12. Interfaces graphiques avec Tkinter

Jusqu'à présent, vos programmes Python interagissaient avec l'utilisateur uniquement via la console (par exemple grâce aux affichages produits par la fonction `print`). Dans ce chapitre, nous allons apprendre à réaliser des *interfaces graphiques* (en anglais *GUI* : *graphic user interface*) tels que celui illustré sur la figure 12.1. Pour cela, nous allons utiliser `tkinter`, l'un des modules (mais pas le seul) permettant de créer des interfaces graphiques avec Python.

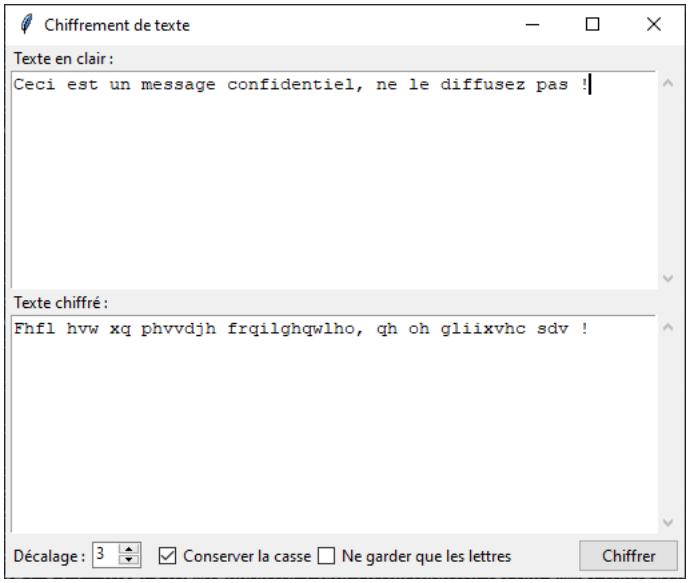


FIGURE 12.1. – Un exemple de programme avec interface graphique.

À tester. Pour vérifier le bon fonctionnement de `tkinter` et voir un exemple basique d'interface graphique, exécutez les deux instructions suivantes :

```
import tkinter
tkinter._test()
```

12.1. Introduction

12.1.1. Fonctionnement interne

Lors de son fonctionnement, un programme qui propose une interface graphique communique en permanence avec le système d'exploitation, afin de réagir notamment aux actions de son utilisateur. Généralement, une boucle tourne durant toute la durée du programme afin de répéter en permanence les tâches suivantes.

1. Le programme récupère auprès du système d'exploitation toutes les actions produite par l'utilisateur depuis l'itération précédente : mouvement de la souris, appui sur ou relâchement d'une touche du clavier, etc.
2. Le programme analyse ces actions en fonction du contenu de l'interface graphique, pour voir si celles-ci correspondent à des événements particuliers. (Par exemple, en cas de « clic », le curseur se situait-il sur un bouton ou un autre élément particulier de l'interface ? En cas d'appui sur une touche du clavier,

est-ce que l'utilisateur était actuellement dans un champ permettant de saisir du texte ? Ou bien a-t-il produit un raccourci clavier particulier ?)

3. Suite à ça et si besoin, le programme met à jour ses variables ou appelle certaines fonctions en réponse aux événements identifiés précédemment.
4. Si l'apparence de l'interface a changé (un bouton devient enfoncé, une case devient cochée, un champ de texte change de contenu, …) alors le programme demande au système d'exploitation de re-dessiner la fenêtre à l'écran en prenant en compte ces changements.

Ces opérations peuvent être répétées plusieurs dizaines de fois par seconde (voire plus pour certain programmes), ce qui garantit que ce fonctionnement est invisible aux yeux de l'utilisateur.

La bonne nouvelle est que le module `tkinter` se charge automatiquement d'effectuer ces opérations. Pour créer une interface graphique avec ce module, il suffit dans votre script d'indiquer quels sont les éléments qui composent votre interface (textes, boutons, cases à cocher, …), leur disposition dans la fenêtre, et les fonctions qui seront appelées lors des différents événements associés. Toute la boucle décrite ci-dessus sera alors cachée dans une méthode nommée `mainloop()` dont vous n'avez pas besoin de connaître le fonctionnement précis. Voici par exemple un code minimaliste qui crée une fenêtre avec `tkinter` et lance la boucle permettant de faire tourner l'interface graphique ainsi créée.

```
1  import tkinter as tk
2
3  window = tk.Tk()
4  window.title("Mon premier GUI")
5  window.geometry("640x480")
6
7  print("Démarrage de la boucle principale")
8  window.mainloop()
9
10 print("Fin du programme")
```

Analysons ce script.

— Ligne 1, on importe le module `tkinter` en lui donnant l'alias `tk` (voir chapitre 5).

— Ligne 3, on utilise la fonction `Tk()` pour créer la fenêtre principale du programme (parfois aussi appelée racine).

— Ligne 4, on modifie le titre de la fenêtre (affiché en haut de la fenêtre mais aussi à d'autres endroits par le système d'exploitation).

— Ligne 5, on impose à la fenêtre une taille initiale de 640 de largeur par 480 de hauteur (en pixels).

Il faut bien comprendre que jusque là, nous avons simplement *initialisé* la fenêtre en donnant ses propriétés, rien de plus. C'est une fois la ligne 8 atteinte que la fenêtre principale « prend vie » grâce à sa méthode `mainloop()`. Cette fonction lance la boucle principale décrite quelques paragraphes plus haut, qui va gérer le comportement de votre fenêtre en réaction aux manipulations de l'utilisateur. Vous pouvez vérifier que vous pouvez déjà déplacer, redimensionner et fermer votre fenêtre sans écrire une seule ligne de code supplémentaire. Il faut bien comprendre que cette boucle, et donc l'exécution de la fonction `mainloop()`, **ne se termine que lorsque la fenêtre principale est fermée**. Vous pouvez vous en convaincre en vérifiant que l'instruction `print` ligne 10 n'est traitée qu'une fois la fenêtre fermée.

Attention. Cela signifie que lorsque vous créez une interface graphique avec `tkinter`, tout son contenu et son comportement doivent être complètement décrits *avant* l'appel à `mainloop()`. Les instructions qui sont placées *après* l'appel à `mainloop()` ne seront exécutées qu'une fois que la fenêtre sera fermée et n'existera plus.

12.1.2. Interfaces modernes avec `tkinter`

Le module `tkinter` se base sur la bibliothèque *Tk* créée en 1988, soit environ trois ans avant la première version publique de Python. En effet, la bibliothèque *Tk* a été originellement conçue pour faire des interfaces graphiques avec un langage de script plus ancien : Tcl. Au fil du temps, elle a été également adaptée à d'autres langages de scripts (comme Ruby, Lua ou encore OCaml). Lors du développement de Python, le choix a été fait d'utiliser également cette bibliothèque comme outil par défaut pour la création d'interfaces graphiques, et un module standard est donc né pour faire le lien entre Python et *Tk* : `tkinter` (pour *Tk interface*).

De 1988 à nos jours, l'apparence des systèmes d'exploitation a énormément évolué. La bibliothèque *Tk* s'est adaptée à cette évolution, avec notamment une grosse mise à jour fin 2007. La version 8.5 de *Tk* sortie cette année-là propose notamment une refonte graphique des composants principaux afin qu'ils s'intégrent mieux dans les environnements de bureau modernes. Toutefois pour des raisons de rétro-compatibilité, ce sont les anciens styles de composants qui restent utilisés par défaut. Si l'on veut utiliser les versions modernes de ceux-ci, il faut aller les chercher explicitement dans le sous-module `tkinter.ttk` (*themed tk*). Pour comparaison, la figure 12.2 donne deux apparences possibles d'un même programme suivant le type de composants utilisé (d'un côté ceux d'origine, de l'autre leurs versions modernes fournies par `ttk`). Dans la section dédiée, on précisera pour chaque type de composant présenté si celui-ci possède une version moderne dans le module `tkinter.ttk` ou non.

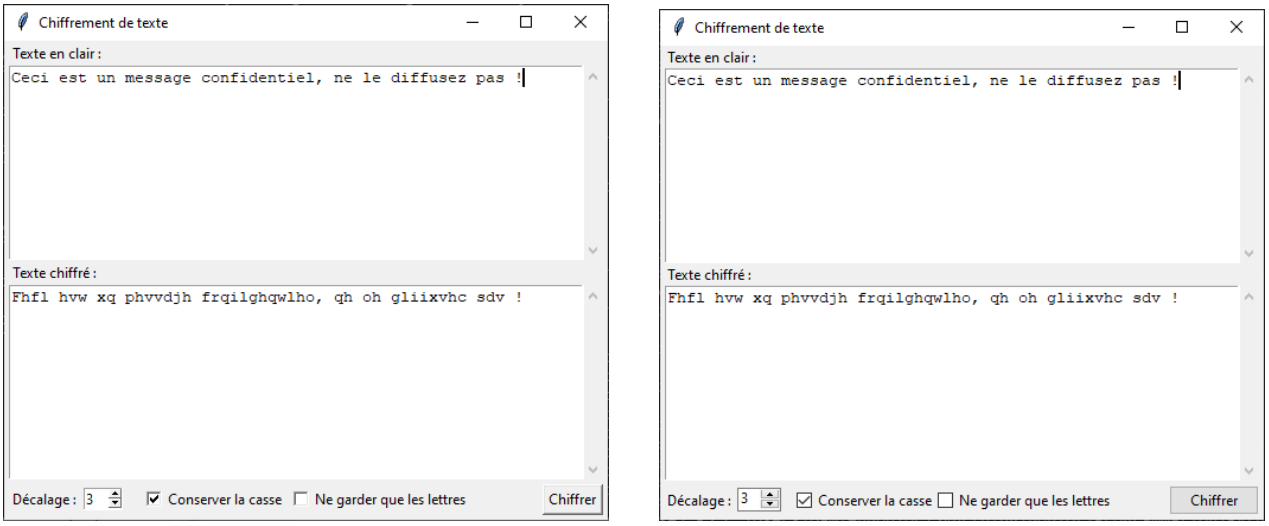


FIGURE 12.2. – Apparence sous Windows 10 d'un même programme Tkinter, à gauche avec les composants d'origine, à droite avec les nouveaux composants (ttk).

Sur Internet se trouvent énormément de ressources concernant Tkinter : attention à certains tutoriels qui peuvent être obsolètes ou proposer de vieilles pratiques inutilement complexes. En ressource complémentaire, nous vous conseillons l'ouvrage *Modern Tkinter for Busy Python Developers* de Mark ROSEMAN, disponible gratuitement en ligne [1] et sur lequel se basent certaines sections de ce cours.

1. `https://tkdocs.com/tutorial/index.html`

12.2. L'organisation de Tkinter

12.2.1. Les widgets

Pour créer une interface graphique avec Tkinter, il vous faut tout d'abord créer une fenêtre principale avec la fonction `Tk` comme vous l'avez vu dans la section précédente. Les composants que vous rajouterez ensuite à cette fenêtre sont appelés des *widgets* (contraction de *window gadget*). La particularité d'un widget est qu'il ne peut pas exister seul : il est forcément rattaché à un élément *parent*, qui peut être soit une fenêtre, soit un autre widget. Pour créer un widget avec `tkinter`, le procédé est toujours le même : on appelle la fonction « *constructeur* » de ce widget en lui donnant en paramètre le parent de l'élément que l'on veut créer, et d'éventuelles *options* servant à configurer ce widget (son apparence, son contenu, son comportement …). Ces constructeurs portent généralement des noms assez explicites, par exemple :

— la fonction `Button` sert à créer un bouton,

— la fonction `Label` sert à créer une *étiquette*, c'est à dire un simple texte à afficher,

— la fonction `Checkbutton` sert à créer une case à cocher …

Le code ci-dessous crée une fenêtre à laquelle on a rattaché deux widgets : une étiquette contenant le texte « Test », et un bouton contenant le texte « Cliquez ici ».

```
1 import tkinter as tk
2
3 window = tk.Tk()
4 widget1 = tk.Label(window, text="Test")
5 widget2 = tk.Button(window, text="Cliquez ici")
```

Remarque. Dans cet exemple on est allé chercher les widgets *Label* et *Button* dans le module `tkinter` pour simplifier le code. Mais en réalité, ces deux widgets existent en versions plus modernes dans le module `tkinter.ttk` et il aurait alors été préférable d'utiliser ce dernier, avec un code de la forme :

```
import tkinter as tk
import tkinter.ttk as ttk

window = tk.Tk()
widget1 = ttk.Label(window, text="Test")
widget2 = ttk.Button(window, text="Cliquez ici")
```

Attention toutefois, si vous ajoutez l'instruction `window.mainloop()` à la suite du code précédent et exécutez le tout, vous constaterez que la fenêtre que vous avez créée est encore vide ! C'est parce que créer un widget rattaché à une fenêtre ne suffit pas, il faut ensuite expliquer où le *placer*. La façon de placer les widgets fera l'objet de la sous-section précédente, en attendant nous admettrons que l'on peut par exemple appeler la méthode pack sans argument. Pour faire apparaître les widgets créer précédemment, nous allons donc rajouter les instructions suivantes :

```
6 widget1.pack()
7 widget2.pack()
8
9 window.mainloop()
```

Attention. On rappelle que, comme toute instruction visant à décrire le contenu de la fenêtre, il faut ajouter les widgets *avant* d'appeler la fonction `mainloop()`.

Remarque. Remarquez que, bien que le bouton n'ait pas encore d'effet particulier dans le programme, il réagit déjà visuellement aux clics de l'utilisateur. C'est le code implémenté dans la fonction `mainloop()` qui gère ce comportement, vous n'avez pas à vous en occuper.

Il est possible d'obtenir ou de changer les options de configuration d'un widget après l'avoir créé. Cela se fait avec la même syntaxe que pour accéder aux éléments d'un dictionnaire, en utilisant comme « clé » le nom de l'option en question.

À tester. Relancez le programme créé dans cette sous-section, en rajoutant au préalable l'instruction suivante n'importe où entre les lignes 4 (création du premier widget) et 9 (démarrage de la boucle principale) l'instruction :

```
widget1["text"] = "Bonjour"
```

12.2.2. Les gestionnaires de géométrie

Tkinter dispose de plusieurs *gestionnaire de géométrie*, c'est-à-dire de façons de placer les widgets dans leurs parents. Nous présentons dans ce cours les deux plus utilisés.

Empilement *L'empilement* sert à placer les widgets les uns sur les autres en prenant comme référence l'un des bords du parents. Cela se fait avec la méthode pack, qui prend notamment comme argument facultatif `side` le bord sur lequel on veut empiler les objets. Le bord choisi est désigné par les chaînes de caractères `"top"`, `"bottom"`, `"left"` et `"right"`. Si l'argument `side` n'est pas donné, il vaut `"top"` par défaut.

À tester. Exécutez le code suivant.

```
1  import tkinter as tk
2
3  window = tk.Tk()
4  window.geometry("320x240")
5  widget1 = tk.Label(window, text="Test")
6  widget2 = tk.Button(window, text="Cliquez ici")
7
8  widget1.pack(side="top")
9  widget2.pack(side="top")
10
11 window.mainloop()
```

Essayez ensuite de remplacer la valeur `"top"` à la ligne 8 ou 9 par `"bottom"`, `"left"` ou `"right"`, et observez à chaque fois le résultat obtenu. Répondez ensuite aux questions suivantes :

1. peut-on empiler les deux widgets chacun sur un bord différent de la fenêtre ?
2. lorsqu'on empile les deux widgets sur un même bord de la fenêtre, que se passe-t-il si on inverse l'ordre des lignes 8 et 9 ?

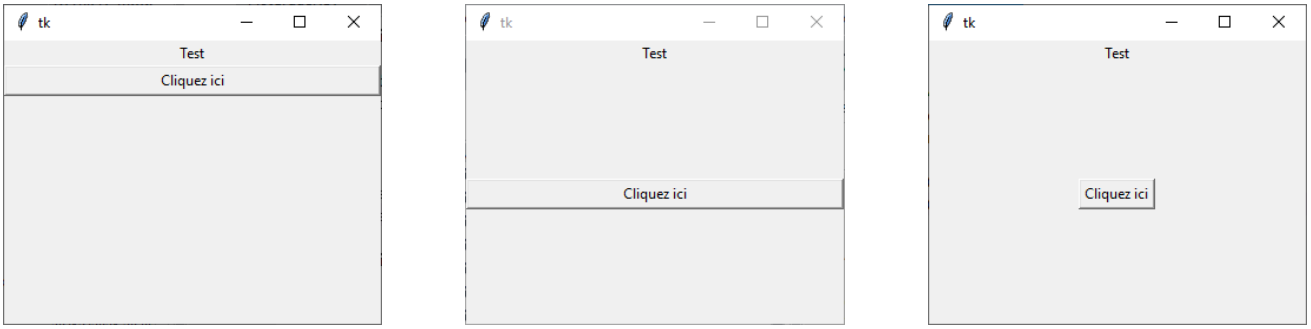


FIGURE 12.3. – Effets des arguments `fill` et expand de la méthode pack.

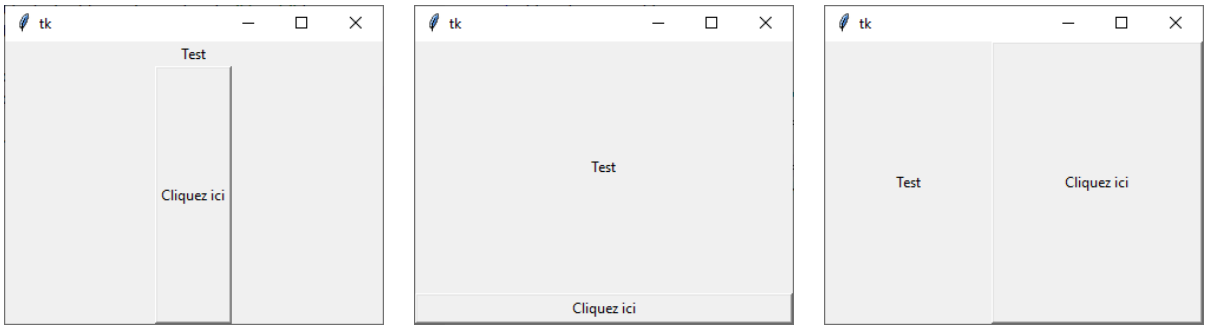
La méthode pack accepte en outre les deux arguments facultatifs suivants :

— `fill` pour indiquer si le widget peut s'étirer horizontalement (argument `"x"`), verticalement (`"y"`) ou dans les deux directions `"both"`),

— expand pour indiquer (via un booléen) si le widget doit occuper autant de place que possible dans l'espace dans lequel il est placé ou non.

L'apparence du widget peut dépendre de la combinaison choisie : si on choisit `expand=True`, c'est la valeur choisie pour `fill` qui déterminera si le widget occupe la place demandée en plaçant de l'espace autour de lui, ou bien en s'étirant. La figure 12.3 illustre les résultats obtenus lorsqu'on ajoute diverses combinaisons de paramètres au code précédent.

Exercice 12.1. En reprenant le code ayant donné la figure 12.3, modifier les arguments des appels à pack de sorte à obtenir les figures suivantes :



Grille On peut aussi décider de placer un widget en imaginant que son parent est « découpé » en un certain nombre de lignes et de colonnes. On passe alors par la méthode `grid` en lui donnant en paramètres :

— `row` : un entier indiquant la ligne dans laquelle placer le widget,

— `column` : un entier indiquant la colonne dans laquelle placer le widget.

La numérotation des lignes et des colonnes commence à partir de zéro.

Le code ci-dessous donne un exemple dans lequel on a disposé quatre boutons suivant une grille constituée de deux lignes et deux colonnes.

```
import tkinter as tk

window = tk.Tk()
window.geometry("320x240")

buttons = dict()
for c in "ABCD":
    buttons[c] = tk.Button(window, text=f"Bouton {c}")

buttons["A"].grid(row=0, column=0)
buttons["B"].grid(row=0, column=1)
buttons["C"].grid(row=1, column=0)
buttons["D"].grid(row=1, column=1)

window.mainloop()
```

Remarque. Ce code illustre aussi le fait que, comme n'importe quel autre objet en Python, un widget peut être stocké dans un dictionnaire, une liste, un tuple …

Par défaut, les lignes et colonnes de la grille ont un *poids* égal à zéro, ce qui signifie qu'elles ne cherchent pas à s'étirer pour occuper l'espace autour. On peut utiliser les fonctions `rowconfigure` et `columnconfigure` pour leur donner un poids (`weight`) strictement positif : elles commenceront à s'étirer, en prenant d'autant plus de place qu'elles ont un poids élevé. Par exemple, ajoutez les instructions ci-dessous avant l'appel à la boucle principale.

```
window.rowconfigure(1, weight=1)
window.columnconfigure(0, weight=1)
window.columnconfigure(1, weight=2)
```

— On a donné un poids de 1 à la ligne d'indice 1, tandis que la ligne d'indice 0 a toujours un poids de zéro par défaut, donc la ligne d'indice 1 occupe tout l'espace disponible verticalement,

— On a donné un poids de 1 à la colonne d'indice 0, et un poids de 2 à la colonne d'indice 1, donc la colonne d'indice 0 occupe un tiers de l'espace horizontal tandis que la colonne 1 en occupe deux tiers.

Cette disposition ne saute pas aux yeux en l'état car par défaut, un widget se place au centre de la case de la grille dans laquelle il a été placé. La figure 12.4 met en évidence ce découpage.

On peut ajuster le positionnement d'un widget au sein de sa cellule en donnant à la méthode `grid` l'argument facultatif `sticky` (littéralement, *collant*). C'est une chaîne de caractères (ou un tuple) formées des lettres `n` (*north*), `s` (*south*), `w` (*west*) et `e` (*east*) représentant les quatre points cardinaux. Chaque lettre qui apparaît indique que le widget doit être « collé » au bord correspondant de sa cellule. Cela permet ainsi de préciser à la fois l'alignement du widget dans sa cellule, mais aussi s'il doit s'étirer ou pas : en effet, si l'on fait apparaître dans l'option `sticky` deux points cardinaux opposés, le widget devra nécessairement s'étirer pour coller simultanément aux deux bords correspondants.

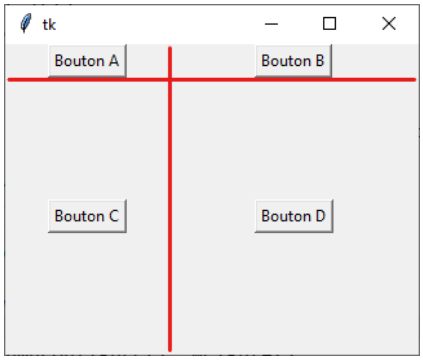


FIGURE 12.4. – Mise en évidence du découpage créé par le gestionnaire `grid`.

À tester. Modifiez le code de l'exemple précédent de sorte à ajouter les paramètres suivants :

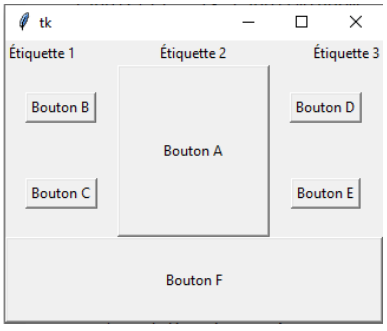
```
buttons["A"].grid(row=0, column=0, sticky="e")
buttons["B"].grid(row=0, column=1, sticky="we")
buttons["C"].grid(row=1, column=0, sticky="nswe")
buttons["D"].grid(row=1, column=1, sticky="se")
```

Enfin, la méthode `grid` accepte aussi les arguments `rowspan` et `columnspan` pour indiquer au widget d'occuper plusieurs lignes et/ou plusieurs colonnes.

À tester. Effectuez maintenant les modifications suivantes et interprétez le résultat obtenu.

```
buttons["A"].grid(row=0, column=0, columnspan=2, sticky="nswe")
buttons["B"].grid(row=0, column=2, rowspan=2, sticky="nswe")
buttons["C"].grid(row=1, column=0, rowspan=2, sticky="nswe")
buttons["D"].grid(row=2, column=1, sticky="nswe")
```

Exercice 12.2. Reproduire l'interface graphique suivante. Le seconde colonne doit occuper deux fois plus de place que les autres colonnes.



Remarque. Un avantage majeur des gestionnaires pack et `grid` présentés dans cette section est qu'ils permettent de créer des dispositions qui s'adaptent automatiquement à la taille de la fenêtre (essayez de la redimensionner pour voir).

12.2.3. La gestion des événements

Comme expliqué en introduction, Tkinter se charge automatiquement de reconnaître des événements-type à partir des informations que lui donne le système d'exploitation (position de la souris, touches enfoncées ou relâchées, etc.). De votre côté, vous devez simplement lui fournir la fonction qui sera appelée lorsqu'un événement précis se produira : cette fonction est généralement appelée *callback*. Il y a deux façons d'associer un callback à un événement dans Tkinter.

1. Certains widgets possèdent un argument `command`, qui prend comme valeur la fonction qui devra être appelée lorsqu'on interagira avec ce widget. C'est le cas par exemple des boutons ou des cases à cocher.

   Attention. Dans ce cas, le callback doit être une fonction qui ne prend pas d'argument.

   À tester. Essayez le programme ci-dessous et expliquez son comportement.

   ```
   import tkinter as tk
   import tkinter.ttk as ttk

   def foo():
       label["text"] = "[" + label["text"] + "]"

   window = tk.Tk()

   label = ttk.Label(window, text="Bonjour")
   button = ttk.Button(window, text="Cliquez ici", command=foo)

   label.pack()
   button.pack()
   window.mainloop()
   ```

   Remarque. En pratique, on choisira bien évidemment des noms de fonctions explicites. Dans l'exemple précédent, la fonction `foo` aurait pu être nommée `on_button_click` ou `add_brackets_to_label` (suivant si vous souhaitez mettre en valeur l'événement auquel elle sera rattachée ou bien l'action qu'elle produira).

   Exercice 12.3. Ajouter à l'exemple précédent un bouton permettant de défaire l'action du bouton déjà présent, c'est-à-dire de retirer une paire de crochets autours du label (uniquement s'il en reste !).

2. Tous les widgets possèdent une méthode `bind` qui permet de relier un événement à son callback. Il suffit de donner en premier argument une chaîne de caractères décrivant un événement (voir ci-après), et en second argument une fonction qui aura été définie au préalable.

   Attention. Dans ce cas, le callback doit être une fonction qui prend exactement un argument. Lorsqu'elle sera appelée, l'argument passé sera un objet de type `Event` contenant des informations sur l'événement qui a déclenché ce callback (par exemple les coordonnées `x` et `y` du curseur au moment de l'événement).

La table 12.1 recense quelques exemples d'événements que l'on peut donner en premier paramètre de la méthode `bind` (mais il en existe bien d'autres).

Événement	**Description**
`"<Enter>"`	Le curseur rentre dans la zone délimitée par le widget.
`"<Leave>"`	Le curseur quitte la zone délimitée par le widget.
`"<1>"`	L'utilisateur clique sur le widget avec le bouton gauche de la souris.
`"<2>"`	L'utilisateur clique sur le widget avec le bouton central de la souris.
`"<3>"`	L'utilisateur clique sur le widget avec le bouton droit de la souris.
`"<Double-1>"`	L'utilisateur double-clique sur le widget avec le bouton gauche de la souris.
On définit de même les événements `"<Double-2>"` *et* `"<Double-3>"`.	
`"<ButtonRelease-1>"`	L'utilisateur relâche le bouton gauche de la souris (après un clic initié sur le widget).
On définit de même les événements `"<ButtonRelease-2>"` *et* `"<ButtonRelease-3>"`.	

TABLE 12.1. – Quelques événements reconnus par la méthode `bind`.

À tester. Essayez le programme ci-dessous et expliquez son comportement.

```
import tkinter as tk
import tkinter.ttk as ttk

def foo(event):
    label["text"] = "Dedans"
    label["background"] = "green"

def bar(event):
    label["text"] = "Dehors"
    label["background"] = "red"

window = tk.Tk()
label = ttk.Label(window, text="Bonjour")
label.pack()

label.bind("<Enter>", foo)
label.bind("<Leave>", bar)

window.mainloop()
```

Exercice 12.4. Créer une interface graphique composée d'une étiquette contenant initialement la valeur « 0 », d'un bouton « Plus » et d'un bouton « Moins ». Faites en sorte qu'à chaque fois que l'on appuie sur l'un de ces deux boutons, cela diminue ou augmente de 1 la valeur affichée dans l'étiquette.

Attention, la propriété `text` *d'une étiquette est une chaîne de caractères. Il faudra effectuer des conversions avant d'effectuer des opérations arithmétiques avec le nombre qu'elle contient.*

12.3. Quelques widgets utiles

12.4. Les cadres : *Frame*

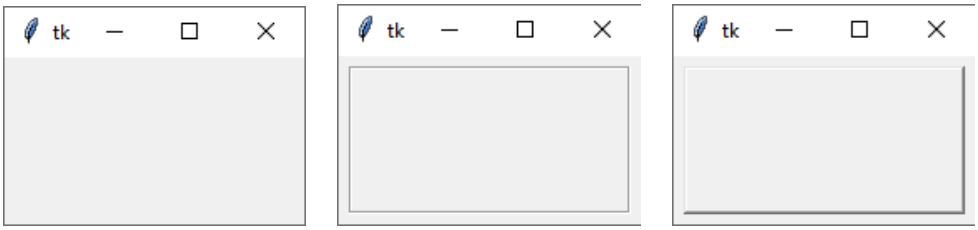
Remarque. Ce widget est disponible en version moderne dans le sous-module `tkinter.ttk`.

Le frame (*cadre* en français) est un widget dont le but est de contenir d'autres widgets. On peut lui affecter une épaisseur et un style de bordure pour bien le délimiter visuellement, ou bien laisser les paramètres par défaut qui le rendent « invisible » au yeux de l'utilisateur.

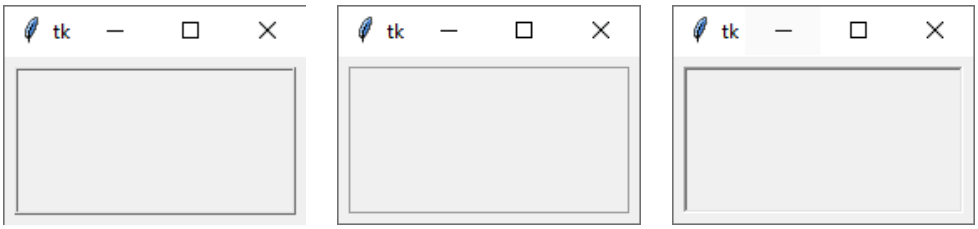
Options utiles.

— `borderwidth` : un entier indiquant l'épaisseur de bordure, en pixels,

— `relief` : le style de bordure, choisi parmi les options ci-dessous,



`flat` (par défaut) `groove` `raised`



`ridge` `solid` `sunken`

— `padding` : la marge (en pixels) à laisser entre le bord du frame et les widgets qu'il contiendra, sous l'une des formes suivantes :

  — un tuple de quatre entiers indiquant la taille des marges à gauche, en haut, à droite et en bas (dans cet ordre),

  — un tuple de deux entiers indiquant la taille des marges horizontales (gauche et droite) et verticales (haut et bas),

  — un unique entier si l'on veut toutes les marges de la même taille.

Les frames permettent de faire en théorie tous les découpages de fenêtre possibles et imaginables. En effet, la façon de placer des widgets dans un frame (`pack` ou `grid`) est totalement indépendante de la façon de placer ce frame lui-même dans son parent, et on peut tout à fait imbriquer les frames les uns dans les autres pour faire des découpages aussi complexes que nécessaire. Le code de la figure 12.5 (page 140) donne un exemple de programme utilisant ce principe de « mise en boîte ». La fenêtre est composée de deux frames empilés verticalement les uns sur les autres, le second s'étirant dans toutes les directions pour occuper tout l'espace disponible. Ensuite, si on regarde plus en détail :

— le premier frame est composé de trois boutons empilés horizontalement à partir du bord gauche,

— le second frame est composé de neuf boutons disposés selon une grille de taille 3×3.

12.4.1. Les étiquettes : *Label*

Remarque. Ce widget est disponible en version moderne dans le sous-module `tkinter.ttk`.

Le label (*étiquette* en français) est un widget dont le but est simplement de contenir du texte [2]. Il y a deux façons de déterminer ce texte :

1. soit en précisant la valeur de l'option `text` lors de la création du label, puis en la redéfinissant à la volée par la suite (cf. section précédente),
2. soit en passant à l'option `textvariable` un objet de type `StringVar`. Il s'agit d'un type implémenté par Tkinter, servant à stocker des « chaînes variables ». Dans ce cas, le label et la chaîne variable seront liés, et modifier le contenu de la chaîne modifiera automatiquement le contenu de tous les widgets qui lui sont associés.

Cette deuxième méthode est un peu plus complexe à mettre en place, mais peut s'avérer utile dans le (rare) cas où plusieurs widgets sont supposés afficher la même chaîne de caractères. Pour créer une chaîne variable, on appelle la fonction `StringVar` en lui donnant en passant en paramètres :

— un widget parent (ça n'est pas nécessairement le widget qui utilisera directement la chaîne variable),

— la valeur de départ de la chaîne.

On peut ensuite récupérer la valeur actuelle de la chaîne variable avec la méthode `get` (sans paramètre), ou bien la modifier avec la méthode `set` (avec en paramètre la nouvelle valeur à donner).

À tester. Analysez puis exécutez le programme ci-dessous.

```
import tkinter as tk
import tkinter.ttk as ttk

def change_text():
    if my_text.get() == "OUI":
        my_text.set("NON")
    else:
        my_text.set("OUI")

window = tk.Tk()

my_text = tk.StringVar(window, "OUI")

btn = ttk.Button(window, textvariable=my_text, command=change_text)
btn.pack()

lbl = ttk.Label(window, textvariable=my_text)
lbl.pack()

window.mainloop()
```

2. … ou éventuellement une image, mais nous n'aborderons pas ce point en cours et laissons les lecteurs curieux se tourner vers des ressources supplémentaires.

12.4.2. Les boutons : *Button*

Remarque. Ce widget est disponible en version moderne dans le sous-module `tkinter.ttk`.

Les boutons sont des widgets prévus pour exécuter une fonction (un *callback*) lorsque l'utilisateur interagit avec.

Options utiles.

— `text` : une chaîne de caractères indiquant le texte affiché sur le bouton (on peut aussi utiliser une chaîne variable en passant par l'option `textvariable` comme pour les labels, mais c'est plus rarement utilisé),

— `command` : la fonction qui sera appelée lorsque l'utilisateur interagira avec le bouton,

Il est également possible de désactiver temporairement un bouton de sorte à ce que l'utilisateur ne puisse plus interagir avec. Cela se fait en ajoutant ou retirant l'attribut `'disabled'` à la liste des *états* du bouton. Concrètement, pour un bouton rattaché à la variable `btn`, cela se fait ainsi :

— `btn.state(['disabled'])` : ajoute le statut *désactivé* au bouton, qui n'est donc plus utilisable,

— `btn.state(['!disabled'])` : retire le statut *désactivé* du bouton, qui redevient donc utilisable.

Remarque. Notez que la fonction `state` ne prend pas directement une chaîne de caractères, mais une liste contenant une ou plusieurs chaînes. C'est parce que l'on pourrait demander à simultanément ajouter ou retirer d'autres statuts au bouton que *désactivé*, mais nous ne rentrerons pas dans les détails dans ce cours.

À tester.

```
import tkinter as tk
import tkinter.ttk as ttk

def set_day():
    btn_day.state(['disabled'])
    btn_night.state(['!disabled'])

def set_night():
    btn_day.state(['!disabled'])
    btn_night.state(['disabled'])

window = tk.Tk()

btn_day = ttk.Button(window, text="Jour", command=set_day)
btn_night = ttk.Button(window, text="Nuit", command=set_night)
btn_night.state(['disabled'])

btn_day.pack()
btn_night.pack()

window.mainloop()
```

Exercice 12.5. Créer une interface graphique contenant les éléments suivants :

— une étiquette contenant initialement le nombre 10,

— un bouton *Action* initialement activé,

— un bouton *Reset* initialement désactivé.

Ces boutons doivent avoir les effets suivants.

— Lorsqu'on clique sur le bouton *Action*, cela diminue de 1 la valeur affichée par l'étiquette. Si la valeur de l'étiquette arrive à zéro, cela doit de plus désactiver le bouton *Action* et réactiver le bouton *Reset*.

— Lorsqu'on clique sur le bouton *Reset*, cela remet la valeur de l'étiquette à 10, cela désactive le bouton *Reset* et réactive le bouton *Action*.

Vous veillerez de plus à proposer une disposition élégante et qui s'adapte à la taille de la fenêtre.

12.4.3. Les cases à cocher : *Checkbutton*

Remarque. Ce widget est disponible en version moderne dans le sous-module `tkinter.ttk`.

Le *checkbutton* est un widget qui peut prendre seulement deux états, représentés visuellement par une case cochée ou non-cochée. L'utilisateur change l'état du widget en cliquant sur celui-ci, et on peut le paramétrer de sorte qu'une fonction soit appelée à ce moment-là. Ce widget est inutile seul : on le couple généralement à une *valeur variable* de Tkinter (par exemple un objet `StringVar`) afin que les deux soient synchronisés, en choisissant la valeur prise par cette variable lorsque la case est cochée, et celle prise lorsque la case n'est pas cochée.

Options utiles.

— `text` : un text affiché à côté de la case à cocher,

— `variable` : l'objet variable (par exemple `StringVar`) qui sera synchronisé avec l'état de la case à cocher,

— `onvalue` : la valeur que prendra cet objet variable lorsque la case est cochée,

— `offvalue` : la valeur que prendra cet objet variable lorsque la case n'est pas cochée,

— `command` : la fonction qui sera appelée lorsque l'utilisateur changera l'état de la case (par exemple en cliquant dessus).

Comme les boutons, les cases à cocher peuvent être désactivées (resp. ré-activées) en leur ajoutant (resp. retirant) le statut `"disabled"` via la méthode `state`.

Remarque. La description ci-dessus et l'exemple suivant utilisent l'objet `StringVar`, mais on peut aussi utiliser les variantes `BooleanVar` (booléen), `IntVar` (entier) ou encore `DoubleVar` (flottant) proposées par Tkinter. Dans ce cas, il faut évidemment que les valeurs passées à `onvalue` et `offvalue` soient du type qui aura été choisi.

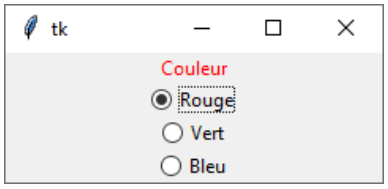
Attention. L'objet variable lié à la case à cocher doit être initialisé à l'une des deux valeurs fixées (`onvalue` ou `offvalue`). Cela n'est pas fait automatiquement par Tkinter ! Si vous donnez à l'objet variable autre chose que l'une de ces deux valeurs, cela place la case à cocher dans un état dit *indéterminé*, parfois représenté par une case partiellement cochée (ou vide suivant le système d'exploitation).

À tester.

```
import tkinter as tk
import tkinter.ttk as ttk

def on_toggle():
    label["text"] = "foo = " + foo.get()

window = tk.Tk()

label = ttk.Label(window, text="Valeur de foo ?")
label.pack()

foo = tk.StringVar(value="Jour")
cb = ttk.Checkbutton(window, text="Test", variable=foo, onvalue="Jour",
    offvalue="Nuit", command=on_toggle)
cb.pack()

window.mainloop()
```

12.4.4. Les boutons radio : *Radiobutton*

Remarque. Ce widget est disponible en version moderne dans le sous-module `tkinter.ttk`.



Les boutons radio sont très similaires aux cases à cocher. La différence réside dans le fait qu'ils fonctionnent par « groupes » au sein desquels un seul bouton peut être coché à la fois. On retrouve les mêmes options que pour les *checkbuttons*, sauf que :

— plusieurs boutons radio peuvent être liés à un même objet variable. Tous les boutons reliés au même objet forment un groupe au sein duquel un seul bouton peut être coché à un instant donné,

— les arguments `onvalue` et `offvalue` sont remplacés par un seul argument `value`, qui indique la valeur prise par l'objet variable relié au bouton lorsque ce dernier est coché.

En effet, il n'y a plus lieu de distinguer `onvalue` et `offvalue` car un bouton radio ne peut pas être directement « dé-coché ». Le seul moyen de faire en sorte qu'un bouton radio ne soit plus coché est de cocher un autre bouton du même groupe. Comme pour les cases à cocher, la variable liée à un groupe de radio boutons doit de préférence être initialisée à l'une des valeurs proposées par les boutons, sinon le groupe de boutons est dans un état *indéterminé* dans lequel aucun des boutons n'est coché.

À tester.

```
import tkinter as tk
import tkinter.ttk as ttk

def change_color():
    label["foreground"] = color.get()

window = tk.Tk()

label = ttk.Label(window, text="Couleur")

color = tk.StringVar()
r = ttk.Radiobutton(window, text="Rouge", variable=color, value="red",
    command=change_color)
g = ttk.Radiobutton(window, text="Vert", variable=color, value="green",
    command=change_color)
b = ttk.Radiobutton(window, text="Bleu", variable=color, value="blue",
    command=change_color)

label.pack()
r.pack()
g.pack()
b.pack()

window.mainloop()
```

Exercice 12.6. Créer une interface graphique contenant les éléments suivants :

- une étiquette contenant initialement la valeur 30,
- un bouton « *Action* »,
- une case à cocher accompagnée du texte « *De 10 en 10* »,
- deux boutons radios appartenant à un même groupe, l'un accompagné du texte « *Incrémenter* » et l'autre du texte « *Décrémenter* ».

Lorsqu'on clique sur le bouton action, la valeur affichée dans le label doit être modifiée, en ajoutant ou soustrayant (suivant le bouton radio choisi) la valeur 10 si la case « *De 10 en 10* » est cochée, et la valeur 1 sinon.

Vous veillerez de plus à proposer une disposition élégante et qui s'adapte à la taille de la fenêtre.

12.4.5. Pour aller plus loin ...

La liste de widgets présentée dans cette section est loin d'être exhaustive. Pour créer des interfaces encore plus élaborées, on laisse le lecteur suivre la documentation proposée en début de chapitre [3]. Mentionnons notamment l'existence des widgets suivants :

- *Entry* : un champ de saisie permettant à l'utilisateur de rentrer le texte de son choix,
- *Spinbox* : un champ permettant à l'utilisateur de saisir plus particulièrement un nombre,

3. https://tkdocs.com/tutorial/

— *Combobox* : un menu déroulant permettant à l'utilisateur de choisir un élément d'une liste prédéfinie,
— *Canvas* : un zone dans laquelle il est possible d'afficher des formes géométriques simples,
— *Scrollbar* : des barres de défilement horizontales ou verticales, à coupler avec des widgets dont le contenu occuperait trop de place,
— ...

```
import tkinter as tk
import tkinter.ttk as ttk

window = tk.Tk()
window.geometry("500x500")

# Frames principaux
top_frame = ttk.Frame(window)
bottom_frame = ttk.Frame(window)

top_frame.pack(fill="x")
bottom_frame.pack(fill="both", expand=True)

# Organisation du frame du haut
top_buttons = dict()
for key in "Menu", "Options", "Quitter":
    top_buttons[key] = ttk.Button(top_frame, text=key)
    top_buttons[key].pack(side="left")

# Organisation du frame du bas
for i in range(3):
    bottom_frame.rowconfigure(i, weight=1)
    bottom_frame.columnconfigure(i, weight=1)

bottom_buttons = []
for i in range(9):
    btn = ttk.Button(bottom_frame, text=f"{i}")
    btn.grid(row=(i // 3), column=(i % 3))
    bottom_buttons.append(btn)

# Démarrage de la boucle principale
window.mainloop()
```

FIGURE 12.5. – Un programme utilisant les frames pour créer une disposition complexe.