Richard Gomez

Le petit Python graphique avec Tkinter

Interfaces graphiques et programmation événementielle avec Python et Tkinter

ellipses

Chapitre 1
Généralités

1.1 Origines de Tkinter

Le module Tkinter est une caisse à outils permettant de créer des interfaces graphiques (*graphical user interfaces*, en anglais, ou plus simplement GUI).

Pour raconter l'histoire de Tkinter, il faut remonter à la fin des années 80. En 1988, John Ousterhout invente le langage Tcl (Tool Command Language). En 1990, il développe une extension de ce langage appelée Tk. Il s'agit d'une bibliothèque permettant de créer des interfaces graphiques (*tool kit* signifie boîte à outils en anglais). Depuis, une importante communauté de développeurs utilise le langage Tcl-Tk.

Plus tard, Steen Lumholt et Guido van Rossum (à qui l'on doit Python) créent le module Tkinter (Tk interface). Il s'agit d'une adaptation de Tk pour Python (un enrobage). La documentation officielle [4] cite également Fredrik Lundh pour sa contribution (les noms sont cités à la section 24.1.2 de la documentation de la vieille version 2.7).

Aujourd'hui, Tkinter est l'interface graphique standard de Python.

Figure 1.1. Autoportrait tramé de John Ousterhout, inventeur de Tcl et Tk, illustrant l'option `dither` de la commande `canvas` de Tk. Source : Wikipedia

Il n'existe pas de documentation complète sur Tkinter et malheureusement on doit parfois aller chercher des informations dans la documentation de Tcl-Tk [9] pour ensuite les traduire en langage Python-Tkinter (les syntaxes sont légèrement différentes). Heureusement pour nous, cela arrive rarement. Normalement, nous n'aurons pas besoin de connaître Tcl-Tk pour apprendre à utiliser Tkinter correctement.

Note. La documentation officielle (mais incomplète) de Tkinter se trouve sur le site officiel de Python, voir [3] et [4].

1.2 Grands principes

Tkinter permet de créer un environnement graphique dans lequel un utilisateur peut, de manière agréable et intuitive, entrer des données, cocher des options, déclencher des calculs, dialoguer, admirer des animations, etc. On peut grâce à ses outils programmer un jeu vidéo, réaliser une simulation graphique et bien d'autres choses encore. Les programmes ainsi créés ressemblent (plus ou moins) aux applications habituellement utilisées sur un ordinateur.

Un programme écrit avec Tkinter crée une *fenêtre principale* (*main window* ou *root window*, en anglais) et éventuellement des fenêtres secondaires (*toplevel window* ou *pop-up windows*). La fenêtre principale est une instance de `Tk`. Les fenêtres secondaires sont des instances de `Toplevel`. On munit bien sûr les fenêtres de plusieurs *widgets* (contraction, semblerait-t-il de *window* et *gadget*). Tkinter offre toutes sortes de widgets : zones de saisie, zones de sélection numérique, boutons, zones de listes, sélecteurs d'options, canevas, etc. Les widgets sont les éléments de base d'une interface graphique.

Notes.

1. Les widgets sont aussi appelés *composants d'interface graphique* ou encore *éléments visuels d'interface graphique*.
2. Les fenêtres sont elles-mêmes des widgets. C'est ainsi, en tout cas, que nous comprenons le mot widget. Par conséquent, quand nous écrivons : « les widgets », nous sous-entendons la plupart du temps « les widgets et les fenêtres ». Dans la documentation Tkinter et de nombreux manuels écrits en anglais, il arrive que le mot *window* désigne à la fois fenêtre et widget. On retrouve cela dans le « langage Tkinter » lui-même. En effet, pour insérer un widget dans un canevas, par exemple, on utilise la méthode `create_window`, et l'item ainsi créé est de type `"window"` (chapitre 27). Quand la documentation parle d'une fenêtre (pas un widget), elle dit « *toplevel window* ». C'est dans ces termes que [4] décrit la fenêtre principale (instance de `Tk`) : « `root` is a toplevel window ».

1.3 Importer Tkinter

Il recommandé de ne pas importer le contenu de `tkinter` mais le module lui-même, afin d'éviter des problèmes d'écrasement et de saturation de l'espace global. De plus, il est d'usage de donner l'alias `tk` à `tkinter` :

```
import tkinter as tk
racine = tk.Tk()
(...)
racine.mainloop()
```

ainsi, tout ce qui vient de `tkinter` porte un nom qualifié commençant par `tk.` Dans le présent document, en revanche, nous ne suivrons presque jamais cette consigne pour des raisons de confort de lecture. Nous importerons « bestialement » le contenu du module :

```
from tkinter import *
```

On peut consulter `TclVersion` et `TkVersion` pour savoir quelle est notre version de Tkinter :

```
>>> from tkinter import TclVersion, TkVersion
>>> TclVersion
8.6
>>> TkVersion
8.6
```

Si on veut connaître la version dans le format *majeur.mineur.correctif* (en anglais *major.minor.patch*), on procède à la manipulation ci-dessous :

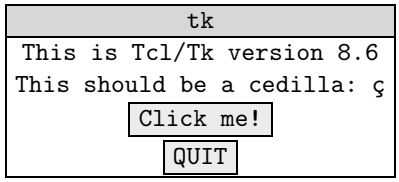
```
>>> racine = Tk()
>>> racine.call("info", "patchlevel")
'8.6.8'
```

(On explique la méthode `call` au chapitre 2.)

La commande « `python3 -m tkinter` » exécutée à partir du `Terminal` de MAC OS, par exemple, permet également de connaître la version utilisée :

```
$ python3 -m tkinter
```

Cette dernière déclenche la création d'une fenêtre ressemblant à ceci :



1.4 Fenêtre et boucle principales

Nous avons dit que la fenêtre principale est une instance de `Tk`. Pour créer un environnement graphique, nous commençons donc par instancier `Tk` :

```
>>> from tkinter import *
>>> racine = Tk()
```

La deuxième instruction fait apparaître une fenêtre ressemblant plus ou moins à ceci :

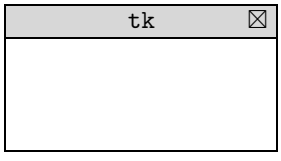


Figure 1.2. La fenêtre principale

Tout programme Tkinter suit le schéma suivant : on importe le module `tkinter`, on instancie `Tk`, on instancie des widgets, on les *place* et on lance la *boucle principale* grâce à la méthode `mainloop` :

```
from tkinter import *
racine = Tk()
(...)
racine.mainloop()
```

Attention. On n'instancie la classe `Tk` qu'une fois.

La boucle principale consiste en ceci : tant que la fenêtre principale n'est pas fermée (détruite), Tkinter est en écoute, ce qui signifie qu'il attend qu'un évènement (au clavier ou à la souris) advienne pour réagir en fonction de qui est écrit dans le programme. On parle ici de *programmation évènementielle*.

Les lignes de code écrites éventuellement après l'appel de `mainloop` sont exécutées lorsque la fenêtre principale est fermée (ou lorsque la méthode `quit` est appelée).

Tous les widgets possèdent des méthodes `mainloop` et `quit`. Nous décrivons ces dernières ci-dessous, sur un widget `w` quelconque.

`w.mainloop()` :

- effet : une fois créés les widgets, on appelle `w.mainloop` afin de lancer la boucle principale. La plupart du temps, `w` est la fenêtre principale, mais cela n'est pas obligatoire. Il est par ailleurs possible d'appeler `mainloop` à l'intérieur d'un gestionnaire d'évènements pour relancer la boucle principale (voir chapitre 39).

`w.quit()` :

- effet : le programme sort de la boucle principale. La plupart du temps, `w` est la fenêtre principale, mais cela n'est pas obligatoire.

Note. Si le programme a lancé `mainloop`, tant que la fenêtre principale n'est pas fermée (ou que `quit` n'est pas appelée), nous n'avons pas la main sur le prompteur (shell). Impossible donc d'afficher manuellement le contenu d'une variable du programme, par exemple. Si on veut pouvoir interroger Python pendant que la fenêtre principale est ouverte, il ne faut pas appeller `mainloop` : c'est ce que nous faisons pendant la phase de débogage de nos programmes. Une fois achevé le débogage, on restaure l'appel de `mainloop`, bien entendu.

Nous testons ces méthodes à la section suivante.

1.5 Widgets conteneurs

Un widget *conteneur* a pour vocation de recevoir des widgets. Les widgets conteneurs sont les fenêtres (principale ou secondaires), les cadres (`Frame` et `Labelframe`), les fenêtres à pans (`PanedWindow`) et d'une certaine manière, les canevas (`Canvas`), ainsi que les zones de texte (`Text`).

On utilise les widgets conteneurs pour organiser l'espace offert par les fenêtres.

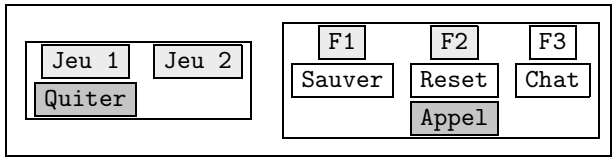


Figure 1.3. Deux groupes de widgets dans une fenêtre (chaque groupe est dans un cadre)

Nous reviendrons sur la notion de conteneur (*master*, *children*, etc.) à la section 1.12. En attendant, nous retiendrons que `Tk`, `Toplevel`, `Frame`, `Labelframe`, `PanedWindow`, `Canvas` et `Text` sont des types de widgets conteneurs.

1.6 Instancier un widget

Les widgets sont les objets graphiques que l'on insère dans la fenêtre principale ou les fenêtres secondaires : affichages, champs de saisie, menus, boutons, etc. L'instanciation se fait en appelant le type désiré.

La signature d'une classe quelconque de widgets est

```
(master=None, cnf={}, *, name=None, **kwargs)
```

où :

- `master` reçoit le nom du conteneur dans lequel est inséré le widget ;
- `cnf` reçoit un dictionnaire d'options (*configuration options*) ;
- `name` reçoit une chaîne de caractères (nom Tkinter, sous-section 1.9.1) ;
- `**kwargs` reçoit les options et leurs valeurs (chaque option est un paramètre avec nom obligatoire).

La plupart du temps, on utilise `**kwargs` pour faire passer des options et non pas `cnf` (dans ce livre nous n'utilisons jamais `cnf`).

Voici différentes manières d'instancier `Button`, par exemple. La manière la plus courante :

```
b = Button(parent, opt1=v1, opt2=v2, etc.)
```

(où `parent` désigne un conteneur).

Certains préfèrent la manière ci-dessous :

```
b = Button(master=parent, opt1=v1, opt2=v2, etc.)
```

ou alors carrément

```
b = Button(opt1=v1, opt2=v2, etc.)
```

ce qui sous-entend que le père est la fenêtre principale (le père par défaut est la fenêtre principale).

On peut reprendre ces 3 manières en y ajoutant le paramètre `name`.

Voici un exemple plus précis (et réel) :

```
b = Button(racine, text="Reeds", bg="red", fg="black", name="b")
```

Voici la liste des widgets offerts par Tkinter : `Tk`, `Toplevel`, `Label`, `Message`, `Entry`, `Spinbox`, `Scale`, `Button`, `Checkbutton`, `Radiobutton`, `Listbox`, `OptionMenu`, `Canvas`, `Text`, `Frame`, `LabelFrame`, `PanedWindow`, `Scrollbar`, `Menu` et `Menubutton`.

Note. Nous incluons `Tk` et `Toplevel` dans la liste des widgets, même si cela peut paraître abusif.

Attention. Il existe deux exceptions à ce que nous avons dit :

- l'instanciation de `Tk` se fait sans argument ;
- l'instanciation de `OptionMenu` se fait en passant obligatoirement le parent, une variable de contrôle et au moins une valeur.

Le cas de `OptionMenu` est un petit caillou dans la chaussure de l'introspection. En effet, tous les programmes où on écrit `w()` déclencheront une erreur pour `w` égal à `OptionMenu`. Nous verrons au chapitre 26 que la signature de cette classe est `(master, variable, value, *values, command=None)`.

1.7 Placer un widget

Une fois instancié, nous devons *placer* notre widget pour qu'il apparaisse à l'écran. Ceci peut être fait avec la méthode `grid`, par exemple :

```
from tkinter import *
racine = Tk()
b = Button(racine, text="Mon beau bouton", bg="red")
b.grid()
racine.mainloop()
```

Nous consacrons le chapitre 13 à cette méthode.

La méthode `grid` n'est pas le seul *gestionnaire de position*, il en existe d'autres comme `pack` et `place` (dans ce livre, c'est `grid` que nous préférons).

Note. Quand on oublie de placer un widget, celui-ci existe en tant qu'objet dans la mémoire mais Tkinter ne le représente pas graphiquement à l'écran. Pour l'utilisateur, c'est comme s'il n'existait pas. Il s'agit d'une erreur courante quand on débute.

1.8 Détruire un widget

Tout widget possède une méthode `destroy`.

`w.destroy()` :

- effet : destruction du widget `w` et ses descendants.

Le programme ci-dessous place trois boutons dans la fenêtre principale nommée `racine`. Le bouton QUIT déclenche `racine.quit()`, le bouton DESTROY RACINE déclenche `racine.destroy()` et le bouton DESTROY BOUTON déclenche `b2.destroy()` :

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

```
print("Affichage commandé avant l'appel de mainloop")
b1 = Button(racine, text="QUIT", command=racine.quit)
b1.grid()
b2 = Button(racine, text="DESTROY RACINE", command=racine.destroy)
b2.grid()
b3 = Button(racine, text="DESTROY BOUTON", command=b2.destroy)
b3.grid()
racine.mainloop()
print("Affichage commandé après l'appel de mainloop")
```

Une fois le programme lancé, il faut soit fermer la fenêtre principale (en cliquant sur ☒), soit cliquer sur `b1` ou `b2` pour pouvoir récupérer la main dans le shell. Le cas échéant, le programme affiche la chaîne `"Affichage commandé après l'appel de mainloop"`.

Tout widget possède une méthode `winfo_exists`.

`w.winfo_exists()` :

– effet : retourne `1` si le widget `w` existe toujours (ie n'a pas été détruit). Retourne `0` dans le cas contraire.

Ne pas perdre de vue que même si on a appelé `w.destroy()`, le nom `w` pointe vers un objet en mémoire. Si on lance le programme précédent, que l'on clique sur DESTROY BOUTON puis sur QUIT, on obtient les réponses suivantes dans le shell :

```
>>> b1.winfo_exists()
1
>>> b2.winfo_exists()
0
```

1.9 Nom Tkinter d'un widget

1.9.1 Nom

Tout widget (hormis la fenêtre principale) possède un attribut `_name` contenant une chaîne de caractères : il s'agit du *nom Tkinter* du widget.

```
>>> from tkinter import *
>>> racine = Tk()
>>> a = Button(racine, text="Pilar")
>>> a.grid()
>>> a._name
'!button'
```

Nous voyons que le nom Tkinter du bouton `a` est la chaîne `"!button"`.

De manière générale, le nom par défaut du premier widget de type `Widget` inséré dans un conteneur donné est `"!widget"`. Le nom du deuxième widget de même type dans le même conteneur reçoit le nom `"!widget2"`, puis le suivant `"!widget3"`, et ainsi de suite. La preuve en poursuivant notre manipulation :

```
>>> b = Button(racine, text="Pedro")
>>> b.grid()
>>> b._name
'!button2'
```

On peut choisir le nom Tkinter d'un widget grâce au paramètre d'instanciation `name` :

```
>>> c = Button(racine, text="Rufino", name="c")
>>> c.grid()
>>> c._name
'c'
```

La chaîne passée dans `name` commence obligatoirement par une lettre minuscule ou un underscore. Le reste des caractères doit obéir aux règles habituelles sur les identificateurs.

Attention. Le paramètre `name` n'est pas une option : il ne fait pas partie des items retournés par la méthode `keys` (1.14) et l'expression `c["name"]` provoque un erreur (1.14), de même que l'utilisation de la clé `"name"` dans `configure` ou `cget` (1.13).

Deux widgets fils d'un même conteneur ne peuvent pas porter le même nom Tkinter. En effet, si on crée une étiquette nommée `"c"`, par exemple :

```
>>> d = Lebel(racine, text="Pepa", name="c")
```

le bouton `c` disparaît instantanément. Ensuite, lorsqu'on place `d` :

```
>>> d.grid()
```

on voit l'étiquette prendre la place du bouton. L'objet `c` existe toujours, mais pas à l'écran. Voici ce qu'on a à l'écran :

```
>>> racine.winfo_children()
[<tkinter.Button object .!button>, <tkinter.Button object .!button2>,
<tkinter.Label object .c>]
```

La fenêtre principale ne possède pas d'attribut `_name` :

```
>>> racine._name
AttributeError: '_tkinter.tkapp' object has no attribute '_name'
```

De manière générale, on déconseille l'invocation de l'attribut `_name` (que cela soit en lecture ou en écriture). Il s'agit d'un attribut interne, comme l'indique le tiret bas (underscore). Si on veut connaître le nom Tkinter d'un widget, on utilise la méthode `winfo_name`.