défaut, une entrée n'a pas le focus autrement dit, elle n'est pas en état de lire des caractères. On l'active :

— soit avec la touche TAB

— soit en cliquant dans la zone de saisie de l'entrée.

Mais on peut forcer l'acquisition du focus par la méthode `focus_set` (ligne 11). Donc, dès que l'application s'ouvre, l'entrée est réceptive aux touches de clavier.

La couleur de fond de la zone d'édition est déterminée par l'option `bg` (ligne 7).

Il est possible de choisir avec l'option `font` la police et la taille de la police de la zone de saisie (ligne 5). On peut décider de limiter le nombre de caractères que peut accepter l'entrée (ligne 5). La taille en pixels de la zone de saisie en sera modifiée mais cette taille dépendra aussi la taille de la police.

Quand le focus est actif, un curseur guide la saisie. Le clignotement du curseur est modifiable avec l'option `insertofftime` (ligne 500) qui désigne la durée en millisecondes entre deux apparition du curseur (si 0 alors aucun clignotement). La couleur du curseur est contrôlée par l'option `insertbackground`.

L'entrée est un entourée d'un bord dont l'épaisseur est contrôlée par l'option `border`.

Le style de relief de l'entrée est contrôlé par l'option `relief`, par défaut, on a `relief=SUNKEN` (ligne 9).

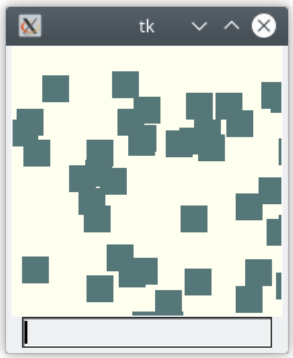
On peut contrôler le texte entré par l'utilisateur grâce à l'option `textvariable` qui doit être initialisée sur une variable de contrôle, par exemple StringVar. Un exemple est fourni au paragraphe Exemple d'utilisation de StringVar. Noter qu'il n'y a pas d'option `text` valide pour un widget Entry.

Une `Entry` ne permet pas d'entrer du texte sur **plusieurs** lignes. Utiliser plutôt le widget `Text` (non décrit sur mon site) ainsi que c'est indiqué par Fredrik Lundh :

The entry widget is used to enter text strings. This widget allows the user to enter one line of text, in a single font. To enter multiple lines of text, use the Text widget.

Effacer une entrée

Dans l'interface ci-dessous, on a écrit un entier dans une entrée :



Après validation de l'entrée (on appuie sur ENTER), des carrés ont été dessinés sur le canevas et *l'entrée a été effacée* si bien que l'utilisateur peut directement réutiliser l'entrée pour générer des carrés sur le canevas.

Pour effacer la zone d'édition d'une entrée, on utilise la méthode `Entry.delete` :

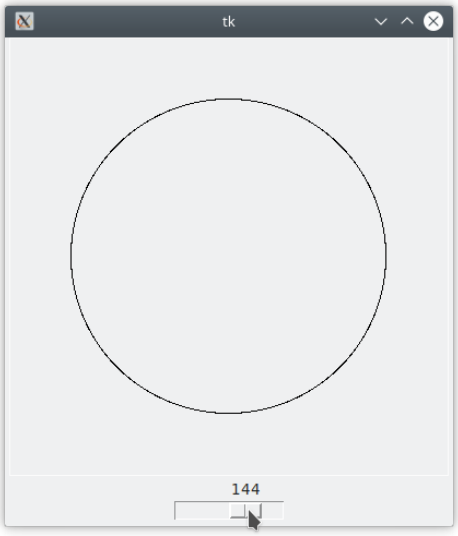
```
1  from tkinter import *
2  from random import randrange
3
4  WIDTH=200
5  HEIGHT=200
6
7  COTE=20
8
9  root = Tk()
10 cnv = Canvas(root, width=WIDTH, height=HEIGHT, background="ivory")
11 cnv.pack()
12
13 def dessiner(event):
14     n=int(my_entry.get())
15     color="#%s%s%s" %(randrange(10),randrange(10),randrange(10))
16     for i in range(n):
17         x=randrange(WIDTH)
18         y=randrange(HEIGHT)
19         cnv.create_rectangle(x, y, x+COTE, y+COTE, fill=color, outline='')
20     my_entry.delete(0, END)
21
22 my_entry = Entry(root)
23 my_entry.pack()
24 my_entry.bind('<Return>', dessiner)
25
26 root.mainloop()
```

— Ligne 21 : on efface l'entrée depuis le caractère d'indice 0 (le premier) jusqu'au dernier (`END`).

Gestion du curseur

Initialiser le placement du curseur

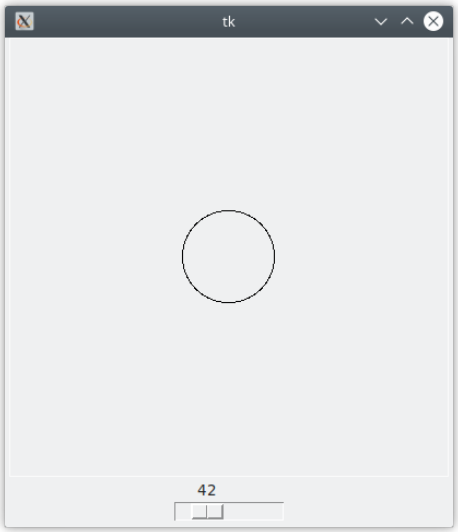
On peut modifier dynamiquement la position du curseur, en particulier à l'initialisation.
On reprend le code d'introduction du widget Slider :



et le code

```
1  from tkinter import *
2
3  root = Tk()
4  cnv = Canvas(root, width=400, height=400)
5  cnv.pack()
6
7  old=None
8
9  def rayon(r):
10     global old
11     r=int(r)
12     cnv.delete(old)
13     old=cnv.create_oval(200-r,200-r,200+r, 200+r)
14
15 curseur = Scale(root, orient = "horizontal", command=rayon, from_=0, to=200)
16 curseur.pack()
17
18 root.mainloop()
```

Initialement (`from_` dans la ligne 15), le curseur est placé à 0. Comment faire en sorte que le curseur soit placé conformément au rayon d'un cercle qui serait tracé au départ ? A l'exemple de la figure ci-dessous



où on voit que le curseur est, à l'ouverture de la fenêtre, positionné à 42 et que le cercle a un rayon de 42.

Deux méthodes sont possibles. La plus simple consiste à utiliser la méthode `set` du widget `Scale` :

```
1  from tkinter import *
2  from random import randrange
3
4  root = Tk()
5  cnv = Canvas(root, width=400, height=400)
6  cnv.pack()
7
8  old=None
9
10 def rayon(r):
11     global old
12     r=int(r)
13     cnv.delete(old)
14     old=cnv.create_oval(200-r,200-r,200+r, 200+r)
15
16 r=42
17 rayon(r)
18 curseur = Scale(root, orient = "horizontal", command=rayon,
19                  from_=0, to=200)
20 curseur.set(r)
21
22 curseur.pack()
23
24 root.mainloop()
```

— Ligne 18 : la position du curseur est initialisée et l'appel correspondant (avec `r=42`) de la fonction de commande est effectué.

Une autre méthode consiste à utiliser les variables dynamiques (ou encore appelées *variables de contrôle*) proposées par Tkinter. Ici, on va utiliser `IntVar` (le rayon est entier).

Voici le code :

```
1  from tkinter import *
2  from random import randrange
3
4  root = Tk()
5  cnv = Canvas(root, width=400, height=400)
6  cnv.pack()
7
8  old=None
9
10 def rayon(r):
11     global old
12     r=int(r)
13     cnv.delete(old)
14     old=cnv.create_oval(200-r,200-r,200+r, 200+r)
15
16 radius=IntVar()
17 r=42
18 radius.set(r)
19 rayon(r)
20 curseur = Scale(root, variable=radius, orient = "horizontal", command=rayon,
21                 from_=0, to=200)
22 curseur.pack()
23
24 root.mainloop()
```

— Ligne 20 : La position du curseur est contrôlée par l'objet défini par l'option `variable` (ligne 20).

— ligne 16 : Initialement, `variable` (ligne 20) pointe vers une variable dynamique `radius`.

— Ligne 18 : `radius` est initialisée à un rayon `r`. La conséquence est qu'à l'ouverture, le curseur pointera vers 42.

— Ligne 19 : L'initialisation de la variable de contrôle `radius` n'entraîne pas d'exécution de la fonction de commande `rayon`. Pour que la dimension du cercle soit en conformité avec la valeur indiquée par le curseur, le cercle est tracé.

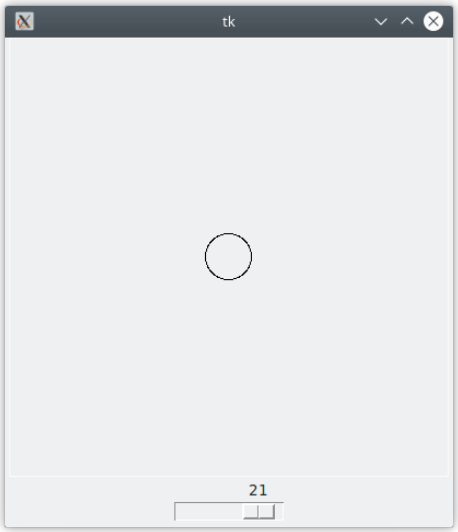
Changer le sens de progression

Par défaut, le sens croissant du curseur est défini par défaut :

— de la gauche vers la droite

— du haut vers le bas.

Pour changer ce sens il suffit d'inverser les valeurs donnée à `from_` et `to`. Ci-dessous, le générateur de cercle avec un curseur progressant vers la gauche et non vers la droite :



Le code correspondant est :

```
from tkinter import *

root = Tk()
cnv = Canvas(root, width=400, height=400)
cnv.pack()

old=None

def rayon(r):
    global old
    r=int(r)
    cnv.delete(old)
    old=cnv.create_oval(200-r,200-r,200+r, 200+r)

curseur = Scale(root, orient = "horizontal",
                command=rayon, from_=200, to=0)
curseur.pack()

root.mainloop()
```

Désactiver le curseur

L'état d'un curseur est déterminé par une variable `state` que l'on peut modifier pour le rendre inactif. Par défaut, l'état est `normal`. L'état déactivé est `disabled`. Par exemple, dans le code ci-dessous :

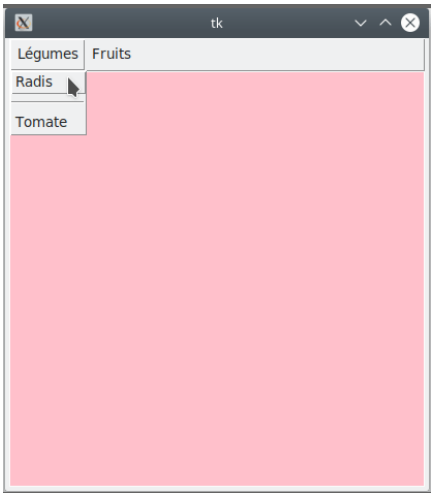
```
from tkinter import *

root = Tk()
cnv = Canvas(root, width=400, height=400)
```

```
5  cnv.pack()
6
7  old=None
8
9  def rayon(r):
10     global old
11     if int(r)>100:
12         curseur["state"]="disabled"
13     r=int(r)
14     cnv.delete(old)
15     old=cnv.create_oval(200-r,200-r,200+r, 200+r)
16
17 curseur = Scale(root, orient = "horizontal",
18                 command=rayon, from_=200, to=0)
19 curseur.pack()
20
21 root.mainloop()
```

dès que le curseur dépasse 100, il est désactivé (lignes 11-12).

Faire un menu déroulant

Voici un menu déroulant :



Le code correspondant est le suivant

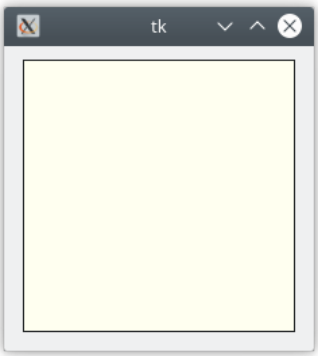
```
1  from tkinter import *
2
3
4  root = Tk()
5
6  cnv=Canvas(root, width=400, height=400, bg="lavender")
7  cnv.pack()
```

```
8
9  def rose():
10     cnv['bg']="pink"
11
12 def rouge():
13     cnv['bg']="red"
14
15 def orange():
16     cnv['bg']="orange"
17
18 def violet():
19     cnv['bg']="purple"
20
21 # Barre de menus
22 mon_menu = Menu(root)
23 root.config(menu=mon_menu)
24
25 # Menu légumes
26 legumes = Menu(mon_menu, tearoff=0)
27 legumes.add_command(label="Radis", command=rose)
28 legumes.add_separator()
29 legumes.add_command(label="Tomate", command=rouge)
30 mon_menu.add_cascade(label="Légumes", menu=legumes)
31
32 # Menu fruits
33 fruits = Menu(mon_menu, tearoff=0)
34 fruits.add_command(label="Raisin", command=violet)
35 fruits.add_command(label="Orange", command=orange)
36 mon_menu.add_cascade(label="Fruits", menu=fruits)
37
38 root.mainloop()
```

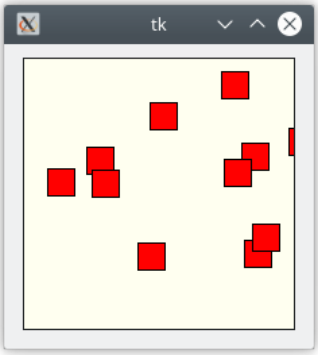
Prise de focus dans un canevas

Si un canevas est une surface devant capturer des événements du clavier, il faut lui donner le focus pour qu'il puisse réagir aux touches de clavier.

Soit l'application suivante qui à l'ouverture se présente ainsi :



Comme le montre le liseré noir autour du canevas, le canevas a le focus : il est en mesure de réceptionner des événements du clavier. En particulier ici, si l'utilisateur appuie sur n'importe quelle touche, cette action dessine un carré rouge aléatoire sur le canevas :



Voici le code correspondant :

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

SIDE=200

root = Tk()
cnv = Canvas(root, width=SIDE, height=SIDE, bg='ivory')
cnv.pack(padx=10, pady=10)
cnv.focus_set()

def dessiner(event):
    a=randrange(SIDE)
    b=randrange(SIDE)
    cnv. create_rectangle(a, b, a+20, b+20, fill="red")

cnv.bind('<Key>', dessiner)
```

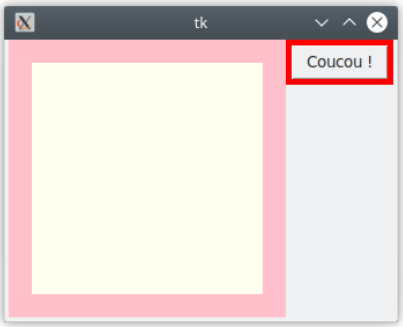
```
root.mainloop()
```

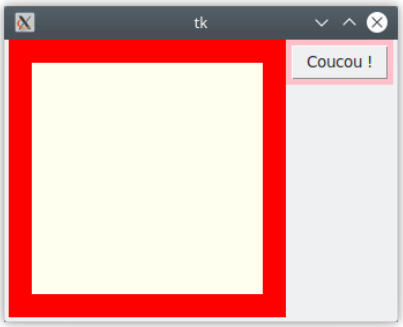
Pour modifier (voire faire disparaître) le liséré noir autour du canevas, il faut modifier une option du canevas nommé `highlightthickness`. Par défaut, il est de 1 pixel et de couleur noire.

Témoin de prise de focus

Les interfaces graphiques permettent de visualiser la partie de l'interface qui a le focus. Sous Tkinter, le widget qui a le focus est entouré d'une bande rectangulaire ;

Ci-dessous une interface composée de deux widgets, un canevas et un bouton, est présentée :





Le widget qui a le focus est entouré d'une bande rouge ; quand il perd le focus, cette bande devient rose. Si on appuie sur la touche TAB, le focus passe des widgets à l'autre et les couleurs rose et rouge sont échangées à chaque changement de focus.

Le code de l'interface précédente montre qu'on peut contrôler les couleurs et la dimension de la bande de témoin de focus :

```
1 from tkinter import Tk, Canvas, Button
2 from random import randrange
3
4 W=H=200
5 root=Tk()
6
7 cnv=Canvas(root, width=W, height=H, bg="ivory",
```

```
8             highlightthickness=20,
9             highlightbackground="pink",
10            highlightcolor="red",
11            takefocus=1)
12 cnv.pack(side="left")
13
14 Button(root, text='Coucou !',
15        highlightthickness=5,
16        highlightbackground="pink",
17        highlightcolor="red").pack()
18
19 def dessiner(event):
20     a=randrange(W)
21     b=randrange(W)
22     cnv.create_rectangle(a, b, a+10, b+10, fill="blue", outline='')
23
24 cnv.bind('<Return>', dessiner)
25
26 root.mainloop()
```

— Lignes 8 et 15 : `highlightthickness` est la largeur en pixels de la bande de focus. La largeur par défaut est de deux pixels.

— Lignes 9 et 16 : `highlightbackground` est la couleur de la bande lorsque le widget n'a pas le focus, ici `pink` ;

— Lignes 10 et 17 : `highlightcolor` est la couleur de la bande lorsque le widget a le focus, ici `red`.

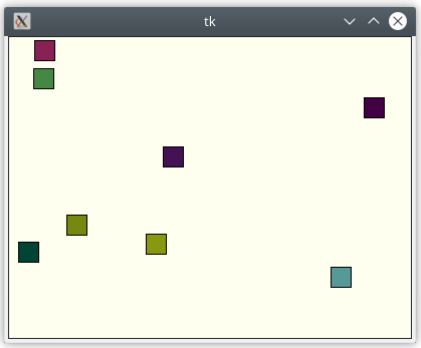
Dans l'interface précédente, lorsque le canevas a le focus, l'appui sur la touche ENTRÉE (cf. ligne 24) dessine un carré aléatoire orange sur le canevas (cf. lignes 18-21). Si le canevas n'a pas le focus, rien n'est dessiné.

Clavier et focus

Soit un jeu contenant plusieurs widgets, par exemple un canevas et une entrée (un widget où on peut entrer du texte au clavier). On veut que le canevas réagisse à certains événements du clavier, par exemple, si on appuie sur la barre d'espace, le joueur saute. Comme le jeu contient plusieurs widgets, chaque widget a sa propre façon d'écouter le clavier, un appui sur ESPACE dans le widget entrée n'aura pas même signification que pour le canevas. Il faut donc qu'il y ait une manière de choisir l'interlocuteur entre le clavier et les widgets. C'est ce qu'on appelle le **focus**.

Le focus est acquis en général par appuis successifs sur la touche TAB ou par clic de souris (si vous cliquez sur un champ d'entrée, le champ prend le focus, comme dans un formulaire d'une page web). Pour faire en sorte qu'un widget prenne automatiquement le focus, on utilise la méthode `focus_set` du widget.

Voici un exemple avec un canevas :



quand on appuie sur la touche ENTRÉE, un carré aléatoire est dessiné sur le canevas. Voici le code correspondant :

```
1  from tkinter import *
2  from random import randrange
3
4
5  WIDTH=400
6  HEIGHT=300
7
8  root = Tk()
9  cnv = Canvas(root, width=WIDTH, height=HEIGHT, background="ivory")
10 cnv.pack()
11
12 COTE= 20
13 cnv.focus_set()
14
15 def f(event):
16     a=randrange(WIDTH)
17     b=randrange(HEIGHT)
18     color="#%s%s%s" %(randrange(10),randrange(10),randrange(10))
19     cnv.create_rectangle(a, b, a+COTE, b+COTE, fill=color)
20
21 cnv.bind('<Return>', f)
22
23 root.mainloop()
```

— Ligne 21 : l'appui sur la touche ENTRÉE (si le canevas a le focus) déclenche l'exécution de la fonction `f`.

— Ligne 13 : la canevas prend le focus ; il pourra réagir au événement du clavier. Sans cette ligne, le canevas serait inerte, il faudrait lui donner le focus manuellement, soit en appuyant sur la touche TAB soit en cliquant dans le clavier.

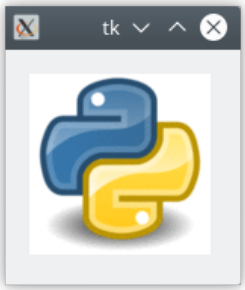
Si la fenêtre contient le canevas comme seul widget, il est plus simple de lier la fonction `f` à la fenêtre tout entière au lieu du seul canevas. Il suffit pour cela de changer la ligne 21 en :

```
1 root.bind('<Return>', f)
```

et on peut alors supprimer la ligne 13 de prise de focus : quand on appuiera sur la touche ENTREE, automatiquement, le canevas capturera l'appui sur la touche.

Image dans un label

Un label peut aussi porter une image au lieu de texte :



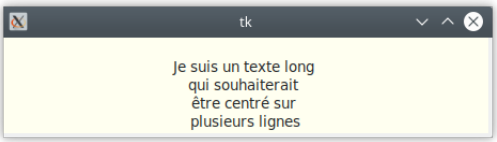
Pour cela, on convertit d'abord avec le fichier image avec `PhotoImage` :

```
1 from tkinter import *
2
3 root = Tk()
4 logo = PhotoImage(file="python.gif")
5 Label(root, image=logo).pack(padx=15, pady=15)
6
7 root.mainloop()
```

et on utilise l'option `image` (ligne 5) de `Label`.
Si l'option `image` est utilisée, l'option `text` sera sans effet.

Centrer un texte dans un label

Un label admet une option d'alignement (option `justify`) permettant le centrage :




```
1 from tkinter import Tk, Label, CENTER
2
3 racine=Tk()
4
5 mon_texte="""
```

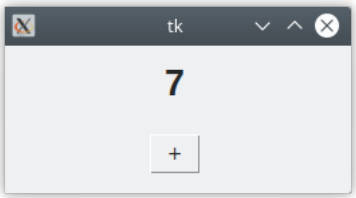
```
6 Je suis un texte long
7 qui souhaiterait
8 être centré sur
9 plusieurs lignes"""
10 annonce=Label(height=5, width= 50, text=mon_texte,
11                justify=CENTER, bg='ivory')
12 annonce.pack()
13 racine.mainloop())
```

— Ligne 11 : l'option `justify` est placée à `CENTER` ce qui centre le texte ligne par ligne.

— ligne 11 : pour rendre le texte plus visible, une couleur de fond a été utilisée dans le widget (option `bg` pour *background*).

Utilisation d'une variable de contrôle dans un label

Soit l'interface suivante :



Quand l'utilisateur clique sur le bouton, un label augmente le compteur d'une unité.

On peut effectuer la mise à jour du label en utilisant une variable de contrôle de Tkinter, ici `StringVar` :

```
1 from tkinter import *
2
3 def plus():
4     cpt.set(int(cpt.get())+1)
5
6 root=Tk()
7 cpt=StringVar()
8 cpt.set('0')
9 lbl=Label(root, width='10', textvariable=cpt, font='Arial 20 bold')
10 lbl.pack(side=TOP, padx=50, pady=10)
11
12 bouton=Button(root, text="+", command=plus)
13 bouton.pack(side=TOP, padx=50, pady=10)
14
15 root.mainloop()
```

— Ligne 9 : le texte du label est couplé à une variable de contrôle `cpt` définie antérieurement (ligne 7).

— Lignes 12 et 3-4 : quand le bouton reçoit un clic, la fonction sans paramètre `plus` est appelée (ligne 3-4) et son code est exécuté. Ce code récupère la valeur de cpt avec la méthode `get` ; cette valeur est une chaîne représentant un entier. On convertit cette chaîne en entier, on incrémente et on met à jour la variable de contrôle `cpt`. **Cela a pour effet de mettre à jour le label**.

On peut aussi se passer de `StrVar` et modifier directement la valeur du label :

```
1 from tkinter import *
2
3 def plus():
4     entree['text']=entree['text']+1
5
6 root=Tk()
7
8 entree=Label(root, width='10', text=0, font='Arial 20 bold',)
9 entree.pack(side=TOP, padx=50, pady=10)
10
11 bouton=Button(root, text="+", command=plus)
12 bouton.pack(side=TOP, padx=50, pady=10)
13
14 root.mainloop()
```

— Ligne 8 : l'option `text` accepte une entrée numérique (qui n'est pas une chaîne de caractères).

— Lignes 8 et 3-4 : la fonction de rappel `plus` met directement à jour le contenu de l'entrée (ligne 4).

Le widget Frame

Le widget `Frame` (*cadre* en français) est un widget qui sert juste de conteneur, un peu comme une fenêtre `Tk`. Ce type de widget est très utile pour regrouper et organiser des interfaces contenant beaucoup de widgets.

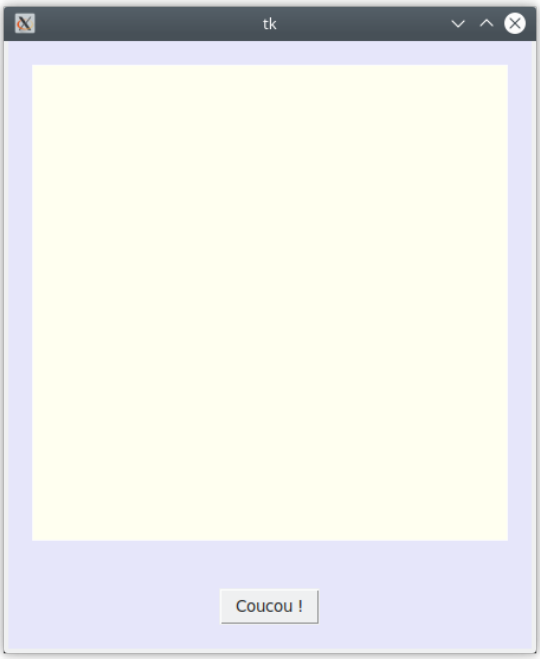
Voici un exemple d'utilisation :

```
1 from tkinter import *
2
3 SIDE = 400
4 root = Tk()
5 frame = Frame(root, background="lavender")
6 frame.pack()
7 cnv = Canvas(frame, width=SIDE, height=SIDE, bg='ivory')
8 cnv.pack(padx=20, pady=20)
9
10 btn = Button(frame, text="Coucou !")
11 btn.pack(pady=20)
12
13 root.mainloop()
```

— ligne 6 : `frame` est un widget comme un autre et il doit donc être placé avec un gestionnaire de géométrie, ici `pack`.

— ligne 5 : on peut donner une couleur de fond à un frame (on ne peut pas pour un fenêtre `Tk`).
— lignes 7 et 10 : un frame est un conteneur et ici, il est le premier argument des constructeurs `Canvas` et `Button`.

qui affiche :



On peut donner une largeur et une hauteur à un frame mais la plupart du temps, ces dimensions sont ignorées (sauf dans certains cas où on inhibe la propagation des dimensions des widgets).

Boutons radio

Typiquement, on utilise des boutons radio dans des situations de choix conduisant à une **unique** réponse comme entre un nombre strictement positif, strictement négatif ou nul :

Voici le code correspondant :

```
1  from tkinter import *
2
3  root = Tk()
4
5  pad = 30
6
7  mineur = Radiobutton(
8      root, variable=StringVar(), text="Positif", font="arial 20")
9  mineur.pack(anchor="w", padx=pad, pady=pad)
10
11 majeur = Radiobutton(
12     root, variable=StringVar(), text="Négatif", font="arial 20")
13 majeur.pack(anchor="w", padx=pad, pady=pad)
14
15 a = Radiobutton(root, variable=StringVar(), text="Nul", font="arial 20")
16 a.pack(anchor="w", padx=pad, pady=pad)
17
18 root.mainloop()
```

— Lignes 7-9 : un bouton radio est un widget comme un autre ; on le crée avec un constructeur, ici `RadioButton` et on le place comme n'importe quel widget, ici avec la méthode `pack` (et des options d'alignement pour que la sortie ne soit pas trop disgrâcieuse).

— Lignes 11-13 et 15-16 : on crée et on place de même deux autres boutons radio.

— Pour chaque bouton, on dispose d'une option de texte descriptif (`text`) et de police pour le texte (`font`).

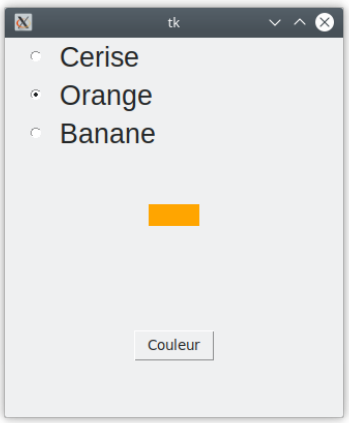
Dans l'exemple ci-dessus, les boutons radio sont purement factices et rien n'est prévu pour écouter s'ils sont cochés ou pas.

Un exemple avec interaction

Les boutons radio fonctionnent, en principe, par groupes. Par exemple, dans un QCM de 10 questions, il y aura 10 groupes de boutons radio. Toutefois, le programmeur va définir autant de boutons radio qu'il a besoin (sans tenir compte des groupes) et c'est ensuite **lui** qui définera les groupes (expliqué plus loin).

Dans ce qui suit, on va examiner une interface qui montre comment le programme peut récupérer les informations fournies par un groupe de boutons radio.

L'interface contient cinq widgets :



— trois boutons radio qui référencent des fruits,
— un label qui va montrer leur couleur,
— un bouton `Couleur` qui lorsqu'on le clique, colorie le label avec la couleur du fruit.

Au départ, aucune couleur n'apparaît et, par défaut, c'est le bouton radio de la cerise qui est activée. Voici le code :

```
from tkinter import *

root = Tk()


def colorer():
    val = fruit.get()
    if val == "cerise":
        lbl_fruit["bg"] = "red"
    elif val == "orange":
        lbl_fruit["bg"] = "orange"
    else:
        lbl_fruit["bg"] = "yellow"


fruit = StringVar()
fruit.set("cerise")

cerise = Radiobutton(
    root,
    text="Cerise",
    variable=fruit,
    value="cerise",
    font="arial 20")
cerise.pack(anchor="w")

orange = Radiobutton(
    root,
```

```
29      text="Orange",
30      variable=fruit,
31      value="orange",
32      font="arial 20")
33  orange.pack(anchor="w")
34
35  banane = Radiobutton(
36      root,
37      text="Banane",
38      variable=fruit,
39      value="banane",
40      font="arial 20")
41  banane.pack(anchor="w")
42
43  lbl_fruit = Label(root, bg="white", width=5)
44  lbl_fruit.pack(padx=50, pady=50)
45
46  btn = Button(root, text="Couleur", command=colorer)
47  btn.pack(padx=50, pady=50)
48
49  root.mainloop()
```

— ligne 16 : une variable de contrôle (spécificité de Tkinter) initialisée à la chaîne `cerise`. Elle initialise les options `variable` des trois boutons radio.

— Lignes 22 et 23 : les boutons radio sont munis de 2 nouvelles options : `variable` et `value`. Si le bouton radio est activé (donc, on a cliqué dessus), la valeur de la variable de contrôle sera `value`.

— Le point qui suit est essentiel : les trois boutons fonctionnent comme un groupe (un seul des trois doit être coché) ; pour le faire comprendre à Tkinter, il faut que leur option `variable` pointe vers la **même** variable de contrôle (lignes 22, 30 et 38), ici appelée `fruit` (ligne 16).

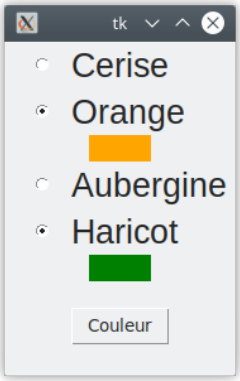
— Ligne 7 : pour savoir quel bouton est coché, il suffit de regarder le contenu de la variable de contrôle et de le comparer aux différentes `value` des boutons.

Si les boutons radio sont nombreux, on utilisera plutôt une boucle `for` pour les créer et les placer. A noter que si votre interface a une certaine complexité et contient des boutons radio, il peut être approprié d'utiliser le placement des widgets en grid.

Cas de plusieurs groupes

Si vous avez plusieurs groupements de boutons radio, il faut associer à **chaque** groupement sa propre variable de contrôle.

Par exemple, si on avait aussi un groupe de boutons radio pour des légumes :




```
from tkinter import *

root = Tk()


def colorer():
    if fruit.get() == "cerise":
        lbl_fruit["bg"] = "red"
    else:
        lbl_fruit["bg"] = "orange"
    if legume.get() == "aubergine":
        lbl_legume["bg"] = "purple"
    else:
        lbl_legume["bg"] = "green"


fruit = StringVar()
fruit.set("cerise")
legume = StringVar()
legume.set("aubergine")

cerise = Radiobutton(
    root,
    text="Cerise",
    variable=fruit,
    value="cerise",
    font="arial 20")
cerise.pack(anchor="w")

orange = Radiobutton(
    root,
    text="Orange",
    variable=fruit,
    value="orange",
    font="arial 20")
```

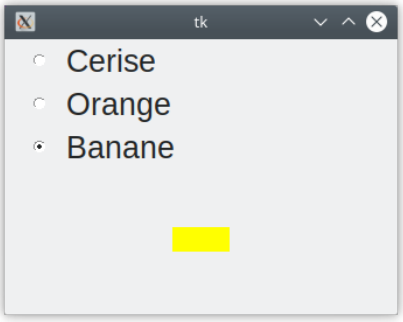
```
36 orange.pack(anchor="w")
37
38 lbl_fruit = Label(root, bg="white", width=5)
39 lbl_fruit.pack()
40
41 aubergine = Radiobutton(
42     root,
43     text="Aubergine",
44     variable=legume,
45     value="aubergine",
46     font="arial 20")
47 aubergine.pack()
48
49 haricot = Radiobutton(
50     root,
51     text="Haricot",
52     variable=legume,
53     value="haricot",
54     font="arial 20")
55 haricot.pack(anchor="w")
56
57 lbl_legume = Label(root, bg="white", width=5)
58 lbl_legume.pack()
59
60 btn = Button(root, text="Couleur", command=colorer)
61 btn.pack(padx=20, pady=20)
62
63 root.mainloop()
```

— Lignes 17 : une première variable de contrôle pour les fruits (cf. lignes 25 et 33).

— Lignes 19 : une seconde variable de contrôle pour les légumes (cf. lignes 44 et 52).

Déclencher une action

Comme pour d'autres widgets (bouton, entrée), un bouton radio dispose d'une option permettant de générer une action (appel d'une fonction) lorsqu'un bouton est modifié. Par exemple, on peut modifier le code ci-dessus pour que, lorsque l'utilisateur clique sur un bouton radio, la couleur du fruit sélectionné apparaise dans le label :



```
from tkinter import *

root = Tk()


def colorer():
    val = fruit.get()
    if val == "cerise":
        lbl_fruit["bg"] = "red"
    elif val == "orange":
        lbl_fruit["bg"] = "orange"
    else:
        lbl_fruit["bg"] = "yellow"


fruit = StringVar()
fruit.set("cerise")

cerise = Radiobutton(
    root,
    text="Cerise",
    variable=fruit,
    value="cerise",
    font="arial 20",
    command=colorer)
cerise.pack(anchor="w")

orange = Radiobutton(
    root,
    text="Orange",
    variable=fruit,
    value="orange",
    font="arial 20",
    command=colorer)
orange.pack(anchor="w")

banane = Radiobutton(
    root,
    text="Banane",
    variable=fruit,
    value="banane",
    font="arial 20",
    command=colorer)
banane.pack(anchor="w")

lbl_fruit = Label(root, bg="white", width=5)
lbl_fruit.pack(padx=50, pady=50)

```

```
49 root.mainloop()
```

— chaque boution radio (par exemple ligne 25) contient une option `command` pointant vers la fonction `colorer` (sans argument, comme l'impose Tkinter)
— Lorsqu'un bouton est cliqué, la valeur commune de la variable de contrôle `fruit` (ligne 16) est examinée et la coloration du label est déclenchée (lignes 9-13).

Le widget case à cocher

Tkinter met à disposition le widget `Checkbutton` montrant du texte et une case à cocher :

Dans l'application ci-dessus, la label affiche dynamiquement la somme des valeurs des cases cochées.

Les widgets `Checkbutton` vont souvent par groupes mais il faut créer autant de widgets `Checkbutton` que de cases à cocher. A la différence des boutons-radio dont un seul est activable, on peut cocher plusieurs cases.

Ci-dessous, on crée 3 simples cases à cocher :


```
from tkinter import *

root=Tk()

check1=Checkbutton(root, text=2, font='arial 30')
check1.grid()

check2=Checkbutton(root, text=3, font='arial 30')
```

```
check2.grid(row=1)

check3=Checkbutton(root, text=4, font='arial 30')
check3.grid(row=2)

root.mainloop()
```

La question qui se pose maintenant est de savoir si une case d'un Checkbutton `check` a été cochée. Pour cela, il suffit d'examiner l'option `variable` ; c'est une variable de contrôle Tkinter qui renvoie l'entier 1 si la case est cochée et 0 sinon. En réalité, il semble qu'il n'y ait pas d'autre façon d'avoir accès à cette information.

Utiliser un callback par case

Par ailleurs, on peut définir, pour chaque case à cocher, une fonction référencée par l'option `command` et qui sera appelée chaque fois que la case sera modifiée. Cela fournit une première méthode pour réaliser l'application présentée tout au début :

```
1  from tkinter import *
2
3  root = Tk()
4
5
6  def f(i):
7      def g():
8          s = sum(check["text"] * v.get() for (check, v) in checks)
9          lbl["text"] = "Somme :  %s" % s
10
11     return g
12
13
14 checks = []
15
16 for i in range(3):
17     v = IntVar()
18     check = Checkbutton(
19         root, text=i + 2, font='arial 30', command=f(i), variable=v)
20     check.grid(row=i)
21     checks.append((check, v))
22
23 lbl = Label(root, text="Somme :  0", font='arial 30')
24 lbl.grid(row=1, column=1, padx=100)
25
26 root.mainloop()
```

— Lignes 16-21 : on crée les 3 cases à cocher ; l'état de la case (cochée ou pas) est contrôlé par la variable de contrôle `v` (c'est indispensable ici). Le fonctionnement des cases à cocher défini par Tkinter stipule que le contenu de cette variable est l'entier 1 si la case est cochée, 0 sinon.

— Ligne 19, option `text` : le texte de la case à l'indice `i` est `i+2`.

— Ligne 20 : on utilise la méthode `grid` pour placer les cases.

— Lignes 14 et 21 : les 3 widgets ainsi que la variable de contrôle correspondante sont placés en tuple dans une liste.

— Ligne 19, option `command` : chaque bouton réagit au clic sur la case. Si le bouton est d'indice `i`, la fonction qui est appelée est `g=f(i)` (ligne 11). Noter que `f` est une fonction qui renvoie une autre fonction.

— Lorsqu'une case à cocher est modifiée, sa fonction de rappel g (ligne 7), qui dispose de l'indice `i` de la case modifiée va exécuter les actions suivantes :

  — elle recalcule (ligne 8) la nouvelle somme : en effet, si une case est décochée, `v.get()` vaut 0 et donc la valeur du texte n'est pas comptée et sinon, `v.get()` vaut 1 et donc la valeur du texte est comptée ;

  — elle met alors le label à jour (ligne 9).

N'utiliser que des variables de contrôle Tkinter

Une autre façon de mettre à jour le label est d'appeler une fonction chaque fois qu'une des variables contrôlant chaque case à cocher est modifiée. Cela se fait en utilisant la méthode `trace` d'une variable de contrôle. Le code suivant montre le principe de la méthode `trace` :

```
def f(*args):
    print(v.get())


v=IntVar()

# Autre code où v est modifiée en écriture

v.trace("w", f)
```

Chaque fois que la variable de contrôle `v` est modifiée ailleurs dans le programme, la fonction `f` est automatiquement appelée ; ici, cette fonction affiche la valeur contenue dans v. L'argument `"w"` de `trace` signifie *write* (modification en écriture de `v`).

Voici le code complet de l'application :

```
1  from tkinter import *
2
3  root=Tk()
4
5  def update(*args):
6      s=sum(check["text"]*v.get() for (check,v) in checks)
7      lbl["text"]="Somme :  %s" %s
8
9  checks=[]
10
11 for i in range(3):
12     v=IntVar()
13     check=Checkbutton(root, text=i+2, font='arial 30', variable=v)
14     check.grid(row=i)
15     v.trace("w", update)
16     checks.append((check,v))
```

```
17
18 lbl=Label(root, text="Somme :  0", font='arial 30')
19 lbl.grid(row=1, column=1, padx=100)
20
21 root.mainloop()
```

— ligne 15 : chaque variable de contrôle v appelera la fonction `update` lorsque la case correspondante sera modifiée.
— Lignes 5-7 : la fonction `update` calcule la somme et met à jour le texte du label.

Le widget Listbox

Le widget Listbox propose une liste dont les éléments sont sélectionnables à la souris ou avec les flèches *Haut* et *Bas* du clavier :

On crée ce widget avec le constructeur `Listbox` (une seule majuscule). Voici le code correspondant à la figure ci-dessus :

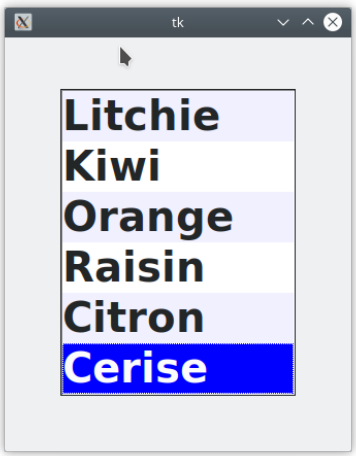
```
1 from tkinter import *
2
3 fruits = ["Litchie", "Kiwi", "Orange", "Raisin", "Citron", "Cerise"]
4 n = len(fruits)
5
6 master = Tk()
7
8 lbox = Listbox(
9     master,
10    width=8,
11    height=n,
12    font="Verdana 30 bold",
13    selectbackground="blue")
14 lbox.pack(padx=50, pady=50)
15
16 for item in fruits:
```

```
17      lbox.insert(END, item)
18
19  mainloop()
```

— Ligne 3 : la liste des chaînes qui vont apparaître dans le widget.
— Lignes 8-14 : construction et placement du widget
— Lignes 9-13. Certaines options du widget :
  — `width` est le nombre de caractères (pas de pixels)
  — `height` est le nombre d'entrées devant apparaître dans la liste
  — `font` : la police utilisée
  — `selectbackground` : couleur du fond d'une cellule sélectionnée.
— Lignes 16-17 : insertion des éléments dans la liste du widget ; `END` signifie qu'on place l'item courant après le dernier placé.

En français, une listbox s'appelle une *liste de sélection*. On ne peut pas placer le texte d'un item sur plusieurs lignes, comme expliqué par Bryan Oakley.

Il existe de nombreuses autres options et ce widget dispose d'un nombre important de méthodes, en particulier de modification des items et dont je ne parlerai pas. Il est possible de faire de la multi-sélection, voir la documentation de `selectmode`. Signalons par exemple la possibilité de :



— placer le focus avec la méthode `focus_set` (et qui n'est pas propre à ce widget) : sans toucher à la souris, on peut se déplacer dans la liste avec la souris ;
— sélectionner une cellule par son indice (à partir de 0)
— modifier la couleur de certaines cellules avec la méthode `itemconfigure`.

Voici un code qui utilise ces méthodes :

```
1  from tkinter import *
2
3  fruits = ["Litchie", "Kiwi", "Orange", "Raisin", "Citron", "Cerise"]
4  n = len(fruits)
5
```

```
6  master = Tk()
7
8  lbox = Listbox(
9      master,
10     width=8,
11     height=n,
12     font="Verdana 20 bold",
13     selectbackground="blue")
14 lbox.pack(padx=20, pady=20)
15
16 for item in fruits:
17     lbox.insert(END, item)
18
19 lbox.focus_set()
20 lbox.selection_set(2)
21
22 for i in range(0, len(fruits), 2):
23     lbox.itemconfigure(i, background='#f0f0ff')
24 for i in range(1, len(fruits), 2):
25     lbox.itemconfigure(i, background='#fff')
26
27 mainloop()
```

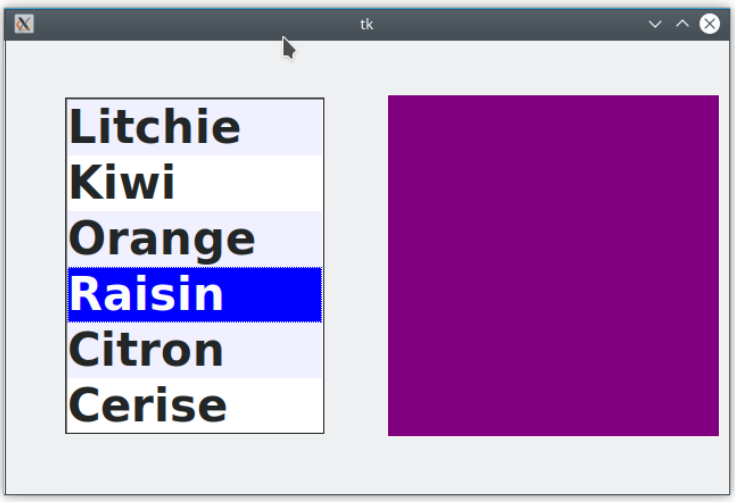
— Lignes 19-20 : placement du focus et sélection du 3e item.

— Lignes 22-25 : on a colorié avec des couleurs alternées les lignes successives de la liste.

Il est également possible de définir les entrées de la liste comme des variables dynamiques Tkinter, voir le paragraphe *Modification des items d'une listbox*.

Action associée

Bien entendu, le déplacement et la sélection d'éléments dans la liste sont souvent associés à une action. Par exemple, dans l'exemple ci-dessous, le déplacement dans la liste affiche dans un canevas une couleur adaptée à la sélection :



Voici le code correspondant :

```
1  from tkinter import *
2
3  fruits = ["Litchie", "Kiwi", "Orange", "Raisin", "Citron", "Cerise"]
4  couleurs = ["pink", "lightgreen", "orange", "purple", "yellow", "red"]
5  n = len(fruits)
6
7  master = Tk()
8
9  lbox = Listbox(
10     master,
11     width=8,
12     height=n,
13     font="Verdana 20 bold",
14     selectbackground="blue")
15 lbox.pack(padx=20, pady=20, side=LEFT)
16
17 for item in fruits:
18     lbox.insert(END, item)
19
20 lbox.focus_set()
21 pos = 1
22 lbox.activate(pos)
23 lbox.selection_set(pos)
24
25 for i in range(0, len(fruits), 2):
26     lbox.itemconfigure(i, background='#f0f0ff')
27 for i in range(1, len(fruits), 2):
28     lbox.itemconfigure(i, background='#fff')
29
30
31 def show(event):
32     index = lbox.curselection()[0]
33     cnv["bg"] = couleurs[index]
34
35
36 lbox.bind('<<ListboxSelect>>', show)
37
38 cnv = Canvas(master, width=200, height=200, bg="ivory")
39 cnv.pack(padx=5, pady=5, side=RIGHT)
40 cnv["bg"] = couleurs[pos]
41
42 mainloop()
```

— Ligne 4 : on crée, en respectant l'ordre, une liste des couleurs associées aux différents fruits.

— Ligne 38 : on crée un canevas qui va afficher la couleur de l'élément sélectionné

— Ligne 36 : le point essentiel : l'événement virtuel `<<ListboxSelect>>` qui réagit chaque fois qu'une cellule est sélectionnée en appelant une fonction, ici la fonction `show` qui va placer la

bonne couleur sur le canevas.

— Ligne 32 : la méthode `curselection` de `Listbox` permet de récupérer l'indice de la cellule sélectionnée (rappel : début à l'indice 0).

Modification des items d'une listbox

On peut aussi modifier le contenu du texte d'une ligne d'une listbox. Dans l'exemple ci-dessous :

initialement le texte est en minuscule et au bout de deux secondes, il est placé en majuscule. Le code correspondant est :

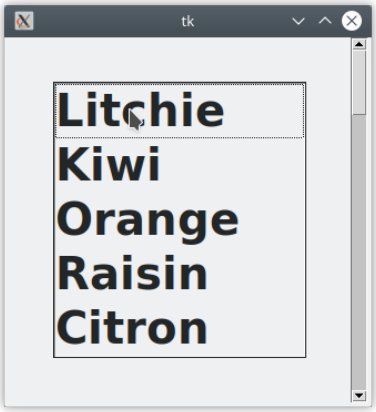
```
1 from tkinter import *
2
3 root=Tk()
4 MOIS=['janvier', 'février', 'mars', 'avril', 'mai', 'juin', 'juillet',
5       'août', 'septembre', 'octobre', 'novembre', 'décembre']
6 z=StringVar(value=MOIS)
7 lb=Listbox(root, width=16, fg='orange', listvar =z,
8            selectbackground='pink', font="Arial 16 bold")
9 lb.pack(side=LEFT, padx=10, pady=10)
```

```
10
11
12 def modif():
13     for i in range(len(MOIS)):
14         MOIS[i]=MOIS[i].upper()
15     z.set(MOIS)
16
17 root.after(2000, modif)
18
19 root.mainloop()
```

— Ligne 17 : au bout de 2,5 secondes après l'ouverture de la fenêtre, la casse des lignes de la listbox est changée.
— Ligne 6 : le texte de chaque ligne est initialement placé dans une variable dynamique Tkinter.
— Ligne 15 : la méthode `set` de la variable dynamique permet une mise à jour automatique.

Une Listbox et une barre de défilement

Si une Listbox contient beaucoup d'items, on peut parcourir son contenu avec une barre de défilement :



Voici le code correspondant :

```
1 from tkinter import *
2 from tkinter import ttk
3
4 fruits = ["Litchie", "Kiwi", "Orange", "Raisin", "Citron", "Cerise"] * 5
5
6 master = Tk()
7
8 lbox = Listbox(
9     master,
10     width=8,
11     height=5,
12     font="Verdana 30 bold",
```

```
13      selectbackground="blue")
14  lbox.pack(side=LEFT, padx=40, pady=40)
15
16  for item in fruits:
17      lbox.insert(END, item)
18
19  sbar = ttk.Scrollbar(master, command=lbox.yview)
20  sbar.pack(side=LEFT, expand=True, fill=Y)
21  lbox.config(yscrollcommand=sbar.set)
22
23  mainloop()
```

— Ligne 1-2 : pour des raisons d'esthétique, j'ai choisi d'utiliser la barre de Ttk.

— Ligne 4 : la liste contient 30 items.

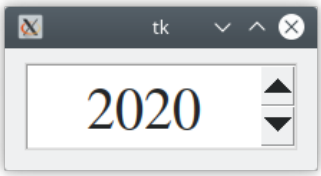
— Ligne 19 : on place une barre (par défaut verticale) ; sa commande pointe vers le déplacement vertical des items de la liste lbox.

— Ligne 20 : on place soigneusement la barre pour qu'elle recouvre tout le côté de la liste.

— Ligne 21 : on apparie cette fois la listbox à la barre.

Le widget spinbox

Un spinbox est un widget hybride :

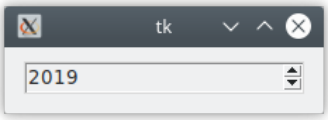


Il dispose d'une **zone de texte** et deux **boutons** de défilement. A partir d'une succession d'éléments textuels, dans l'exemple les nombres de 2019 à 2024, ce widget permet de faire défiler ces élements dans la zone de texte en progressant dans un sens ou dans l'autre selon les boutons qui vont recevoir un clic. Voici le code d'une version minimaliste :

spinbox_minimal.py

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

sp= Spinbox(root, from_=2019, to=2024)
sp.pack(padx=10, pady=10)

root.mainloop()
```

qui produit :

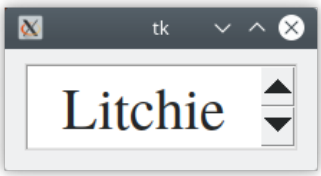


On peut aussi parcourir une liste de chaînes dans la zone de texte :

spinbox_minimal_texte.py

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

fruits = ["Litchie", "Kiwi", "Orange", "Raisin", "Citron", "Cerise"]

sp= Spinbox(root, values=fruits)
sp.pack(padx=10, pady=10)

root.mainloop()
```

qui produit :



L'intérêt de ce widget est la compacité : il permet de circuler relativement facilement entre tous les éléments d'une succession tout en prenant le minimum de place.

Ce widget contient de nombreuses options. Voici un exemple de l'usage de quelques unes :

spinbox_simple_texte.py

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

fruits = ["Litchie", "Kiwi", "Orange", "Raisin", "Citron", "Cerise"]

sp = Spinbox(
    root,
    values=fruits,
    width=8,
    bg="white",
    justify=CENTER,
    wrap=True,
    font="times 30")
sp.pack(padx=10, pady=10)
```

```
15
16 root.mainloop()
```

— Ligne 9 : l'option `width` est le nombre de caractères visibles dans la zone de texte.
— Ligne 10 : `bg` est la couleur de fond.
— Ligne 11 : `justify` permet de disposer le texte dans la zone
— Ligne 13 : `font` permet de modifier la police et la taille du texte.
— Ligne 12 : par défaut, quand le défilement arrive en bout de succession, le défilement est bloqué. L'option `wrap` permet d'aller en un clic à l'autre extrémité de la succession.

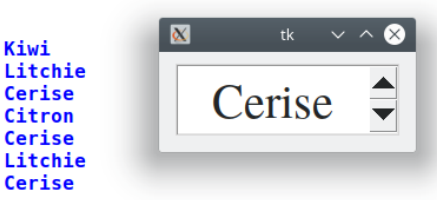
La zone de texte est en fait une entrée (widget de la classe `Entry`) et est donc modifiable. Toutefois, comme une liste de chaînes à afficher est en général prédéfinie, l'intérêt de pouvoir modifier directement la zone de texte n'est pas immédiat. Sans compter la possibilité d'écraser accidentellement une valeur. Et justement, il est possible de faire en sorte que la zone de texte soit non modifiable. Il suffit pour cela d'utiliser l'option `state="readonly"`. Toutefois, ce changement d'état modifie la couleur de fond de la zone de texte pour bien montrer qu'elle est désactivée, ce qui n'est pas forcément souhaité. On peut y remédier en définissant une option `readonlybackground`. Voici un exemple :

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

fruits = ["Litchie", "Kiwi", "Orange", "Raisin", "Citron", "Cerise"]

sp = Spinbox(
    root,
    values=fruits,
    width=8,
    justify=CENTER,
    wrap=True,
    font="times 30",
    state="readonly",
    readonlybackground="white")
sp.pack(padx=10, pady=10)

root.mainloop()
```

La sortie est analogue à la sortie précédente.

Exécution d'actions

On peut faire en sorte qu'une action soit exécutée chaque fois que la zone de texte est modifiée par appui sur un des deux boutons. Pour cela, comme pour la plupart des widgets de Tkinter, on utilise l'option `command`. Pour faire simple, on va parcourir des nombres et on va afficher dans la console le nombre placé dans la zone de texte du spinbox après clic :



Voici le code correspondant :

spinbox_command.py

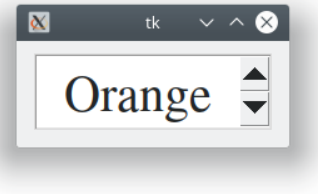
```
1  from tkinter import *
2
3
4  def f():
5      print(sp.get())
6
7
8  root = Tk()
9
10 fruits = ["Litchie", "Kiwi", "Orange", "Raisin", "Citron", "Cerise"]
11
12 sp = Spinbox(
13     root,
14     values=fruits,
15     width=8,
16     bg="white",
17     justify=CENTER,
18     wrap=True,
19     font="times 30",
20     command=f)
21 sp.pack(padx=10, pady=10)
22
23 root.mainloop()
```

— Ligne 20 : l'option `command` qui pointe vers une fonction `f`

— Ligne 4 : la fonction `f` utilisée lorsque un des deux boutons du spinbox est pressé.

Si on veut différencier le bouton sur lequel un clic a été reçu (le haut ou le bas), il faut procéder autrement (trouvé sur SO) : on définit toujours une option `command` mais la commande indiquée va être encapsulée avec la méthode `register`. Avec un exemple, ce sera plus simple à comprendre :

```
up
up
up
up
up
down
down
down
up
down
```



spinbox_command_plus.py

```
1  from tkinter import *
2
3  def f(drn):
4      print(drn)
5
6
7  root = Tk()
8
9  g=root.register(f)
10
11 fruits = ["Litchie", "Kiwi", "Orange", "Raisin", "Citron", "Cerise"]
12
13 sp = Spinbox(
14     root,
15     values=fruits,
16     width=8,
17     bg="white",
18     justify=CENTER,
19     wrap=True,
20     font="times 30",
21     command=(g, "%d"))
22 sp.pack(padx=10, pady=10)
23
24 root.mainloop()
```

— Ligne 22 : la commande est fournie sous forme de tuple

— Ligne 22 : le premier élément du tuple est le retour d'un appel à la méthode `register`, cf. ligne 9. Cette dernière encapsule une fonction, ici `f` (ligne 3).

— Ligne 22 : les éléments suivants du tuple représentent le type des paramètres que la fonction encapsulée va recevoir (ici, un seul paramètre). Ces types sont conventionnelement définis par Tkinter. Ici par exemple, le type `%d` représente une chaîne de caractère valant `up` si la flèche *haut* a été pressée et `down` si c'est la flèche *bas*.

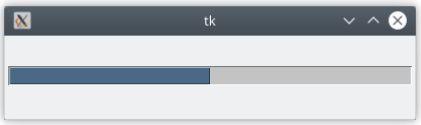
Bien qu'on parcoure une liste ou une succession, le widget ne semble pas donner la possibilité d'accéder à l'indice courant dans la succession de la zone de texte visible.

Il n'existe pas de version ttk du widget spinbox.

En français, la widget spinbox devrait se dire *saisie rotative*.

Barre de progression Ttk

Le module standard Ttk propose une barre de progression. Voici un exemple minimal :




```
1 from tkinter import Tk, ttk
2
3 root=Tk()
4
5 progress = ttk.Progressbar(root, length=400,  maximum= 300)
6 progress.pack(pady=30)
7
8 progress.start(10)
9
10 root.mainloop()
```

— Ligne 1 : il faut impérativement importer `ttk` **explicitement** de `tkinter`. Une importation de la forme `from tkinter import *` ne donnera pas accès à Ttk.

— Lignes 5-6 : une barre de progression se crée comme tout autre widget : construction et placement.

— Ligne 5 : on peut définir la longueur de la barre avec l'option `length`. L'option `maximum` sera expliquée ci-dessous.

— Ligne 8 : lancement de la progression dans la barre.

Par défaut, une barre de progression est horizontale (il existe une option pour la placer verticale). On peut changer la couleur par défaut en créant un style.

Principe de fonctionnnement d'une barre de progression Ttk

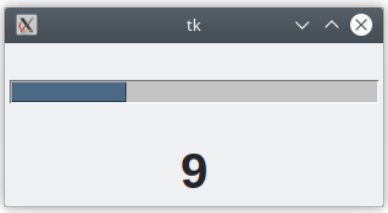
Il existe deux modes de fonctionnement d'une barre de progression : l'option `indeterminate` et l'option `determinate` qui est l'option par défaut. La première option **ne** prend **pas** en compte la **proportion** de progression d'une tâche, elle sert à indiquer que, par exemple, une tâche est en cours d'exécution mais pour une durée indéterminée. Cette option ne sera pas examinée.

Quand une barre de progression évolue, une valeur interne à la barre parcourt, à un certain rythme, des valeurs décimales entre 0 et une valeur `maximum` que l'on indique en option. Lorsque cette valeur est atteinte, la barre arrive visuellement à la fin de son rectangle de progression. Mais, curieusement, au lieu de s'arrêter, la progression **reprend automatiquement** de zéro. Ce comportement n'est pas modifiable sans utiliser la méthode `after` d'un widget.

On peut contrôler la valeur en cours de progression avec l'option `"value"`, accessible par `progress["value"]` où `progress` est le nom de la barre de progression.

Enfin, si on lance la barre de progression par `progress.start(10)` (cf. ligne 8), cela signifie que la valeur de référence de la barre va évoluer d'une unité toutes les 10 ms. Dans l'exemple ci-dessus, comme la valeur maximale est à 300, cela signifie que la barre se remplit en 3 secondes.

On peut d'ailleurs visualiser la duréee de remplissage avec un label :




```
from tkinter import Tk, ttk, StringVar, Label

root = Tk()

progress = ttk.Progressbar(root, length=400, maximum=300)
progress.pack(pady=30)

progress.start(10)

message = StringVar()
w = Label(root, textvariable=message, font='Arial 30 bold')
w.pack()

seconds = 0
delay = 1000


def anim(ms):
    message.set(str(ms // 1000))
    root.after(delay, anim, ms + delay)


anim(0)

root.mainloop()
```

Un appel de `start` sans paramètre effectue une progression toutes les 50 ms.

Barre de progression qui s'arrête

Lorsque la barre de progression est complètement remplie, on souhaite que la progression ne reparte pas à zero.

Pour cela, il faut utiliser `after` et surveiller l'évolution de la variable interne `value` jusqu'au moment où elle atteint la valeur de `maximum`. Voici un code possible :

```
1  from tkinter import *
2  from tkinter import ttk
3
4  root = Tk()
5
6  progress = ttk.Progressbar(root, maximum=300, length=400)
7  progress.pack(pady=30)
8
9  unit = 10
10 progress.start(10)
11
12
13 def anim(value):
14     if progress["value"] < value:
15         progress.stop()
16         progress["value"] = progress["maximum"]
17     root.after(10, anim, progress["value"])
18
19
20 anim(0)
21
22 root.mainloop()
```

— Ligne 14 : on surveille la variable `value` de la barre. Si la barre redémarre à zéro, c'est que la valeur courante de `value` diminue.

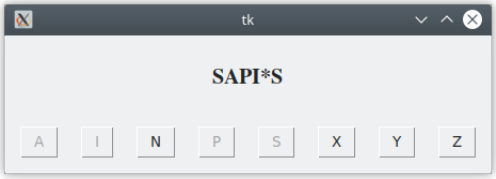
— Ligne 17 : on redémarre l'animation avec l'ancienne `value`.

— Lignes 15-16 : on arrête la barre de progression et pour qu'elle remplisse la zone, `value` est affecté à la valeur de remplissage maximum.

Modifier les options d'un widget

Ce qui suit est essentiel pour faire fonctionner une interface graphique sous Tkinter. Il suppose que vous avez un minimum de familiarité avec les widgets.

Pour illustrer, considérons un jeu du Pendu en version très simplifiée : le joueur choisit des lettres en cliquant sur des boutons à la recherche de lettres cachées dans une zone de texte.



On souhaite que :

— lorsqu'un clic découvre une lettre présente dans le mot inconnu, la lettre soit rendue visible et donc que l'astérisque qui couvre la lettre disparaisse ;

— tout clic (gagnant ou perdant) sur une lettre jamais encore choisie provoque la désactivation du bouton sur lequel le joueur a cliqué (c'est assez naturel puisqu'il n'y a aucune raison que le joueur re-clique sur le même bouton).

Chaque lettre de la zone de texte est un label. On veut donc qu'à chaque clic :

— le texte du label bascule de * vers la lettre trouvée ;

— le bouton soit modifié (et désactivé).

On doit donc **changer** l'état des deux widgets. Tous les widgets d'une interface Tkinter ont des options, souvent très nombreuses. Par exemple, un bouton a une option `state` qui peut prendre trois valeurs selon l'état de réceptivité du widget. De même, un label a une option `text` qui indique le contenu du texte qu'on lit sur le label ou encore une option `bg` pour la couleur de fond du label. L'état d'un widget est déterminé par l'état de ses options. Ce qui anime une interface graphique est que ses widgets changent d'état.

Il existe essentiellement deux syntaxes pour changer une option (il existe une 3e façon moins usuelle qui ne sera que très brièvement évoquée). Ainsi, dans le jeu du Pendu, pour chager l'astérisque d'un label appelé disons `lbl` en, par exemple, la lettre `E`, on pourra écrire :

```
lbl["text"] = "E"
```

C'est la syntaxe **la plus simple**, sous forme de dictionnaire. Bien noter que `lbl` est ici une référence vers le widget, le nom de l'option `text` est placé entre guillement pour obtenir une chaîne de caractère. Et `"E"` est la nouvelle valeur de l'option du widget.

Dans l'exemple du jeu du Pendu, pour désactiver un bouton nommé `btn`, on écrira par exemple :

```
btn["state"]=DISABLED
```

Il existe une 2e syntaxe qui consiste à utiliser la méthode `configure` de chaque widget :

```
lbl.configure(text="E")
```

Notons que le nom de l'option est placé en argument nommé, sans guillemet. Cette syntaxe est plus indiquée lorsqu'il y a beaucoup d'options à changer simultanément. Par exemple, supposons que nous voulions changer la lettre en `E` mais aussi la police et la couleur de fond. On pourrait alors écrire :

```
lbl.configure(text="E", bg="red", font="Times 20 Bold")
```

Enfin, il est aussi possible de changer l'état d'un widget de manière indirecte et automatique en faisant en sorte qu'une option de ce widget soit au départ enregistrée sous la forme d'une variable de contrôle Tkinter telle que `StringVar`. Une telle variable est mise à jour automatiquement, sans qu'on réaffecte comme ci-dessus. Pour voir une situation typique, consulter Exemple d'utilisation de StringVar. Cette utilisation toutefois peut être évitée dans de nombreuses situations.

Chapitre III

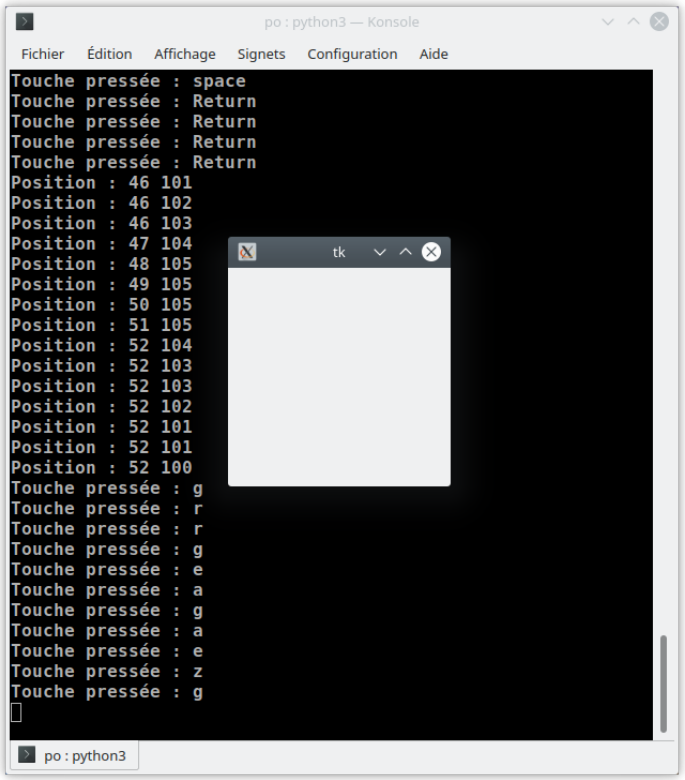
Les événements

Capturer des événements

Comme il a été vu lors de la présentation de la programmation événementielle, une interface Tkinter est à l'écoute de certains événements liés à la souris ou au clavier.

Un événement du clavier sera le fait d'appuyer ou de relâcher une touche du clavier ; un événement de la souris consiste en le fait de cliquer, de déplacer (en cliquant ou pas) la souris, de tourner la molette ou de relâcher un bouton.

Le point essentiel est qu'il est possible de **lier** un des événements précédents à une fenêtre ou un widget. Voici un exemple d'une simple fenêtre Tkinter qui réagit par un message dans la console si elle perçoit un clic de souris ou la pression sur une touche du clavier. Le message indique, selon les cas, la position de la souris dans la fenêtre ou alors la touche qui a été pressée :



Le code correspondant :

```
1 from tkinter import *
2
3 def f(event):
4     t=event.keysym
5     print("Touche pressée :", t)
6
7 def g(event):
8     x=event.x
9     y=event.y
10    print("Position :", x, y)
11
12 root = Tk()
13
14 root.bind("<Key>", f)
15 root.bind("<Motion>",g)
16 root.mainloop()
```

— Ligne 12 : on crée une fenêtre Tkinter ; elle est à l'écoute des événements.

— Ligne 14 : la chaîne `"<Key>"` représente l'événement de pression quelconque sur une touche du clavier. Cette ligne de code **lie** (*bind* en anglais) la fonction `f` définie lignes 3-5 et ces événements. Chaque fois qu'un tel événement se produit, la fonction f est exécutée.

— Lignes 3-5 : Lorqu'on presse une touche du clavier, la fonction `f` reçoit l'événement correspondant. Cet événement, appelé ici `event` (mais qu'on pourrait appeler par tout autre nom) est un objet dont un des attributs `keysym` est le nom de la touche sur laquelle on a appuyé. La fonction `f` affiche le nom de cette touche.

— Ligne 15 : la chaîne `"<Motion>"` représente l'événement de déplacement de la souris dans la fenêtre. Chaque point de la fenêtre `root` a une position exprimée en pixels. La position du coin supérieur gauche est `(0,0)`. Cette ligne de code **lie** la fonction g définie lignes 7-10 et ces déplacements. Chaque fois que la souris bouge, la fonction g est exécutée.

— Ligne 7-10 : lorque la souris bouge, la fonction g reçoit l'événement correspondant (ligne 7). Cet événement est un objet dont les attributs `x` et `y` représentent la position courante de la souris (lignes 8-9). La fonction g affiche (ligne 10) alors cette position.

— Des fonctions telles que `f` et g sont dites des *fonctions de rappel* (en anglais, *callback* function).

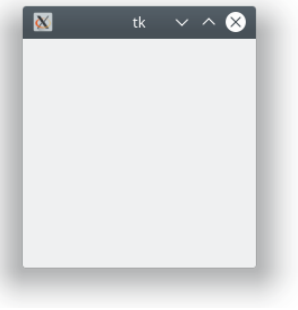
Il existe une liste d'événements du clavier et de la souris reconnus par Tkinter, désignés par une chaîne de caractères littérale, comme `"<Key>"` ou `"<Motion>"`. D'une manière générale, les événements standard sont accessibles par une syntaxe de la forme `"<event>"` où `event` est une chaîne de caractères comme **Key** ou **Motion** ou encore **space**, cette dernière désignant l'appui sur la touche ESPACE.

Pour une information plus complète sur les événements sous Tkinter, cf. la documentation de Fredrik Lundh.

Événement du clavier

Ci-dessous, on aperçoit une fenêtre Tkinter sans contenu. Mais lorsqu'on appuie sur une touche du clavier, la console affiche un message indiquant sur quelle touche on a pressé.

```
Touche j pressée
Touche KP_8 pressée
Touche KP_Divide pressée
Touche Alt_L pressée
Touche Super_L pressée
Touche h pressée
Touche space pressée
Touche Return pressée
```



Le code correspondant est :

```
1 from tkinter import *
2
3 def touche(event):
4     t=event.keysym
5     print("Touche %s pressée" %t)
6
7 root = Tk()
8
9 root.bind('<Key>', touche)
10
11 root.mainloop()
```

```
12 Touche space pressée
13 Touche p pressée
14 Touche KP_4 pressée
```

— Ligne 7 : une fenêtre est créée

— Ligne 9 : les événements du clavier perçus par l'application sont **liés** (cf. `touch`) à la fonction `touche`.

— Lignes 3-5 : lorsqu'on appuie sur n'importe quelle touche, la fonction `touche` s'exécute. Elle reçoit l'événement du clavier correspondant. Cet événement (nommé ici `event`) contient un attribut `keysym` qui est le nom de la touche sur laquelle on a appuyé. Un message est lisible dans la console indiquant la touchée qui a atét choisie.

L'événement de clavier que l'on cherche généralement à détecter est l'appui sur une touche (en anglais *key press*) ; parfois, on cherche aussi à détecter le relâchement d'une touche (*key release*).

L'événement associé à la pression d'une touche **quelconque** du clavier est nommé `Key` cf ; ligne 9.

Codes de quelques touches

Un code d'événement est une chaîne littérale de la forme `"<event>"` où `event` est une chaîne de caractères telle que **Enter**. Ci-dessous, le nom des événements représentant `"<event>"` et associés à l'appui sur certaines touches :

— Flèches : `Left`, `Right`, `Up`, `Down`
— ESPACE : `space`
— Touche ENTRÉE : `Return` ou, sur le pavé numérique, `KP_Enter`
— Touche ECHAP : `Escape`

L'appui sur une touche de caractère, par exemple **z**, se nomme `KeyPress-z` mais on peut l'abréger tout simplement en `z`. Pour la majuscule `Z`, on peut écrire `Z`. Pour le relâchement de la touche `z`, on écrira `KeyRelease-z`.

Pour une combinaison de touches comme ALT+CTRL-a (appui simultané), l'événement sera nommé `<Control-Shift-KeyPress-a>` :

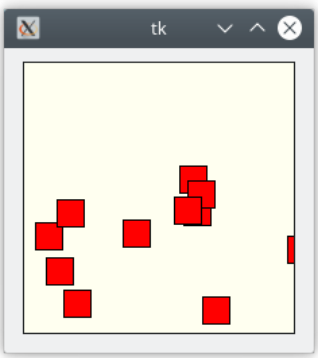
```
from tkinter import Tk, Canvas

root = Tk()

def f(event):
    print("OK")

root.bind('<Control-Shift-KeyPress-a>', f)

root.mainloop()
```

Par exemple, l'événement à déclarer si on appuie sur la touche ESPACE est `<space>`. On trouvera la liste complète dans la documentation : Nom des touches.

Les nombres du pavé numérique

L'appui sur une touche du pavé numérique génère un événement dont le début du nom est `KP_` qui signifie *KeyPad* et qui est suivi d'une chaîne précisant la touche. Par exemple, l'appui sur la touche 3 du pavé numérique génère l'événement `KP_3`. Cette syntaxe permet en analysant le retour du nom `event.keysym` de savoir la valeur numérique corresponsante.

Voici un exemple d'application : si on appuie sur la touche valant le chiffre `N` du pavé numérique avec N valant 1, 2 ou 3 alors sont dessinés N carrés aléatoires sur un canevas :

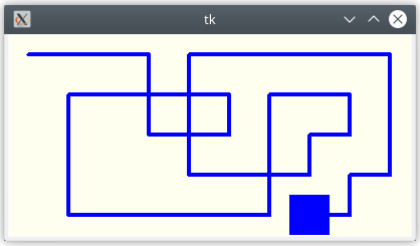


```
1  from tkinter import Tk, Canvas
2  from random import randrange
3
4  SIDE=200
5
6  root = Tk()
7  cnv = Canvas(root, width=SIDE, height=SIDE, bg='ivory')
8  cnv.pack(padx=10, pady=10)
9  cnv.focus_set()
10
11 def dessiner(event):
12     n=int(event.keysym[3])
13     for i in range(n):
14         a=randrange(SIDE)
15         b=randrange(SIDE)
16         cnv. create_rectangle(a, b, a+20, b+20, fill="red")
17
18 cnv.bind('<KP_1>', dessiner)
19 cnv.bind('<KP_2>', dessiner)
20 cnv.bind('<KP_3>', dessiner)
21
22 root.mainloop()
```

— Ligne 9 : on donne le focus au canevas, comme ça, l'application réagit à l'appui sur une touche.
— Lignes 18-20 : seules les touches 1, 2 et 3 du pavé numérique vont appeler la fonction `dessiner`
— ligne 12 : on filtre sur l'appui des touches 1, 2 et 3 du pavé numérique qui est le caractère d'indice 3 dans la chaîne représentant l'événemlent, comme la chaîne `"KP_1"`.
— Ligne 12 : on capture la **chaîne** représentant l'appui, elle commence par `KP_` ; le caractère suivant, d'indice 3 donc, représente le chiffre sur lequel on a appuyé. Ce **caractère** est converti en **vrai entier** avec la fonction `int`.
— Ligne 13 : on est en mesure, avec `range(n)` de dessiner le bon nombre de carrés.

Tracer un chemin réversible au clavier

Le programme suivant vise à illustrer les événements du clavier agissant sur un canevas. L'interface ci-dessous



permet de tracer avec les flèches du clavier un chemin sur un canevas, le chemin pouvant être reparcouru en arrière (avec effacement du chemin de retour).

Le code :

```
1  from tkinter import *
2
3  WIDTH=400
4  HEIGHT=200
5  COTE=40
6
7  root = Tk()
8  cnv = Canvas(root, width=WIDTH, height=HEIGHT, background="ivory")
9  cnv.pack()
10
11 DIR={'Left':(-1,0),'Right':(1,0),'Up':(0,-1),'Down':(0,1)}
12
13 def bouge(event):
14     key=event.keysym
15     dx, dy=DIR[key]
16     a,b, segment=pile[-1]
17     if len(pile)>=2 and a+dx==pile[-2][0] and b+dy==pile[-2][1] :
18             pile.pop()
19             print("back")
20             cnv.delete(segment)
21     else:
22         segment=cnv.create_line(a*COTE+COTE//2, b*COTE+COTE//2,
23                                 a*COTE+COTE//2+dx*COTE,
24                                 b*COTE+COTE//2+dy*COTE,
25                                 fill='blue', width=4, capstyle=ROUND)
26         pile.append([a+dx, b+dy, segment])
27
28     cnv.move("perso", dx*COTE, dy*COTE)
29
30 perso=cnv.create_rectangle(0, 0,COTE, COTE, fill="blue",
31                            outline='', tag='perso')
32 pile=[(0,0, perso)]
33
34 for key in ["<Left>", "<Right>", "<Up>", "<Down>"]:
35     root.bind(key, bouge)
36
37 root.mainloop()
```

— Ligne 34 : l'appui sur une des 4 flèches du clavier est détecté par le canevas.

— Ligne 30 : un carré bleu indique la position courante sur le canevas (utile si on repasse sur un chemin déjà parcouru)

— Ligne 35 : le canevas a le focus car la fenêtre entière (`root`) a le focus quand l'application est lancée.

— Lignes 34-35 : tout appui sur une des touches du clavier exécute la fonction `bouge`.

— Ligne 11 : chaque appui sur une des flèches du clavier est associé à un décalage suivant le repère du canevas. Par exemple, à un appui sur la flèche bas est associé la direction correspondante, ici le couple `(0, -1)`. C'est juste une facilité pour coder la fonction `bouge`.

— Ligne 32 : les différents mouvements de l'utilisateur sont enregistrés dans une pile. La pile permet (en dépilant) de revenir en arrière et d'effacer le trajet de retour.
— Ligne 17-19 : la détection d'une marche arrière. le segment est effacé (ligne 19)
— Lignes 22-26 : si pas de retour en arrière, le segment pour le nouveau déplacement est rendu visible et rajouté dans la pile.
— Ligne 28 : la sortie du canevas n'est pas gérée par le programme.

Evénement du clavier : press vs release

On a parfois besoin de distinguer entre une pression sur une touche du clavier et un relâchement de cette touche. Cela correspond à deux événements différents, comme le montre le code ci-dessous (en mode texte, pas véritablement de fenêtre pertinente ici) :

```
1 from tkinter import Tk, Canvas
2
3 root = Tk()
4
5 def press(event):
6     print("press:", event.keysym)
7
8 def release(event):
9     print("release:", event.keysym)
10
11 root.bind('<KeyPress>', press)
12 root.bind('<KeyRelease>', release)
13 root.mainloop()
```

qui peut afficher

```
1 release: Return
2 press: a
3 release: a
4 press: space
5 release: space
6 press: Return
7 release: Return
```

Un évènement `press` correspond à l'appui sur une touche. Quand on relâche la touche, on obtient un évènement `release`. Dans la sortie ci-dessus, la première ligne correspond au relâchement de la touche `ENTRÉE` qui a servi à lancer le programme en console.

Concernant la question des événements d'appui vs relâchement, on pourra lire le fil de discussion TkInter keypress, keyrelease events.

Evénement du clavier : majuscule vs minuscule

On peut distinguer (automatiquement) l'appui sur une touche et l'appui sur la même touche mais en majuscule (on appuie simultanément sur la touche `MAJ`), comme le montre le code ci-dessous :

```
1 from tkinter import Tk, Canvas
2
3 root = Tk()
4 cnv = Canvas(root, width=400, height=100, bg="ivory")
5 cnv.pack()
6 cnv.focus_set()
7 x=0
8
9 def texte(event):
10     global x
11     cnv.create_text(x, 40, text =event.keysym, font="Arial 50 bold")
12     x+=30
13
14 cnv.bind('<Key-a>', texte)
15 cnv.bind('<Key-B>', texte)
16
17 root.mainloop()
```

— Ligne 14 : détection de l'appui simple sur une touche.

— Ligne 15 : détection de l'appui simultané sur une touche et la touche MAJ.

Relâchement d'une touche (sous Linux, auto-repeat)

Il est parfois important d'observer quand l'utilisateur relâche une touche. C'est assez délicat à détecter sous Linux : une possibilité est de désactiver l'auto-repeat (et penser à le réactiver en fin de script sinon il sera perdu pour toutes les applications, y compris en dehors de Python !), comme indiqué dans TkInter keypress, keyrelease events. Voici un exemple :

```
1 from tkinter import Tk, Canvas
2 import os
3
4 os.system('xset r off')
5
6 root = Tk()
7 cnv = Canvas(root, width=400, height=100, bg="ivory")
8 cnv.pack()
9 cnv.focus_set()
10 x=0
11
12 def texte(event):
13     global x
14     cnv.create_text(x, 40, text =event.keysym, font="Arial 50 bold")
15     x+=30
16
17 cnv.bind('<KeyRelease-a>', texte)
18
19 root.mainloop()
20 os.system('xset r on')
```

— Ligne 17 : chaque fois que l'utilisateur relâche la touche A minuscule du clavier, la lettre a est écrite en minuscule sur le canevas.
— Lignes 12-15 : on vérifie que si laisse la touche A appuyé, aucun événement n'est détecté.
— Ligne 4 : si cette ligne est supprimée, on vérifie que le relâchement de touche n'est pas détecté.
— Ligne 20 : on réactive la détection de répétition de touche.

Pression continue et simultanée sur deux touches

L'appui continu et simultané sur *deux* touches engendre des événements qui dépendent du système d'exploitation. Le code suivant

```
1  from tkinter import *
2  from time import time
3
4  root = Tk()
5
6  def press(event):
7      print("press:", event.keysym, time())
8
9  def release(event):
10     print("release:", event.keysym, time())
11
12 for key in ["a", "z"]:
13     root.bind('<KeyPress-%s>' %key, press)
14     root.bind('<KeyRelease-%s>' %key, release)
15
16 root.mainloop()
```

indique si la touche **a** ou **z** a été pressée ou relâchée et à quel moment cela se produit. Plus précisément, lors de l'exécution, on va

— appuyer continûment sur la touche **a**,
— appuyer continûment sur la touche **z**, sans relâcher **a**
— relâcher la touche **z** et laisser enfoncée un certain temps la touche **a**
— relâcher enfin la touche **a**.

Sous Linux, on lira ceci (code en partie tronqué)

```
1  press: a 1555710017.386128
2  release: a 1555710017.986468
3  press: a 1555710017.9867318
4  release: a 1555710018.026511
5  press: a 1555710018.0267715
6  release: a 1555710018.066585
7  press: a 1555710018.066844
8  ...
9  release: a 1555710018.3864462
10 press: a 1555710018.3866968
11 release: a 1555710018.4267566
```

```
12 press: a 1555710018.4269776
13 press: z 1555710018.434321
14 release: z 1555710019.0345645
15 press: z 1555710019.034829
16 release: z 1555710019.0747504
17 press: z 1555710019.0750027
18 ...
19 release: z 1555710019.5153375
20 press: z 1555710019.515587
21 release: z 1555710019.5553439
22 press: z 1555710019.5556386
23 release: z 1555710019.5705464
24 release: a 1555710020.0423253
```

On observe, comme pour le cas d'une unique touche maintenue enfoncée, qu'est générée, pour chacune des touches, une suite d'événements très rapprochés `release` puis `press`(par exemple lignes 6-7 pour **a** et lignes 19-20 pour **z**). On obseve aussi que lorsque survient l'appui sur la touche **z** (ligne 13), l'appui maintenu sur **a** n'engendre plus aucun événement. Lorsque la pression sur **z** est relâchée (cf. ligne 23), même si la pression sur **a** est prolongée, aucun événement venant de **a** n'est généré et c'est seulement lorsque la touche **a** est relâché qu'un événement est émis (ligne 24).

Sous Windows, on lira plutôt ceci (code en partie tronqué)

```
1 press: a 1555711099.043254
2 press: a 1555711099.5322323
3 press: a 1555711099.6011677
4 press: a 1555711099.6329112
5 ...
6 press: a 1555711100.1418097
7 press: a 1555711100.171071
8 press: z 1555711100.195916
9 press: z 1555711100.666682
10 press: z 1555711100.7084837
11 ...
12 press: z 1555711101.175861
13 press: z 1555711101.2056215
14 release: z 1555711101.2358758
15 release: a 1555711101.7203972
```

Cette fois, un appui continu engendre un événement `press` (lignes 1-13). Le 2^{e} appui cache le premier (lignes 8-14). La fin se termine par deux événements `release` (lignes 14-15).

Voici deux approches pour gérer un appui simultané sur deux touches :

— Bryan Oakley : Machine à état fini

— unutbu : désativation de l'auto-repeat : constate la génération d'événement `release` même si la touche n'est pas relachée. Serait dû à l'auto-repeat du clavier. Il propose un code pour désactiver l'auto-repeat sous X-windows (Linux).

Déplacement amélioré avec deux touches (Windows)

Une première version de déplacement oblique d'un objet par appui simultané et continu sur deux touches a été fournie mais l'exécution souffre d'une latence assez sensible au lancement du mouvement. Voici une version qui corrige ce problème sous Windows.

Pour bien comprendre le code, il faut s'être penché sur la gestion des événements `press` et `release` lors d'un appui continu sur une ou deux touches. Sous Windows, c'est très simple, un événement `press` est généré pour tout appui continu et un événement `release` uniquement si l'utilisateur relâche la touche. Il suffit donc de capturer les appuis et ordonner le déplacement qui en résulte

L'idée est d'enregistrer en permanence dans un dictionnaire l'état (pressée ou pas) des 4 touches de déplacement. Cet état des touches est ensuite examiné toutes les 20 ms par une fonction qui décide de déplacer ou non l'objet.

Voici le code

```
from tkinter import *

WIDTH = 800
HEIGHT = 500

root = Tk()
cnv = Canvas(root, width=WIDTH, height=HEIGHT, bg="ivory")
cnv.pack()
cnv.focus_set()

SIDE = 30
UNIT = 2

rect = cnv.create_rectangle(
    (WIDTH / 2 - SIDE / 2, HEIGHT / 2 - SIDE / 2),
    (WIDTH / 2 + SIDE / 2, HEIGHT / 2 + SIDE / 2),
    fill="black",
)


def handler():
    unit = UNIT

    if sum(keys.values()) > 1:
        unit = UNIT/1.5

    for key in drn:
        if keys[key]:
            if key == "Up":
                cnv.move(rect, 0, -unit)
            elif key == "Right":
                cnv.move(rect, unit, 0)
            elif key == "Left":
```

```
34                 cnv.move(rect, -unit, 0)
35             elif key == "Down":
36                 cnv.move(rect, 0, unit)
37
38     root.after(20, handler)
39
40
41 def press(event):
42     keys[event.keysym] = True
43
44
45 def release(event):
46     keys[event.keysym] = False
47
48
49 drn = ["Up", "Right", "Left", "Down"]
50 keys = dict.fromkeys(drn, False)
51
52 for key in drn:
53     cnv.bind("<KeyPress-%s>" % key, press)
54     cnv.bind("<KeyRelease-%s>" % key, release)
55
56 handler()
57
58 root.mainloop()
```

— Lignes 49-50 : l'état de chacune des 4 flèches du clavier est enregistré dans le dictionnaire `keys`. Pour chacune des flèches, on enregistre si elle est activée ou pas.

— Lignes 41 et 45 : les états sont enregistrés par les fonctions `press` et `release` qui sont appelées automatiquement par Tkinter après liaison (lignes 52-54)

— Ligne 38 : toutes les 20 ms, la fonction `handler` est appelée. Elle examine l'état de chacune des 4 touches (ligne 27).

— Lignes 28-36 : si la touche est active (elle est pressée), le mouvement correspondant est exécuté.

— Lignes 24-25 : si deux touches sont pressées simultanément, la vitesse de déplacement est rectifiée pour qu'elle corresponde à peu près au déplacement horizontal ou vertical.

Déplacement amélioré avec deux touches (Linux)

Une première version de déplacement oblique d'un objet par appui simultané et continu sur deux touches a été fournie mais l'exécution souffre d'une latence assez sensible au lancement du mouvement. Voici une version qui corrige ce problème sous Linux.

Pour bien comprendre le code, il faut s'être penché sur la gestion des événements `press` et `release` lors d'un appui continu sur une ou deux touches. Sous Linux (et macOS aussi) la difficulté provient du fait qu'un appui continu engendre automatiquement un événement `release` qu'il faut arriver à discerner d'un relâchement provoqué par l'utilisateur. Par ailleurs, il faut arriver à contourner le problème de latence au démarrage en provoquant le déplacement **dès qu**'une pression est enregistrée.

L'idée est d'enregistrer en permanence dans une liste l'état des 4 touches de déplacement : la touche a-t-elle été pressée ? à quel moment reçoit-elle un événement `press` ? à quel moment reçoit-elle un événement `release` ? Cet état des touches est ensuite examiné toutes les 20 ms par une fonction qui décide de déplacer ou non l'objet.

Voici le code

```
from tkinter import *
import time

WIDTH = 800
HEIGHT = 500

root = Tk()
cnv = Canvas(root, width=WIDTH, height=HEIGHT, bg="ivory")
cnv.pack()
cnv.focus_set()

SIDE = 30
UNIT = 2

rect = cnv.create_rectangle(
    (WIDTH / 2 - SIDE / 2, HEIGHT / 2 - SIDE / 2),
    (WIDTH / 2 + SIDE / 2, HEIGHT / 2 + SIDE / 2),
    fill="black",
)


def handler():
    for i in range(4):

        if keys[i][0] and keys[i][2] is not None:
            if keys[i][1] is None or keys[i][1] - keys[i][
                    2] > 3 or keys[i][1] - keys[i][2] < 0:
                keys[i][0] = False

        elif not keys[i][
                0] and keys[i][1] is not None and keys[i][2] is None:
            keys[i][0] = True

        if keys[i][0]:
            if i == 0:
                cnv.move(rect, 0, -UNIT)
            if i == 1:
                cnv.move(rect, UNIT, 0)
            if i == 2:
                cnv.move(rect, -UNIT, 0)
            if i == 3:
                cnv.move(rect, 0, UNIT)
        keys[i][1] = None
```

```
44          keys[i][2] = None
45
46      root.after(20, handler)
47
48
49  def press(event):
50      key = event.keysym
51      if key == "Up":
52          keys[0][1] = time.time()
53      elif key == "Right":
54          keys[1][1] = time.time()
55      elif key == "Left":
56          keys[2][1] = time.time()
57      elif key == "Down":
58          keys[3][1] = time.time()
59
60
61  def release(event):
62      key = event.keysym
63      if key == "Up":
64          keys[0][2] = time.time()
65      elif key == "Right":
66          keys[1][2] = time.time()
67      elif key == "Left":
68          keys[2][2] = time.time()
69      elif key == "Down":
70          keys[3][2] = time.time()
71
72
73  for key in ["Up", "Right", "Left", "Down"]:
74      cnv.bind("<KeyPress-%s>" % key, press)
75      cnv.bind("<KeyRelease-%s>" % key, release)
76
77  keys = [
78      [False, None, None],  # up
79      [False, None, None],  # right
80      [False, None, None],  # left
81      [False, None, None],  # doum
82  ]
83  handler()
84
85  root.mainloop()
```

— Ligne 2 : La fonction `time` du module `time` permet de savoir à la microseconde quand un événement a lieu (cf. ligne 49 par exemple)

— lignes 73 : l'état de chacune des 4 flèches du clavier est enregistré dans la liste `keys`. Pour chacune des flèches, on enregistre :

  — si elle a été pressée par l'utilisateur

  — l'instant où un événement `press` est capturé

— l'instant où un événement `release` est capturé.

— lignes 49 et 61 : les états sont enregistrés par les fonctions `press` et `release` qui sont appelées automatiquement par Tkinter après liaison (lignes 73-75)

— Ligne 46 : toutes les 20 ms, la fonction `handler` est appelée. Elle examine l'état de chacune des 4 touches (cf. ligne 23).

— Si la touche a été pressée initialement (ligne 25), elle détermine si cette touche est relâchée (ligne 28) par l'utilisateur. Une succession trop proche (moins de 3 ms, cf. ligne 27) entre un événement `release` et un événement `press` n'est pas considérée comme un relâchement de l'utilisateur.

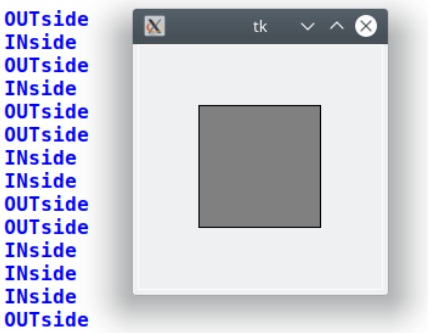
— Si la touche n'a pas été pressée (ligne 30), la fonction détermine si la touche est pressée par l'utilisateur (ligne 32). En particulier, si dans l'interalle de 20 ms, une pression est enregistrée sans événement `release`, on considère qu'une pression continue sur la touche est exercée.

— lignes 34-42 : une fois l'appui sur les touches connu, les mouvement qui s'en déduisent sont exécutés.

Pour contrôler la vitesse de déplacement, on peut jouer sur la valeur de `UNIT` (ligne 13) ; en particulier, on pourrait modifier le code pour que les déplacements obliques et verticaux/horizontaux se fassent à la même vitesse (actuellement, ils sont plus rapides).

Événements de la souris

Un widget peut capturer des actions de la souris. Par exemple, dans l'interface ci-dessous :



si l'utilisateur clique à l'extérieur du rectangle, la console affiche `OUTside`, sinon elle affiche `INside`.

Voici le code correspondant :

```
1 from tkinter import Tk, Canvas
2
3 root = Tk()
4 cnv = Canvas(root, width=200, height=200)
5 cnv.pack()
6
7 cnv.create_rectangle(50, 50, 150, 150, fill='gray')
```

```
8
9  def je_clique(event):
10     if 50< event.x <150 and 50< event.y <150:
11         print("INside")
12     else:
13         print("OUTside")
14
15 cnv.bind("<Button-1>",je_clique)
16
17 root.mainloop()
```

— Ligne 7 : création d'un rectangle gris.

— Ligne 15 : tout appui sur le bouton gauche de la souris (événement `<Button-1>`) entraîne l'exécution de la la fonction `je_clique`.

— Lignes 9-13 : la fonction `je_clique`, quand elle est automatiquement appelée suite à un clic gauche dans le canevas, reçoit un argument représenté par le paramètre `event` traduisant le clic de souris. Cet événement contient en attributs les coordonnées `(event.x, event.y)` dans le canevas du clic de souris. Un simple calcul permet de savoir si le clic est dans le rectangle ou à l'extérieur.

Récapitulatif des événement de la souris

Voici un résumé des principaux événements liés à la souris :

Nom de l'évenement de souris	Action
Button-1	Clic gauche
Button-3	Clic bouton droit
Button-2	Clic bouton central
Button-4	Molette vers le haut
Button-5	Molette vers le bas
ButtonRelease	Relâchement d'un bouton
Motion	Déplacement du curseur de la souris
MouseWheel	Roulette (pas sous Linux)
Button	Un des boutons

L'événement du clic de souris

La programmation d'interfaces graphiques est de la programmation événementielle : une boucle infinie (la mainloop) attend des **événements** (c'est comme ça qu'on dit) tel qu'un appui sur une touche du clavier, un clic de souris ; éventuellement, le programme réagit à ces événements.

Pour surveiller un clic de souris sur le canevas, il suffit

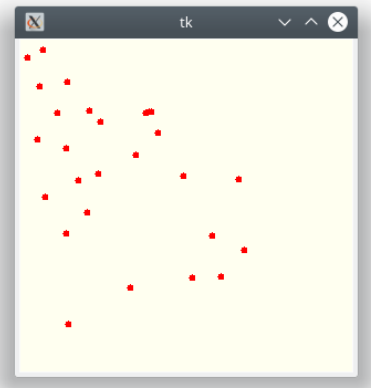
— d'écrire une fonction (disons `clic`) qui sera appelée (automatiquement) chaque fois qu'un clic aura lieu sur le canevas ;

— de **lier** l'événement clic de souris à l'appel de la fonction `clic`.

On dit que la fonction `clic` est une fonction de rappel (*callback function* en anglais). La *liaison* est appelée *binding* en anglais, cf. le code ci-dessous.

Le programme présenté ci-dessous :

```
100 225
44 258
18 43
7 17
21 10
43 39
42 99
23 143
42 176
61 157
71 122
73 75
63 65
34 67
16 91
53 128
105 105
198 127
174 178
156 216
182 215
203 191
148 124
125 85
114 67
117 66
119 66
```



montre la réaction à un événement de clic de souris sur un canevas affichant les coordonnées du clic et dessinant un disque là où on a cliqué. Le code correspondant est :

```
from tkinter import *

root = Tk()
cnv = Canvas(root, width=300, height=300, bg="ivory")
cnv.pack()

def clic(event):
    x, y = event.x, event.y
    print(x, y)
    cnv.create_oval(x-3, y-3, x+3, y+3,
                    fill='red', outline='')

cnv.bind("<Button-1>", clic)
root.mainloop()
```

```
151 244
121 225
101 155
92 110
```

```
19 86 89
20 61 63
```

— Ligne 9 : la fonction de rappel `clic` ; chaque fois qu'on clique sur le canevas, cette fonction est appelée et reçoit en argument, sans que le programmeur ait la main dessus, un objet appelé ici `event` qui représente (et donne accès) à toutes les propriétés du clic qui a été effectié.

— Ligne 12 : la **liaison** de l'événement clic gauche de souris, qui se code en Tkinter par `"<Button-1>"` et de la fonction `clic` : chaque fois qu'un événement clic de souris est intercepté sur le canevas, la fonction `clic` est appelée. -

— La fonction `clic` effectue ici deux actions :

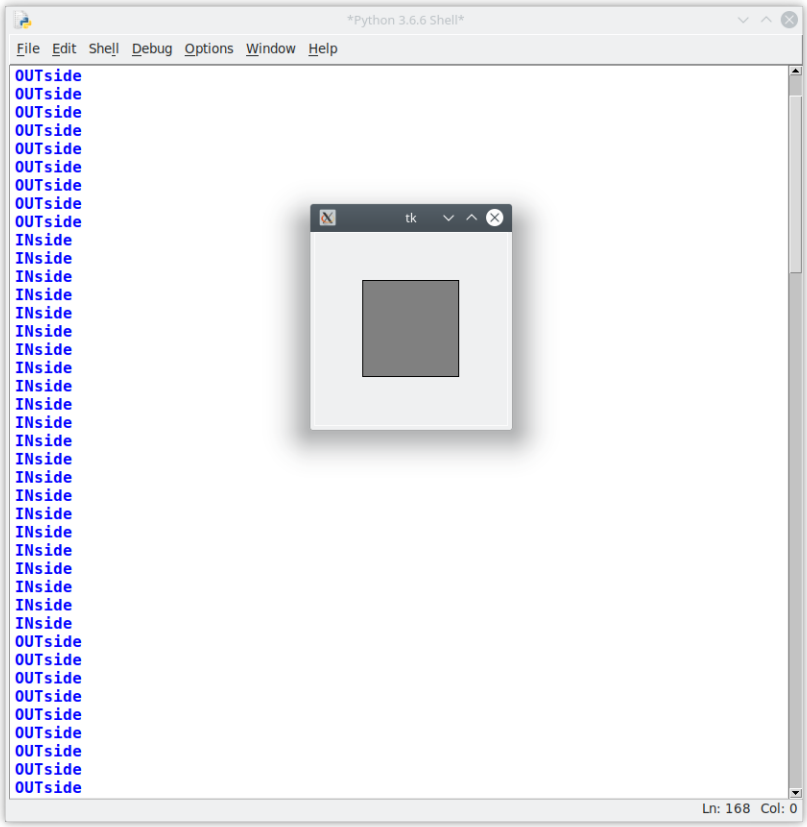
  — elle affiche dans la console les coordonnées sur le canevas du pixel qui a été cliqué. Dans l'exemple (lignes 15-19), on voit que l'utilisateur a cliqué 6 fois sur le canevas. Remarquez que les valeurs lues sont, bien sûr, entre 0 et 300 ;

  — elle dessine quelque chose au point du clic (ici, un petit disque rouge, c'est juste pour que l'image soit compréhensible).

L'événement `"<Button>"` (sans numéro) s'applique à tout événement relatif à un bouton de souris, le clic gauche comme le clic droit ou l'action sur la molette. Si on veut capturer exactement un clic **gauche** de souris, il faut utiliser l'événement `"<Button-1>"`.

Événement du déplacement de la souris

Le code ci-dessous donne un exemple de capture de tout mouvement de la souris (en javascript, c'est "onmouseover") :



```
1  from tkinter import Tk, Canvas
2
3  root = Tk()
4  cnv = Canvas(root, width=200, height=200)
5  cnv.pack()
6
7  cnv.create_rectangle(50, 50, 150, 150, fill='gray')
8
9  def mouvement(event):
10     if 50< event.x <150 and 50< event.y <150:
11         print("INside")
12     else:
13         print("OUTside")
14
15 cnv.bind("<Motion>",mouvement)
16
17 root.mainloop()
```

— Ligne 15 : tout mouvement de la souris sur le canevas entraîne l'exécution de la fonction `mouvement`.

— Lignes 9-13 : la position du curseur (`event.x, event.y`) est capturée. En fonction de sa valeur, un message est affiché dans la console pour indiquer si le curseur est à l'intérieur ou à l'extérieur du rectangle gris.

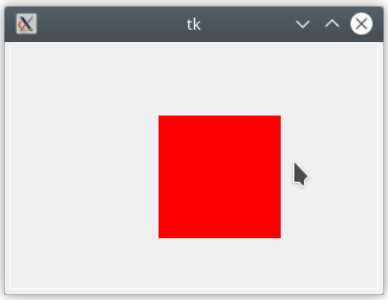
On peut aussi capturer tout mouvement de la souris mais avec le bouton gauche enfoncé :

```
1  from tkinter import Tk, Canvas
2
3  root = Tk()
4  cnv = Canvas(root, width=200, height=200)
5  cnv.pack()
6
7  cnv.create_rectangle(50, 50, 150, 150, fill='gray')
8
9  def glisser(event):
10     if 50< event.x <150 and 50< event.y <150:
11         print("INside")
12     else:
13         print("OUTside")
14
15 cnv.bind("<B1-Motion>",glisser)
16
17 root.mainloop()
```

— Ligne 15 : l'événement "<B1-Motion>" capture le bouton 1 enfoncé et le mouvement de la souris.

Déplacer un objet à la souris

L'interface ci-dessous



montre un carré rouge dans un canevas que la souris peut déplacer par glisser-déposer (le bouton gauche de la souris reste appuyé).

Voici le code :

```
1  from tkinter import Tk, Canvas
2
3  root = Tk()
4  cnv = Canvas(root, width=300, height=200)
5  cnv.pack()
6
7  rect=cnv.create_rectangle(30, 30, 130, 130, fill="red", outline='')
8
9  old=[None, None]
10
11 def clic(event):
12     old[0]=event.x
13     old[1]=event.y
14
15 def glisser(event):
16     cnv.move(rect, event.x-old[0], event.y-old[1])
17     old[0]=event.x
18     old[1]=event.y
19
20
21 cnv.bind("<B1-Motion>",glisser)
22 cnv.bind("<Button-1>",clic)
23
24 root.mainloop()
```

— Lignes 9-22 : le principe est qu'une variable `old` (ligne 9) enregistre la dernière position de la souris, ce qui permet de savoir, à l'instant courant, de combien (ligne 16) on doit déplacer l'objet. Cette position est initialisée au clic qui précède le glisser-déposer (ligne 12-13).

— Ligne 21 : chaque fois que la souris bouge dans le canevas, la fonction `glisser` est appelée.

— Ligne 22 : chaque fois que le bouton gauche est enfoncé, la fonction `clic` est appelée.

— Ligne 11-13 : permet de mémoriser la position `(event.x, event.y)` du clic qui précède le déplacement du carré

— Ligne 15-18 : `( event.x-old[0], event.y-old[1])` représente le vecteur de déplacement entre le moment courant et le dernier enregistrement de position.

— Lignes 17-18 : on met à jour la position de la souris.

Si on veut faire un déplacement uniquement dans une direction et en bloquant le déplacement :

```
1  from tkinter import Tk, Canvas
2
3  RIGHT=300
4
5  root = Tk()
6  cnv = Canvas(root, width=RIGHT, height=200)
7  cnv.pack()
8
9  rect=cnv.create_rectangle(30, 30, 130, 130, fill="red", outline='')
10
11 old=[None, None]
12
13 def clic(event):
14     old[0]=event.x
15     old[1]=event.y
16
17 def glisser(event):
18     a,b,c,d=cnv.coords(rect)
19     if c< RIGHT or event.x<old[0]:
20         cnv.move(rect, event.x-old[0], 0)
21     old[0]=event.x
22     old[1]=event.y
23
24 cnv.bind("<B1-Motion>",glisser)
25 cnv.bind("<Button-1>",clic)
26
27 root.mainloop()
```

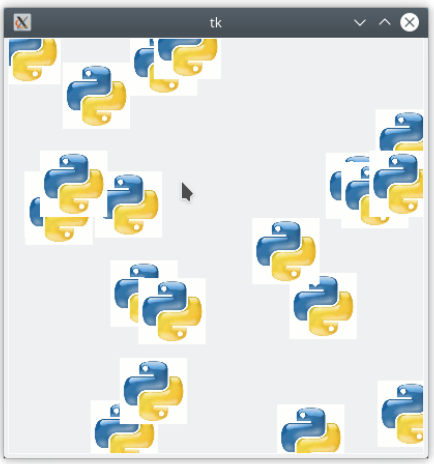
— Lignes 3, 6 et 19 : RIGHT permet de bloquer le déplacement vers la droite

— Lignes 18 : c donne l'abscisse courante du bord droit du rectangle.

— Ligne 20 : à cause du dernier paramètre qui vaut 0, il n'y a pas de déplacement du rectangle suivant la composante verticale.

Supprimer des images à la souris

Soit l'interface suivante :



où un canevas contient des images placées aléatoirement du logo Python et que l'on peut supprimer à la souris en se plaçant assez proche de l'image.

Le code correspondant est :

```
1  from tkinter import *
2  from random import randrange
3
4  SIDE=400
5  root = Tk()
6  cnv = Canvas(root, width=SIDE, height=SIDE)
7  cnv.pack()
8
9  logo = PhotoImage(file="python64.gif")
10
11 for i in range(20):
12     x, y= randrange(SIDE),randrange(SIDE)
13     cnv.create_image(x, y, image=logo)
14
15 def clic(event):
16     x=event.x
17     y=event.y
18     t=cnv.find_closest(x, y)
19     if t:
20         cnv.delete(t[0])
21
22 cnv.bind("<Button>",clic)
23
24 root.mainloop()
```

— Ligne 22 : tout clic venant d'un des boutons de la souris appelle la fonction `clic`.

— Ligne 15-20 : la fonction clic enregistre la position `(x, y)` du clic (lignes 16-17) et appelle la méthode `find_closest` du canevas(ligne 18) et recherche l'item `t` le plus proche de la position `(x, y)`. L'appel récupère l'item en question (ou `None` si le canevas est vide) et le

supprime (ligne 20) grâce à son id qui est le premier élément du tuple (ligne 20) représentant l'item.

Position de la souris

Si un événement `event` est lié à la souris, par exemple un clic de souris :

```
1 from tkinter import *
2
3 def clic(event):
4     x=event.x
5     y=event.y
6     print("Position :", x, y)
7
8 root = Tk()
9 root.bind("<Button-1>",clic)
10 root.mainloop()
```

il est possible de capturer la position de la souris lors de l'exécution de l'événement via les prises d'attributs `event.x` et `event.y` (lignes 4 et 5 dans l'exemple).

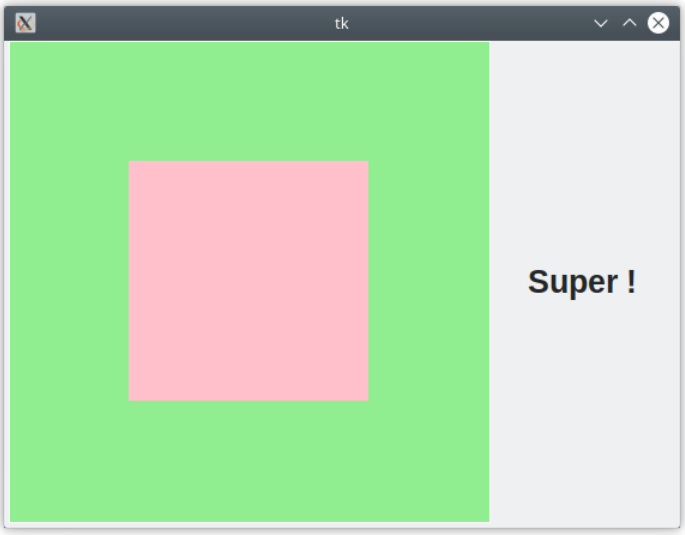
Bien que la docstring de Tkinter ne le précise pas, il semble que ces positions soient de type entier et non pas flottant, comme semble le montrer cet extrait du code-source de Tkinter :

```
1 def _substitute(self, *args):
2
3         # ....
4
5         getint = self.tk.getint
6         def getint_event(s):
7             """Tk changed behavior in 8.4.2, returning "??" rather more often."""
8             try:
9                 return getint(s)
10            except (ValueError, TclError):
11                return s
12
13        # ....
14
15        e.x = getint_event(x)
16        e.y = getint_event(y)
```

Désassocier un événement

Quand une partie dans un jeu est terminée (par exemple, le joueur a perdu), on souhaite désactiver la capture de certains événements du clavier ou de la souris et qui avaient été initialement associés à certains widget. On utilise pour cela la méthode `unbind` et elle peut s'appliquer à n'importe quel type d'événement.

Voici un exemple :



si le joueur clique sur la zone verte, il reçoit un message et la partie peut alors continuer, sinon, un message indique **Perdu !** et la zone de clic est désactivée.

Le code est ci-dessous :

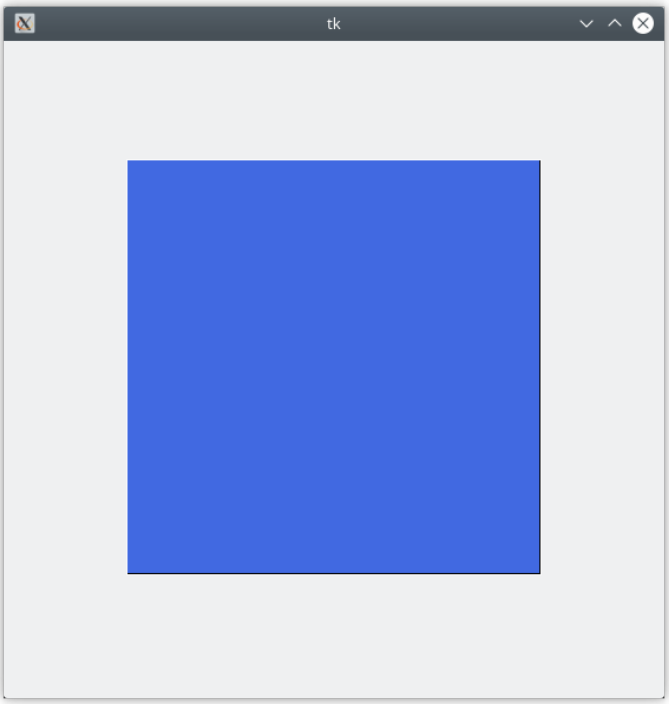
```
from random import randrange
from tkinter import *

root = Tk()
cnv=Canvas(width=400, height=400, bg="lightgreen")
cnv.pack(side="left")
lbl=Label(text='', width=10, font="arial 20 bold")
lbl.pack(side='right')

MESSAGE=["Bien\n joué !", "Nice!", "Super !", "Bravo !", "Champion !", "OK !"]

def f(event):
    x, y=event.x, event.y
    if 100<x<300 and 100<y<300:
        lbl['text']="Perdu !"
        root.unbind("<Button-1>")
    else:
        lbl['text']=MESSAGE[randrange(len(MESSAGE))]


A=(100,100)
B=(300, 300)
cnv.create_rectangle(A, B, fill='pink', outline='')

root.bind("<Button-1>", f)

root.mainloop()
```

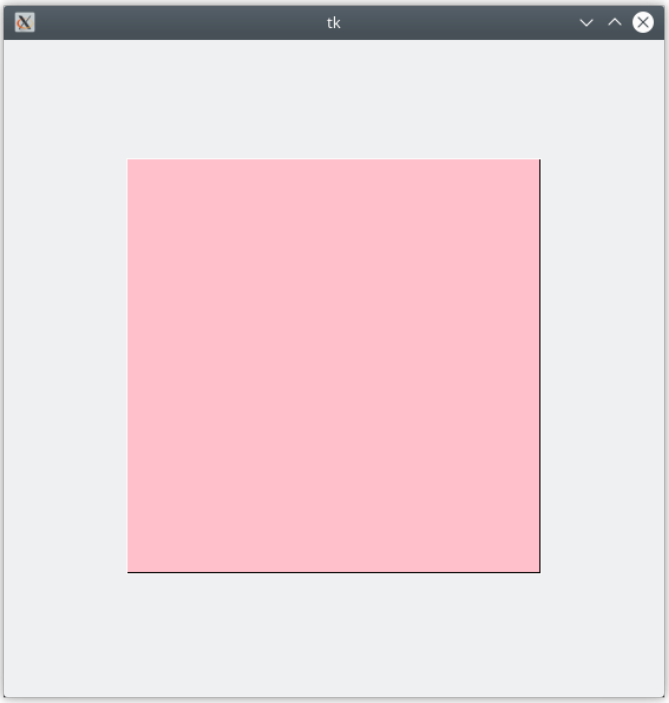
— Ligne 25 : la fonction `f` est associée à un clic gauche dans la fenêtre.
— Ligne 16 : si le joueur clique dans la zone en rose (cf. ligne 14 et ligne 23), on lui annonce qu'il a perdu (ligne 15) et le clic gauche dans la fenêtre ne réagira plus (à cause de `unbind`).

Si la méthode `unbind` est appelée sur un événement sans nom de fonction particlulière (comme c'est le cas dans l'exemple ci-dessus), *toutes* les fonctions du widget associées à l'événement sont désactivées. Mais il serait possible de faire une désactivation sélective.

Modifier un widget par survol de la souris

Un événement du type `<Enter>` ne doit pas être confondu avec un événement d'appui sur la touche `Enter` du clavier (qui est plutôt l'événement `<Return>`). L'événement `<Enter>` est activé lorsque la souris **entre** dans la zone délimitée par un widget. Il existe aussi l'événement inverse `<Leave>` qui est activé lorsque la souris **quitte** la zone. Voici un exemple :






```
from tkinter import *

root = Tk()
side=350
pad=30
cnv = Canvas(root, width=side, height=side)
cnv.pack(padx=100, pady=100)

def go_in(event):
    cnv['bg']="pink"

def go_out(event):
    cnv['bg']="royal blue"

cnv.bind("<Enter>", go_in)
cnv.bind("<Leave>", go_out)

root.mainloop()
```

Quand la souris entre dans le canevas `cnv`, la couleur de fond passe en rose. Quand la souris quitte le canevas,la couleur de fond passe en bleu.

Chapitre IV

Le canevas

Le widget Canvas

Tkinter dispose d'un widget canevas : c'est une surface permettant de dessiner des formes géométriques (rectangles, disques, du texte, etc) et de les manipuler (personnalisation, déplacement, suppression, etc). Ce type de widget permet de créer des jeux de toutes sortes.

Le widget `Canvas` est très souple d'emploi grâce à son système de tags et d'id. Il est robuste car, selon cet expert de TkInter, il peut supporter la gestion de plusieurs dizaines de milliers d'items.

Dans le jargon `Tkinter`, les formes géométriques sont appelés des **items**. A la différences des widgets Tkinter tels qu'un bouton ou un label ou une fenêtre qui sont définis par des classes comme la classe `Button`, la classe `Label`, etc et qui contiennent d'autres widgets, les formes géométriques que l'on peut dessiner sur le canvas ne sont ni des widgets ni des instances de classes.

Voici en résumé les items les plus utiles qu'un canevas peut générer :

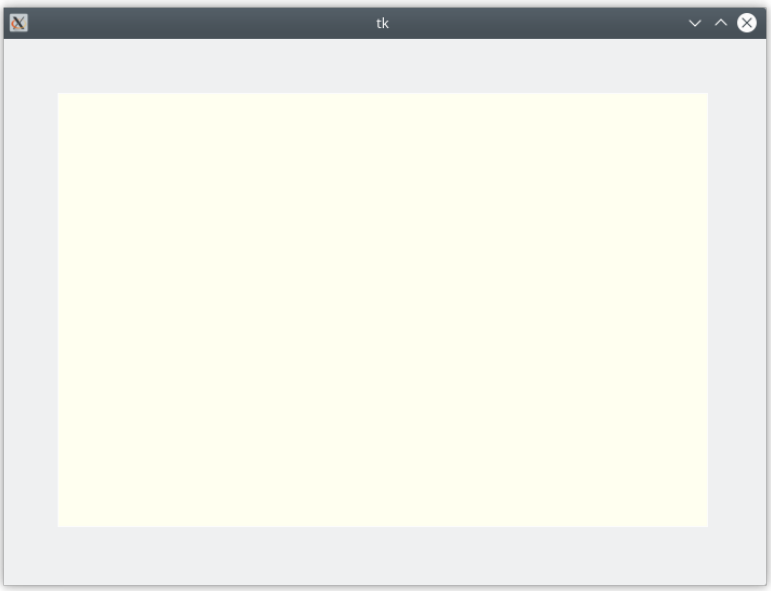
Motif géométrique	**Item Tkinter**
segment de droite, flèche	`create_line`
rectangle carré	`create_rectangle`
cercle ellipse	`create_oval`
texte	`create_text`
image png, gif	`create_image`
polygone	`create_polygon`
arcs de cercle, d'ellipse	`create_arc`

Création d'un canevas

Voici un code qui crée un simple canevas :

```
1 from tkinter import Tk, Canvas
2
3 root=Tk()
4 cnv=Canvas(root, width=600, height=400, bg="ivory")
5 cnv.pack(padx=50, pady=50)
6 root.mainloop()
```

ce qui affiche :



On crée une surface canevas avec le constructeur `Canvas` (c'est une classe). Comme pour tout widget, il faut lui indiquer comme premier argument le widget qui va contenir le canevas à créer, dans l'exemple ci-dessus, c'est la fenêtre appelée `root`.

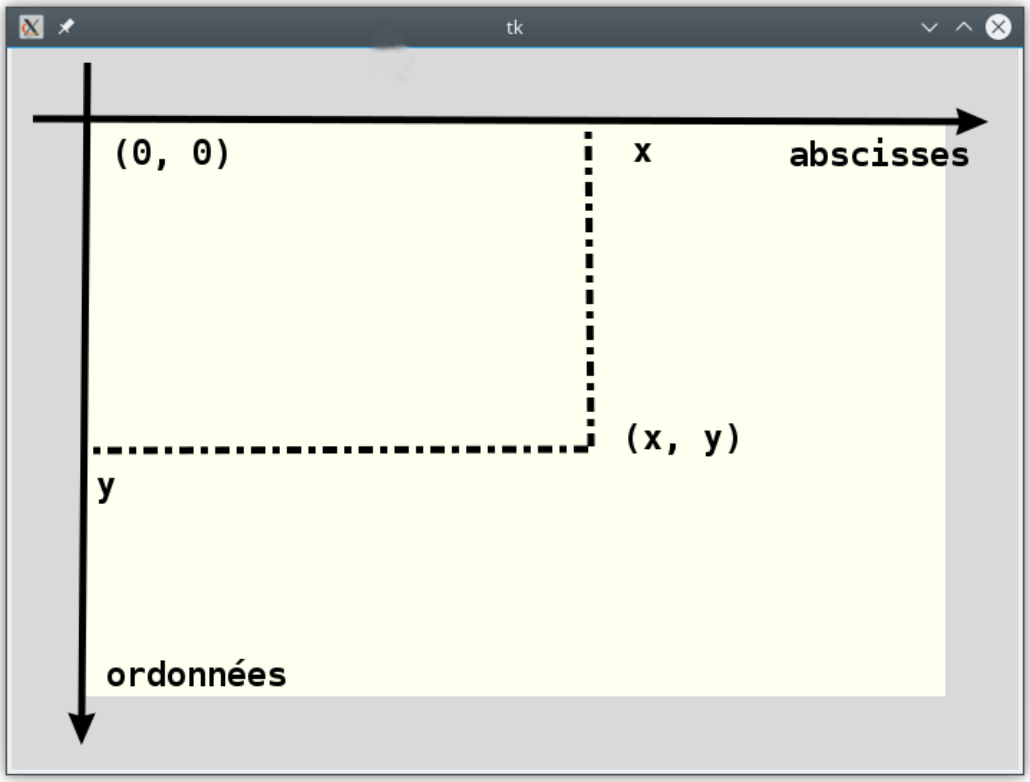
On lui indique les dimensions du canevas avec les arguments nommés `width` et `height`. On peut indiquer une couleur de fond avec l'argument nommé `bg` (qui signifie *background*). Si on ne met rien, par défaut c'est une couleur grise qui se confond avec la couleur par défaut du widget parent, ce qui ne permet pas de bien distinguer la surface (et voilà pourquoi, en principe, je placerai une couleur de fond dans mes canevas).

Repérage dans un canevas

Soit le code suivant construisant un canevas :

```
from tkinter import Tk, Canvas

root=Tk()
cnv=Canvas(root, width=200, height=200, bg="ivory")
cnv.pack()
root.mainloop()
```

Un canevas (défini par le constructeur `Canvas`) est muni d'un système de repérage (invisible) dans lequel chaque point du canevas a deux coordonnées :



Attention, le système de repérage est orienté **différemment** du système utilisé en mathématiques :

— l'origine est le coin en haut à gauche
— l'axe des abscisses est, comme en math, l'axe horizontal orienté de gauche à droite
— l'axe des ordonnées est, comme en math, un axe vertical mais il est orienté vers le bas (assez logique vue la configuration, non ?).

Les coordonnées repèrent des pixels. Pour être plus précis, le premier pixel est numéroté 0, le suivant 1, etc. Si un canevas est créé avec l'option `width=200` le pixel le plus à droite du canevas est numéroté 199.

Bords d'un canevas

Le contenu de ce paragraphe peut être réservé à un approfondissment.

Lorsqu'on définit un canevas, on peut lui fournir deux options `highlightthickness` et `bd` qui prennent chacune des pixels :

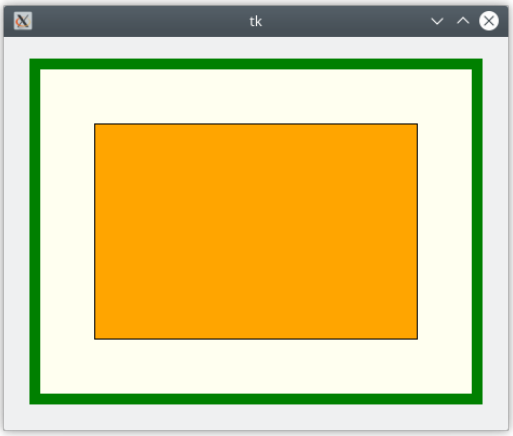
— l'option `bd` est la largeur d'une bande autour du canevas et qui a même couleur de fond ; l'option `borderwidth` est synonyme de `bd` ;
— l'option `highlightthickness` est la largeur d'une bande autour du canevas (et de son bord) et qui est activé lorsque le canevas a le focus (cad que le canevas est susceptible d'enregistrer des touches au clavier ou des activités de la souris comme cela est utile dans un jeu) ; par défaut, cette bande d'activation du canevas est blanche et de deux pixels de large.

La plupart du temps, on peut ignorer ces options et si vous débutez complètement vous pouvez ignorer ce qui suit. Mais, parfois le comportement par défaut de ces options, justifie qu'on les modifie. La documentation n'explique pas clairement comment ces options affectent la zone active du canevas (celle où on peut dessiner), avec pour conséquence que parfois, certaines zones, prêts des bords en particulier, sont rendues invisibles.

Soit le code suivant

```
from tkinter import Tk, Canvas

root=Tk()

# Largeur du canevas
W=300

# Hauteur du canevas
H=200

# Largeur du bord
B=50

# Largeur de la zone de focus
F=10

cnv=Canvas(root, width=W, height=H, bg="ivory",
           bd=B,
           highlightthickness=F,
           highlightbackground="green")

cnv.pack(side="left", padx=20, pady=20)

# Partie active du canevas
z=B+F
w=W-1
h=H-1
cnv.create_rectangle(z,z,z+w,z+h, fill="orange")

root.mainloop()
```

Il produit la fenêtre suivante :



que l'on va décrire en liaison avec le code :

— l'origine (0,0) du canevas se trouve dans la zone verte, au coin en haut à gauche ;
— en vert, la bande d'activation du focus. Son épaisseur est `highlightthickness` (ligne 19). La couleur provient de l'option `highlightbackground` donnée au canevas ;
— en ivoire, le bord du canevas, de couleur `bg="ivory"`, cf. ligne 17. La largeur du bord est donnée par `bd` (ligne 18)
— en orange, un rectangle dont les dimensions sont calculées ici pour recouvrir exactement la totalité de la zone du canevas sur laquelle on peut dessiner de **manière visible** ; en dehors de cette zone, tout item généré (rectangle, ligne, etc) sera **caché**.
— le rectangle orange (lignes 25-28) a les dimensions déclarées dans le contructeur `Canvas` ligne 17 c'est-à-dire `width=W` et `height=H`.
— Noter le comportement **non intuitif** suivant :
  — un item placé en un point `(x,y)` où $0 <$ `x` $<$ `width` et $0 <$ `y` $<$ `height` peut être invisible ;
  — un item placé en un point `(x,y)` ne vérifiant pas $0 <$ `x` $<$ `width` ou $0 <$ `y` $<$ `height` peut être visible ;

La plupart du temps, le comportement n'utilisant pas les options `highlightthickness` et `bd` convient car les valeurs par défaut des options sont de 1 ou 2 pixels, ce qui est peu visible.

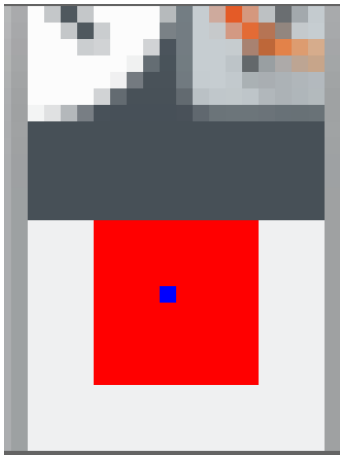
Mais si on veut faire des dessins **très précis** sur un canevas, et un comportement plus intuitif selon lequel les coordonnées visibles sont exactement `range(W)` et `range(H)` où `W` et `H` sont les dimensions données au canevas pour `width` et `height`, je conseille de mettre `highlightthickness` et `bd` à 0.

Voici le même dessin avec ces options placées à zéro :

```
1 from tkinter import Tk, Canvas
2
3 root=Tk()
4
5 # Largeur du canevas
6 W=300
7
8 # Hauteur du canevas
```

```
H=200

# Largeur du bord
B=0

# Largeur de la zone de focus
F=0

cnv=Canvas(root, width=W, height=H, bg="ivory",
           bd=B,
           highlightthickness=F)

cnv.pack(side="left", padx=20, pady=20)

# Partie active du canevas
z=B+F
w=W-1
h=H-1
cnv.create_rectangle(z,z,z+w,z+h, fill="orange")

root.mainloop()
```

et la sortie :



Limites des objets dessinés sur le canevas

Les positions données lors de la construction d'objets tels que des rectangles, etc sont toujours comprises au sens large. Par exemple, un rectangle créé par

```
cnv.create_rectangle((10, 20), (100, 200))
```

commence en largeur au pixel du canevas numéroté 10 et se termine au pixel numéroté 100, ce pixel étant inclus. Donc les dimensions du rectangle sont, **si on inclut les bords**, de 91 pixels.

Sauvegarder le contenu du canevas

Vous avez dessiné quelque chose sur un canevas et vous souhaitez le sauvegarder dans un format image, png par exemple. Sous Tkinter, la seule sortie possible est le format eps (du postscript encapsulé) qui peut se lire sous Gimp facilement et se convertir en png par exemple.

On a besoin de la méthode `postscript` du canevas. Voici un exemple d'utilisation :

```
1  from tkinter import *
2
3  SIDE=300
4  SEP=50
5
6  def print_cnv():
7      cnv.postscript(file="rect.eps", colormode='color')
8
9  master=Tk()
10
11 cnv=Canvas(master, width=SIDE, height=SIDE, bg='lavender')
12 cnv.pack()
13
14 cnv.create_rectangle(SEP, SEP, SIDE-SEP, SIDE-SEP, fill="gray")
15 cnv.create_text(SIDE/2, SIDE/2, text="Tkinter",
16                 fill="white",
17                 font="Times 30 bold")
18
19 cnv.after(1000, print_cnv)
20
21 master.mainloop()
```

— Ligne 7 : la méthode `postscript` du canevas permet d'exporter au format `eps`. On désigne un fichier de sortie avec l'argument nommé `file`.

— Lignes 14-15 : on peut capturer les items de Tkinter comme des rectangles ou du texte. La capture d'image Photoimage ne fonctionne pas.

— Ligne 11 : la couleur de fond du canevas (qui n'est pas un item) n'est pas capturé.

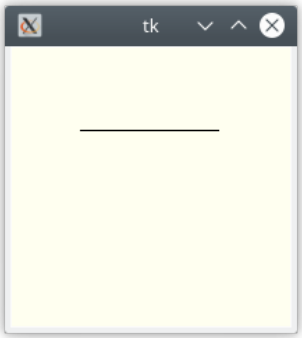
— Ligne 19 : il faut attendre que le canevas soit dessiné.

Voici la fenêtre et son image capturée (et convertie en png) :

1 Items par catégories

Dessiner un segment, une ligne brisée

Un segment est généré par la méthode `create_line` du `Canvas` :



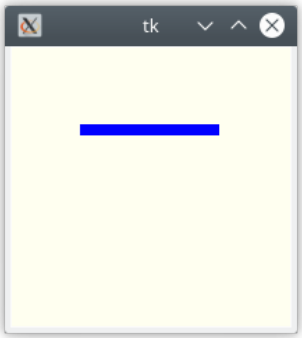
Voici le code qui trace un segment d'extrémités les points A et B :

```
from tkinter import Tk, Canvas

root=Tk()
cnv=Canvas(root, width=200, height=200, bg="ivory")
cnv.pack()

A=(50, 60)
B=(150, 60)
cnv.create_line(A, B)
root.mainloop()
```

On peut aussi utiliser la syntaxe moins lisible `cnv.create_line(50, 60, 150, 60)`.

Pour la couleur du trait (noire par défaut), utiliser l'option `fill` et l'épaisseur de la ligne (1 pixel par défaut), utiliser l'option `width` :

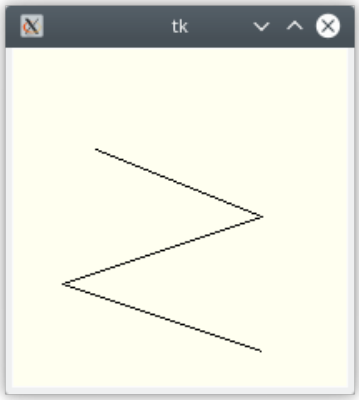
```
1 from tkinter import Tk, Canvas
2
3 root=Tk()
4 cnv=Canvas(root, width=200, height=200, bg="ivory")
5 cnv.pack()
6
7 cnv.create_line(50, 60,150, 60, width=8, fill="blue")
8 root.mainloop()
```



Plus généralement, `create_line` permet de tracer une ligne polygonale :

```
from tkinter import Tk, Canvas

root=Tk()
cnv=Canvas(root, width=200, height=200, bg="ivory")
cnv.pack()

A=50, 60
B=150, 100
C= 30,140
```

```
D=150, 180

cnv.create_line(A, B, C, D)
root.mainloop()
```



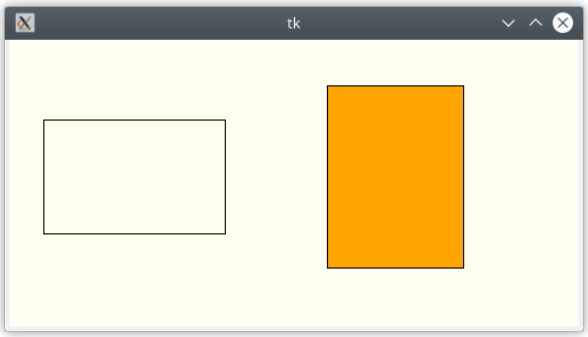
On peut utiliser la syntaxe moins lisible

```
cnv.create_line(50, 60,150, 100, 30,140, 150, 180)
```

Dessiner un rectangle

Pour dessiner un rectangle sur un canevas cnv, on appelle la méthode `cnv.create_rectangle` :

```
1 from tkinter import Tk, Canvas
2
3 root = Tk()
4 cnv = Canvas(root, width=500, height=250, bg='ivory')
5 cnv.pack()
6
7 cnv.create_rectangle(30, 70, 190, 170)
8 cnv.create_rectangle((280, 40), (400, 200), fill='orange')
9
10 root.mainloop()
```

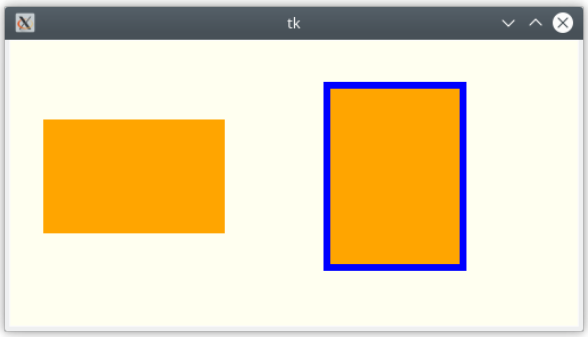


Pour le positionnement du rectangle sur le canevas, il faut donner les coordonnées des extrémités d'une des deux diagonales du rectangle :

— soit en les plaçant côte à côte (ligne 7)

— soit en plaçant les coordonnées entre parenthèses (ligne 8), ce qui est plus lisible.

Un rectangle peut être dessiné « rempli » avec l'argument nommé `fill` prenant un nom de couleur (ligne 8). Par défaut, un rectangle est toujours délimité par des côtés qui sont formés d'une ligne noire d'un pixel d'épaisseur. Si on veut supprimer cette bordure, on donne l'argument `outline=''`. Si on veut colorier par exemple en bleu cette bordure, on écrira `outline="blue"` :




```
from tkinter import Tk, Canvas

root = Tk()
cnv = Canvas(root, width=500, height=250, bg='ivory')
cnv.pack()

cnv.create_rectangle(30, 70, 190, 170, fill="orange", outline="")
cnv.create_rectangle((280, 40), (400, 200),
                     fill='orange',
                     outline="blue",
                     width=6)

root.mainloop()
```

Ordre des sommets et `create_rectangle`

Si on construit un rectangle par un appel `cnv.create_rectangle((a, b), (c, d))`, il n'est pas nécessaire que `a` $\leq$ `c` ou que `b` $\leq$ `d`; en fait Tkinter construit un rectangle de diagonales les points de coordonnées `(a, b)` et `(c,d)` ce qui bien sûr, permet de tracer le rectangle.

```
from tkinter import Tk, Canvas

root = Tk()
cnv = Canvas(root, width=400, height=400)
cnv.pack()

cnv.create_rectangle((50, 50), (150, 100), fill='red', outline='')
cnv.create_rectangle((250, 150), (350, 50), fill='green')

cnv.create_rectangle((150, 300), (50, 250), fill='orange')
cnv.create_rectangle((350, 250), (250, 350), fill='brown', outline='')

root.mainloop()
```

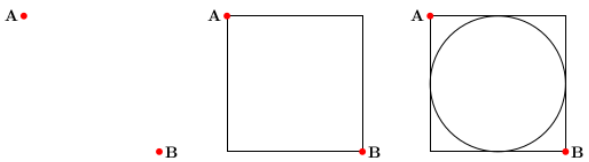
Une précision toutefois. Par convention,

— le bord gauche et le bord supérieur du rectangle ainsi identifiés sont considérés comme faisant partie du rectangle

— le bord droit et le bord inférieur est considéré comme NE faisant PAS partie du rectangle bien que ces bords soient déssinés (si l'option `outline` n'est pas marquée `outline=''`).

Dessiner un cercle, un disque

Contrairement à ce qu'on s'attendrait, un cercle n'est pas construit en donnant son centre et son rayon. En fait, le cercle est produit comme si on suivait les trois étapes ci-dessous :

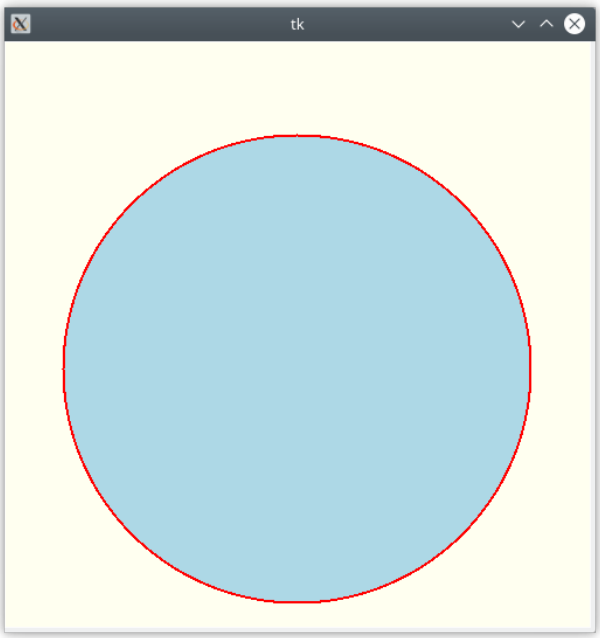


On donne à Tkinter deux points `A` et `B` qui sont deux sommets opposés du carré ayant leurs côtés parallèles aux axes et **circonscrit** au cercle que l'on veut dessiner.

Voici un code de dessin :

```
1 from tkinter import Tk, Canvas
2 SIDE=500
3
4 root=Tk()
```

```
5 cnv=Canvas(root, width=SIDE, height=SIDE, bg="ivory")
6 cnv.pack()
7
8 diam=400
9 A=(a,b)=(50, 80)
10 B=(a+diam, b+diam)
11
12 cnv.create_oval(A, B, fill='light blue', outline='red', width=2)
13
14 root.mainloop()
```



— Lignes 8-10 : le côté du carré et le diamètre du cercle ont même longueur.

— Ligne 12 :

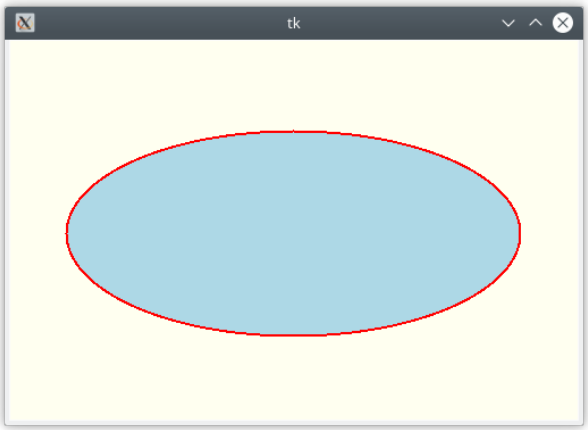
  — l'option `fill` détermine la couleur de remplissage ; en l'absence de cette option, un cercle plutôt qu'un disque est dessiné (absence de remplissage) ;

  — l'option `outline` définit la couleur du cercle ; en l'absence de cette option, un cercle de 1 pixel d'épaisseur entoure le disque ;

  — l'option `width` détermine, en pixel, l'épaisseur du bord.

Noter que les deux espacements, vertical et horizontal, entre `A` et `B` doivent être égaux. Cet espacement est le diamètre du cercle (dans le code, c'est `diam`). Le centre du cercle est le milieu des points `A` et `B`.

Si les espacements ne sont pas identiques, on obtient plus un cercle mais une ellipse :




```
from tkinter import Tk, Canvas
SIDE=500

root=Tk()
cnv=Canvas(root, width=SIDE, height=SIDE//1.5, bg="ivory")
cnv.pack()

da=400
db=180
A=(a,b)=(50, 80)
B=(a+da, b+db)

cnv.create_oval(A, B, fill='light blue', outline='red', width=2)

root.mainloop()
```

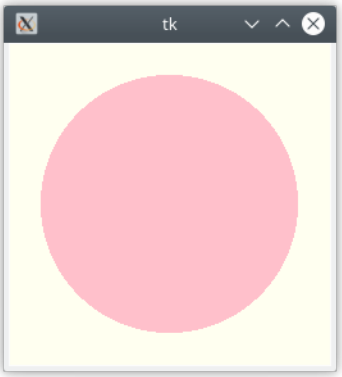
Disque de centre et de rayon donnés

Tkinter ne dispose pas de fonction de tracé de cercle mais il est facile d'en créer une :

```
from tkinter import *

def dot(cnv, C, R=6, color='red'):
    xC, yC=C
    A=(xC-R, yC-R)
    B=(xC+R, yC+R)
    return cnv.create_oval(A,B, fill=color, outline=color)

PAD=50
DIM=200
WIDTH=DIM+PAD
HEIGHT=DIM+PAD

```

```
14 root = Tk()
15 cnv = Canvas(root, width=WIDTH, height=HEIGHT, background="ivory")
16 cnv.pack()
17
18 C = (WIDTH//2, HEIGHT//2)
19
20 dot(cnv, C, R=DIM//2, color="pink")
21
22 root.mainloop()
```

ce qui produit :

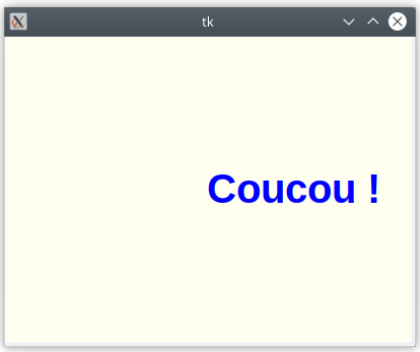


Quelques explications :

- Lignes 5-6 : si le centre du cercle est $\texttt{C} = (\texttt{xC}, \texttt{yC})$ et que son rayon est `R`, c'est que le sommet `A` en haut à gauche du carré extérieur au cercle est $\texttt{A} = (\texttt{xC} - \texttt{R}, \texttt{yC} - \texttt{R})$ et de même, le sommet `B` en bas à droite est $\texttt{B} = (\texttt{xC} + \texttt{R}, \texttt{yC} + \texttt{R})$;
- Ligne 3 : la fonction `dot` recevra comme argument le canevas (`cnv`) sur lequel le disque sera dessiné, comme cela la fonction facilement réutilisable ; on donne, bien sûr, le centre `C` et le rayon `R` et une couleur de fond, par défaut rouge ;
- Ligne 7 : on utilise la méthode `create_oval` pour dessiner le disque ; le bord de cercle (`outline`) a même couleur que le fond (ça évite un liseré noir comme bord).
- Ligne 10 : `DIM` sera le diamètre du disque, `PAD` est une marge pour que le disque ne soit pas trop collé au bord du canevas.
- Ligne 18 : les coordonnées du centre choisi. On a centré le point dans le canevas.
- Ligne 20 : dessin d'un disque.

Placer du texte dans le canevas

De la même façon que l'on peut créer des figures géométriques sur un canevas (rectangles, etc), on peut créer des items de texte :

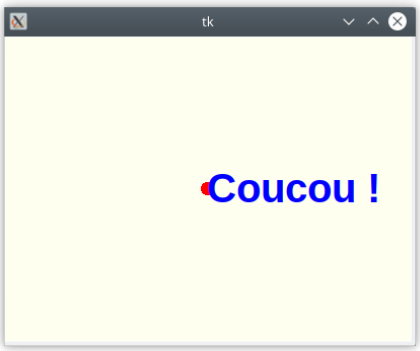


qui est produit par le code suivant :

```
1  from tkinter import *
2
3  WIDTH=400
4  HEIGHT=300
5
6  root = Tk()
7  cnv = Canvas(root, width=WIDTH, height=HEIGHT, background="ivory")
8  cnv.pack()
9  C=(WIDTH//2, HEIGHT//2)
10
11 cnv.create_text(C, anchor =W,
12                  text ="Coucou !", fill ="blue", font="Arial 30 bold")
13
14 root.mainloop()
```

On indique en premier argument de `create_text` (point `C` de la ligne 11 qui est défini en ligne 9) un point de référence (une ancre) par rapport auquel le texte sera positionné sur le canevas.

On peut donner une option d'ancre (`anchor`) qui indique à quel point cardinal le point de référence se trouve par rapport au texte. Le point cardinal est le nord, sud, est, ouest, etc mais abrégé en anglais et en majuscule ou sinon entre quotes. Dans l'exemple, le point de référence est placé à l'ouest du texte comme on peut le voir sur la sortie ci-dessous où le point de référence est marqué en rouge :



C'est assez peu intuitif : c'est le point indiqué avec ses coordonnées (ici `C`) qui est à l'ouest du texte et non pas le contraire !

En l'absence d'option `anchor` le texte est centré au point de référence.

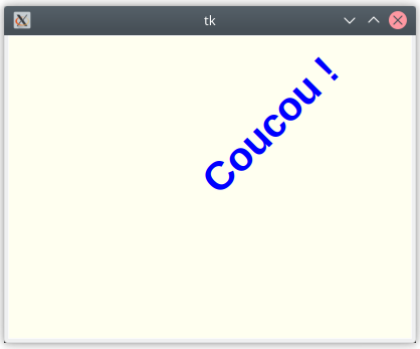
Le contenu du texte (une chaîne de caractères) est placé dans l'argument nommé `text`. Parmi les options se trouvent :

— le remplissage (`fill` affecté à un nom de couleur comme `'blue'` dans l'exemple)

— le choix de la fonte avec l'option `font` (ligne 12).

Si le texte est trop long par rapport à la surface du canevas, il sera coupé et sortira (donc invisible).

Il existe une option `angle` permettant d'incliner le texte (ligne 13 ci-dessous) :




```
1  from tkinter import *
2
3  WIDTH=400
4  HEIGHT=300
5
6  root = Tk()
7  cnv = Canvas(root, width=WIDTH, height=HEIGHT, background="ivory")
8  cnv.pack()
9  C=(WIDTH//2, HEIGHT//2)
10
11 cnv.create_text(C, anchor =W,
12                  text ="Coucou !", fill ="blue", font="Arial 30 bold",
13                  angle=45)
14
15 root.mainloop()
```

Inclusion d'images sur le canevas

Le canevas supporte le placement d'images présentes sur une unité de stockage. Seuls les formats

— png (depuis Python 3.4 sous Windows et peut-être Python 3.6 pour Mac Os)

— gif

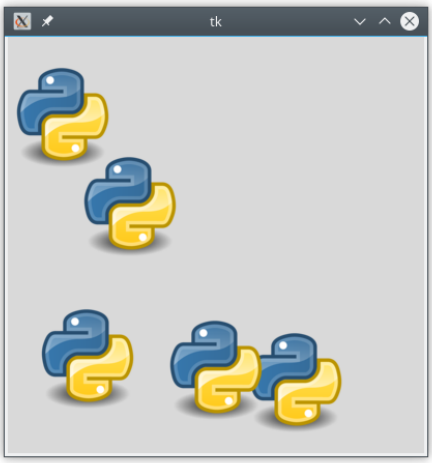
sont pris en charge. Le format `jpeg` ne semble pas pris en charge, cf. ce message Tkinter error: Couldn't recognize data in image file.

Exemple de code :

inclusion_images.py

```
1  from tkinter import *
2  from random import randrange
3
4  SIDE=400
5  root = Tk()
6  cnv = Canvas(root, width=SIDE, height=SIDE)
7  cnv.pack()
8
9  logo = PhotoImage(file="python.gif")
10
11 for i in range(5):
12     centre= (randrange(SIDE),randrange(SIDE))
13     cnv.create_image(centre, image=logo)
14
15 root.mainloop()
```

et qui affiche



Le code ci-dessus place aléatoirement 5 images sur un canevas.

Avant de pouvoir utiliser une image (ligne 13), il faut la convertir dans un format propre à Tkinter. Pour cela (ligne 9), on doit utiliser la classe `PhotoImage` en lui précisant l'adresse du fichier image sur le disque. Une fois l'image convertie par appel à `PhotoImage`, on peut l'incorporer dans un canevas en utilisant la méthode du canevas appelée `create_image`. Comme les dimensions d'une image sont invariables, il suffit de donner à `create_image` les coordonnées du point où on souhaite que le centre de l'image se trouve sur le canevas.

Remarquer qu'on a besoin de créer juste **une seule** instance de `PhotoImage` même si elle sert à la création de **plusieurs** images.

L'image ne s'adapte pas automatiquement au canevas où l'image est placée, si l'image est plus

large que le canevas, elle ne sera qu'en partie visible dans le canevas.

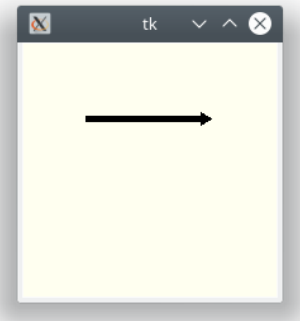
Noter qu'il existe une syntaxe peut-être moins lisible pour désigner les coordonnées du centre de l'image : au lieu d'écrire `cnv.create_image(c, image=logo)` après avoir défini la variable `c=(x,y)`, on peut écrire `cnv.create_image(x, y, image=logo)`.

Pour le support du png sous macOS, voir IDLE and tkinter with Tcl/Tk on macOS.

Dessiner un segment fléché

Pour dessiner un segment avec une extrémité en forme de flèche, utiliser l'argument nommé `arrow` de la méthode `cnv.create_line` :

```
1 from tkinter import Tk, Canvas
2
3 root=Tk()
4 cnv=Canvas(root, width=200, height=200, bg="ivory")
5 cnv.pack()
6
7 cnv.create_line(50, 60,150, 60, width=5, arrow='last')
8 root.mainloop()
```



L'option `'last'` signifie que c'est l'extrémité donnée en 2e qui est fléchée. Pour que ce soit l'origine qui soit fléché, l'option à donner est `"first"` et pour les deux extrémités soient fléchées, utiliser la valeur `"both"`. La largeur de la flèche s'adapte à la largeur du segment (`width`). Il semble qu'on ne puisse générer des flèches que pour des segments, pas pour des acs.

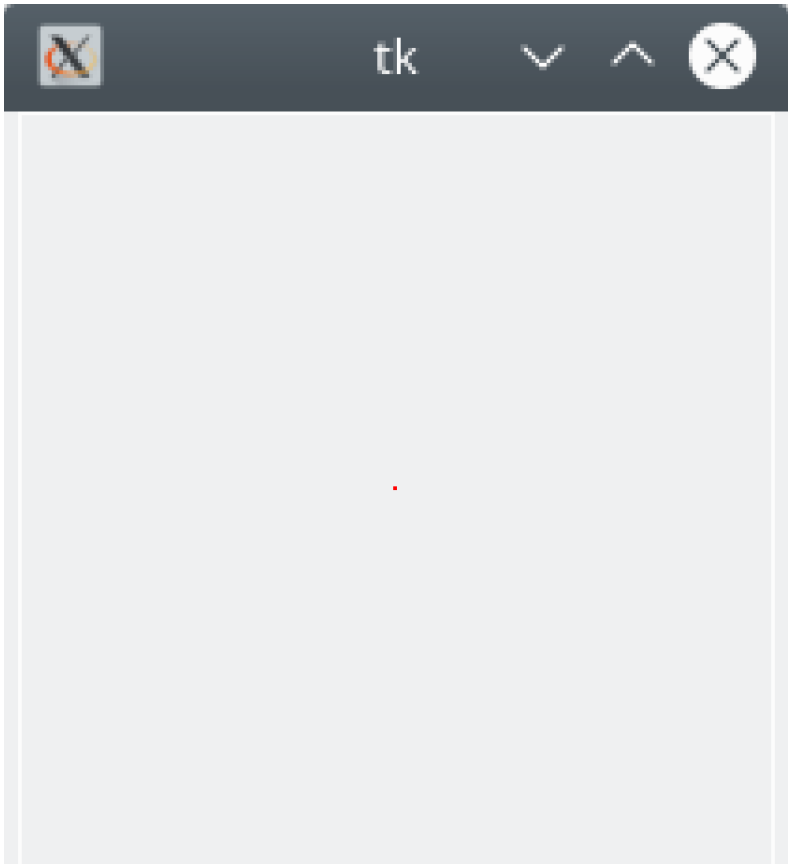
Dessiner un unique pixel

Pour se repérer (lorsqu'on débogue un code par exemple), on peut avoir besoin de placer un point ayant une taille d'un **unique pixel** :

```
1 from tkinter import Tk, Canvas
2
3 root = Tk()
```

```
cnv = Canvas(root, width=200, height=200)
cnv.pack()
cnv.create_rectangle(100, 100, 100, 100, fill='red', outline='')

root.mainloop()
```

ce qui, agrandi, donne :



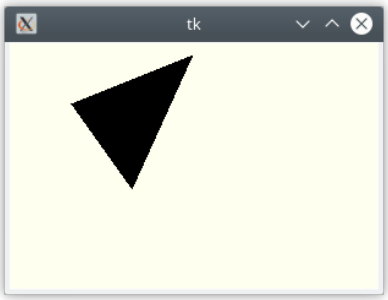
Tracer un polygone

Un polygône est tracé par la méthode `create_polygon` à qui on donne les coordonnées des sommets :

```
from tkinter import *

WIDTH=300
HEIGHT=200

root = Tk()
cnv = Canvas(root, width=WIDTH, height=HEIGHT, background="ivory")
cnv.pack()

cnv.create_polygon(50,50, 100, 120, 150, 10)

```

```
12 root.mainloop()
```

ce qui poduit :



Par défaut, les polygones sont remplis en noir et sans bordure, ie `fill="black"` et `outline=''`, cf. code ci-dessous pour une alternative.

Le dernier sommet est relié au premier.

Il est aussi possible de placer un sommet par son couple de coordonnées ; le code est alors plus lisible :

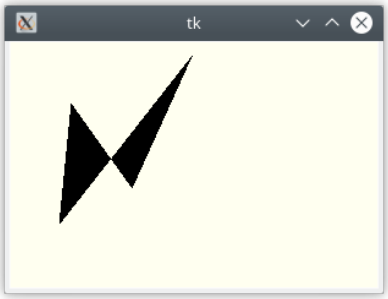
```
1  from tkinter import *
2  from random import randrange, sample
3
4  WIDTH=300
5  HEIGHT=200
6
7  root = Tk()
8  cnv = Canvas(root, width=WIDTH, height=HEIGHT, background="ivory")
9  cnv.pack()
10
11 A=50,50
12 B=100, 120
13 C=150, 10
14
15 triangle=cnv.create_polygon(A, B, C, fill='lavender', outline='red')
16 L=cnv.coords(triangle)
17
18 root.mainloop()
```

La suite des coordonnées des sommets détermine le caractère convexe ou concave du polygone :

```
1  from tkinter import *
2
3  WIDTH=300
4  HEIGHT=200
5
6  root = Tk()
7  cnv = Canvas(root, width=WIDTH, height=HEIGHT, background="ivory")
```

```
8 cnv.pack()
9
10 cnv.create_polygon((50,50), (100, 120), (150, 10), (40,150))
11
12 root.mainloop()
```

ce qui produit



Bord d'un rectangle

Par défaut, un bord noir de 1 pixel d'épaisseur est rajouté à chaque rectangle. On peut agir sur la couleur du bord avec l'option `bd` pour *outline* :

```
1 from tkinter import Tk, Canvas
2
3 root = Tk()
4 cnv = Canvas(root, width=480, height=120, bg='ivory')
5 cnv.pack()
6
7 cnv.create_rectangle(10, 10, 110, 110, fill='light blue')
8 cnv.create_rectangle(120, 10, 220, 110, fill='light blue', outline='')
9 cnv.create_rectangle(230, 10, 340, 110, fill='light blue', outline='red')
10 cnv.create_rectangle(350, 10, 460, 110, fill='light blue',
11                      outline='light blue')
12
13 root.mainloop()
```

On voit donc qu'on peut faire disparaître de deux façons le bord d'un rectangle : soit en plaçant `outline=''` (ligne 8), soit en confondant sa couleur avec la couleur de fond (ligne 11).

Bord, intérieur et dimensions exactes d'un rectangle

Le contenu de cette unité peut être examiné en seconde lecture.

Le premier pixel en haut à gauche d'un rectangle est toujours dessiné.

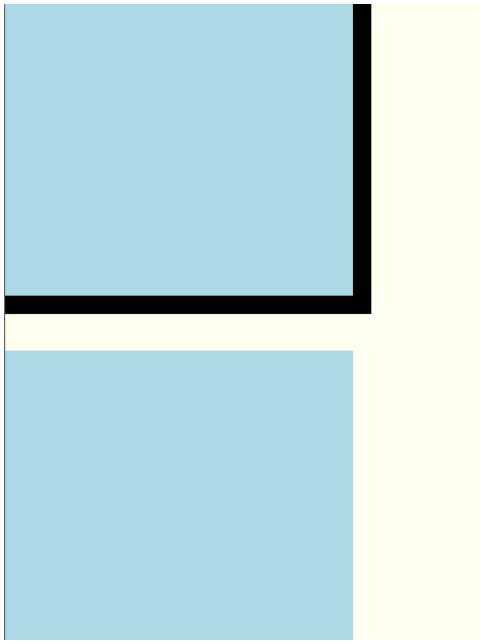
Par défaut, un bord de 1 pixel d'épaisseur est rajouté à chaque rectangle mais de manière dissymétrique :

— le bord en bas et le bord à droite sont extérieurs au rectangle,
— le bord en haut et le bord à gauche sont intérieurs au rectangle et viennent recouvrir la couleur de fond.

Ainsi, si un côté a pour abscisse a et le côté parallèle a pour abscisse b, la largeur par défaut du rectangle est $|a-b|+1$ et de $|a-b|$ si le bord est exclu. Ne pas oublier toutefois que si $a=b$ le pixel est quand même dessiné.

Voici un code et l'image agrandie qui en résulte :

```
from tkinter import Tk, Canvas

root = Tk()
cnv = Canvas(root, width=130, height=250, bg='ivory')

cnv.pack()

cnv.create_rectangle(10, 10, 110, 110, fill='light blue')
cnv.create_rectangle(10, 113, 110, 223, fill='light blue', outline='')

root.mainloop()
```



On voit bien le décalage de 1 pixel entre la version avec bord et la version sans bord.

Lorsque le bord est retiré, les dimensions d'un rectangle sont faciles à calculer : c'est la différence entre les abscisses et la différence entre les ordonnées. Par exemple ci-dessous le rectangle est de largeur 100 et de hauteur 50 :

```
from tkinter import Tk, Canvas

root = Tk()
cnv = Canvas(root, width=120, height=60, bg='ivory')
```

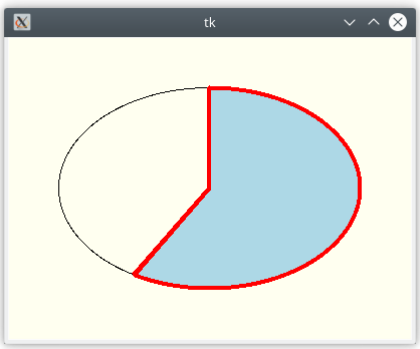
```
5
6 cnv.pack()
7
8 cnv.create_rectangle(10, 10, 110, 60, fill='light blue', outline='')
9
10
11 root.mainloop()
```

Créer un arc

Voici un exemple de tracé de secteur angulaire dans une ellipse :

```
1 from tkinter import *
2
3 WIDTH=400
4 HEIGHT=300
5
6 root = Tk()
7 cnv = Canvas(root, width=WIDTH, height=HEIGHT, background="ivory")
8 cnv.pack()
9 A=(50,50)
10 B=(350, 250)
11
12 cnv.create_oval(A, B)
13 cnv.create_arc(A, B, outline="red", extent=210, start=-120,
14                fill="light blue",  width=4)
15 root.mainloop()
```

ce qui produit



— Ligne 12-14 : le secteur est tracé sur la base de l'ellipse qui contient le secteur. on appelera O le centre de l'ellipse et on reprend les noms de sommets A et B.

— Ligne 13-14 :

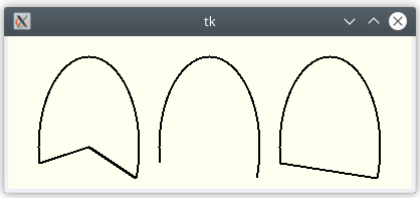
  — option `extent` : la totalité de l'angle (OA, OB), en degrés ;

  — option `start` : l'angle en degrés (u, OA) où u représente une demi-droite horizontale dirigée vers l'Est et A le point de départ sur l'ellipse ;

Les trois styles possibles de fermeture du secteur sont :

```
from tkinter import *

WIDTH=400
HEIGHT=150

root = Tk()
cnv = Canvas(root, width=WIDTH, height=HEIGHT, background="ivory")
cnv.pack()

x=30
y=20
COTEx=100
COTEy=180

cnv.create_arc(x,y, x+COTEx, y+COTEy, extent=210, start=-20, width=2)
x+=20+COTEx
cnv.create_arc(x,y, x+COTEx, y+COTEy, extent=210, start=-20, width=2,
               style=ARC)

x+=20+COTEx
cnv.create_arc(x,y, x+COTEx, y+COTEy, extent=210, start=-20, width=2,
               style=CHORD)

root.mainloop()
```

ce qui produit



Bord d'un cercle

Un cercle possède par défaut un bord noir d'une épaisseur `width` de 1 pixel. La question se pose de savoir

```
cnv.create_line(a, b, a+2*R, b+2*R)
```

s'étend sur 2R+1 pixels, coupant la verticale numérotée a et la verticale numérotée a+2*R. Si l'option `width` est placée à 0 alors le cercle ne possède plus de bord visible et est situé strictement entre les deux droites. Si `width` est mise à 2 alors, de chaque côté, on a un pixel à l'intérieur et 1 pixel sur les droites. Ensuite, ça se répartit équitablement entre l'intérieur et l'extérieur.

Pour remplir, option `fill`, par exemple `fill='red'`.

```
from tkinter import Tk, Canvas

R=400
x=20
SIDE=2*(R+x)

root=Tk()
cnv=Canvas(root, width=SIDE, height=SIDE, bg="ivory")
cnv.pack()

u=x
v=x


# le bord du cercle passe sur les droites
cnv.create_line(u,0,u, SIDE)
cnv.create_line(u+2*R,0,u+2*R, SIDE)
cnv.create_oval(u,v,u+2*R, v+2*R, fill='green', outline='red', width=2)

root.mainloop()
```

Ci-dessous un cercle de centre le point (100, 100) et de rayon 5. Bords compris, le diamètre du cercle s'étend sur 11 pixels :

— le centre : 1 pixel

— le diamètre intérieur (hors centre) : 4 pixels

— le bord : 1 pixel

Les bords du cercle sont sur les droites d'équations x=xC+R, x=xC-R. Si on passe l'option `fill=''` alors la seule différence est que le bord est supprimé.

```
from tkinter import *

WIDTH=200
HEIGHT=200

root = Tk()
cnv = Canvas(root, width=WIDTH, height=HEIGHT, background="ivory")
cnv.pack()

xC, yC= WIDTH//2, HEIGHT//2
R=5

# cercle de centre C et de rayon R
cnv.create_rectangle(xC, yC, xC, yC, fill='black', outline='')

cnv.create_oval(xC-R,yC-R,xC+R, yC+R,  outline='red')

root.mainloop()
```

Les deux points donnés à la construction sont les extrémités de la diagonale d'un rectangle qui contient le disque.

Attention, la boite est à côtés parallèles aux axes ce qui peut poser problème en cas de transformation par rotation..

Dessiner une courbe

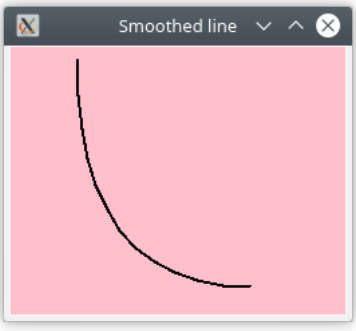
On peut être intéressé par tracer une courbe passant par certains points.

Si la courbe est connue par son équation, on peut utiliser Matplotlib qui s'interface assez bien avec Tkinter.

Voici une solution utilisant des segments, d'après draw a smooth curve :

```
from tkinter import Canvas, Tk
#from tkinter import * # For Python 3.2.3 and higher.
root = Tk()
root.title('Smoothed line')
cw = 250 # canvas width
ch = 200 # canvas height
canvas_1 = Canvas(root, width=cw, height=ch, background="pink")
canvas_1.grid(row=0, column=1)

x1 = 50
y1 = 10
x2 = 50
y2 = 180
x3 = 180
y3 = 180

canvas_1.create_line(x1,y1, x2,y2, x3,y3, smooth="true", width= 2)
root.mainloop()
```

ce qui produit :

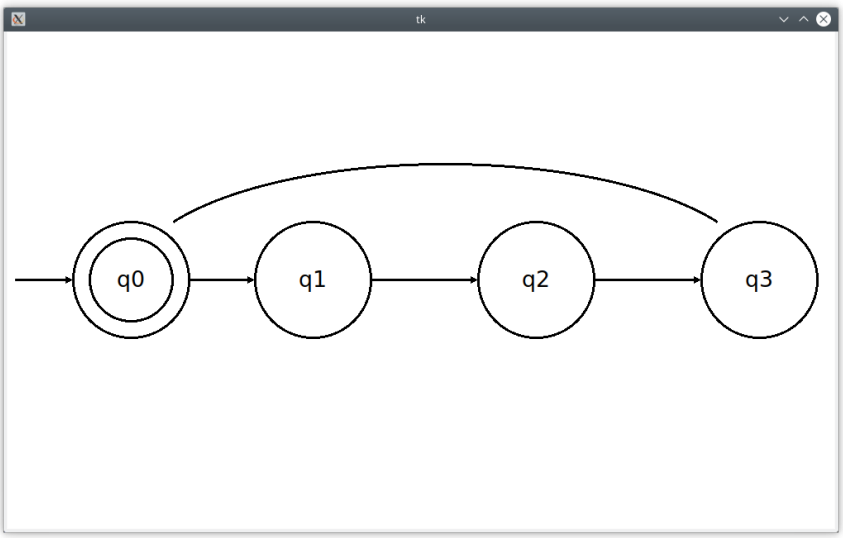


Voici une autre possibilité, utilisant `create_arc` :

```
from tkinter import Canvas, mainloop, Tk

def circle(canvas, x, y, r, width):
    return canvas.create_oval(x+r, y+r, x-r, y-r, width=width)

def circular_arc(canvas, x, y, r, t0, t1, width):
    return canvas.create_arc(x-r, y-r, x+r, y+r, start=t0,
                             extent=t1-t0, style='arc', width=width)

def ellipse(canvas, x, y, r1, r2, width):
    return canvas.create_oval(x+r1, y+r2, x-r1, y-r2, width=width)

def elliptical_arc(canvas, x, y, r1, r2, t0, t1, width):
    return canvas.create_arc(x-r1, y-r2, x+r1, y+r2, start=t0,
                             extent=t1-t0, style='arc', width=width)

def line(canvas, x1, y1, x2, y2, width, start_arrow=0, end_arrow=0):
    arrow_opts = start_arrow << 1 | end_arrow
    arrows = {0b10: 'first', 0b01: 'last', 0b11: 'both'}.get(
        arrow_opts, None)
    return canvas.create_line(x1, y1, x2, y2, width=width,
                              arrow=arrows)

def text(canvas, x, y, text):
    return canvas.create_text(x, y, text=text, font=('bold', 20))


w = Canvas(Tk(), width=1000, height=600, bg='white')

circle(w, 150, 300, 70, 3)  # q0 outer edge
circle(w, 150, 300, 50, 3)  # q0 inner edge
circle(w, 370, 300, 70, 3)  # q1
circle(w, 640, 300, 70, 3)  # q2
circle(w, 910, 300, 70, 3)  # q3

# Draw arc from circle q3 to q0.
midx, midy = (150+910) / 2, 300
r1, r2 = 910-midx, 70+70
elliptical_arc(w, midx, midy, r1, r2, 30, 180-30, 3)

line(w,  10, 300,  80, 300, 3, end_arrow=1)
line(w, 220, 300, 300, 300, 3, end_arrow=1)
line(w, 440, 300, 570, 300, 3, end_arrow=1)
line(w, 710, 300, 840, 300, 3, end_arrow=1)

text(w, 150, 300, 'q0')
text(w, 370, 300, 'q1')
text(w, 640, 300, 'q2')
```

```
text(w, 910, 300, 'q3')

w.pack()
mainloop()
```

ce qui produit

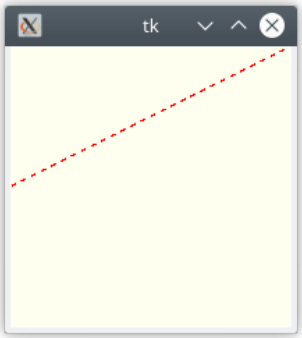


Dessiner une ligne en pointillé

Il s'agit plutôt de tirets que de pointillés. Il faut utiliser l'option dash. L'exemple ci-dessous provient du site de Fredrick lundh :

```
1 from tkinter import Tk, Canvas
2
3 root = Tk()
4 cnv = Canvas(root, width=200, height=200, background="ivory")
5 cnv.pack()
6
7 cnv.create_line(0, 100, 200, 0, fill="red", dash=(4, 4))
8
9 root.mainloop()
```

qui affiche :



Dessiner un pixel invisible

Créer un pixel caché, cela peut-être utile pour déboguer un code : le pixel n'est pas visible mais l'item existe sur le canevas, il a donc une id :

```
from tkinter import *
from random import randrange, sample

WIDTH=300
HEIGHT=200

root = Tk()
cnv = Canvas(root, width=WIDTH, height=HEIGHT, background="ivory")
cnv.pack()

point=cnv.create_polygon(50,50, fill='', outline='')
print(cnv.coords(point))

cnv.move(point, 100, 100)
print(cnv.coords(point))

L=cnv.coords(point)
root.mainloop()
```

qui affiche :

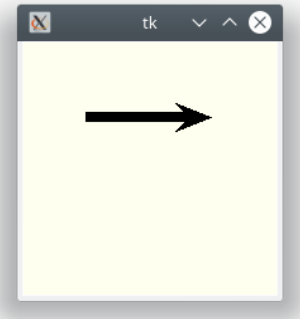
```
[50.0, 50.0]
[150.0, 150.0]
```

Modifier le profil de la flèche

On souhaite parfois modifier l'allure de la pointe d'une flèche. C'est possible avec l'option `arrowshape` :

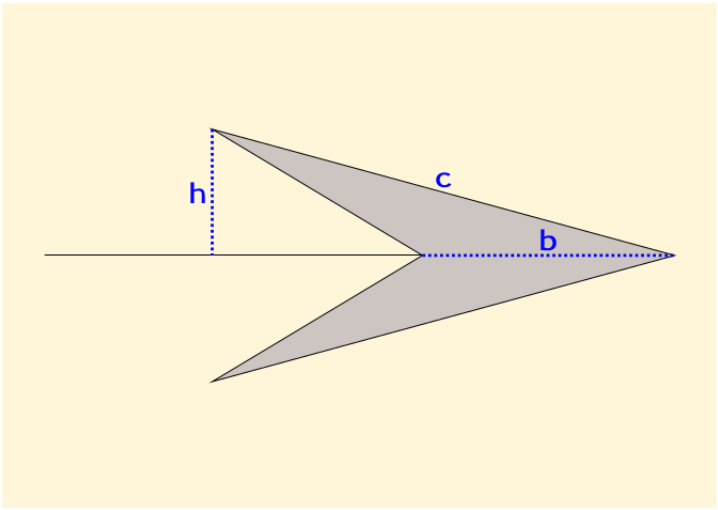
```
from tkinter import Tk, Canvas

```

```
root=Tk()
cnv=Canvas(root, width=200, height=200, bg="ivory")
cnv.pack()

cnv.create_line(50, 60,150, 60, width=8, arrow='last', arrowshape=(18,30, 8))
root.mainloop()
```



Voici la description de l'option `arrowshape=(h, b, c)` :

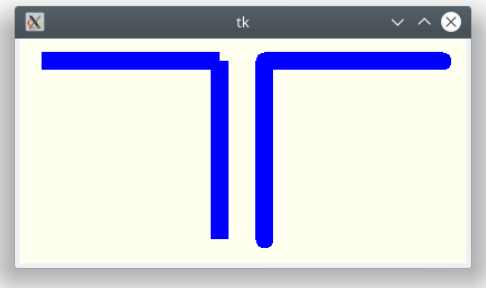
— `h` est la distance à l'axe de l'aile de la flèche,
— `b` est la largeur de la base de la flèche (la partie qui est sur l'axe),
— `c` est la longueur de la partie extérieure de la flèche.



Extrémité, jonction de segments

L'option `capstyle` de `create_line` permet de gérer le raccordement de segments, en particulier de l'arrondir :

```
from tkinter import Tk, Canvas, ROUND
WIDTH=400
HEIGHT=200

root = Tk()
cnv = Canvas(root, width=WIDTH, height=HEIGHT, background="ivory")
cnv.pack()

cnv.create_line(20, 20, 180, 20, fill='blue', width=16)
cnv.create_line(180, 20, 180, 180, fill='blue', width=16)

cnv.create_line(380, 20, 220, 20, fill='blue', width=16, capstyle=ROUND)
cnv.create_line(220, 20, 220, 180, fill='blue', width=16, capstyle=ROUND)

root.mainloop()
```



Voir la documentation pour d'autres possibilités.

Image vs rectangle : positionnement

On fera attention que pour la méthode `create_image`, les arguments de dimensions n'ont pas même signification que pour, par exemple, `create_rectangle`. Ainsi,

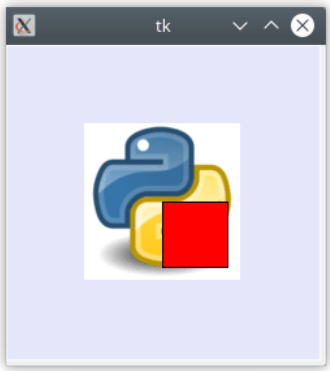
— `cnv.create_image(a, a, image=test)` va afficher le **centre** de l'image au point `(a, a)` ;
— `cnv.create_rectangle(a, a, a+50, a+50, fill="red")` va afficher le coin supérieur gauche du rectangle au point `(a, a)`.

Ainsi le code suivant :

```
from tkinter import *

side=120
root = Tk()
cnv = Canvas(root, width=2*side, height=2*side, bg="lavender")
cnv.pack()

a=side
```

```
test = PhotoImage(file="python.gif")
cnv.create_image(a, a, image=test)
cnv.create_rectangle(a, a, a+50, a+50, fill="red")

root.mainloop()
```

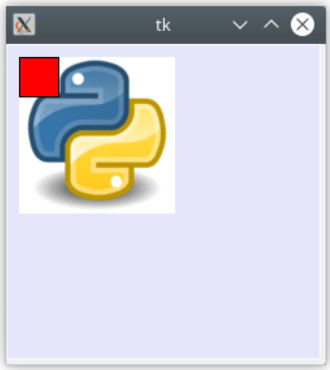
affiche-t-il



On peut remédier à cela en donnant l'argument nommé `anchor=nw` à `create_image` :

```
from tkinter import *

side=120
root = Tk()
cnv = Canvas(root, width=2*side, height=2*side, bg="lavender")
cnv.pack()

a=side
test = PhotoImage(file="python.gif")
cnv.create_image(a, a, image=test, anchor=NW)
cnv.create_rectangle(a, a, a+50, a+50, fill="red")

root.mainloop()
```

qui affiche :



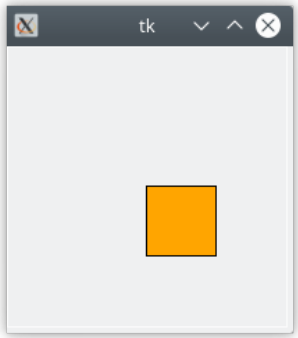
On voit alors que les coins supérieurs gauches du carré et de l'image, cette fois, coïncident.

Image qui n'apparaît pas

Cette section traite d'un souci très fréquemment rencontré par les utilisateurs du canevas. Les items de type PhotoImage souffrent d'une anomalie très curieuse, illustrée par le code suivant (il faut disposer dans le répertoire courant d'une image python.gif qui est un logo du langage Python)

```
1  from tkinter import *
2
3  def run():
4      root = Tk()
5      cnv = Canvas(root, width=200, height=200)
6      cnv.pack()
7      photo(cnv)
8      root.mainloop()
9
10 def photo(cnv):
11     test =PhotoImage(file="python.gif")
12     cnv.create_image(100, 100, image=test)
13     cnv.create_rectangle(100, 100, 150,150,fill="orange")
14
15 run()
```

qui affiche



Quand le code ci-dessus s'exécute, l'image de la ligne 12 reste invisible tandis que le rectangle à la ligne 13 lui est bien visible. Pourquoi ? Parce que `test` est une variable locale à la fonction `photo` et qu'elle est ensuite éliminée par le garbage collector de Python. En réalité, c'est peut-être même plus compliqué puisque le code suivant ne contient aucune variable vers `PhotoImage(file="python.gif")` et pourtant l'image n'est pas affichée :

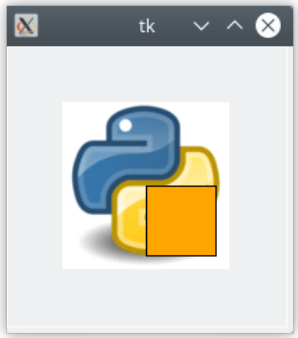
```
1  from tkinter import *
2
3  def run():
4      root = Tk()
5      cnv = Canvas(root, width=200, height=200)
6      cnv.pack()
7      photo(cnv)
8      root.mainloop()
9
10 def photo(cnv):
11     cnv.create_image(100, 100, image=PhotoImage(file="python.gif"))
12     cnv.create_rectangle(100, 100, 150,150,fill="orange")
13
14 run()
```

Pour y remédier, il suffit qu'un widget Tkinter qui contient l'image en garde une référence, par exemple en définissant l'image comme attribut du canevas, ligne 12 ci-dessous :

```
1  from tkinter import *
2
3  def run():
4      root = Tk()
5      cnv = Canvas(root, width=200, height=200)
6      cnv.pack()
7      photo(cnv)
8      root.mainloop()
9
10 def photo(cnv):
11     test =PhotoImage(file="python.gif")
12     cnv.test=test
13     cnv.create_image(100, 100, image=test)
```

```
    cnv.create_rectangle(100, 100, 150,150,fill="orange")

run()
```

qui affiche



Cette anomalie est bien connue, par exemple voir le message de furas sur StackOverFlow ou le site de Fredrik Lunth. Voir aussi cette partie de la documentation officielle et qui explique ainsi le comportement : *Tk will not keep a reference to the image.*

2 Les items et les tags

Identifiant d'items du canevas

Tout objet placé sur un canevas Tkinter est créé avec une méthode du canevas dont le nom commence par `create`, comme `create_rectangle`. Lors de la création, chaque objet reçoit une id unique. Cette id est un identifiant entier strictement positif. Il permet de garder trace de l'objet créé et de le modifier dynamiquement.

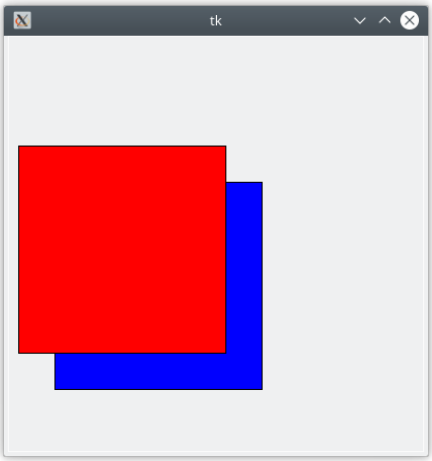
Exemple :

```
from tkinter import *
from random import randrange

SIDE=400
root = Tk()
cnv = Canvas(root, width=SIDE, height=SIDE)
cnv.pack()

t=SIDE//2
xA, yA=A=(randrange(t),randrange(t))
xB, yB=B=(randrange(t),randrange(t))

rect1=cnv.create_rectangle(A, (xA+t,yA+t), fill='blue')
rect2=cnv.create_rectangle(B, (xB+t,yB+t), fill='red')

```

```
16 print(rect1)
17 print(rect2)
18
19 root.mainloop()
```

```
20 1
21 2
```

et qui produit :



Dans ce code, on fait apparaître, à des endroits aléatoires (lignes 10 et 11 les points A et B), deux rectangles sur le canevas. L'appel à `cnv.create_rectangle` (lignes 16 et 17) est récupéré dans une variable et qui contient alors l'id du rectangle créé. Les ids sont alors affichées (ligne 16 et 17). On remarque (lignes 20-21) que les id ne sont pas des entiers aléatoires mais qu'il s'agit d'entiers consécutifs croissants.

Identifiant d'image

Tout image placée sur un canevas Tkinter et créée avec la méthode du canevas `create_image`, lors de la création, reçoit une id unique. Cette id est un identifiant entier strictement positif. Il permet de garder trace de l'image créée.

Exemple :

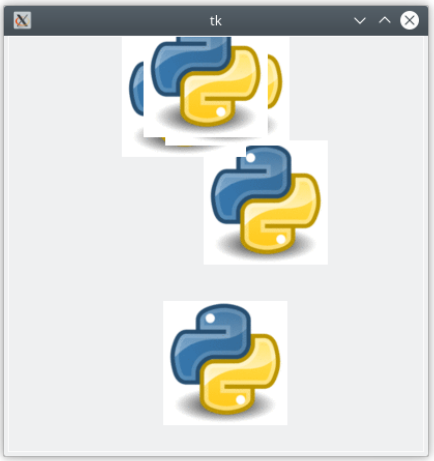
`id_images.py`

```
from tkinter import *
from random import randrange

SIDE=400
root = Tk()
cnv = Canvas(root, width=SIDE, height=SIDE)
cnv.pack()

logo = PhotoImage(file="python.gif")
```

```
for i in range(5):
    x, y= randrange(SIDE),randrange(SIDE)
    id_image=cnv.create_image(x, y, image=logo)
    print(id_image)

root.mainloop()
```

```
1
2
3
4
5
```

qui produit :

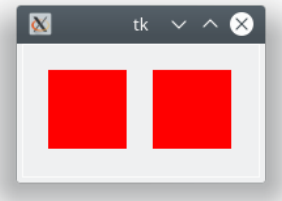


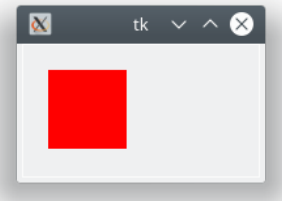
Dans ce code, on fait apparaître, à des endroits aléatoires, 5 images sur le canevas. L'appel à `cnv.create_image` est récupéré dans une variable et qui contient alors l'id de l'image créée. Les ids sont alors affichées. On remarque que les id ne sont pas des entiers aléatoires mais qu'il s'agit d'entiers consécutifs croissants.

La méthode `delete`

Le canevas offre une méthode permettant de supprimer des items présents sur le canevas. Supprimer signifie que les items ne sont plus visibles mais aussi que la mémoire qu'ils occupaient est libérée.

Soit l'animation minimale suivante :





qui crée deux carrés (lignes 7 et 8 ci-dessous) dans un canevas et, au bout de deux secondes, fait disparaître un des deux carrés. Voici le code :

```
1 from tkinter import Tk, Canvas
2
3 root = Tk()
4 cnv = Canvas(root, width=180, height=100)
5 cnv.pack()
6
7 cnv.create_rectangle(20, 20, 80, 80, fill='red', outline='')
8 rect=cnv.create_rectangle(100, 20, 160, 80, fill='red', outline='')
9
10 def effacer(ident):
11     cnv.delete(ident)
12
13 cnv.after(2000, effacer, rect)
14
15 root.mainloop()
```

C'est à la ligne 11 qu'on observe l'effacement et à la ligne 13 qu'on observe la temporisation d'une seconde.

L'effacement se fait en appelant la méthode `delete` du canevas sur l'identificateur de l'item que l'on souhaite supprimer.

Pour **tout** effacer tous les items présents sur le canevas, utiliser un appel de la forme `Canvas.delete('all')`. C'est une technique fréquente lorsqu'on cherche à rafraichir le plateau d'un jeu (on efface et on redessine).

Le pemier usage de `delete` est lorsqu'on veut faire disparaître visuellement un item, par exemple un projectile qui a atteint sa cible.

Toutefois, il n'y a pas que la disparition visuelle : imaginons un missile qui se déplace puis sort de la zone du canevas ; il n'est plus visible du joueur mais est toujours référencé comme item. Si de tels missiles étaient très nombreux, ils pourraient utiliser inutilement des ressources : calcul (détermination de la nouvelle position du missile) et mémoire (car un item nécessite une allocation mémoire effectué par la bibliothèque TCL/Tk sous-jacente).

Pour déplacer un objet et donc le faire disparaître d'une position pour le faire apparaître à une autre, on n'utilise pas la méthode `delete` mais les méthodes de Canvas du nom de `move` ou de `coords`.

Suppression d'images du canevas

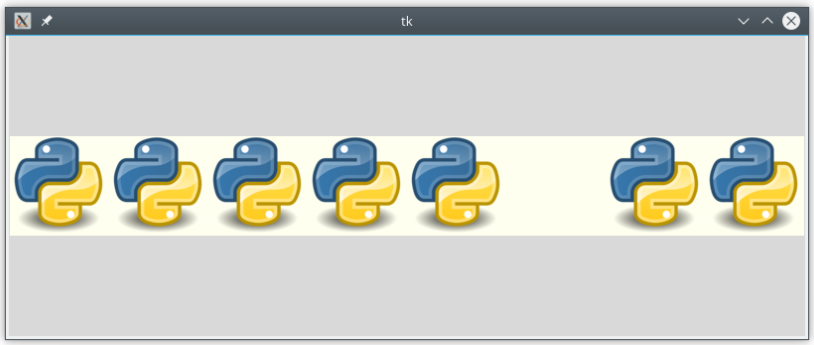
Il est possible de supprimer n'importe quelle image créée sur le canevas. Pour cela, il suffit de connaître l'id de l'image à supprimer et d'utiliser la méthode `delete` du canevas.

Exemple :

suppression_image.py

```
1  from tkinter import *
2  from random import randrange
3
4  NB_IMG=8
5  SIDE=100
6  WIDTH=SIDE*NB_IMG
7  X0=Y0=SIDE//2
8
9  root = Tk()
10 logo = PhotoImage(file="python.gif")
11 cnv = Canvas(root, width=WIDTH, height=SIDE, bg="ivory")
12 cnv.pack(pady=100)
13
14 ids=[]
15 for k in range(NB_IMG):
16     id_image=cnv.create_image(X0+k*SIDE, Y0, image=logo)
17     ids.append(id_image)
18
19 j=randrange(NB_IMG)
20 print(j)
21 mon_id=ids[j]
22
23 cnv.delete(mon_id)
24
25 root.mainloop()
```

```
26 5
```



On utilise une même image `python.gif` carrée de taille `SIDE=100`. Dans la boucle (lignes 15-17) :

— on dessine côte à côte `NB_IMG=8` images

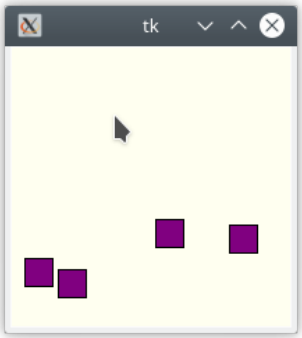
— on mémorise l'id de chaque image dans une liste `ids`

Puis, on choisit au hasard une id d'image (ligne 19 et ligne 20 pour l'affichage de l'indice) et on supprime du canevas l'objet ayant cet id (ligne 23). Comme on le voit sur la copie d'écran, il y a bien un trou dans l'alignement des images à l'indice choisi.

La liste de tous les items du canevas

Il est possible de récupérer toutes les id des objets créés sur un canevas (disons `cnv`). Cette liste s'obtient en appelant la méthode `cnv.find_all`.

Ci-dessous, on tire un entier aléatoire `N` entre 1 et 10 et on dessine sur le canevas `N` carrés aléatoires. Puis on demande l'affichage de tous les items :




```
1 from tkinter import *
2 from random import randrange
3
4 WIDTH=200
5 HEIGHT=200
6
```

```
7  COTE=20
8
9  root = Tk()
10 cnv = Canvas(root, width=WIDTH, height=HEIGHT, background="ivory")
11 cnv.pack()
12
13 N=randrange(10)
14 print(N)
15
16 for _ in range(N):
17     x=randrange(WIDTH)
18     y=randrange(HEIGHT)
19     cnv.create_rectangle(x, y, x+COTE, y+COTE, fill="purple")
20
21 print(cnv.find_all())
22
23 root.mainloop()
```

```
24 4
25 (1, 2, 3, 4)
```

— Lignes 13-14 et 24 : le nombre aléatoire de carrés à dessiner.
— Lignes 16-19 : le dessin des carrés
— Lignes 21 et 25 : la liste de toutes les id des objets présents sur le canevas. ici, il n'y a que les `N` carrés construits.

La génération d'identifiants d'items du canevas

Les identifiants des items générés d'un canevas donné sont uniques, y compris si on supprime des items. Les identifiants générés sont des entiers consécutifs croissants à partir de 1.

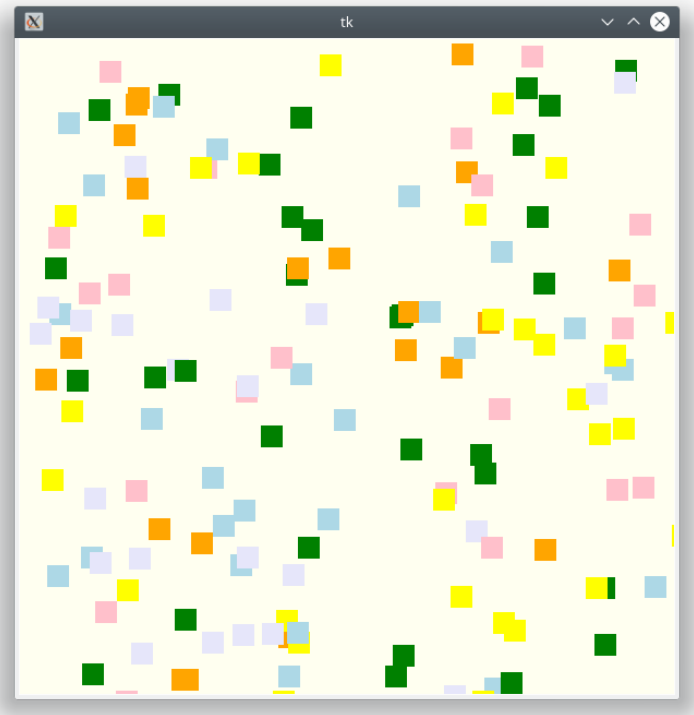
Cela signifie que par exemple les identifiants 1, 2, 3, etc sont générés. Si l'item d'identifiant par exemple 42 est supprimé, l'identifiant 42 ne sera pas réutilisé.

Pour observer le phénomène, soit le programme ci-dessous :

```
1  from tkinter import *
2  from random import randrange, sample
3
4  COLORS=["orange", "pink", "green", "yellow", "lavender", "lightblue"]
5  NCOLORS=len(COLORS)
6
7  WIDTH=600
8  HEIGHT=600
9  COTE=20
10
11 root = Tk()
12 cnv = Canvas(root, width=WIDTH, height=HEIGHT, background="ivory")
13 cnv.pack()
14
15 N=100
```

```
16
17 items_ids=[]
18
19 for i in range(N):
20     x=randrange(WIDTH)
21     y=randrange(HEIGHT)
22     color=COLORS[randrange(NCOLORS)]
23     rect=cnv.create_rectangle(x, y, x+COTE, y+COTE, fill=color, outline="")
24     items_ids.append(rect)
25
26 print(items_ids==list(range(1, N+1)))
27
28
29 M=sample(items_ids, N//2)
30
31 for m in M:
32     cnv.delete(m)
33
34 new_items_ids=[]
35
36 for i in range(N):
37     x=randrange(WIDTH)
38     y=randrange(HEIGHT)
39     color=COLORS[randrange(NCOLORS)]
40     rect=cnv.create_rectangle(x, y, x+COTE, y+COTE, fill=color, outline="")
41     new_items_ids.append(rect)
42
43 print(new_items_ids==list(range(N+1, 2*N+1)))
44
45 root.mainloop()
```

qui affiche ceci :



Le programme génère 100 carrés placés et colorés aléatoirement (lignes 23 et 22) dont les id sont stockés dans une liste (ligne 24). Ensuite, on vérifie (ligne 26) que les id générés sont les entiers de 1 à 100, dans cet ordre.

Puis on supprime (à la fois de la mémoire du canevas et visuellement) au hasard la moitié des items (ligne 29). Et on recrée 100 items dont les id sont stockés dans une liste (ligne 41).

On vérifie alors (ligne 43) que les id générés sont dans l'ordre, les entiers de 101 à 200.

Lire les options d'un item du canevas

La méthode `itemconfigure` du canevas permet de lire les options d'un item :

```
1 om tkinter import Tk, Canvas
2
3 root = Tk()
4 cnv = Canvas(root, width=300, height=200)
5 cnv.pack()
6
7 rect=cnv.create_rectangle(30, 30, 130, 130, fill="red", outline='green')
8
9 options=cnv.itemconfigure(rect)
10 print(*options.items(), sep='\n')
11
12 root.mainloop()
```

```
13 ('activedash', ('activedash', '', '', '', ''))
14 ('activefill', ('activefill', '', '', '', ''))
15 ('activeoutline', ('activeoutline', '', '', '', ''))
16 ('activeoutlinestipple', ('activeoutlinestipple', '', '', '', ''))
17 ('activestipple', ('activestipple', '', '', '', ''))
18 ('activewidth', ('activewidth', '', '', '0.0', '0.0'))
19 ('dash', ('dash', '', '', '', ''))
20 ('dashoffset', ('dashoffset', '', '', '0', '0'))
21 ('disableddash', ('disableddash', '', '', '', ''))
22 ('disabledfill', ('disabledfill', '', '', '', ''))
23 ('disabledoutline', ('disabledoutline', '', '', '', ''))
24 ('disabledoutlinestipple', ('disabledoutlinestipple', '', '', '', ''))
25 ('disabledstipple', ('disabledstipple', '', '', '', ''))
26 ('disabledwidth', ('disabledwidth', '', '', '0.0', '0'))
27 ('fill', ('fill', '', '', '', 'red'))
28 ('offset', ('offset', '', '', '0,0', '0,0'))
29 ('outline', ('outline', '', '', 'black', 'green'))
30 ('outlineoffset', ('outlineoffset', '', '', '0,0', '0,0'))
31 ('outlinestipple', ('outlinestipple', '', '', '', ''))
32 ('state', ('state', '', '', '', ''))
33 ('stipple', ('stipple', '', '', '', ''))
34 ('tags', ('tags', '', '', '', ''))
35 ('width', ('width', '', '', '1.0', '1.0'))
```

— Ligne 9 : les options du rectangle (ligne 7) apparaissent comme un dictionnaire.

— Lignes 10 et 13-35 : les couples (clé, valeur) du dictionnaire sont affichés grâce à la méthode `items`. Par exemple, on lit la couleur de remplissage ligne 27.

La méthode `itemcget` permet de lire une option particulière (lignes 9 et 13) :

```
1 from tkinter import Tk, Canvas
2
3 root = Tk()
4 cnv = Canvas(root, width=300, height=200)
5 cnv.pack()
6
7 rect=cnv.create_rectangle(30, 30, 130, 130, fill="red", outline='green')
```

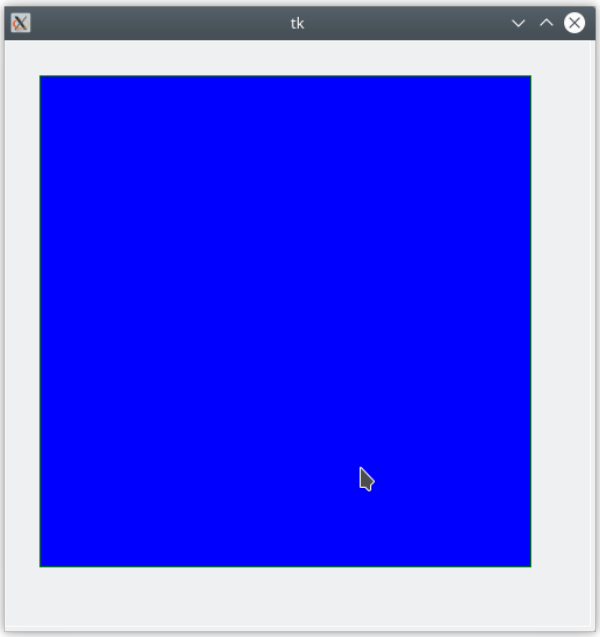
```
8
9 color=cnv.itemcget(rect, 'fill')
10 print(color)
11
12 root.mainloop()
```

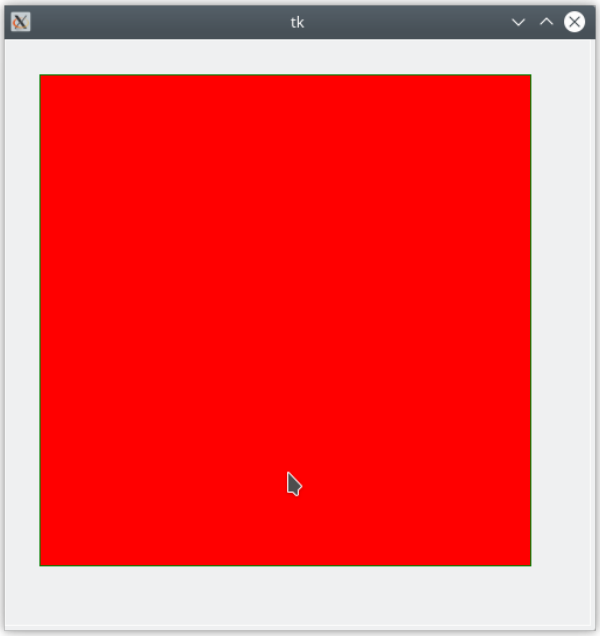
```
13 red
```

Modifier les options d'un item du canevas

Parfois, on cherche à modifier les propriétés d'un item, par exemple changer du texte, changer la couleur de fond ou les dimensions sans pour autant recréer complètement l'item. La méthode `itemconfigure` (ou `itemconfig`) qui permettait déjà de lire les options permet aussi de les modifier.

Le programme visible ci-dessous :





crée d'abord un carré rouge dans un canevas puis, au bout de deux secondes, change en bleu la couleur du carré :

Le code correspondant :

```
1 from tkinter import Tk, Canvas
2
3 def action():
4     cnv.itemconfigure(rect, fill="lavender")
5
6
7 root = Tk()
8 cnv = Canvas(root, width=500, height=500)
9 cnv.pack()
10
11 rect=cnv.create_rectangle(30, 30, 450, 450, fill="pink", outline='green')
12
13
14 cnv.after(1000, action)
15 root.mainloop()
```

La modification de la couleur est lancée par la ligne 14 et est effectuée ligne 4 où l'option `fill` est passée à `"lavender"` sur l'item du rectangle par la méthode `itemconfigure`.

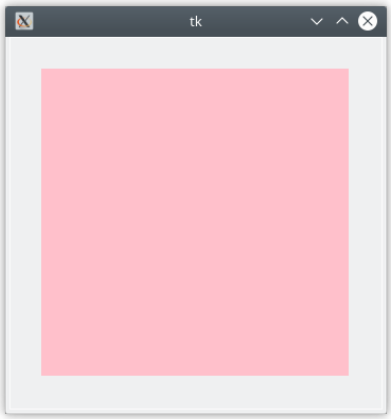
Noter qu'on ne peut pas modifier ainsi la position de l'item sur le canevas puisque `itemconfigure` donne seulement accès aux **options ayant un nom** comme `fill`. Pour déplacer ou agrandir un item, utiliser les méthodes `move` ou `coords`.

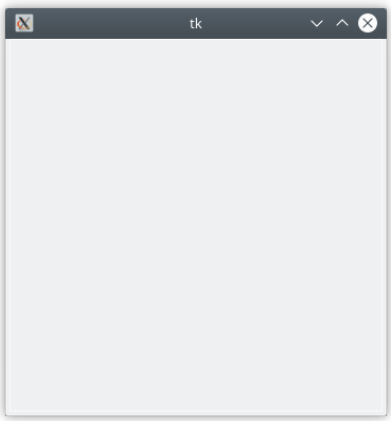
Cacher/montrer des items sur le canevas

On peut avoir besoin de faire disparaître *provisoirement* un item présent sur le canevas, par exemple un rectangle. Il serait possible de l'effacer avec la méthode `delete` du canevas puis de le recréer quand on en a à nouveau besoin. En réalité, il est possible de le faire disparaître et de le faire réapparaître sans le détruire ni le reconstruire. Il suffit pour cela d'utiliser l'option `state` de l'item et qui prend deux valeurs :

— `hidden` : pour cacher l'item
— `normal` : pour rendre visible l'item.

Pour modifier cette option, il suffit d'utiliser la méthode du canevas `itemconfigure`. Voici un exemple :






```
1 from tkinter import Tk, Canvas
2
3 def hide_my_rect():
4     cnv.itemconfigure(rect, state="hidden")
5
6 def show_my_rect():
```

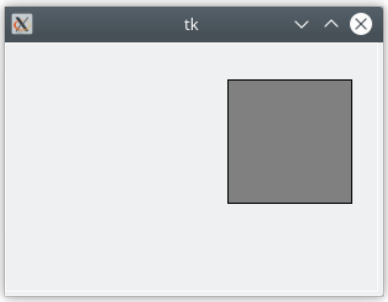
```
    cnv.itemconfigure(rect, state="normal")

root = Tk()
cnv = Canvas(root, width=350, height=350)
cnv.pack()

rect=cnv.create_rectangle(30, 30, 270, 270, fill="red")

cnv.after(1500, hide_my_rect)
cnv.after(3000, show_my_rect)

root.mainloop()
```

Le canevas contient un rectangle rouge. Au bout d'une seconde et demi, le rectangle est caché. Une seconde et demi après, le rectangle réapparaît.

L'options `state` peut aussi être utile si on veut créer avant exécution un certain nombre d'items qu'on aura plus qu'à afficher le moment venu. Dans un jeu , il est souvent préférable de créer des objets à l'initialisation du jeu plutôt qu'en cours de jeu car la création d'objets peut être source de ralentissments.

Déplacer un item avec la méthode `move`

On peut déplacer un item du canevas en utilisant la méthode `move` :



Pour cela on suppose que l'on connait le vecteur de déplacement. Voici un exemple :

```
1 from tkinter import Tk, Canvas
2
3 root = Tk()
4 cnv = Canvas(root, width=300, height=200)
5 cnv.pack()
6
7 rect=cnv.create_rectangle(30, 30, 130, 130, fill='gray')
8
9 def deplacer(x):
10     cnv.move(rect, x, 0)
11
12 cnv.after(1000, deplacer, 150)
13
14 root.mainloop()
```

— Ligne 7 : le rectangle a une id appelée `rect`

— Ligne 10 : le déplacement se fait en utilisant de l'id de l'item. Le vecteur de déplacement est `(x, 0)` et à l'appel, `x` vaudra 150 (cf. ligne 12).

— Ligne 12 : le déplacement se fait une seconde après l'ouverture de la fenêtre.

— Ligne 10 : le déplacement est instantané, c'est plutôt une translation qu'un déplacement.

Variante

Ce qui suit est d'importance secondaire, en particulier si vous êtes débutant. Bien comprendre qu'un déplacement d'un item est en fait une **translation** d'un item. Et une translation dépend d'un **vecteur**. Si vous voulez bouger un item du point $A = (x_A, y_A)$ vers le point $B = (x_B, y_B)$ alors la translation est de vecteur $\vec{AB} = (x_B - x_A, y_B - y_A)$ (pensez à la formule « extrémité moins origine »). Donc pour pouvoir bouger un item sur le canevas, au lieu d'écrire

```
v=(xB - xA, yB - yA)
cnv.move(item, v[0], v[1])
```

qui n'est pas très parlant, on aimerait pouvoir écrire le vecteur directement :

```
v=(xB - xA, yB - yA)
cnv.move(item, v)
```

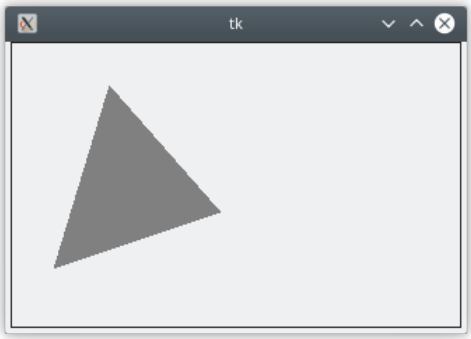
Mais ce code ne sera pas accepté car la méthode `move` attend 3 arguments et pas 2. On peut toutefois écrire un code assez proche :

```
v=(xB - xA, yB - yA)
cnv.move(item, *v)
```

En effet, l'« opérateur » `*` en Python permet de décompacter les éléments d'un tuple (comme `v` ci-dessus) et de placer ces éléments comme arguments d'un appel de fonction.

Déplacer un item avec la méthode `coords`

On peut déplacer un item du canevas en utilisant la méthode coords :



Cela suppose que l'on connaisse la position finale de l'objet. Voici un exemple avec pour objet un triangle :

```
1  from tkinter import Tk, Canvas
2
3  root = Tk()
4  cnv = Canvas(root, width=300, height=200)
5  cnv.pack()
6
7  A=(70, 30)
8  B=(30, 160)
9  C=(150, 120)
10
11 tr=cnv.create_polygon(A, B, C, fill='gray')
12 print(cnv.coords(tr))
13
14 def deplacer(x, y):
15     cnv.coords(tr, A[0]+x, A[1]+y,  B[0]+x, B[1]+y, C[0]+x, C[1]+y)
16
17 cnv.after(1000, deplacer, 150,50)
18
19 root.mainloop()
```

— Ligne 11 : le triangle a une id appelée `tr`

— Ligne 12 : les coordonnées des sommets du triangle.
— Ligne 15 : le déplacement se fait à partir de l'id de l'item. La position finale des 3 sommets du triangle est indiquée dans l'appel. Ici, on a décalé chaque abscisse de `x` et chaque ordonnée de `y`.
— Ligne 17 : le déplacement se fait une seconde après l'ouverture de la fenêtre.
— Ligne 15 : le déplacement est instantané, c'est plutôt une translation qu'un déplacement.
— Lignes 12 et 15 : on notera les deux rôles possibles de `coords` (informer ou déplacer).

Contour d'un item

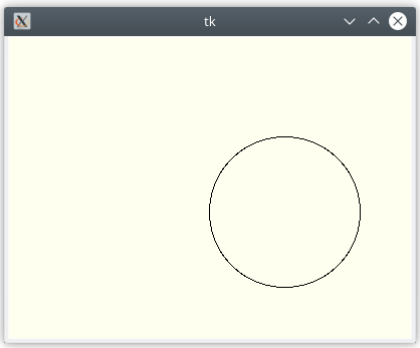
On a parfois besoin de connaître la position précise d'un item sur le canevas, surtout si cet item est mobile.

Un appel du type `cnv.coords(item)` renvoie une liste de coordonnées de points qui entourent l'item. La liste renvoyée dépend du type d'item. Dans le cas d'un item de type rectangle, ellipse, ligne, ou une image, une liste `[a, b, c, d]` de 4 nombres flottants est renvoyée et qui correspond aux coordonnées des extrémités d'une diagonale du rectangle qui entoure l'item ; plus précisément, `(a, b)` est le coin supérieur gauche et `(c, d)` est le coin inférieur droit. Voici un exemple d'utilisation :

```
1  from tkinter import *
2
3  WIDTH=400
4  HEIGHT=300
5
6  root = Tk()
7  cnv = Canvas(root, width=WIDTH, height=HEIGHT, background="ivory")
8  cnv.pack()
9
10 ellipse= cnv.create_oval(200, 250, 350, 100)
11
12 print(cnv.coords(ellipse))
13
14 root.mainloop()
```

```
15 [200.0, 100.0, 350.0, 250.0]
```

et qui produit :



Pour un segment, ce sera les coordonnées des deux extrémités. Si l'item est du texte, ce sera les coordonnées du centre du texte. Si l'item est un arc, l'appel `cnv.coords(item)` renvoie les sommets du rectangle ayant servi de support à la création de l'arc. Si l'item est un polygone, l'appel `cnv.coords(item)` renvoie les sommets du polygone :

```
1  from tkinter import *
2
3  WIDTH=400
4  HEIGHT=300
5
6  root = Tk()
7  cnv = Canvas(root, width=WIDTH, height=HEIGHT, background="ivory")
8  cnv.pack()
9
10 triangle = cnv.create_polygon(50,50, 300, 250, 390, 10)
11
12 print(cnv.coords(triangle))
13
14 root.mainloop()
```

```
15 [50.0, 50.0, 300.0, 250.0, 390.0, 10.0]
```

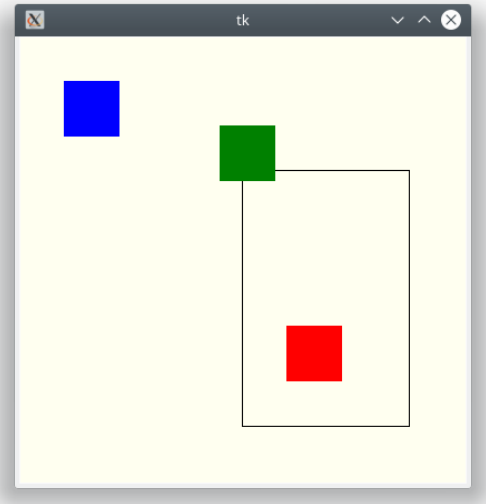
Il est aussi possible d'utiliser la méthode `bbox` (*bounding box*) :

```
1  from tkinter import *
2
3  SIDE=400
4  item=None
5  root=Tk()
6  cnv=Canvas(root, width=SIDE, height=SIDE, background="ivory")
7  cnv.pack()
8
9  item = cnv.create_text(SIDE/2, SIDE/2, text ="2024", font="Arial 100 bold")
10 print(cnv.bbox(item))
11
12 root.mainloop()
```

```
13 (51, 125, 349, 276)
```

Items touchant une zone rectangulaire

La méthode `find_overlapping` permet une forme de détection de collisions (autrement dit, le fait que deux items se touchent). Dans le programme ci-dessous



produit par le code suivant

```
from tkinter import *

WIDTH=400
HEIGHT=400

root = Tk()
cnv = Canvas(root, width=WIDTH, height=HEIGHT, background="ivory")
cnv.pack()

# rectangle de référence
a, b=U=(200, 120)
c, d=V=(350, 350)
cnv.create_rectangle(U, V)

COTE=50

# carré bleu
x=y=40
cnv.create_rectangle(x, y, x+COTE, y+COTE, fill="blue", outline='')

# carré vert
x=180
y=80
```

```
24 cnv.create_rectangle(x, y, x+COTE, y+COTE, fill="green", outline='')
25
26 # carré rouge
27 x=240
28 y=260
29 cnv.create_rectangle(x, y, x+COTE, y+COTE, fill="red", outline='')
30
31 print(cnv.find_all())
32 print(cnv.find_overlapping(a+1, b+1, c-1, d-1))
33
34 root.mainloop()
```

```
35 (1, 2, 3, 4)
36 (3, 4)
```

on crée

— un rectangle de référence, de diagonale `AB` où `A=(a, b)` et `B=(c,d)`, non coloré et recevant l'id numéro 1 (ligne 13)

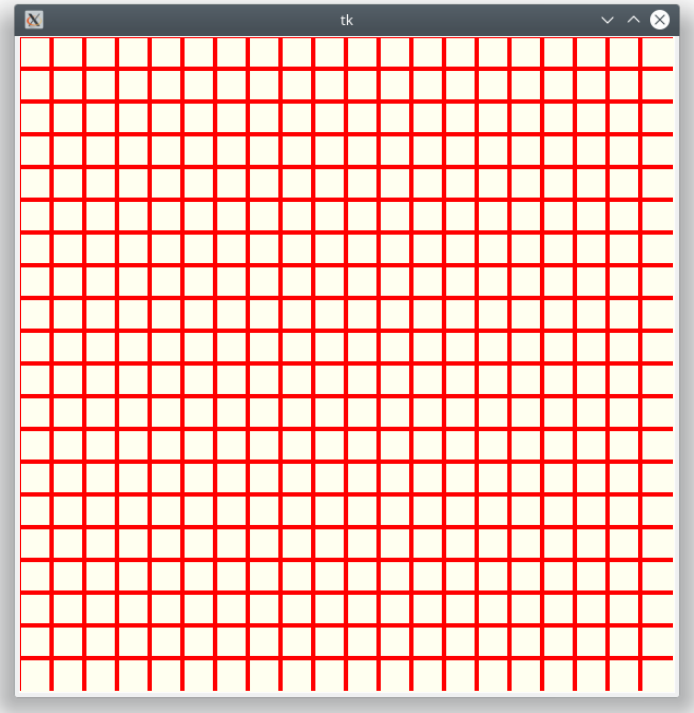
— trois carrés de couleur d'id 2, 3 et 4 (cf. lignes 17-29 et lignes 31 et 35 pour l'affichage des id)

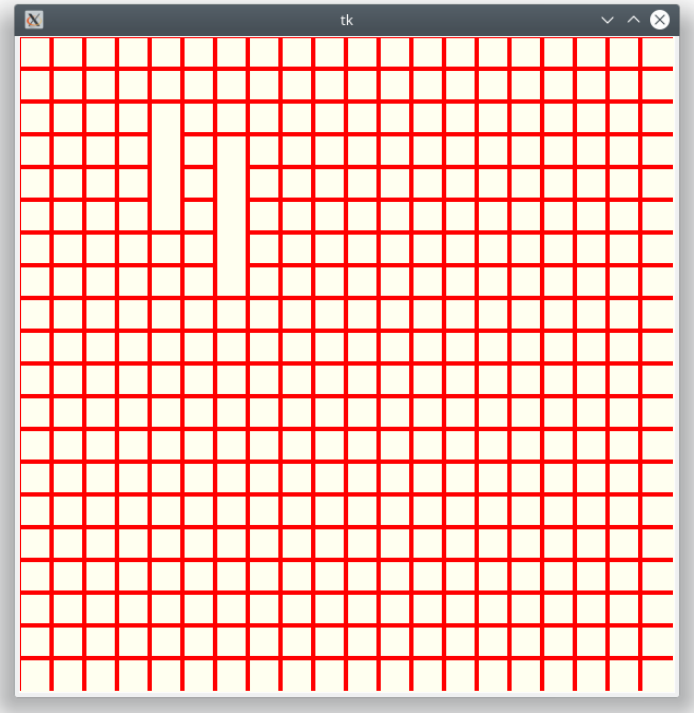
et on demande à la méthode `find_overlapping` (ligne 32) de donner les items du canevas qui touchent l'intérieur du rectangle. La figure montre bien que le carré rouge et le carré vert touchent l'intérieur du rectangle mais que c'est pas le cas du bleu. Et la méthode `find_overlapping` renvoie les id 3 et 4 correspondant au vert et au rouge (ligne 36).

Noter que `cnv.find_overlapping` ne renvoie pas l'id 1 du rectangle de référence car en écrivant `a+1, b+1, c-1, d-1` à la ligne 32, on est garanti, à cause de `+1` et `-1`, que la zone est **strictement** à l'intérieur du rectange de diagonale `AB`.

Capture de l'item le plus proche

Le canevas dispose d'une méthode permettant de connaître l'id de l'item placé sur le canevas et le plus proche d'un point donné. Par exemple, soit le programme suivant où on peut défaire à la souris les segments d'une grille :





Le code correspondant est :

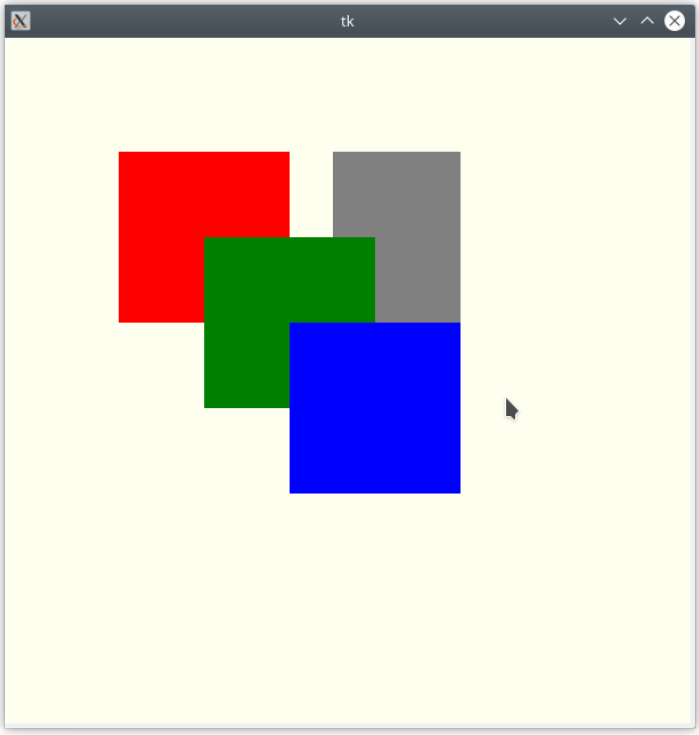
```
1  from tkinter import *
2
3  COTE=30
4  WIDTH=20*COTE
5  HEIGHT=20*COTE
6
7  root = Tk()
8  cnv = Canvas(root, width=WIDTH, height=HEIGHT, background="ivory")
9  cnv.pack()
10
11 n=WIDTH//COTE
12
13 for i in range(n):
14     for j in range(n):
15         cnv.create_line(i*COTE, j*COTE, i*COTE + COTE, j*COTE,
16                         fill='red', width=4)
17         cnv.create_line(j*COTE, i*COTE,  j*COTE, i*COTE + COTE,
18                         fill='red', width=4)
19
20 def suppr(event):
21     clic=event.x, event.y
22     bord = cnv.find_closest(*clic)
23     cnv.delete(bord)
24
25 cnv.bind("<Button-1>",suppr)
26
27 root.mainloop()
```

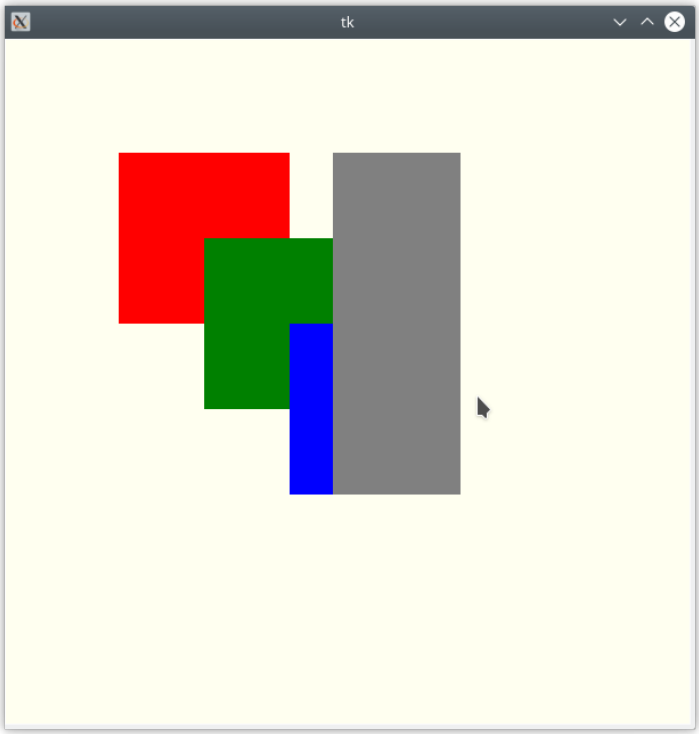
Voici une description du programme : on dispose d'une grille dont chaque cellule constitué de 4 segments rouges indépendants (lignes 15-16). Chaque clic de souris de l'utilisateur (cf. ligne 25) déclenche l'exécution de la fonction `suppr` (lignes 20-23) ; la fonction `suppr` récupère alors les coordonnées `clic` (ligne 21) du point où l'utilisateur a cliqué puis détermine (ligne 22), avec la méthode `find_closest`, l'item le plus proche de ce point. L'item trouvé est ensuite effacé (ligne 23).

Superposition des items sur le canevas

Lorsque des items sont créés sur un canevas, certains nouveaux items peuvent en partie intersecter des items déjà présents ; les nouveaux items sont toujours placés **au-dessus** des anciens. Les nouveaux items viennent alors cacher (masquer) ceux sur lesquels il se superposent.

Soit le programme suivant :





correspondant au code ci-dessous :

```
1  from tkinter import *
2  from random import randrange, sample
3
4  WIDTH=600
5  HEIGHT=600
6
7  COTE=150
8
9  root = Tk()
10 cnv = Canvas(root, width=WIDTH, height=HEIGHT, background="ivory")
11 cnv.pack()
12 x=y=100
13
14 rect1=cnv.create_rectangle(x, y, x+COTE, y+COTE, fill="red",
15                              outline='')
16 rect2=cnv.create_rectangle(x+1.25*COTE, y, x+2*COTE, y+2*COTE,
17                              fill="gray", outline='')
18 x=x+COTE/2
19 y=y+COTE/2
20 rect3=cnv.create_rectangle(x, y, x+COTE, y+COTE, fill="green",
21                              outline='')
22 x=x+COTE/2
23 y=y+COTE/2
24 rect4=cnv.create_rectangle(x, y, x+COTE, y+COTE, fill="blue",
25                              outline='')
26
27 print(cnv.find_all())
28
29 def f():
30     cnv.tag_raise(rect2, rect4)
31     print(cnv.find_all())
32
33 cnv.after(1000, f)
34
35 root.mainloop()
```

```
36 (1, 2, 3, 4)
37 (1, 3, 4, 2)
```

On crée un carré rouge puis, à sa droite un rectangle gris puis un carré vert et un carré bleu qui se superposent au-dessus du rectangle et viennent donc le cacher en partie. Au bout de 1,5 seconde, le rectangle gris est placé **au-dessus** des deux derniers carrés. Noter que la position relative des carrés elle ne change pas.

Expliquons le code.

— Ligne 27 : avant l'animation. La méthode `find_all` renvoie les items en commençant par les plus profonds et en remontant dans la pile des items.

— Ligne 33 : l'animation est déclenchée avec la méthode `after` qui exécute la fonction up

— ligne 30 : la méthode `tag_raise` est appelée. Le rectangle `rect2`, de couleur grise est déplacé dans la pile des items, au-dessus de l'item `rect4` qui est le dernier carré, celui de couleur bleue.

— Lignes 31 et 33 : on affiche à nouveau la pile des items sur le canevas. On voit bien que l'item 2 a été placé **au-dessus** de l'item d'id 4. L'ordre relatif des autres items dans la pile lui n'a pas bougé.

On peut appeler `tag_raise` sans le second argument et dans ce cas, l'item est placé en dernière position dans la liste d'affichage des items du canevas et donc au-dessus de tous les autres.

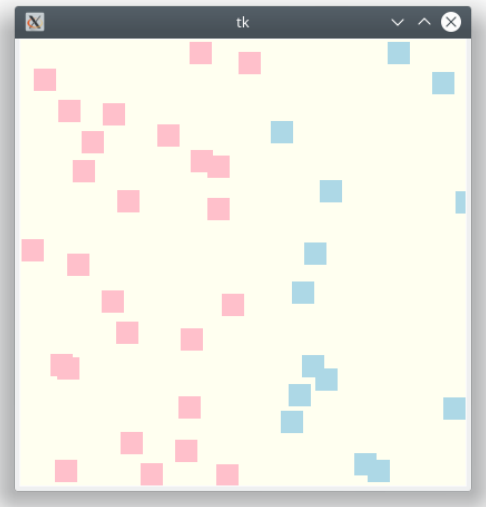
Les items plus récents sont au-dessus. En pratique, un item qu'on déplace doit être visible ce qui peut nécessiter de le hisser.

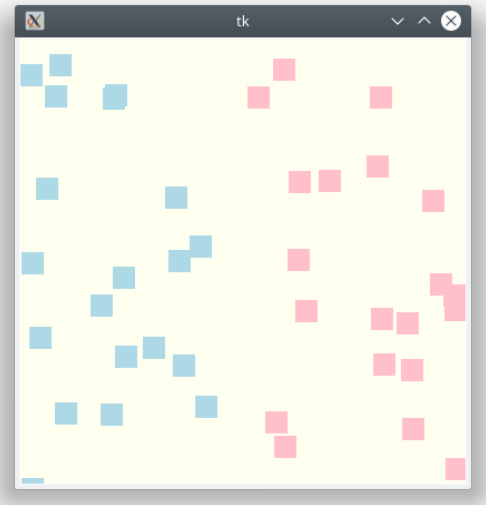
La méthode inverse de `tag_raise` est `tag_lower`. Ainsi, une instruction du type `cnv.tag_lower(mon_item)` va placer `mon_item` tout **en bas** de la pile des items.

Tag sur un item

Chaque item créé sur un canevas peut recevoir un tag sous forme d'une chaîne de caractères. Le canevas permet ensuite d'agir sur tous les items ayant un tag donné.

A titre d'illustration, soit par exemple le programme suivant





où on place des petits carrés aléatoires sur un canevas, des bleus à droite et des rouges à gauche ; les carrés à gauche sont marqués avec le tag `"left"` et ceux de droite sont marqués avec le tag `"right"` ; au bout de deux secondes, tous les carrés bleus se déplacent dans la zone de gauche et les rouges dans la zone de droite.

Voici le code correspondant :

```
from tkinter import *
from random import randrange

WIDTH=400
HEIGHT=400

COTE=20

root = Tk()
cnv = Canvas(root, width=WIDTH, height=HEIGHT, background="ivory")
cnv.pack()

for i in range(40):
    x=randrange(WIDTH)
    y=randrange(HEIGHT)
    if x>WIDTH//2:
        cnv.create_rectangle(x, y, x+COTE, y+COTE, fill='light blue',
                             outline='', tag='right')
    else:
        cnv.create_rectangle(x, y, x+COTE, y+COTE, fill='pink',
                             outline='', tag='left')

def move():
    cnv.move("right", -WIDTH//2, 0)
```

```
25     cnv.move("left", WIDTH//2, 0)
26
27 cnv.after(1000, move)
28
29 root.mainloop()
```

— Ligne 18 : les items à droite sont marqués `"right"` et initialement de couleur bleu.
— Ligne 21 : les items à gauche sont marqués `"left"` et initialement de couleur rouge.
— Ligne 27 : lancement dfe l'animation au bout de 1 seconde.
— Ligne 24 : les items qui sont marqués du tag `"right"` bougent vers la gauche. de la moitié de la largeur du canevas.
— Ligne 25 : les items qui sont marqués du tag `"left"` bougent vers la croite.

Il est possible de placer une option `tags` (au pluriel) associant **plusieurs** tags à un item. La syntaxe est :

```
1 cv. create_rectangle(80, 80, 100, 100, tags=("clickable", "right"))
```

Ci-dessous, on a utilisé un tuple mais on pourrait utiliser une liste.

Les tags permettent de filtrer des items et de lier sélectivement certains items portant un tag à des événements.

Lecture de l'option tags

Les `tags` attribués un item sont accessibles via l'option `tags` à laquelle on peut accéder avec la méthode `itemconfig`.

Cependant, quand on appelle cette option, au lieu de récupérer un tuple des différents tags, on récupère une chaîne qui est la concaténation, séparée par des espaces, des tags donnés en options.

Exemple :

```
1 from tkinter import Tk, Canvas
2
3 root = Tk()
4 cnv = Canvas(root, width=200, height=200)
5 cnv.pack()
6
7 rect=cnv.create_rectangle(100, 100, 150, 150, fill='blue', tags=["blue", 42])
8
9 options=cnv.itemconfig(rect)
10 print(*options.items(), sep='\n')
11
12 root.mainloop()
```

```
13 ('activedash', ('activedash', '', '', '', ''))
14 ('activefill', ('activefill', '', '', '', ''))
15 ('activeoutline', ('activeoutline', '', '', '', ''))
16 ('activeoutlinestipple', ('activeoutlinestipple', '', '', '', ''))
17 ('activestipple', ('activestipple', '', '', '', ''))
```

```
('activewidth', ('activewidth', '', '', '0.0', '0.0'))
('dash', ('dash', '', '', '', ''))
('dashoffset', ('dashoffset', '', '', '0', '0'))
('disableddash', ('disableddash', '', '', '', ''))
('disabledfill', ('disabledfill', '', '', '', ''))
('disabledoutline', ('disabledoutline', '', '', '', ''))
('disabledoutlinestipple', ('disabledoutlinestipple', '', '', '', ''))
('disabledstipple', ('disabledstipple', '', '', '', ''))
('disabledwidth', ('disabledwidth', '', '', '0.0', '0'))
('fill', ('fill', '', '', '', 'blue'))
('offset', ('offset', '', '', '0,0', '0,0'))
('outline', ('outline', '', '', 'black', 'black'))
('outlineoffset', ('outlineoffset', '', '', '0,0', '0,0'))
('outlinestipple', ('outlinestipple', '', '', '', ''))
('state', ('state', '', '', '', ''))
('stipple', ('stipple', '', '', '', ''))
('tags', ('tags', '', '', '', 'blue 42'))
('width', ('width', '', '', '1.0', '1.0'))
```

— Ligne 7 : on passe sous forme de liste les tags `tags=["blue", 42]` au rectangle.

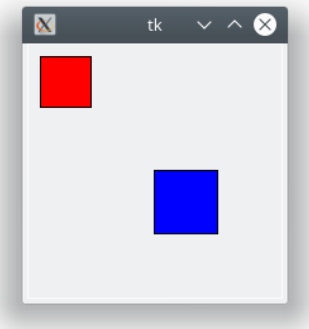
— Lignes 9 et 34 : on accède au dictionaire des options et la liste de tags est devenue la chaîne de caractères `"blue 42"`.

Récupérer les tags d'un item

La méthode `gettags` appliquée à un item du canevas permet de récupérer les tags que l'on a affectés à cet item. Exemple :

```
from tkinter import Tk, Canvas

root = Tk()
cnv = Canvas(root, width=200, height=200)
cnv.pack()

rect1=cnv.create_rectangle(10, 10, 50, 50, fill='red', tags=("red", 24))
rect2=cnv.create_rectangle(100, 100, 150, 150, fill='blue', tags=["blue", 42])

print(cnv.gettags(rect2))

root.mainloop()
```

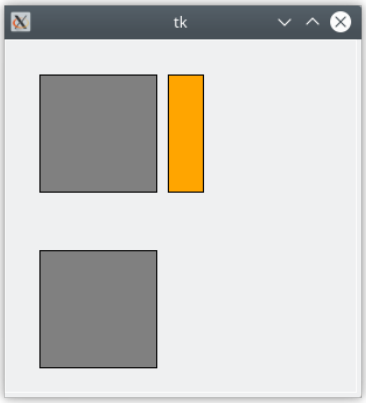
```
('blue', '42')
```

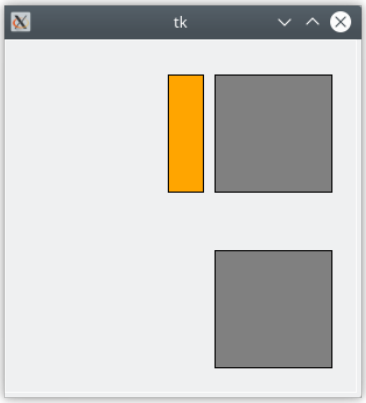


— Lignes 7-8 : on affecte des tags (une couleur, un numéro) à chaque rectangle.
— Lignes 10 et 13 : on récupère les tags de l'item d'id `rect2`.

Déplacement multiple avec `tags` et `move`

Les tags permettent d'agir simultanément sur plusieurs items ayant un tag donné. L'exemple ci-dessous montre le déplacement de tous les items du canevas dont un tag est la chaîne `"rect"` :






```
1  from tkinter import Tk, Canvas
2
3  root = Tk()
4  cnv = Canvas(root, width=300, height=300)
5  cnv.pack()
6
7  cnv.create_rectangle(30, 30, 130, 130, fill='gray', tags=("A", "rect"))
8  cnv.create_rectangle(30, 30+150, 130, 130+150, fill='gray', tags=("rect", "B"))
9  cnv.create_rectangle(140, 30, 170, 130, fill='orange', tags=("small rect",))
10
11 def deplacer(x):
12     cnv.move("rect", x, 0)
13
14 cnv.after(1000, deplacer, 150)
15
16 root.mainloop()
```

— Ligne 12 : tout item dont un des tag est la chaîne `"rect"` sera déplacé vers la droite de `x` pixels.

— Ligne 14 : le déplacement est activé une seconde et demi après ouverture du programme.

— Lignes 7-8 : les deux deux rectangles ont un tag `"rect"` : ils seront déplacés.

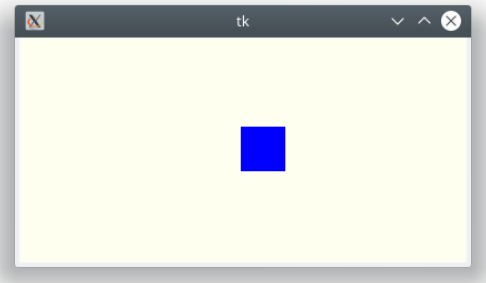
— Ligne 9 : ce rectangle orange a un tag qui est une chaîne **contenant** le mot `"rect"` mais sans être exactement le mot `"rect"` donc il ne sera pas déplacé.

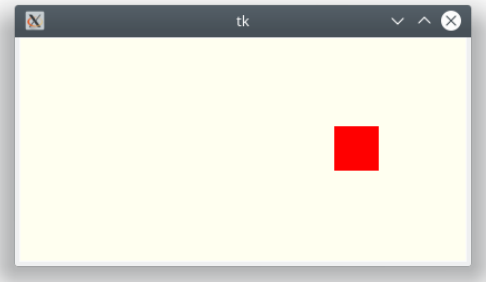
On pourrait également effectuer des suppressions multiples avec la méthode `delete` du canevas.

Événement associé à un tag

Il est possible de faire en sorte qu'un événement de la souris ne soit associé qu'à certains items du canevas portant un tag donné.

Dans le programme ci-dessous :





un carré fait l'aller et retour entre le bord gauche et la bord droit du canevas. Lorsque l'utilisateur clique sur le carré, il change de couleur

```
from tkinter import *

WIDTH=400
HEIGHT=200
COTE=40

root = Tk()
cnv = Canvas(root, width=WIDTH, height=HEIGHT, background="ivory")
cnv.pack()

milieu=HEIGHT/2-COTE/2
debut=10
fin = WIDTH-debut

cnv.create_rectangle(debut, milieu, debut+COTE, milieu+COTE, fill="blue",
                     outline='', tag="mobile")

def animate(v):
    a,b,c,d=cnv.coords("mobile")
    if a< debut or c> fin:
        v=-v
```

```
22      cnv.move("mobile", v, 0)
23      cnv.after(25, animate, v)
24
25  def changer_couleur(event):
26      print("click !")
27      color=cnv.itemconfigure("mobile", "fill")[-1]
28      if color=="red":
29          color="blue"
30      else:
31          color="red"
32      cnv.itemconfigure("mobile", fill=color)
33
34  cnv.tag_bind("mobile", "<Button-1>", changer_couleur)
35
36  animate(v=3)
37  root.mainloop()
```

— Un carré portant un tag "mobile" est créé lignes 15-16.

— L'animation est lancée (ligne 36) avec une vitesse valant 3.

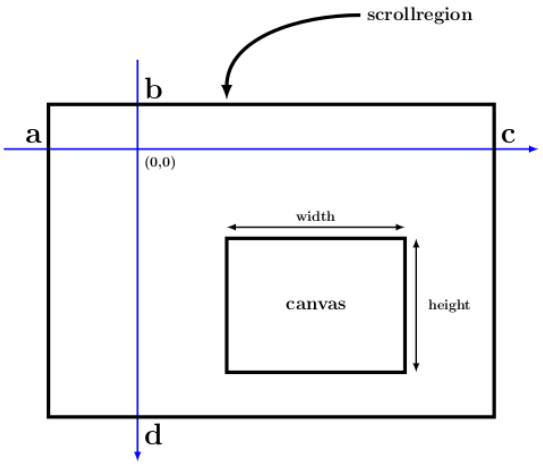
— Un événement de la souris est défini ligne 34 mais il est attaché **uniquement** à un clic sur un item portant un tag "mobile". Si on clique **ailleurs** que sur un tel objet, la fonction `changer_couleur` **n'est pas** lancée et l'affichage ligne 25 n'apparaîtra pas.

— Si on clique sur l'objet mobile, la fonction `changer_couleur` est lancée : après avoir affiché "click!" dans la console (ligne 26), elle examine la couleur de l'objet cliqué (ligne 28) et la change (lignes 28-32).

3 Barres de défilement

La zone `scrollregion` d'un canevas

La partie visible d'un canevas peut, en fait, être incluse dans une zone plus étendue :



Lorsque le canevas est construit, cette zone est donnée par une option `scrollregion` :

```
cnv=Canvas(root,  scrollregion =(-100, 200, 600, 400),
           width=300, height=200, bg='ivory')
```

Ici, `cnv` est un canevas, s'étendant sur une zone rectangulaire de diagonale les points `(-100, 200)` et `(600, 400)`. Le sens et la direction des axes sont comme dans toute interface graphique :

— axe des abscisses horizontal et orienté de gauche à droite,

— axe des ordonnées vertical orienté de haut en bas.

L'origine des axes s'en déduit par intersection. Il en résulte que l'origine des axes **n'est pas forcément** en haut à gauche de la région.

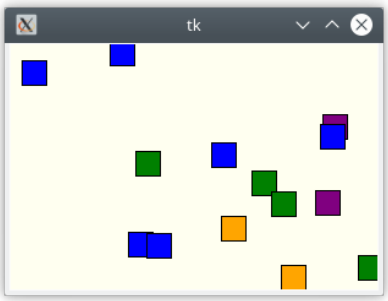
Je parlerai de *zone visible* pour le rectangle du canevas et de *zone de défilement* pour la zone définie par `scrollregion`.

Ici, la zone visible (le canevas à proprement parler) est un rectangle de largeur `width=300` et de hauteur `height=200` inclus dans la zone de défilement. Par défaut, la zone visible et la zone de défilement ont des coins supérieurs gauches qui coïncident.

Voici un code complet définissant une zone de défilement :

```
from tkinter import *
from random import randrange

def dot(cnv, C, R=6, color='red'):
    xC, yC=C
    A=(xC-R, yC-R)
    B=(xC+R, yC+R)
    return cnv.create_oval(A,B, fill=color, outline=color)

root=Tk()

COLORS=["orange", "blue", "green", "purple"]
R=20
```

```
cnv=Canvas(root,  scrollregion =(-100, 200, 600, 400),
           width=300, height=200, bg='ivory')
cnv.pack()

for _ in range(100):
    A=[randrange(600) for _ in range(2)]
    B=A[0]+R, A[1]+R
    cnv.create_rectangle(A, B, fill=COLORS[randrange(4)])
dot(cnv, (10, 10))

root.mainloop()
```



Le code ci-dessus ne montre pas véritablement l'intérêt de l'option `scrollregion`. Néanmoins, il faut savoir qu'autour du canevas visible, le canevas se poursuit. L'intérêt provient du fait que des méthodes du canevas permettent d'explorer (`xview` et `yview`) la totalité de la zone, avec barres de défilement (*scrollbar*, d'où le nom de `scrollregion`) ou même, sans barre de défilement.

Robustesse

On pourra observer qu'un canevas peut admettre une zone de défilement extrêmement grande tout en restant fluide (si on essayait de l'explorer). Le programme ci-dessous crée une zone de défilement gigantesque :

```
from tkinter import *
from random import randrange

root=Tk()

COLORS=["orange", "blue", "green", "purple"]
R=20

cnv=Canvas(root,  scrollregion =(0, 0, 10**6, 10**6),
           width=600, height=600, bg='ivory')
cnv.pack()

for _ in range(10**6):
```

```
    A=[randrange(600) for _ in range(2)]
    B=A[0]+R, A[1]+R
    cnv.create_rectangle(A, B, fill=COLORS[randrange(4)])

root.mainloop()
```

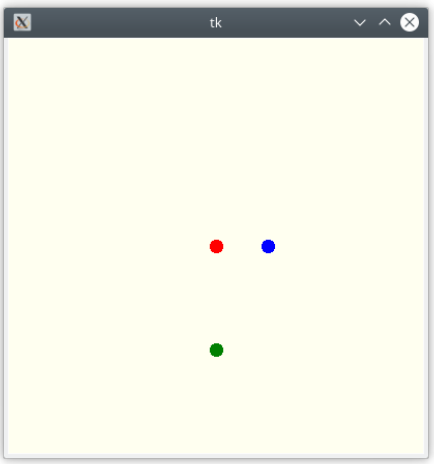
La taille *potentielle* est de 1000 milliards de pixels. Seul le temps de chargement des items (la boucle `for` dans le code ci-dessus) est un peu long.

Changement d'origine, d'unité

Tkinter ne permet pas de changer les unités sur les axes, en particulier, on ne peut pas changer de sens l'axe des ordonnées (pour obtenir le même sens qu'en mathématiques).

En revanche, l'existence d'une zone de défilement `scrollregion` paramétrable permet de changer l' **origine** des axes. Voici un exemple :

```
from tkinter import *

def dot(cnv, C, R=6, color='red'):
    xC, yC=C
    A=(xC-R, yC-R)
    B=(xC+R, yC+R)
    return cnv.create_oval(A,B, fill=color, outline=color)

root=Tk()

SIDE=200

cnv=Canvas(root,  scrollregion =(-SIDE, -SIDE, SIDE, SIDE),
           width=2*SIDE, height=2*SIDE, bg='ivory')
cnv.pack()

dot(cnv, (0,0))
dot(cnv, (SIDE//4,0), color="blue")
dot(cnv, (0,SIDE//2), color="green")

root.mainloop()
```



La fonction `dot` permet de dessiner un disque, par défaut rouge, de 6 pixels de rayon et centré au point C donné.

Comme le montre ligne 17, on dessine un disque rouge à l'origine `(0,0)` et on observe que le disque n'est pas placé en haut à gauche du canevas mais au centre de celui-ci : on a donc bien changé l'origine du repère. Le placement des deux autres points montre que les axes ont leurs direction et sens habituels.

En effet, la définition de `scrollregion` contraint la position de l'origine. Ici, comme `scrollregion =(-200, -200, 200, 200)`, c'est que l'origine `(0,0)` est au centre de cette région. Comme, la fenêtre est de dimension 400 x 400 (même dimension que la zone de défilement), c'est que l'origine est au centre du canevas visible.

Scroller un canevas sans barre de défilement

On dispose d'un canevas ayant une option `scrollregion` (zone de défilement) dont on voudrait explorer le contenu, typiquement :

```
cnv=Canvas(root,  scrollregion =(-SIDE, -SIDE, SIDE, SIDE),
           width=2*SIDE, height=2*SIDE, bg='ivory')
```

On rappelle que, par défaut, le canevas à proprement parler permet juste de voir le rectangle

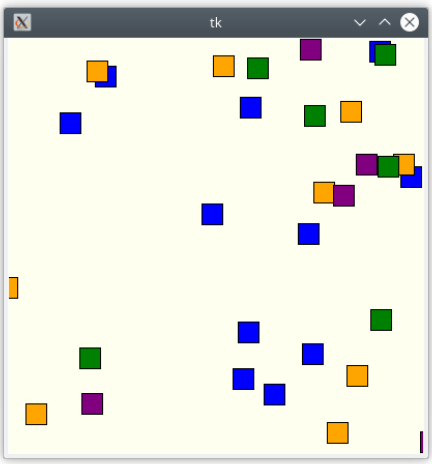
— de largeur `width`

— de hauteur `height`

— placé dans le coin supérieur gauche de la zone `scrollregion`.

Pour voir un **autre** rectangle, il suffit de translater le canevas dans la zone `scrollregion` ; pour cela on utilise les méthodes `xview_scroll` (pour un décalage horizontal) et `yview_scroll` (pour un décalage vertical) de `Canvas`.

Dans le programme ci-dessous



initialement la zone de défilement est remplie de carrés colorés aléatoires. Une seconde après le début du programme, une animation permet de parcourir la zone de défilement de gauche à droite par pas de 100 pixels.

Voici le code correspondant :

```
1  from tkinter import *
2  from random import randrange
3
4  root=Tk()
5
6  COLORS=["orange", "blue", "green", "purple"]
7  R=20
8
9  cnv=Canvas(root,  scrollregion =(0, 0, 800, 800),
10            width=200, height=200, bg='ivory')
11 cnv.pack()
12
13 cnv["xscrollincrement"]=100
14
15 for _ in range(500):
16     A=[randrange(800) for _ in range(2)]
17     B=A[0]+R, A[1]+R
18     cnv.create_rectangle(A, B, fill=COLORS[randrange(4)])
19
20 def xscroll():
21     cnv.xview_scroll(1, "units")
22     cnv.after(1500, xscroll)
23
24 xscroll()
25
26 root.mainloop()
```

— Ligne 10 : le canevas visible est de taille 200 x 200

— Lignes 15-18 : toute la zone de défilement (actuellement visible ou pas) est remplie de carrés

aléatoires colorés.

— Lignes 20-22 : l'animation du défilement : un décalage toutes les 1 seconde 1/2.

— Ligne 21 : chaque scroll décale d'une unité, l'unité étant indiquée par la chaîne `"units"`. Il semblerait, d'après la documentation de Tcl/Tk que l'unité par défaut soit 1/10 de la largeur du canevas.

Il est aussi possible de modifier le décalage de scroll (l'unité) par défaut. Par exemple, pour un décalage horizontal de 100 pixels, il suffit pour cela de définir l'option `"xscrollincrement"` :

```
cnv=Canvas(root,  scrollregion =(0, 0, 800, 800),
           width=200, height=200, bg='ivory',
           xscrollincrement=100)
```

et de continuer à déclarer le décalage par

```
cnv.xview_scroll(1, "units")
```

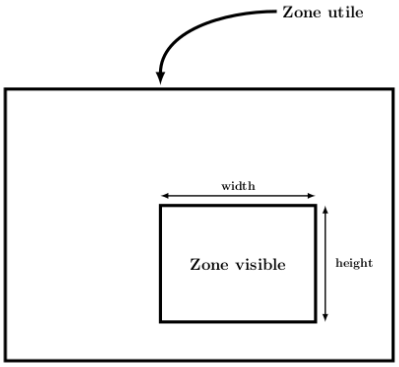
Il est même possible d'effectuer un décalage dynamique en modifiant `cnv["xscrollincrement"]`.

Fonctionnement d'une barre de défilement

Une barre de défilement permet de rendre accessible une zone a priori invisible. Plus précisément, une barre de défilement s'utilise lorsqu'on dispose de deux zones rectangulaires

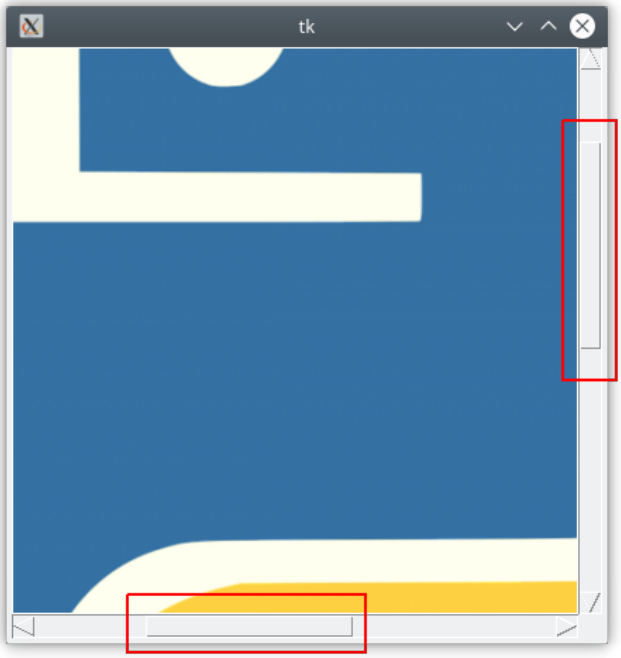
— une zone « utile » (par exemple, du texte, du dessin)

— une zone « de vue » (un canevas par exemple)



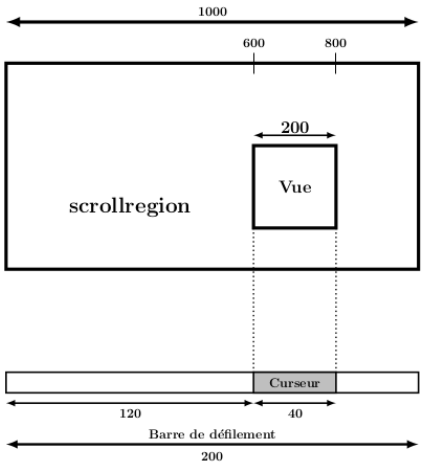
et que la zone utile déborde de la zone de vue et donc que certaines parties de la zone utile ne sont pas visibles. Une barre de défilement peut être horiontale ou verticale. Pour visualiser la zone utile dans sa largeur, une barre de défilement horizontale permet de déplacer la zone de vue suivant un couloir de la gauche vers la droite de la zone utile. De même si on dispose d'une barre de défilement verticale, on peut explorer la zone utile de haut en bas et sur un couloir. Avec deux barres, on peut donc explorer **toute** la zone utile.

Pour fixer les idées, prenons le cas d'une barre de défilement horizontale. Elle possède un curseur de défilement qui se déplace de la gauche à la droite de la barre. Ce curseur de défilement se présente sous la forme d'une segment mobile en relief :



Comment la barre de défilement corrèle-t-elle la zone de vue et la zone utile ? Réponse :

— la totalité de la barre représente la largeur de la zone utile ;
— le curseur représente la largeur de la vue ;
— la position du curseur dans la barre représente **proportionnellement** la position du début de la vue par rapport à la longueur de la barre ;
— la largeur du curseur est **proportionnelle** à la largeur de la vue.



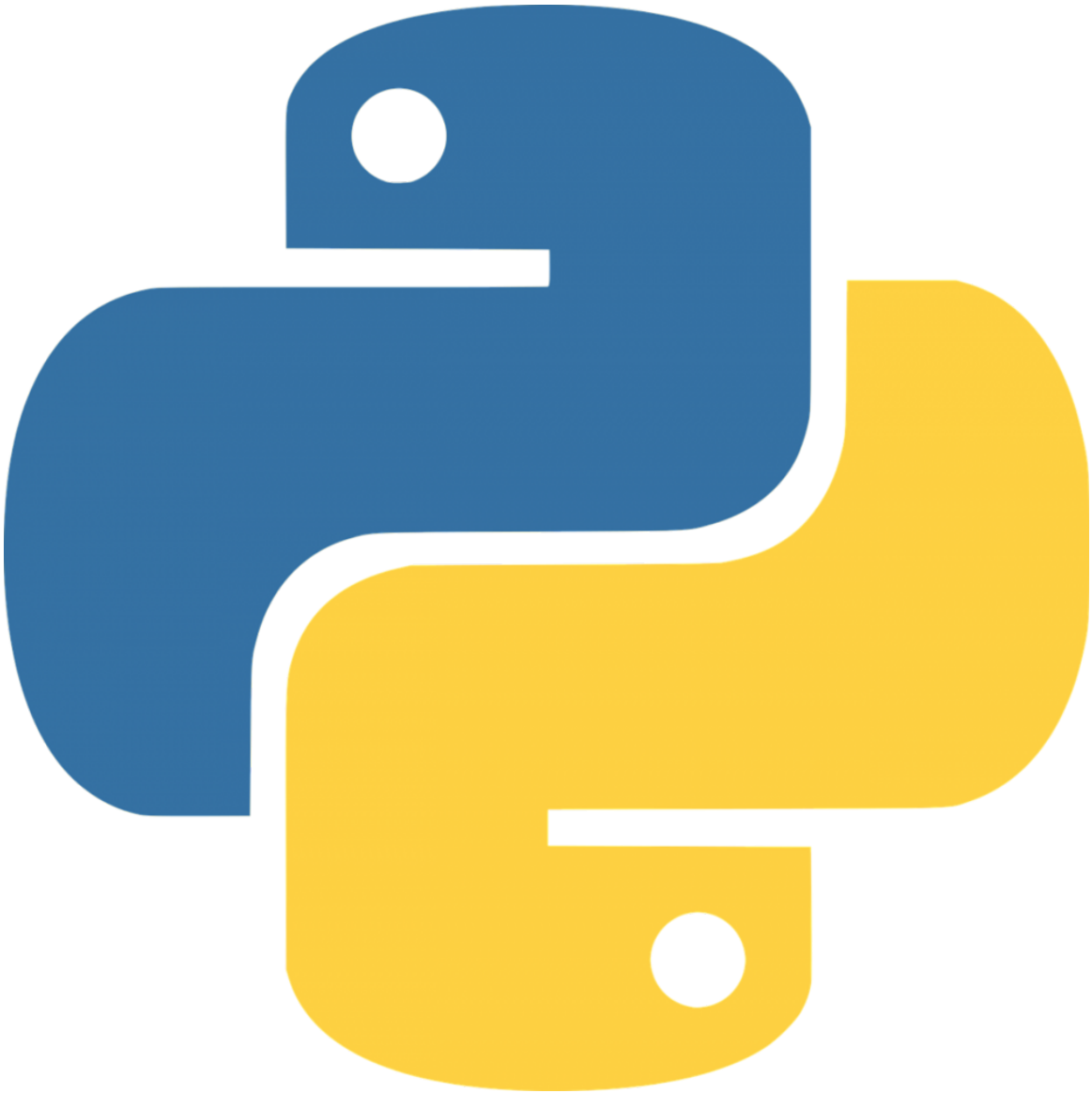
Ainsi, pour fixer les idées, considérons une zone utile de 1000 pixels de large admettant une vue de 200 pixels de large. La barre de défilement, collée sous la zone de vue, est aussi de 200 pixels

de large. Comme il y a un rapport de 1 à 5 entre la vue et l'utile, la vue sera représentée dans la barre de défilement par un segment mobile de longueur 200/5=40 pixels. Si le côté gauche de la vue commence au pixel 600 de la zone utile (c'est-à-dire aux 3/5), c'est que le segment mobile commencera aux 3/5 de la barre de défilement et donc sera positionné à partir du début de la barre entre le pixel 120 et le pixel 120+40=160.

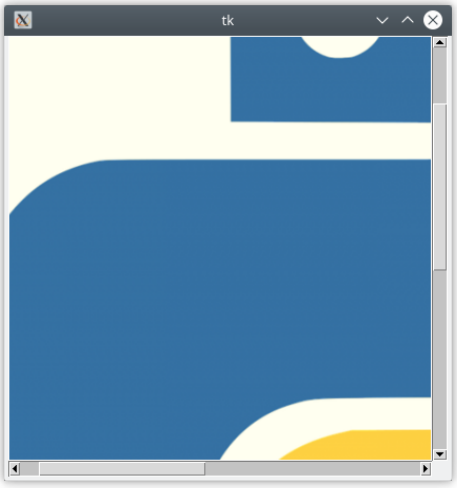
Sous Tkinter, une barre de défilement est créée avec le widget `Scrollbar`. Ce widget est, dans un premier temps, construit de manière **indépendante** du widget qu'il visualisera (par exemple un canevas ou une zone de texte). La tâche essentielle d'un widget `Scrollbar` est de gérer la position du segment mobile de déplacement de la vue. En interne, il le gère par deux valeurs qui sont deux ratios représentant la vue par rapport à la dimension de la zone utile. Dans l'exemple précédent, le curseur est déterminé par son ratio gauche qui est de $120/200 = 0.6$ et son ratio droit qui est $160/200 = 0.8$.

Barres de défilement autour d'un canevas

Assez classiquement, on peut entourer un canevas de barres de défilement, une barre horizontale, une barre verticale. Mais on va voir que ce n'est pas aussi immédiat que la création d'autres widgets. En guise d'illustration, un logo du langage Python en taille 1000 x 1000 :



on va placer une (grosse) image `logo.png` de taille 1000 x 1000 dans un canevas de même taille. La **vue** du canevas sera de taille 400 x 400 et deux barres de défilement permettront d'explorer la totalité de l'image :



Le code final sera le suivant :

```
from tkinter import *
from tkinter import ttk

root=Tk()
SIZE=1000
size=400

cnv=Canvas(root,  scrollregion =(0, 0, SIZE, SIZE),
           width=size, height=size, bg='ivory')

cnv.grid(row=0,column=0)

xscroll=ttk.Scrollbar(root, orient=HORIZONTAL)
yscroll=ttk.Scrollbar(root, orient=VERTICAL)


xscroll.grid(row=1, column=0, sticky=E+W)
yscroll.grid(row=0, column=1,  sticky=S+N)

xscroll["command"]=cnv.xview
yscroll["command"]=cnv.yview
cnv['xscrollcommand']=xscroll.set
cnv['yscrollcommand']=yscroll.set

logo=PhotoImage(file="logo.png")

cnv.create_image((SIZE//2, SIZE//2), image=logo)

root.mainloop()
```

Dans ce qui suit, on décrit ce code. D'abord, pour des raisons esthétiques, je n'ai pas utilisé les barres de défilement par défaut de Tkinter mais les barres utilisables par son extension standard

Ttk. Il faut donc importer Ttk comme montré dans les deux premières lignes. D'autre part, on a une zone `scrollregion` de taille 1000 x 1000 dans un canevas de taille visible 400 x 400 :

```
SIZE=1000
size=400

cnv=Canvas(root,  scrollregion =(0, 0, SIZE, SIZE),
           width=size, height=size, bg='ivory')
```

Une barre de défilement dans un canevas est un widget de type `Scrollbar` qui, dans un premier temps, est construit indépendamment du canevas. On construit deux barres, une horizontale et l'autre verticale :

```
1 xscroll=ttk.Scrollbar(root, orient=HORIZONTAL)
2 yscroll=ttk.Scrollbar(root, orient=VERTICAL)
3
4
5 xscroll.grid(row=1, column=0, sticky=E+W)
6 yscroll.grid(row=0, column=1,  sticky=S+N)
```

Comme j'ai choisi d'utiliser l'extension Ttk de Tkinter, le nom de la classe est ici `ttk.Scrollbar`.

On utilise la méthode `grid` de placement des widgets, elle est ici assez naturelle. Pour ajuster les barres, on utilise l'option d'étirement `sticky`. Les barres de défilement sont alors bien présentes mais inactives. Si on voulait placer les barres autrement, par exemple une en haut au lieu d'en bas, il suffit de modifier en conséquence l'emplacement des widgets dans les cellules de la grille (`grid`) utilisée.

Il faut maintenant :

— connecter chaque barre au canevas ;

— connecter le canevas à chaque barre ;

Les connexions se font en définissant des options pour les widgets. Rappelons que si un widget, disons `mon_widget`, admet une option portant le nom `mon_option`, alors Tkinter permet d'initialiser ou de modifier l'option en affectant `mon_widget[mon_option]` à ce qui est souhaité. Par exemple, si on veut changer en rouge la couleur de fond d'un canevas `cnv` alors on écrira `cnv["background"]="red"`. C'est avec cette syntaxe que l'on va modifier les options du canevas et des barres. Par ailleurs, on va limiter les explications au cas de la barre horizontale, nommée `xscroll` ci-dessus, le cas de la barre verticale étant analogue.

Lorsque le curseur de défilement de la barre est modifié par l'utilisateur, une fonction référencée par l'option `command` de la barre de défilement `xscroll` est automatiquelent appelée. Cette fonction va transmettre le déplacement au canevas ; le scroll horizontal du canevas est géré par la méthode `xview` ; d'où la « connexion » suivante :

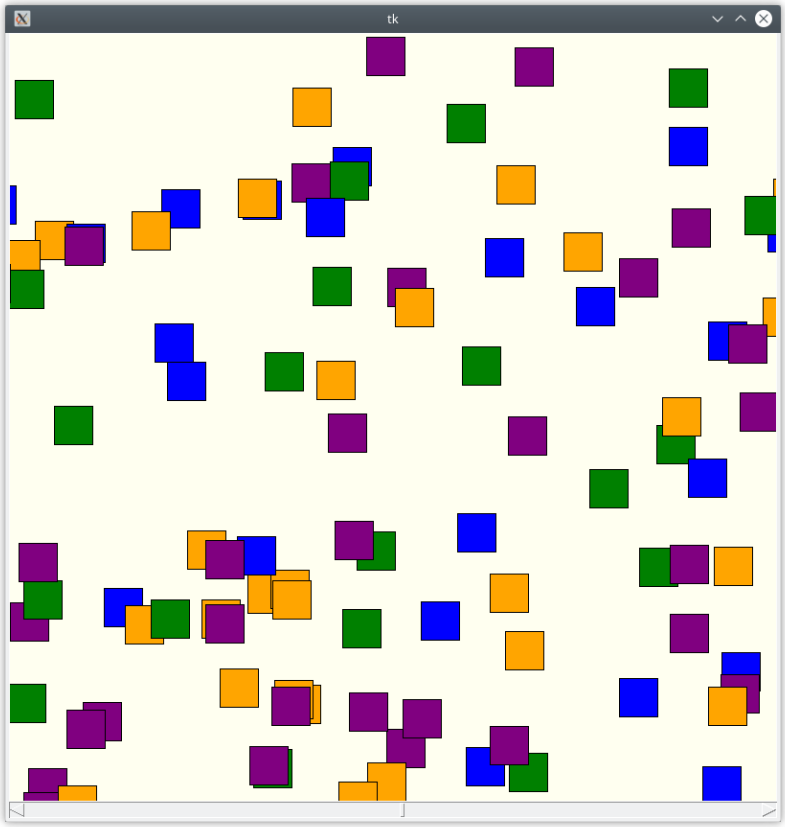
```
xscroll["command"]=cnv.xview
```

Mais le canevas doit lui-même avertir la barre de comment il va scroller (par exemple, le scroll pourrait être bloqué et seul le canevas le sait). Pour cela, le canevas dispose d'une option `xscrollcommand` ; enfin, une barre de défilement dispose d'une méthode qui lui permet de définir la position de son curseur de défilement, la méthode `set`. D'où la connexion suivante :

```
cnv['xscrollcommand']=xscroll.set
```

Barre de défilement et canevas : performances

Les barres permettent de défiler une zone extrêmement grande sans perte de performance. Ci-dessous, le canevas est de taille un rectangle de longueur un million de pixels et de 800 de hauteur et rempli avec un million de carrés aléatoires :



Une fois le canevas visible (ce qui prend un peu de temps à cause de la création aléatoire des items), la barre de défilement permet de le parcourir de manière très fluide.

Le code correspondant est :

```
from tkinter import *
from random import randrange

root=Tk()

SIDE=10**7
WINDOW=800
nitems=10**6
COLORS=["orange", "blue", "green", "purple"]
R=40

cnv=Canvas(root,  scrollregion =(0, 0, SIDE, WINDOW),
           width=WINDOW, height=WINDOW, bg='ivory')
cnv.grid(row=0,column=0)
```

```
15
16 xscroll=Scrollbar(root, orient=HORIZONTAL, command=cnv.xview)
17 xscroll.grid(row=1, column=0, sticky=E+W)
18
19 cnv['xscrollcommand']=xscroll.set
20
21 for _ in range(nitems):
22     A=[randrange(SIDE), randrange(WINDOW)]
23     B=A[0]+R, A[1]+R
24     cnv.create_rectangle(A, B, fill=COLORS[randrange(4)])
25
26 root.mainloop()
```

Il serait même possible de choisir deux barres de défilement sur une aire virtuelle d'un million par un million et d'obtenir un déplacement encore très fluide.

Barre de défilement et canevas : les fonctions de commande

Voici un code de placement d'une unique barre de défilement sous un canevas :

```
from tkinter import *
from random import randrange

root=Tk()

SIDE=1000
WINDOW=500
nitems=300
COLORS=["orange", "blue", "green", "purple"]
R=20

cnv=Canvas(root,  scrollregion =(0, 0, 1000, 1000),
           width=WINDOW, height=WINDOW, bg='ivory')
cnv.grid(row=0,column=0)

scroll=Scrollbar(root, orient=HORIZONTAL, command=cnv.xview)
scroll.grid(row=1, column=0, sticky=E+W)

cnv['xscrollcommand']=scroll.set

for _ in range(300):
    A=[randrange(1000) for _ in range(2)]
    B=A[0]+R, A[1]+R
    cnv.create_rectangle(A, B, fill=COLORS[randrange(4)])

root.mainloop()
```

On va ici décrire les arguments transmis lors des appels automatiques des fonctions de rappel des widgets ; pour le widget `Canvas`, on va décrire :

— l'option `xscrollcommand`
— la méthode `xview`

et pour le widget `Scrollbar` :

— l'option `command`
— la méthode set

Lorsqu'un élément d'une barre de défilement est modifié, son option `command` est appelée. Dans le code ci-dessus, cette option pointe vers la méthode `xview` du canevas. Lors de l'appel, `command` transmet à `xview` la nature du mouvement (`scroll` ou `moveto`) et un flottant entre 0 et 1 représentant le ratio entre la nouvelle position du curseur et la longueur de la barre de défilement (par exemple, 0.5 si le début du curseur est au milieu de la barre). Mais, le canevas doit examiner si le déplacement proposé est valide (par exemple, le curseur peut être bloqué). Le canevas appelle donc son option `xscrollcommand` ; elle pointe ici vers la méthode `set` de la barre. L'option ne transmet rien si le curseur est bloqué et sinon, elle transmet deux valeurs à la méthode `set` lui permettant de positionner le début et la fin du curseur.
Pour bien observer le comportement des appels automatiques, il pourra être utile de les intercepter comme ceci :

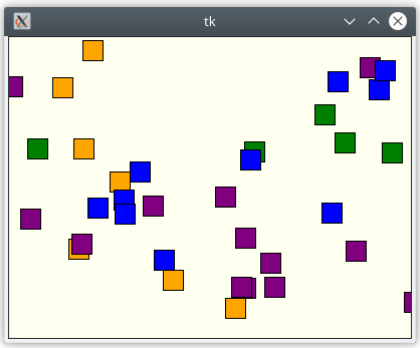
```
scroll=Scrollbar(root, orient=HORIZONTAL, command=f)
cnv['xscrollcommand']=ff
```

où on définira préalablement les fonctions interceptrices par :

```
def f(*z):
    print("f")
    print(*z)
    print("---------------")
    cnv.xview(*z)

def ff(*z):
    print("ff")
    print(*z)
    print("---------------")
    scroll.set(*z)
```

Défilement du canevas avec le clavier

Il est possible de scroller un canevas en utilisant les flèches du clavier :



Le principe est assez simple. On associe (méthode `bind`) les flèches du clavier à une fonction qui, selon la direction choisie, appelle les méthodes de scroll `xview_scroll` et `xview_scroll`. Voici le code :

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

root=Tk()

COLORS=["orange", "blue", "green", "purple"]
R=20
SIDE=1200
wcnv=400
hcnv=300

cnv=Canvas(root,  scrollregion =(0, 0, SIDE, SIDE),
           width=wcnv, height=hcnv, bg='ivory')
cnv.pack()
cnv.focus_set()

for _ in range(400):
    A=[randrange(SIDE) for _ in range(2)]
    B=A[0]+R, A[1]+R
    cnv.create_rectangle(A, B, fill=COLORS[randrange(4)])

def scroll(event):
    arrow=event.keysym
    if arrow=='Right':
        cnv.xview_scroll(1, "units")
    elif arrow=='Left':
        cnv.xview_scroll(-1, "units")
    elif arrow=='Down':
        cnv.yview_scroll(1, "units")
    elif arrow=='Up':
        cnv.yview_scroll(-1, "units")

```

```
33 for arrow in ["Left", "Right", "Down", "Up"]:
34     cnv.bind('<%s>' %arrow, scroll)
35
36 root.mainloop()
```

— Ligne 12 : on définit une zone de défilement carrée de côté 1200 pixels.

— Ligne 17-20 : on dessine des carrés aléatoires colorés pour remplir la zone de défilement.

— Ligne 15 : on donne le focus au canevas sinon, il n'écoute pas le canevas.

— Lignes 33-34 : les flèches du clavier sont associées à la fonction `scroll` de la ligne 22.

— Lignes 23-31 : selon la direction de flèche qui est repérée, une méthode `xview_scroll` ou `yview_scroll` est appelée ; cette méthode décale d'une demi-fenêtre la vue. Arrivée en fin de canevas, la vue est bloquée. Si la valeur de décalage est 1, le décalage a lieu vers la droite ou vers le bas. Si c'est -1, c'est dans le sens inverse.

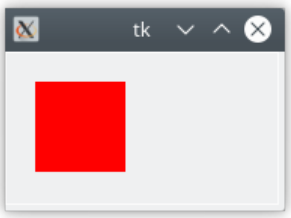
Chapitre V

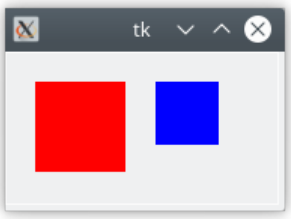
Les animations

La méthode `after`

La technique pour créer des animations en Tkinter est basée sur l'usage de la méthode `after`. Il s'agit d'une méthode associée à un widget et permettant d'appeler avec un certain délai (ou même périodiquement) l'exécution d'une fonction qui se charge d'animer (en déplaçant ou supprimant des objets, en en changeant les propriétés, ou la vitesse de déplacement, etc).

Voici un exemple très simplifié permettant de voir comment fonctionne et s'utilise la méthode ‘`after`‘





Quand l'interface s'ouvre, on observe un rectangle rouge et au bout d'une seconde, un rectangle bleu est ajouté.

Le code corresponsant est :

```
1 from tkinter import Tk, Canvas
2
3 root = Tk()
```

```
4 cnv = Canvas(root, width=180, height=100)
5 cnv.pack()
6
7 cnv.create_rectangle(20, 20, 80, 80, fill='red', outline='')
8
9 def dessiner(c):
10     cnv.create_rectangle(100, 20, 100+c, 20+c, fill='blue', outline='')
11
12 cnv.after(1000, dessiner, 42)
13
14 root.mainloop()
```

— Ligne 7 : le rectangle rouge visible au départ.

— Lignes 9-10 : fonction qui va dessiner un carré bleu de côté `c` qui sera donné en argument à la fonction.

— Ligne 12 : appel de la méthode `after` en tant que méthode du canevas. L'appel prend ici 3 arguments

  — le premier argument `1000` qui correspond à une durée en millisecondes

  — le 2e argument `dessiner` qui est la fonction qui sera appelée après le délai de 1000 ms

  — les autres arguments (ici juste 42), les arguments, dans l'ordre qu'il faudra passer à la fonction `dessiner` donnée en 2e argument de l'appel de la méthode `after`.

On peut appeler `after` depuis n'importe quel widget ou fenêtre :

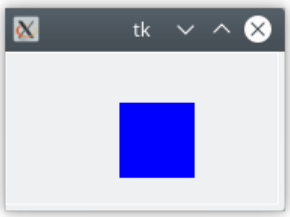
```
1 from tkinter import Tk, Canvas
2
3 root = Tk()
4 cnv = Canvas(root, width=180, height=100)
5 cnv.pack()
6
7 cnv.create_rectangle(20, 20, 80, 80, fill='red', outline='')
8
9 def dessiner(c):
10     print(c)
11     cnv.create_rectangle(100, 20, 100+c, 20+c, fill='blue', outline='')
12
13 root.after(1000, dessiner, 42)
14
15 root.mainloop()
```

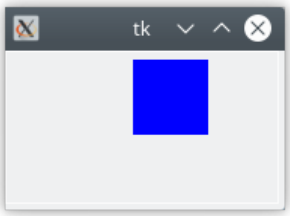
— Ligne 13 : appel de la méthode `after` comme une méthode de la fenêtre `root`.

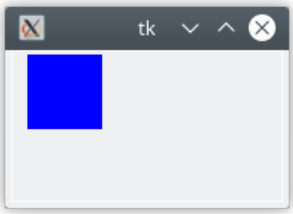
La méthode `after` : usage typique

L'usage typique de la méthode `after` est qu'elle appelle à intervalle régulier (et non plus une seule fois) une fonction d'animation.

Voici un exemple. Le programme ci-dessous montre un carré bleu se déplaçant aléatoirement chaque seconde :







Le code correspondant est :

```
from tkinter import Tk, Canvas
from random import randint


root = Tk()
cnv = Canvas(root, width=180, height=100)
cnv.pack()

def dessiner(item=None):
    cnv.delete(item)
    a=randint(1,90)
    b=randint(1,50)
    c=50
    item=cnv.create_rectangle(a, b, a+c, b+c, fill='blue', outline='')
    root.after(1000, dessiner, item)

dessiner()
```

```
18
19 root.mainloop()
```

— Ligne 9-15 : fonction d'animation qui est appelée périodiquement.

— Ligne 10 : la fonction efface le carré présent sur le canvas.

— Ligne 11-14 : la fonction `dessiner` crée un carré, nommé encore `item`, aléatoire (lignes 11-12), de 50 pixels de côté (ligne 13) et le dessine (ligne 14).

— **Ligne 15** : l'élément fondamental de l'animation qui permet de boucler l'animation. La méthode `after` temporise 1000 ms et fait en sorte que la fonction `dessiner` soit appelée en prenant le nouveau carré comme argument.

Illustrer `after` en créant des images

La méthode `after` permet de déclencher avec un certain délai de temps une action définie par une fonction.

Soit par exemple le code ci-dessous :

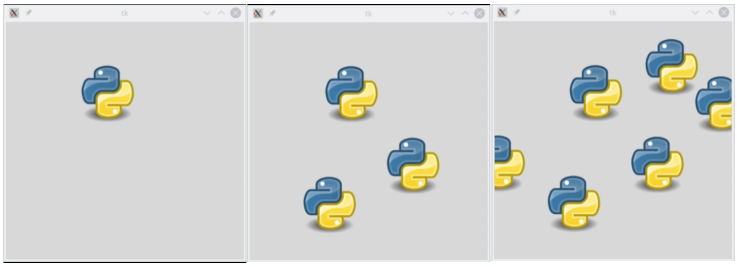
after_images.py

```
1  from tkinter import *
2  from random import randrange
3
4  SIDE=400
5  root = Tk()
6  cnv = Canvas(root, width=SIDE, height=SIDE)
7  cnv.pack()
8
9  logo = PhotoImage(file="python.gif")
10
11 def action(x, y):
12     cnv.create_image((x, y), image=logo)
13
14 x, y= randrange(SIDE),randrange(SIDE)
15 cnv.after(1000, action, x, y)
16
17 x, y= randrange(SIDE),randrange(SIDE)
18 cnv.after(2000, action, x, y)
19
20 x, y= randrange(SIDE),randrange(SIDE)
21 cnv.after(3000, action, x, y)
22
23 x, y= randrange(SIDE),randrange(SIDE)
24 cnv.after(4000, action, x, y)
25
26 x, y= randrange(SIDE),randrange(SIDE)
27 cnv.after(5000, action, x, y)
28
29 x, y= randrange(SIDE),randrange(SIDE)
30 cnv.after(6000, action, x, y)
```

```
31
32 root.mainloop()
```

On a défini une fonction `action` qui prend deux paramètres (le nom `action` n'a rien d'obligatoire).

Voici l'exécution :



Lorsque le programme est lancé, les 6 instructions `cnv.after(ms, action, x, y)` vont être exécutées les unes après les autres et instantanément. Chacune de ces instructions va déclencher l'exécution de la fonction `action` mais après le délai en ms indiqué par l'appel `cnv.after(ms, action, x, y`
Par exemple, une instruction telle que `cnv.after(4000, action, 200, 300)` va provoquer, 4 secondes (4000 ms) après avoir été activée, l'exécution de l'appel `action(200, 300)`. D'où l'effet d'animation.

La méthode `after` n'est pas spécifique de Canvas : dans le code ci-dessus, on pourrait remplacer les appels `cnv.after` par des appels `root.after`.

Annuler la méthode `after`

Un appel à la méthode `after` renvoie un identifiant de la tâche qui va être relancée. Cet identifiant, disons `id_anim`, peut être utilisé pour annuler cette tâche. Pour cela, il suffit d'appeler `after_cancel(id_anim)` où `after_cancel` est une méthode d'un widget.

Dans l'exemple ci-dessous, un compteur est lancé :

Un bouton permet (artificiellement) d'annuler l'animation et d'en relancer une nouvelle avec un nouveau compteur.

Le code est :

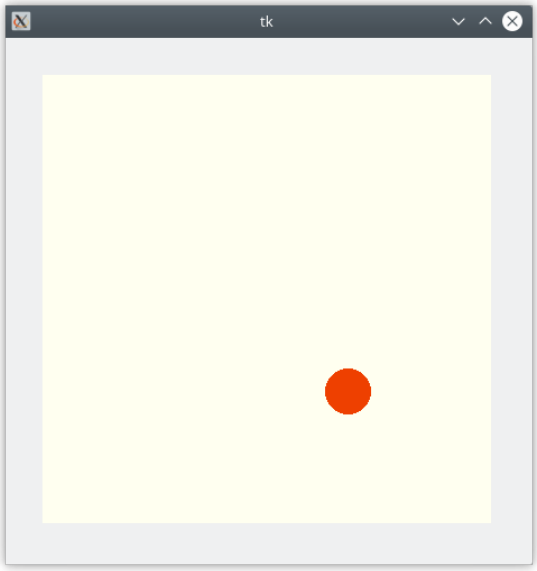
```
1  from tkinter import Tk, Canvas, Button
2  from random import randrange
3
4  def anim(cpt):
5      global id_anim
6      cnv.delete("all")
7      cnv.create_text(SIDE//2, SIDE//2, text =int(cpt), font="Arial 200 bold")
8      id_anim=cnv.after(500, anim, cpt+1)
9
10 def go():
11     cnv.after_cancel(id_anim)
12     anim(1)
13
14 SIDE=400
15 root = Tk()
16 cnv = Canvas(root, width=SIDE, height=SIDE, bg='ivory')
17 cnv.pack()
18 btn=Button(root, text="Reset", command=go)
19 btn.pack()
20
21 id_anim=None
22 anim(1)
23
24 root.mainloop()
```

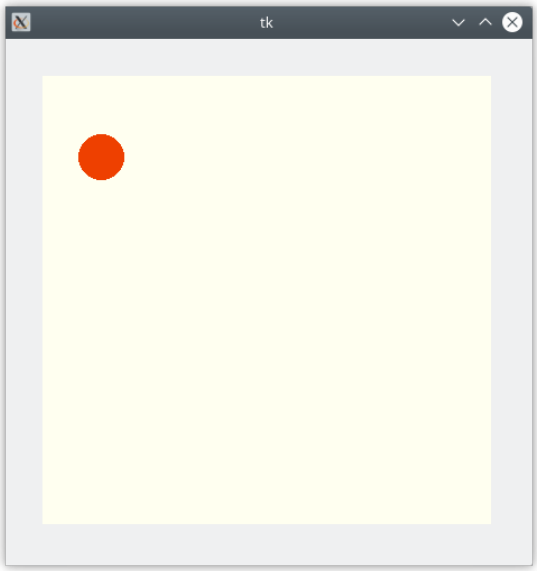
— Lignes 4-8 : la fonction d'animation : elle efface tout (ligne 6) et crée le nombre suivant du compteur. La foncion est relancée toutes les demi secondes (ligne 8). Le rappel provoque la création d'un identifiant (ligne 8). Pour pouvoir agir sur cet identifiant, il est placé dans une variable globale (ligne 5).

— Ligne 18 : Création d'un bouton. Lorsqu'on clique sur ce bouton, il lance la fonction go.
— Lignes 10-12 : le clic sur le bouton entraîne l'annulation de l'animation en cours : la méthode d'annulation `after_cancel` (ligne 11) est appelée. Puis une animation est aussitôt relancée (ligne 12).

Balle rebondissante

Voici une animation classique :





une balle est lancée dans un rectangle et quand elle touche un mur, elle rebondit et repart en position symétrique.

Voici le code correspondant :

```
1  from tkinter import *
2
3  SIDE=400
4  PAD=SIDE//12
5
6  def move(dx, dy):
7      ulx, uly, lrx, lry = list(map(int, cvs.coords(ball)))
8      # bords verticaux
9      if ulx <=0 or lrx >=SIDE:
10         dx=-dx
11     # bords horizontaux
12     elif uly<=0 or lry>=SIDE:
13         dy=-dy
14     cvs.move(ball, dx, dy)
15     cvs.after(30, move, dx, dy)
16
17 # Création d'un canevas
18 w=Tk()
19 cvs=Canvas(w, width=SIDE, height=SIDE, highlightthickness=0,
20            bg="ivory")
21 cvs.pack(padx=PAD, pady=PAD)
22
23 # Création d'une balle
24 n=20
25 R=SIDE//n
26 x0=200
27 y0=40
28 ball=cvs.create_oval(x0, y0, 2*R+x0, 2*R+y0, outline='OrangeRed2',
29                       fill='OrangeRed2')
30
31 # Lancement de l'animation
32 move(4, 6)
33 w.mainloop()
```

— Lignes 18-21 : création du rectangle (un canevas en fait)

— Lignes 24-29 : création de la balle rebondissante. La balle a une id (`ball`, ligne 28) qui sert pour détecter la position de la balle sur le canevas (ligne 7) et la déplacer (ligne 14)

— Ligne 25 : le rayon de la balle qui s'ajuste en fonction de la taille du canevas pour garder des proportions correctes.

— Lignes 26-27 : le point où la balle est lâchée.

— Lignes 6-15 : la fonction d'animation

— Ligne 32 : le lancement de l'animation.

— Ligne 6 : la fonction d'animation déplace la balle toujours de la même façon : 4 pixels horizontalement, 6 pixels verticalement (cf. ligne 32).

— Ligne 15 : l'animation est relancée toutes les 30 millisecondes.

— Ligne 14 : le déplacement de la balle après rectification éventuelle si la balle a touché le mur.

— Lignes 8-13 : gestion des collisions avec les murs. Quand la balle touche un mur, le sens de son mouvement est inversé (ce qui explique pourquoi `dx` ou `dy` est changé en son opposé aux lignes 10 et 13)
— Ligne 7 : la balle a une id (créée ligne 28). Grâce à cette id, le canevas est en mesure de fournir les quatre coins du carré qui encadre la balle. C'est ainsi que l'on sait si la balle est en train de rebondir ou pas (lignes 9 et 12).

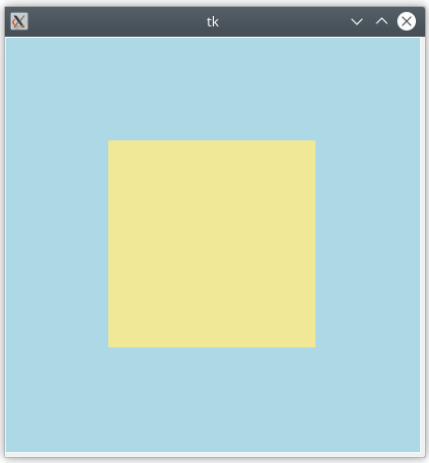
Pour modifier la vitesse de la balle, on peut jouer sur :

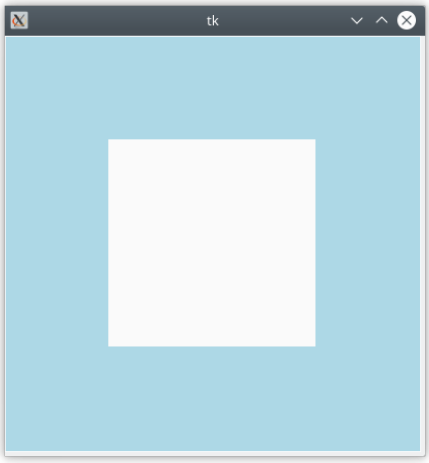
— le délai de rafraîchissement (ligne 15)
— l'amplitude de déplacement (ligne 32).

Certains choix de ces deux paramètres peuvent produire des animations disgrâcieuses.

Effet de fading

Un effet de fading (atténuation progressive d'une couleur) est obtenu en faisant une animation. On passe graduellement (ici en deux secondes et 10 étapes) d'un carré jaune à un carré blanc :





Le code correspondant est :

```
1  from tkinter import *
2
3  SIDE=400
4
5  root = Tk()
6  cnv = Canvas(root, width=SIDE, height=SIDE,
7               bg='light blue')
8  cnv.pack()
9
10 def fading_color(r,g,b, N):
11     return (255-r)//N,(255-g)//N,  (255-b)//N
12
13 def fading(rect, r, g, b, N):
14     if N<=0:
15         return
16     s=cnv.itemcget(rect,"fill")[1:]
17     R,G,B=[int(''.join(u),base=16) for u in zip(s[0::2],s[1::2])]
18     rr=str(hex(min(R+r, 255)))[2:]
19     gg=str(hex(min(G+g, 255)))[2:]
20     bb=str(hex(min(B+b, 255)))[2:]
21     color="#"+str(rr+gg+bb)
22     cnv.delete(rect)
23     rect=cnv.create_rectangle(100, 100, 300, 300,
24                               fill=color, outline='')
25     cnv.after(200, fading, rect, r, g, b, N-1)
26
27 rect= cnv.create_rectangle(100, 100, 300, 300,
28                            fill='#f0e68c', outline='')
29
30 r, g, b=fading_color(0xf0,0xe6, 0x8c, 10)
31 fading(rect, r, g, b, 10)
```

— Lignes 13-23 : la fonction d'animation.

— Ligne 30 : on calcule au préalalble trois tranches `r`, `g` et `b` de composantes RGB que l'on va additionner (lignes 18-20) à la couleur courante à chaque étape de l'animation.

— Ligne 25 : l'animation est relancée tous les 200 ms.

— Ligne 16 : on lit les couleurs courantes du rectangle en accédant aux options de l'item grâce à son id (ici `rect`) qui est un paramètre de la fonction `fading`.

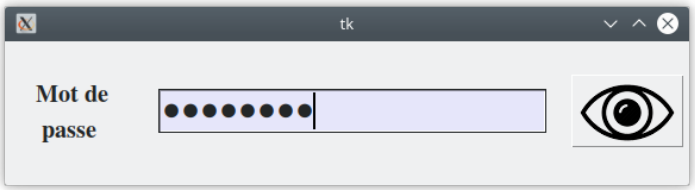
— Ligne 22-24 : on efface le carré courant et on le remplace par le carré avec la nouvelle couleur.

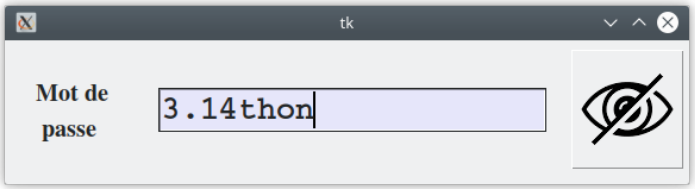
Chapitre VI

Illustrations

Voir/cacher un mot de passe

Comme dans l'enregistreur de mots de passe de Google Chrome, on souhaite réaliser une mini-application :





qui affiche ou cache un mot de passe dans un champ si on clique sur le bouton.
Pour les icones *masquer* et *démasquer* sur le bouton, il faut disposer des deux images `view.png` et `hide.png` :

Voici le code correspondant :

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

lbl = Label(root, text="Mot de\npasse ", font='Times 15 bold')
lbl.pack(side='left', padx=20)

my_entry = Entry(
    root, font='Courier 20 bold', width=20, bg='lavender', show='')

my_entry.pack(padx=20, pady=40, side="left")
my_entry.focus_set()


def update_entry():
    global hidden
    if hidden:
        my_entry['show'] = ''
        btn['image'] = hide
    else:
        my_entry['show'] = ''
        btn['image'] = view
    hidden = not hidden


hidden = True

hide = PhotoImage(file='hide.png')
view = PhotoImage(file='view.png')

btn = Button(
    root,
    image=hide,
    width=90,
    font='Times 15 bold',
    command=update_entry)
btn.pack(side="left")
root.mainloop()
```

Il faut créer trois widgets :

— ligne 7 : une entrée dans laquelle est écrit le mot de passe
— ligne 30 : un bouton qui va démasquer/masquer le mot de passe
— ligne 4 : un label pour indiquer qu'il faut entrer un mot de passe.

Lorsque le mot de passe est caché, chaque caractère est remplacé par un disque noir ●. Pour que l'écriture du mot de passe ne montre que le caractère ●, il faut activer l'option `show` de `Entry` en posant `show="●"`, cf. ligne 8. On peut écrire tel quel le caractère entre guillemets, par exemple en faisant un copier-coller. Ce caractère est connu sous le nom Unicode de Black Circle. En Python 3, on peut aussi produire ce caractère avec une séquence d'échappement unicode :

```
print("\u25CF")
```

L'option `command` du bouton (ligne 35) référence une fonction `update_entry` (ligne 14) qui va se charger :

— de masquer/démasquer l'entrée (lignes 20 et 17)
— de changer l'icône du bouton (lignes 18 et 21).

L'état de visibilité du mot de passe est enregistré dans une variable globale `hidden` (ligne 25) qui vaut initialement `False` afin que le mot de passe soit invisible. Comme cette variable est **modifiée** par la fonction `update_entry` (ligne 22), elle y est déclarée `global` (ligne 15).

Pour cacher chaque caractère de l'entrée, on a vu qu'il fallait que l'option `show` du bouton référence la chaîne qui va remplacer chaque caractère. Pour que les caractères soient normalement visible, il faut que cette option soit à la chaine vide `""`. Comme on le ferait pour toute option de widget, on peut changer l'option `show` avec une affectation de `my_entry["show"]` (lignes 17 et 20).

Pour changer l'image du bouton, c'est le même principe, on réaffecte `btn["image"]` (lignes 18 et 21). L'image doit pointer vers un objet de type `PhotoImage` que l'on aura créé au préalable (lignes 27 et 28).

Interface pour vérifier un mot de passe

Voici une interface où un utilisateur entre un mot de passe (caché par des astérisques) et le système lui répond si son login est valide ou pas :

Le code correspondant est :

```
1 from tkinter import *
2
3 def estValide() :
```

```
4     if user.get() == "moi" and mdp.get()=="3.14":
5         label_login["text"]="Login correct"
6     else:
7         label_login["text"]="Login incorrect"
8
9 app = Tk()
10
11 user=StringVar()
12 mdp=StringVar()
13
14 label_user = Label(app,text="Identifiant")
15 label_mdp = Label(app,text="Mot de passe")
16 label_login = Label(app, font="Arial 20 bold")
17
18 btn = Button(app,text="Valider",command=estValide, width=20)
19 entry_user = Entry(app, textvariable=user)
20 entry_mdp = Entry(app, textvariable=mdp, show="*")
21
22
23 label_user.pack()
24 entry_user.pack()
25
26 label_mdp.pack()
27 entry_mdp.pack()
28
29 btn.pack(padx=100)
30 label_login.pack()
31
32 app.mainloop()
```

— Ligne 11-12 : les textes des entrées sont enregistrées dans des variables de contrôle.

— Lignes 3 et 18 : quand l'utilisateur clique sur le bouton, la fonction `estValide` est lancée et elle examine si le login est valide ; la réponse est alors affichée dans un label prévu à cet effet (ligne 16).

En pratique, un mot de passe n'est jamais stocké en clair. Cet exemple est basé en partie sur une question posée sur le forum Python d'OpenClassroom.

De quel bouton vient le clic ?

Imaginons une situation où plusieurs boutons peuvent modifier un widget, comme ci-dessous :

où si on clique sur un des boutons, le numéro du bouton est dessiné sur un canevas. Comme expliqué dans la FAQ pour ISN de developpez.com, une situation typique serait le cas des boutons d'une calculatrice.

Pour réaliser cela, on se dit qu'on aimerait pouvoir ne créer qu' **une seule** fonction de rappel et qu'elle réagisse en fonction du bouton sur lequel on a cliqué. Mais, comme la fonction de rappel d'un bouton ne reçoit aucun argument, on ne peut pas savoir quel bouton a appelé. Par exemple, dans le code ci-dessous où le clic entraîne juste l'affichage d'un message :

```
from tkinter import *
from random import randrange

WIDTH=400
HEIGHT=200
NBUTTONS=5

root = Tk()
cnv = Canvas(root, width=WIDTH, height=HEIGHT, background="ivory")
cnv.grid(row=0,columnspan=NBUTTONS)

def clic():
    print("Clic")

for i in range(NBUTTONS):
    Button(root, text="Bouton\n%s" %i, command=clic).grid(row=1, column=i)

root.mainloop()
```

si on clique sur un des 5 boutons, la fonction `clic` est appelée et rien n'indique dans sa définition qui l'aurait appelée (d'ailleurs, n'importe quelle portion de code pourrait appeler cette fonction).

On va donc créer 5 fonctions de rappel, une par bouton. Pour cela, on peut utiliser une fonction qui génère, pour chaque bouton, des fonctions de rappel. Voici un code réalisant cela :

```
from tkinter import *


SIZE=200
```

```
5  NBUTTONS=5
6
7  root = Tk()
8  cnv = Canvas(root, width=SIZE, height=SIZE, background="ivory")
9  cnv.grid(row=0,columnspan=NBUTTONS)
10
11 def make_clic(i):
12     def clic():
13         cnv.delete(ALL)
14         cnv.create_text(SIZE/2, SIZE/2, text=i, font="Arial 90 bold")
15     return clic
16
17 for i in range(NBUTTONS):
18     btni=Button(root, text=i, command=make_clic(i))
19     btni.grid(row=1, column=i)
20
21 root.mainloop()
```

— Lignes 17-19 : dans une boucle parcourue par un indice `i`, on génère le bouton portant la valeur `i`. Non seulement le texte dépend de `i` mais aussi la façon de réagir du bouton. En effet, chaque commande est le retour d'un appel de fonction qui renvoie une fonction mémorisant i.

— Lignes 11-15 : cette fonction génère une fonction de rappel personnalisée pour chaque bouton. Cette fonction est appelée avec un numéro de bouton. Pour chaque numéro, elle crée une fonction (lignes 12-14) qui réagit au clic sur le bouton `i`. Cette fonction est créée dans le corps de la fonction et est renvoyée (ligne 15) après sa création. Cette fonction, nommée `clic` mémorise un contexte, en particulier le numéro `i`.

— Ligne 11 : une telle fonction est appelée une clôture (**closure**).

Alternative avec paramètre par défaut

J'ai vu cette autre façon de faire dans un message de RedTenZ sur le forum Python d'OpenClassrooms. Voici l'adaptation de son code :

```
from tkinter import *

SIZE=200
NBUTTONS=5

root = Tk()
cnv = Canvas(root, width=SIZE, height=SIZE, background="ivory")
cnv.grid(row=0,columnspan=NBUTTONS)

for i in range(NBUTTONS):
    def clic(i=i):
        cnv.delete(ALL)
        cnv.create_text(SIZE/2, SIZE/2, text=i, font="Arial 90 bold")
    btni=Button(root, text=i, command=clic)
    btni.grid(row=1, column=i)
```

```
root.mainloop()
```

Toute l'astuce est dans le paramètre par défaut `i` dans la définiton de `clic` (l'auteur avait utilisé une fonction `lambda` qui risque d'être peu lisible dans le code ci-dessus). Cette méthode est signalée dans la documentation de Shipman au §54.7.

Alternative identifiant le widget

Une autre façon de faire est d'associer avec `bind` chaque bouton au clic de souris sur le bouton à une même fonction (ci-dessous `clic`). Cette fonction sera appelée lorsqu'on cliquera sur un des boutons en générant un événement, disons `event`. Et `event` permet d'identifier le widget associé par l'attribut `event.widget`. Il ne reste plus qu'à récupérer le texte du bouton via `event.widget["text"]` pour identifier la valeur. D'où le code :

```
from tkinter import *

SIZE=200
NBUTTONS=5

root = Tk()
cnv = Canvas(root, width=SIZE, height=SIZE, background="ivory")
cnv.grid(row=0,columnspan=NBUTTONS)

def clic(event):
    i=event.widget["text"]
    cnv.delete(ALL)
    cnv.create_text(SIZE/2, SIZE/2, text=i, font="Arial 90 bold")


for i in range(NBUTTONS):
    btni=Button(root, text=i)
    btni.grid(row=1, column=i)
    btni.bind("<Button>", clic)

root.mainloop()
```

Alternative utilisant `partial`

Si on veut utiliser la fonction de rappel suivante qui affiche un numéro sur le canevas :

```
def clic(nro):
    cnv.delete(ALL)
    cnv.create_text(SIZE/2, SIZE/2, text=nro, font="Arial 90 bold")
```

on a le problème que cette fonction prend déjà un paramètre alors que la fontion passée à `command` dans un bouton n'en prend pas. On peut toutefois transmettre l'argument en utilisant la fonction `partial` du module standard `functools`, ce qui donne le code suivant :

```
from tkinter import *
from functools import partial

SIZE=200
```

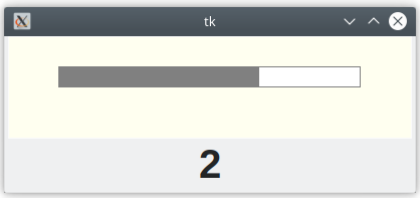
```
NBUTTONS=5

root = Tk()
cnv = Canvas(root, width=SIZE, height=SIZE, background="ivory")
cnv.grid(row=0,columnspan=NBUTTONS)

def clic(nro):
    cnv.delete(ALL)
    cnv.create_text(SIZE/2, SIZE/2, text=nro, font="Arial 90 bold")


for i in range(NBUTTONS):
    btni=Button(root, text=i, command=partial(clic,i))
    btni.grid(row=1, column=i)

root.mainloop()
```

J'ai vu cette méthode indiquée dans un message de LeCobriste128 sur le forum Python d'OpenClassrooms.

Plus généralement, on pourra consulter cette discussion sur Stack Overflow.

Construire sa propre barre de progression

Tkinter ne propose pas de barre de progression, pour cela il faut utiliser son extension (standard) Ttk. Néanmoins, la barre de Ttk n'est pas facilement personnalisable (il faut utiliser un style Ttk et cela n'empêche pas certaines limitations). On peut donc être tenté de créer sa propre barre de progression. Pour cela, on utilise un canevas et, périodiquement, on modifie la longueur d'un rectangle gris emboîté dans un rectangle blanc :



Voici une base de code :

```
1 from tkinter import *
2
3 DELAY = 100
4
5 root = Tk()
6
7 # Le canevas
8 cnv = Canvas(root, width=400, height=100, bg='ivory')
9 cnv.pack()
```

```
10
11 # Label
12 message = StringVar()
13 w = Label(root, textvariable=message, font='Arial 30 bold')
14 w.pack()
15
16 # Largeur de la barre
17 W = 300
18
19 # Durées
20 period_s = 10
21 period_ms = period_s * 1000
22 DELAY_BAR = int(round(period_ms / W))
23
24 # Les deux rectangles
25 A = (50, 50)
26 B = (50 + W , 30)
27 bg = cnv.create_rectangle(A, B, outline="gray", fill="white")
28 bar = cnv.create_rectangle(A, B, outline="gray", fill="gray", width=0)
29
30
31 def animate(L, bar):
32     if L >= 0:
33         cnv.delete(bar)
34         newbar = cnv.create_rectangle(
35             50, 50, 50 + L, 30, outline="gray", fill="gray", width=0)
36         L -= 1
37         cnv.after(DELAY_BAR, animate, L, newbar)
38
39
40 def chrono(s, message):
41     if s >= 0:
42         message.set(str(s))
43         root.after(1000, chrono, s - 1, message)
44
45
46 animate(W, bar)
47 chrono(period_s, message)
48
49 root.mainloop()
```

— Lignes 27-28 : Le rectangle gris fait office de barre de progression. Quand il évolue, le fond blanc de la barre se découvre.

— Ligne 22 : on précalcule la durée de rafraîchissement du rectangle pour 1 pixel.

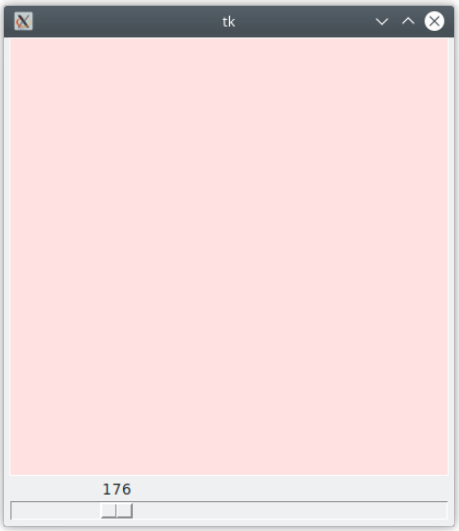
— Lignes 31-37 : on fait évoluer la barre pixel par pixel (la longueur L). Pour cela, on remplace le rectangle gris par un rectangle ayant 1 pixel de moins de longueur.

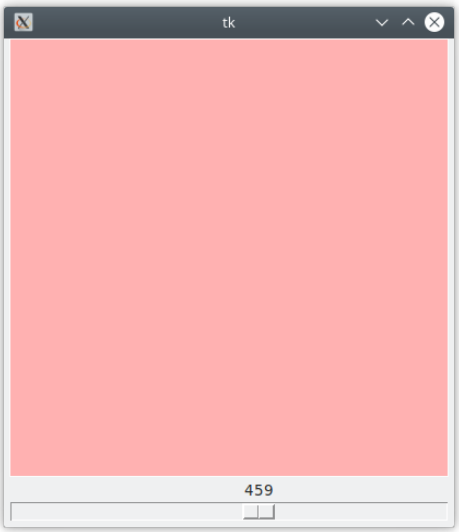
— Lignes 40-43 : on fait évoluer le compteur seconde par seconde jusqu'à écoulement de la période donnée initialement (`period_s`, ligne 20).

Dessiner un dégradé

Par défaut, Tkinter ne permet pas de créer des gradients. Toutefois, on peut réaliser une forme de gradient en écrivant soi-même le code.

Voici un exemple avec un gradient rouge :






```
from tkinter import *

LENGTH = 800
TOP_RED = 120

root = Tk()
cnv = Canvas(root, width=400, height=400, bg="white")
```

```
cnv.pack()


def rgb_10to16(r, g, b):
    return ("#%.02x" % r) + ("%.02x" % g) + ("%.02x" % b)


def gradient(x):
    y = int((-255 + TOP_RED) / LENGTH * (int(x)) + 255)
    cnv["bg"] = rgb_10to16(255, y, y)


curseur = Scale(
    root,
    orient="horizontal",
    command=gradient,
    length=400,
    from_=0,
    to=LENGTH)
curseur.pack()

root.mainloop()
```

Voici un exemple avec des niveaux de gris

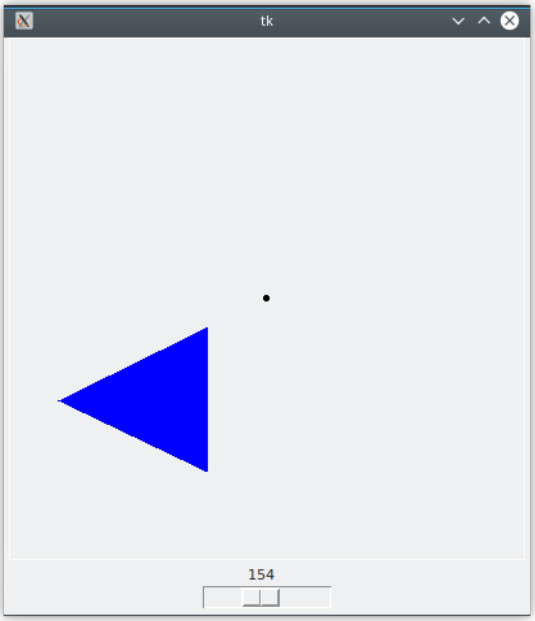
```
from tkinter import Tk, Canvas

SIDE=600

root=Tk()
cnv=Canvas(root, width=SIDE, height=SIDE*0.3, background="ivory")
cnv.pack()
top=50
H=100

a=10
N=100
W=0.95*SIDE//N

gris0=40
gris1=200
h=(gris1-gris0)//N


def toHex(v):
    z=hex(v)[2:]
    return '0'*(len(z)==1)+z
```

```
for i in range(N):
    color=toHex(gris1-i*h)
    cnv.create_rectangle(a, top, a+W, top+H, outline='',
                         fill="#%s%s%s" %(color, color, color))
    a+=W

root.mainloop()
```

ce qui produit :

Rotation d'un item

Ce qui suit va montrer comment faire tourner un item géométrique du canevas autour d'un point d'un certain angle :



On utilise pour cela la rotation vectorielle fournie par la classe `Vec2D` du module Turtle. Voici le code :

```
1  from tkinter import *
2  from turtle import Vec2D
3
4  def rot(C, t, M):
5      CM=Vec2D(*M)-Vec2D(*C)
6      return Vec2D(*C)+CM.rotate(t)
7
8  root = Tk()
9  cnv = Canvas(root, width=400, height=400)
10 cnv.pack()
11
12 # Le centre
13 C=(200, 200)
14
15 # Un triangle
16 P=(250, 200)
17 Q=(300, 100)
18 R=(380, 200)
19 tr=cnv.create_polygon(P, Q, R, width=8, fill="blue")
20
21 cnv.create_oval(200-2, 200-2, 200+2, 200+2, fill="black")
22
23 def rotate(t):
24     PP=rot(C, float(t), P)
25     QQ=rot(C, float(t), Q)
26     RR=rot(C, float(t), R)
27     cnv.coords(tr, *PP, *QQ, *RR)
28
29 curseur = Scale(root, orient = "horizontal", command=rotate, from_=0, to=360)
30 curseur.pack()
31
32 root.mainloop()
```

— Lignes 4-6 : la fonction `rot` calcule les coordonnées du transformé de `M` par la rotation de centre `C` et d'angle `t`.

— Lignes 13 : on crée un centre de rotation `C` et (ligne 21) on le marque avec un petit disque de 2 pixels de rayon.

— Lignes 16-19 : on crée un motif triangulaire `PQR` qui va tourner autour du centre.

— Ligne 29 : un curseur permettant de choisir l'angle de rotation (entre 0 et 360°) autour de `C` par rapport à la position initiale.

— ligne 23 : chaque fois que le curseur est déplacé (`command` à la ligne 29), le triangle doit tourner depuis sa position au lancement de l'application d'un certain angle `t`. On calcule la nouvelle position PP, QQ et RR de chacun des sommets (ligne 24-26) et on déplace le triangle avec la méthode `coords` à sa nouvelle position (ligne 27).

Le code ci-dessus est adapté d'un message du forum Python d'OpenClassroom.