1 Notion de *binding*

Tous les widgets obéissent à des évènements : lorsqu'un bouton possède le focus, par exemple, le fait d'appuyer sur la barre d'espace a pour effet d'appuyer sur le bouton, ce qui enclenche l'appel de la fonction associée à l'option `command`. Il se passe bien sûr la même chose si l'utilisateur clique sur le bouton. Ici, nous avons affaire à 3 entités « associées » :

clic ⟶ [Bouton] ⟶ appel d'une fonction

De manière générale, une *association* concerne :

- un type d'évènement ;
- un widget (ou une classe de widgets, ou l'application entière) ;
- un *callback* (fonction, méthode, appelable).

Non seulement Tkinter est truffé d'*associations* (presque tous les widgets en possèdent plusieurs), mais il permet d'en créer autant que l'on veut. Dans le jargon Tkinter, *association* se dit *binding* (le verbe anglais *to bind* signifie lier, connecter). Un binding concerne toujours un type d'évènement, un ou plusieurs widgets et un gestionnaire d'évènement (*event handler*).

Un évènement est soit une action réalisée par l'utilisateur (cliquer quelque part ou appuyer sur la touche RETURN, par exemple), soit un changement se produisant chez un widget (changement d'état, perte du focus, redimensionnement, etc.) La notion d'évènement est assez large et correspond à : « quelque chose qui vient de se produire et qui intéresse notre programme ».

2 Binding concernant un widget

2.1 Methode `bind`

La méthode permettant de créer un binding s'appelle `bind`.

`w.bind(ch=None, func=None, add=None)` :

- `ch` : séquence représentant un type d'évènement (chaîne) ;
- `func` : fonction (callback) ;
- `add` : une seule valeur possible, `"+"` ;
- effet : un appel `w.bind()` sans argument retourne l'uplet des séquences impliquées dans un binding avec `w`. Un appel `w.bind(ch)` retourne la chaîne-séquence des callbacks associés au widget `w` et la séquence `ch`. Un appel `w.bind(ch, func)` associe le widget `w`, la séquence `ch` et la fonction `func`. Chaque fois que l'évènement de type `ch` aura lieu sur `w`, le callback `func` sera appelé. Tkinter appelle le callback en faisant passer un objet de type `Event` contenant toutes les propriétés de l'évènement détecté. S'il existe déjà un binding entre `w` et `ch`, celui-ci vient le remplacer (le nouveau callback écrase l'ancien), sauf si on a pris « `add = "+"` », auquel cas, le nouveau binding s'ajoute à l'ancien. L'appel `w.bind(ch, func)` retourne une chaîne de caractères appelée « identité du callback ». Cette chaîne contient un code suivi du nom de la fonction `func` passée en argument.

En fait, l'argument `add` peut-être n'importe quel objet ayant `True` pour valeur booléenne.

L'identité du callback est utilisée par la méthode `unbind` (sous-section suivante).

Le programme ci-dessous fabrique quatre associations :

- « clic », cadre `a` et `action1` ;
- « clic gauche », cadre `b` et `action2` ;
- « clic gauche », cadre `c` et `action3` ;
- « clic droit », cadre `a` et `action4` ;

Ces quatre callbacks affichent les séquences associées avec le widget correspondant (par exemple `action1` affiche les séquences liées au widget `a`) :

```
from tkinter import *
def action1(event):
    print("Clic quelconque sur a (rouge en haut)")
    print(a.bind())
def action2(event):
    print("Clic gauche sur b (jaune)")
    print(b.bind())
def action3(event):
    print("Clic gauche sur c (rouge en bas)")
    print(c.bind())
def action4(event):
    print("Clic droit sur c (rouge en bas)")
    print(c.bind())
racine = Tk()
a = Frame(racine, bg="red", width=450, height=75)
a.grid()
b = Frame(racine, bg="yellow", width=450, height=150)
b.grid()
c = Frame(racine,bg="red", width=450, height=75)
c.grid()
identité = a.bind("<ButtonPress>", action1)
print(identité)
b.bind("<ButtonPress-1>", action2)
c.bind("<ButtonPress-1>", action3)
c.bind("<ButtonPress-2>", action4)          # 3 sur PC Windows
racine.mainloop()
```

On notera la présence du paramètre `event` dans les callbacks. En effet, comme cela a été dit, lorsque Tkinter appelle un callback, il fait passer un objet de type `Event` en argument. Cet objet contient toutes les propriétés de l'évènement ayant déclenché le callback : abscisse du lieu du clic (si clic), ordonnée (si clic), caractère de la touche (si touche pressée), etc.

Le programme provoque l'affichage de l'identité du binding `(a, "<ButtonPress>", action1)` :

```
140695824123968action1
```

2.2 Chaîne-séquence des callbacks

Nous avons dit à la sous-section précédente que l'appel `w.bind(ch)` retourne la chaîne-séquence des callbacks associés au widget `w` et la séquence `ch`, mais nous n'avons pas expliqué ce qu'est cette « chaîne-séquence ».

Supposons que `w` et `"<1>"` sont associés aux fonctions `f`, `g` et `h`. Alors, moralement, `w.bind("<1>")` retourne la séquence `f, g, h`. Dans la pratique, c'est un peu plus compliqué.

Exécutons le programme ci-dessous :

```
from tkinter import *
def action(evt): print("action")
def faire(evt): print("faire")
racine = Tk()
frame = Frame(racine, width=300, height=300, bg="pink")
frame.grid()
frame.bind("<1>", action)
frame.bind("<1>", faire, add="+")
```

et regardons ce que retourne `frame.bind("<1>")` :

```
>>> frame.bind("<1>")
'if {"[140194455207872action %# %b %f %h %k %s %t %w %x %y %A %E %K %N
%W %T %X %Y %D]" == "break"} break\n\nif {"[140194457330176faire %# %b
%f %h %k %s %t %w %x %y %A %E %K %N %W %T %X %Y %D]" =="break"} break\n'
```

Le résultat est une chaîne de caractères contenant un enrobage incompréhensible de l'identité du binding `(frame, "<1>", action)`, suivie d'un double saut de ligne `"\n\n"`, suivi d'un enrobage de l'identité du binding `(frame, "<1>", faire)`, suivi de `"\n"`.

Si on découpe cette chaîne selon `"\n"`, on obtient un item correspondant au binding `action`, une chaîne vide, un item correspondant à `faire` et une chaîne vide :

```
>>> chaîne = frame.bind("<1>")
>>> bindings = chaîne.split("\n")
```

sauf le dernier item qui ne contient rien :

```
>>> for k in bindings: print(repr(k[:30]))
'if {"[140194455207872action %#'
''
'if {"[140194457330176faire %# '
''
```

Observons l'item correspondant à `action` (le premier item) :

```
>>> bindings[0]
'if {"[140194455207872action %# %b %f %h %k %s %t %w %x %y %A %E %K %N
%W %T %X %Y %D]" == "break"} break'
```

L'identité de `(frame, "<1>", action)` est la chaîne `"140194455207872action"` et on remarque qu'elle commence au caractère numéro `[6]` et s'arrête au premier espace rencontré.

La fonction `extraire` ci-dessous est capable d'extraire l'identité d'un binding contenu dans un item de chaîne-séquence :

```
def extraire(callback):
    i = callback.find(" ", 6)
    return callback[6:i]
```

Testons (après reconstruction de `chaîne` et `bindings`) :

```
>>> for k in bindings: print(repr(extraire(k)))
'140194455207872action'
''
'140194457330176faire'
''
```

Nous utiliserons ces chaînes-séquences et la fonction `extraire` à la sous-sous-section 39.2.4.2.

2.3 Méthode `unbind`

On peut défaire un binding avec `unbind` :

`w.unbind(ch, funcid=None)` :

- `ch` : séquence représentant un type d'évènement (chaîne) ;
- `funcid` : identité d'un callback (chaîne) ;
- effet : si `funcid` est la chaîne retournée par un appel `w.bind(ch, func)`, l'appel `w.unbind(ch, funcid)` désactive le binding `(w, ch, func)`. Un appel `w.unbind(ch)` désactive tous les bindings `(w, ch, *)`.

Notes.

1. La chaîne `funcid` retournée par `bind` pourrait s'appeler « identité du binding » au lieu de « identité du callback ». C'est en effet en tant que trace d'un binding que cette chaîne est utilisée.

2. C'est dommage que Tkinter n'ait pas prévu d'appel `w.unbind()` sans argument pour désactiver tous les bindings sur `w`. Si on effectue un tel appel, on déclenche une erreur.

Le programme ci-dessous crée un cadre gris et un bouton DÉFAIRE. Le cadre est associé à l'évènement « clic gauche » et au callback `action`. Le bouton a pour callback une fonction qui affiche l'identité du binding `(frame, "<1>", action)` et le défait :

```
from tkinter import *
def action(event): print("Clic")
def défaire():
    print(binding)
    frame.unbind("<1>", binding)
racine = Tk()
```

```
bouton = Button(racine, text="Défaire", command=défaire)
bouton.grid()
frame = Frame(racine, width=300, height=300, bg="grey95")
frame.grid()
binding = frame.bind("<1>", action)
racine.mainloop()
```

Quand nous avons cliqué sur le bouton, nous avons constaté que l'identité du binding `(frame, "<1>", action)` était `"140228858960320action"`, mais le lecteur constatera que le nombre change à chaque exécution du programme.

2.4 Dysfonctionnement de `unbind`

Il arrive sur certains systèmes qu'un appel `w.unbind(séq, funcid)` efface toutes associations liant `w` à la séquence `séq` alors que seule l'association `(w, séq, funcid)` est visée. Pour savoir si notre système fonctionne correctement, on exécute le programme ci-dessous :

```
# "unbindtest.py"
from tkinter import *
def f(event): print("f")
def g(event): print("g")
def h(event): print("h")
def défaire():
    print(binding)
    frame.unbind("<1>", binding)
racine = Tk()
bouton = Button(racine, text="Défaire", command=défaire)
bouton.grid()
frame = Frame(racine, width=300, height=300, bg="grey95")
frame.grid()
binding = frame.bind("<1>", f)
frame.bind("<1>", g, add="+")
frame.bind("<1>", h, add="+")
racine.mainloop()
```

Ce programme permet de voir si `frame.unbind("<1>", binding)` désactive le binding d'identité `binding` ou tous les bindings `(frame, "<1>", *)`.

Il existe plusieurs manières de corriger cet éventuel problème.

2.4.1 Premier remède

Une première manière consiste à remplacer la classe `Frame` par une classe dérivée `Frame_b` surchargeant les méthodes `bind` et `unbind` :

```
class Frame_b(Frame):
    def __init__(self, *args, **kwargs):
        self.bindings = {}
```

```
        super().__init__(*args, **kwargs)
    def bind(self, sequence=None, func=None, add=None):
        if (sequence, func, add) == (None, None, None):
            return self.bindings
        if (func, add) == (None, None): return self.bindings[sequence]
        super().bind(sequence, func, add)
        if add and sequence in self.bindings:
            self.bindings[sequence].append(func)
        else: self.bindings[sequence] = [func]
        return func
    def unbind(self, sequence, func=None):
        super().unbind(sequence)
        if func is None:
            del self.bindings[sequence]
            return
        if func in self.bindings[sequence]:
            self.bindings[sequence].remove(func)
        for f in self.bindings[sequence]:
            super().bind(sequence, f, add="+")
```

Un cadre de type `Frame_b` possède un dictionnaire `bindings` dont les clés sont des séquences et les valeurs des listes de fonctions. Si par exemple on a

```
bindings["<1>"] = [f, g]
```

alors le cadre est associé à `("<1>", f)` et `("<1>", g)` et tout clic gauche sur le cadre déclenche `f` puis `g`. Notre méthode `bind` ne retourne pas une identité de binding mais le callback lui-même.

Par symétrie, notre méthode `unbind` n'attend pas l'identité d'un binding mais un callback déjà associé.

Note. Le lecteur comblera, s'il le désire, une lacune signalée à la sous-section précédente en implémentant les appels `unbind()` sans argument (on prendra « `sequence = None` » dans la signature, pour commencer).

Pour tester notre classe `Frame_b`, on reprend le programme `"unbindtest.py"` (début de sous-section) en remplaçant `Frame` par `Frame_b`.

On fera attention à un détail : si on écrit `"<ButtonPress-1>"` dans un `bind`, il faudra écrire `"<ButtonPress-1>"` dans le `unbind` correspondant, et non son abréviation `"<1>"`. Pour retrouver la souplesse offerte par Tkinter, il faudrait écrire une fonction qui à chaque séquence associe son écriture canonique.

Note. Si `unbind` n'existait pas, on écrirait une fonction `unbind` dans notre classe `Frame_b` qui ferait la chose suivante. Pour défaire un lien `(frame, séq, action)`, on exécuterait un `frame.bind(séq, rien)` où `rien` est une fonction de signature `(event)` qui ne fait rien. On peut prendre `rien` égal à « `lambda event: None` », par exemple. Bien sûr, il ne faudrait pas oublier de réactiver tous les `(frame, séq, func)` pour tous les `func` de `frame.bindings[séq]`, en prenant « `add = "+"` ». On laisse au lecteur le plaisir d'écrire une telle fonction `unbind` en guise d'exercice.

Tout ce que nous avons fait en dérivant `Frame` peut être appliqué à la dérivation de toute classe de widgets, bien entendu.

2.4.2 Deuxième remède

La deuxième manière de corriger un `unbind` défaillant requiert des connaissances précises sur le langage TCL.

Supposons que `frame` est un widget de nom complet `".frame"`. Alors l'instruction TCL

```
bind ".frame" <1> ""
```

fait que `frame` n'est plus lié à `"<1>"`. En effet, cette instruction associe `frame` et `"<1>"` à une séquence `""` vide de callbacks.

Nous savons par ailleurs qu'un interpréteur TCL possède une méthode `call` permettant d'exécuter des instructions écrites en langage TCL. Ainsi, si `tcl` est un interpréteur, l'appel

```
tcl.call("bind", ".frame", "<1>", "")
```

désactive tous les bindings associant `frame` et la séquence `"<1>"`.

Poursuivons notre tour du langage TCL. L'instruction

```
bind ".frame" <1>
```

ne crée aucun binding mais retourne la chaîne-séquence des callbacks associés à `frame` et `"<1>"`. Nous avons vu à la sous-section 39.2.2 comment est encodée cette chaîne-séquence et comment on en extrait les identités des callbacks impliqués.

Pour faire passer cette instruction à un interpréteur, on écrit :

```
chaîne = racine.call("bind", ".frame", "<1>")
```

et la chaîne-séquence se retrouve dans `chaîne`.

Poursuivons notre promenade dans TCL. Si on veut, par exemple, que `frame` et `"<1>"` soient associés à des fonctions `f`, `g` et `h`, on encode ces 3 callbacks sous la forme d'une chaîne-séquence, et on exécute l'instruction TCL ci-dessous :

```
bind ".frame" <1> "<codage de f>\n<codage de g>\n<codage de h>"
```

(un simple saut de ligne suffit). Nous ne savons pas faire cet encodage, mais en vérité, nous n'en avons pas besoin pour exécuter un *unbind*. En effet, pour désactiver un binding `(frame, "<1>", f)`, par exemple, on récupère la chaîne-séquence des callbacks de `(frame, "<1>")`, on la découpe selon `"\n"`, on retire l'item correspondant à `f`, on recompose la chaîne et on commande un

```
bind ".frame" <1> "<chaîne recomposée>"
```

tout simplement !

Dernier point : tout widget possède un attribut `tk` pointant vers un interpéteur TCL et un attribut `_w` pointant vers son nom complet. Avec toutes ces informations, nous pouvons écrire une classe `Frame_b` surchargeant et corrigeant la méthode `unbind` :

```
class Frame_b(Frame):
    @staticmethod
    def extraire(callback):
        i = callback.find(" ", 6)
        return callback[6:i]
    @staticmethod
    def égal(identité, funcid):
        chaîne = Frame_b.extraire(identité)
        return  chaîne == funcid
    def unbind(self, sequence, funcid=None):
        if not funcid:
            self.tk.call("bind", self._w, sequence, "")
            return
        identités = self.tk.call("bind", self._w, sequence)
        identités = identités.split("\n")
        égal = Frame_b.égal
        nouvelles = [ch for ch in identités if not égal(ch,funcid)]
        nouvelles = "\n".join(nouvelles)
        self.tk.call("bind", self._w, sequence, nouvelles)
        self.deletecommand(funcid)
```

Notre méthode `unbind` extrait la chaîne-séquence `identités` des identités de bindings (Tkinter dirait la chaîne-séquence des callbacks). Elle fabrique la chaîne-séquence `nouvelles` en supprimant l'identité correspondant à `funcid`. Pour ce faire, la compréhension définissant `nouvelles` parcourt les `ch` de `identités` en comparant `ch` avec `funcid`. La comparaison est effectuée avec la fonction `égal` (méthode statique). Cette fonction utilise la fonction `extraire` (méthode statique aussi) qui permet d'extraire l'identité à proprement parler de tout le texte caractérisant le binding en question (on a déjà écrit cette fonction à la sous-section 39.2.2). La dernière ligne n'est pas obligatoire. Elle demande à Tkinter, via la méthode universelle `deletecommand`, d'oublier la commande `funcid`.

2.4.3 Troisième remède

La manière la plus simple mais aussi la moins universelle consiste à réactiver à la main tous les bindings malencontreusement désactivés par `unbind`.

C'est facile à mettre en place mais cela souffre d'un assez grave inconvénient : toute modification du programme (ajoutant des bindings) nous oblige à mettre à jour la partie du programme qui réactive les bindings désactivés.

C'est cette solution que nous avons choisi dans le projet Traceur (chapitre 42).

3 Binding concernant une classe de widgets

3.1 Methode `bind_class`

La méthode permettant de créer un binding avec une classe de widgets est `bind_class`.

`w.bind_class(cn, ch=None, func=None, add=None)` :

– `cn` : nom Tkinter d'une classe (chaîne) ;
– `ch` : séquence représentant un type d'évènement (chaîne) ;
– `func` : fonction (callback) ;
– `add` : une seule valeur possible, `"+"` ;
– effet : Un appel `w.bind_class(cn)` retourne la liste des séquences impliquées dans un *bind class* avec la classe de widgets nommée `cn`. Un appel `w.bind_class(cn, ch)` retourne la chaîne-séquence des callbacks associés à la classe `cn` et la séquence `ch`. Un appel `w.bind_class(cn, ch, func)` associe la classe `cn`, la séquence `ch` et la fonction `func`. Chaque fois qu'un évènement de type `ch` aura lieu sur un widget de type `cn`, le callback `func` sera appelé. S'il existe déjà un binding entre `cn` et `ch`, celui-ci vient le remplacer (le nouveau callback écrase l'ancien), sauf si on a pris « `add = "+"` », auquel cas, le nouveau binding s'ajoute à l'ancien. La méthode retourne l'identité du binding sous la forme d'une chaîne. Cette chaîne contient un code suivi du nom du callback. Tkinter appelle le callback en faisant passer un objet de type `Event` contenant toutes les propriétés de l'évènement détecté.

Notes. 1. L'identité du binding (ie la chaîne retournée) ne sert à rien. En effet, la méthode `unbind_class` (qui désactive les bindings de classe, voir plus bas) ne prévoit pas la désactivation d'un callback en particulier.

2. En vérité, `cn` peut recevoir un *bind tag*. Nous reviendrons sur ce point et sur `bind_class` à la section 39.9.

Le programme ci-dessous associe l'événement « clic droit », la classe `Button` et la fonction `change_langue`, de sorte que le clic droit sur n'importe quel bouton change la langue utilisée par l'application :

```
from tkinter import *
from functools import partial
def couleur(color): cadre.configure(bg=color)
def change_langue(event):
    global langue
    if langue == "français": langue = "espagnol"
    else: langue = "français"
    if langue == "français": dico = français
    else: dico = espagnol
    for color in couleurs:
        var = StringVar(name=color)
        valeur = dico[color]
        var.set(valeur)
espagnol = dict(red="Rojo", green="Verde", blue="Azul")
français = dict(red="Rouge", green="Vert", blue="bleu")
couleurs = ["red", "green", "blue"]
langue = "français"
racine = Tk()
racine.configure(bg="grey95")
```

```
i = 1
for color in couleurs:
    valeur = français[color]
    var = StringVar(racine, valeur, name=color)
    callback = partial(couleur, color)
    bouton = Button(racine, width=10, bg=color, textvariable=var,
                    command=callback)
    bouton.grid(row=1, column=i)
    i = i + 1
cadre = Frame(racine, bg="grey", width=500, height=100)
cadre.grid(row=2, column=1, columnspan=3)
racine.bind_class("Button", "<Button-2>", change_langue)
racine.mainloop()
```

La séquence `"<Button-2>"` correspond à « clic droit ».

La boucle `for` fabrique 3 variables de contrôle et 3 boutons (sans se soucier de leur donner un identifiant Python). En revanche, chacune des variables possède un nom Tkinter.

Voici ce que fait `change_langue`. Tout d'abord, la fonction change la valeur de la variable globale `langue` (cette variable alterne entre `"français"` et `"espagnol"`). Ensuite, elle définit le dictionnaire `dico` en fonction de la valeur de `langue`. Par exemple si `langue` est `"français"`, `dico["red"]` donnera `"rouge"`. Ensuite, pour `color` égal successivement à `"red"`, `"green"`, `"blue"` (ce sont les noms Tkinter des variables de contrôle), on fait une copie de la variable correspondante grâce à l'astuce « `var = StringVar(name=color)` ». Le fait de changer ensuite la valeur de cette variable modifie également la valeur de la variable copiée, ce qui met à jour automatiquement la légende du bouton correspondant, et le tour est joué.

Attention. Le callback doit avoir pour signature `(event)` car Tkinter l'appelle en faisant passer un objet de type `Event` en argument.

3.2 Methode `unbind_class`

On peut défaire un binding de classe avec `unbind_class` :

`w.unbind_class(cn, ch)` :

- `cn` : nom d'une classe (chaîne) ;
- `ch` : séquence représentant un évènement (chaîne) ;
- effet : désactive tous les bindings `(cn,ch,*)`.

Cette méthode ne permet pas de supprimer un binding `(cn, ch, func)` en particulier.

Exercice 1. Écrire une classe `Tk_b` dérivée de `Tk` qui surcharge la méthode `unbind_class` afin qu'elle puisse désactiver un binding `(cn, ch, func)` en particulier. La méthode aura pour signature `(cn, ch, funcid=None)`.

4 Binding concernant toute l'application

4.1 Methode `bind_all`

La méthode permettant de créer un binding avec l'application toute entière est `bind_all` :

`w.bind_all(ch=None, func=None, add=None)` :

- `ch` : séquence représentant un type d'évènement (chaîne) ;
- `func` : fonction (callback) ;
- `add` : une seule valeur possible, `"+"` ;
- effet : Un appel `w.bind_all()` sans argument retourne la liste des séquences impliquées dans un *bind all*. Un appel `w.bind_all(ch)` retourne la chaîne-séquence des callbacks associés à la séquence `ch` dans un *bind all*. Un appel `w.bind_all(ch, func)` associe tous les widgets de l'application à la séquence `ch` et la fonction `func`. Chaque fois que l'évènement `ch` aura lieu, le callback `func` sera appelé. S'il existe déjà un binding entre l'application et `ch`, celui-ci vient le remplacer (le nouveau callback écrase l'ancien), sauf si on a pris « `add = "+"` », auquel cas, le nouveau binding s'ajoute à l'ancien. La méthode retourne l'identité du binding sous la forme d'une chaîne. Cette chaîne contient un code suivi du nom du callback. Tkinter appelle le callback en faisant passer un objet de type `Event` contenant toutes les propriétés de l'évènement détecté.

Note. L'identité du binding (ie la chaîne retournée) ne sert à rien. En effet, la méthode `unbind_all` (qui désactive un *bind all*, voir plus bas) ne prévoit pas la désactivation d'un callback en particulier.

Le programme ci-dessous associe l'évènement « appuyer sur ESCAPE », l'application toute entière et la fonction `action` :

```
from tkinter import *
def action(event): racine.destroy()
racine = Tk()
label = Label(racine, text="Touche ESC pour quitter")
label.pack()
racine.bind_all("<KeyPress-Escape>", action)
racine.mainloop()
```

Le lecteur essaiera ce programme en prenant `racine.quit`, pour voir la différence.

La séquence `"<KeyPress-Escape>"` signifie « on appuie sur la touche ESCAPE ». On aurait pu se passer de `action` et écrire :

```
racine.bind_all("<KeyPress-Escape>", racine.destroy)
```

tout simplement (nous avons voulu donner une forme plus générale).

4.2 Methode `unbind_all`

On peut défaire un binding général avec `unbind_all` :

`w.unbind_all(ch)` :

- `ch` : séquence représentant un type d'évènement (chaîne) ;
- effet : désactive tous les bindings `(all, ch, *)`.

5 Séquences

Dans le contexte de la programmation évènementielle, le mot « séquence » désigne une chaîne de caractères décrivant un type d'évènement. Cette section montre comment écrire une séquence.

5.1 Syntaxe générale

Nous savons qu'un *binding* relie :

- une séquence (chaîne désignant un évènement) ;
- une cible (widget, classe de widgets ou l'application entière) ;
- un gestionnaire d'évènement (fonction ou méthode).

De manière générale, une séquence est une chaîne ressemblant à ceci :

"<modificateurs-type-détail>"

Il peut y avoir zéro, un ou plusieurs modificateurs séparés par un tiret.

Exemples. 1. La séquence correspondant à l'évènement « on presse CTRL-Q » est "<Control-KeyPress-q>". Le modificateur est "Control", le type est "KeyPress" et le détail est "q".

2. La séquence correspondant à « CTRL-ALT-SUPPR » est "<Control-Alt-KeyPress-Delete>" (deux modificateurs).

3. La séquence correspondant à « widget reçoit le focus » est "<FocusIn>" (pas de modificateur, ni de détail).

5.2 Modificateurs

Sur un clavier, les modificateurs sont les touches qui modifient le comportement des autres touches lorsqu'elles sont enfoncées simultanément. La touche SHIFT (appelée aussi MAJ), par exemple, est un modificateur : lorsqu'on presse SHIFT et A en même temps, on obtient le A majuscule. On dit aussi que SHIFT est une touche de combinaison.

Chez Tkinter, les modificateurs correspondent aux touches de combinaison mais aussi à des indications comme par exemple "Any", "Double" ou "Triple" :

Modificateur	**Signification**
"Alt"	On appuie sur la touche ALT (*alternate key*)
"Control"	On appuie sur la touche CONTROL
"Lock"	On appuie sur la touche SHIFT LOCK
"Shift"	On appuie sur la touche SHIFT, appelée aussi MAJ
"Meta"	On appuie sur la touche portant le logo WINDOWS ou APPLE
"Mod1"	MAC OS : on appuie sur la touche COMMAND (*pomme*)
"Mod2"	MAC OS : on appuie sur la touche ALT
"Any"	Permet de généraliser (*)
"Double"	L'évènement a lieu 2 fois de manière très rapprochée
"Triple"	L'évènement a lieu 3 fois de manière très rapprochée
"Quadruple"	L'évènement a lieu 4 fois de manière très rapprochée.
"B1"	On appuie sur le bouton 1 de la souris.

(*) "<Any-KeyPress>", par exemple, signifie « on appuie sur une touche quelconque ».

Exemples. 1. La séquence correspondant à « double clic gauche » est "<Double-Button-1>" qu'on peut aussi écrire "<Double-1>".

2. Si MAJ n'est pas verrouillé, la séquence correspondant à « on appuie sur SHIFT-A » est "<Shift-A>", mais aussi "<A>".

3. Si MAJ est verrouillé, la séquence correspondant à « on appuie sur SHIFT-A » est "<Shift-a>", mais aussi "<a>". Dans cette situation, "<Shift-A>" ne correspond à rien.

5.3 Types

Nous ne donnons ici que les types les plus importants :

Type		**Signification**
"KeyPress" ou "Key"	2	On appuie sur une touche du clavier
"KeyRelease"	3	On relâche une touche du clavier
"ButtonPress" ou "Button"	4	On appuie sur un bouton de la souris (clic)
"ButtonRelease"	5	On relâche un bouton de la souris
"MouseWheel"	38	On fait tourner la molette de la souris (pas LINUX)
"Enter"	7	Le pointeur de la souris pénètre la surface du widget
"Motion"	6	Le pointeur de la souris bouge au dessus du widget
"Leave"	8	Le pointeur de la souris quitte le widget
"FocusIn"	9	Le widget reçoit le focus
"FocusOut"	10	Le widget perd le focus
"Activate"	36	L'état du widget (state) passe à *active*
"Deactivate"	37	L'état du widget (state) passe à *inactive*
"Expose"	12	Le widget est un peu moins recouvert
"Visibility"	15	Une partie du widget redevient visible
"Map"	19	Le widget est affiché (pack, grid, etc.)
"Unmap"	18	Le widget est retiré de l'affichage (grid_remove, etc.)
"Configure"	22	Le widget est redimensionné
"Destroy"	17	Le widget est détruit (destroy).

Notes. 1. La séquence "<MouseWheel>" fonctionne sous PC WINDOWS et MAC OS, mais pas LINUX. Sous LINUX, on utilise "<Button-4>" pour « molette tourne vers le haut » et "<Button-5>" pour « molette tourne vers le bas ». De manière générale, la distinction entre ces deux manières de tourner se fait grâce à l'attribut delta de l'évènement instancié (voir sous-section 39.7.2).

2. Les boutons de la souris n'ont pas le même numéro selon le système. Sur PC WINDOWS, bouton à gauche : 1, molette : 2, bouton à droite : 3. Sur MAC OS, clic gauche : 1, clic droit : 2, etc.

Exemples. 1. La séquence "<Key-a>" correspond à « on a pressé sur A ». Idem "<a>".

2. "<Alt-Key-a>" correspond à « on a pressé sur ALT-A ». Idem "<Alt-a>".

3. "<Any-Button>" correspond à « on a cliqué ». Idem "<Button>", "<ButtonPress>".

4. "<B1-Motion>" correspond à « la souris bouge tout en pressant le bouton gauche ».

5.4 Détails

Pour écrire la séquence correspondant à une touche régulière pressée, comme par exemple « on appuie sur la touche A », on prend comme type "KeyPress" et comme détail "a", ce qui donne : "<KeyPress-a>".

Si la séquence concerne une touche spéciale, comme par exemple « on appuie sur la touche SHIFT », le détail est le nom Tkinter de cette touche spéciale, à savoir "Shift_L" ou "Shift_R" (il y a deux touches SHIFT), ce qui donne "<KeyPress-Shift_L>" ou "<KeyPress-Shift_R>".

Voici le nom que donne Tkinter à chaque touche spéciale :

Touche	Nom anglais	Détail		
ENTRÉE	*Return*	"Return"	36	65293
MAJ GAUCHE	*Shift (left)*	"Shift_L"	50	65505
MAJ DROIT	*Shift (right)*	"Shift_R"	62	65506
CTRL GAUCHE	*Left-hand control*	"Control_L"	37	65507
CTRL DROIT	*Right-hand control*	"Control_R"	109	65508
ALT GAUCHE	*Left-hand alt*	"Alt_L"	64	65513
ALT DROIT	*Right-hand alt*	"Alt_R"	113	65514
ÉCHAPPEMENT	*Escape*	"Escape"	9	65307
TAB	*Tab*	"Tab"	23	65289
RETOUR ARRIÈRE	*Backspace* (*)	"BackSpace"	22	65288
SUPPRIMER	*Delete*	"Delete"	107	65535
INSERTION	*Insert*	"Insert"	106	65379
VERR. MAJ	*Caps lock*	"Caps_Lock"	66	65549
VERR. NUM	*Num lock*	"Num_Lock"	77	65407
↑		"Up"	98	65362
↓		"Down"	104	65364
←		"Left"	100	65361
→		"Right"	102	65363
DÉBUT ↖	*Home*	"Home"	97	65360
FIN	*End*	"End"	103	65367
PAGE PRÉC.	*Page up*	"Prior"	99	65365

PAGE SUIV.	*Page down*	"Next"	105	65366
F1	*Function 1*	"F1"	67	65470
F2	*Function 2*	"F2"	68	65471
F3	*Function 3*	"F3"	69	65472
F4	*Function 4*	"F4"	70	65473
F5	*Function 5*	"F5"	71	65474
F6	*Function 6*	"F6"	72	65475
F7	*Function 7*	"F7"	73	65476
F8	*Function 8*	"F8"	74	65477
F9	*Function 9*	"F9"	75	65478
F10	*Function 10*	"F10"	76	65479
F11	*Function 11*	"F11"	77	65480
F12	*Function 12*	"F12"	96	65781
IMPRIME ÉCRAN	*Print screen*	"Print"	111	65377
SYSTÈME	*System request*	"Execute"	111	65378
ARRÊT DÉFIL.	*Scroll lock*	"Scroll_lock"	78	65300
PAUSE	*Pause*	"Pause"	110	65299
PAUSE BREAK	*Pause Break*	"Cancel"	110	65387
CRLF (CTRL-J)	*Carriage return line feed*	"Linefeed"	54	106
ESPACE	*Space*	"space"		32
<		"less"		60
>		"greater"		62

(*) Touche parfois appelée DELETE en France. Das les pays anglosaxons, c'est la touche SUPPRIMER qui est appelée DELETE.

Note. Le détail s'appelle aussi `keysym` (abréviation de *key symbol*). Les deux colonnes de droite donnent les numéros `keycode` et `keysym_num`.

Certaines touches régulières possèdent un nom, comme par exemple [<] qui s'appelle "less". On peut donc écrire "<KeyPress-less>" ou "<KeyPress-<>", cela revient au même. La touche [@] s'appelle "at", la touche [&] s'appelle "ampersand", etc. Nous n'avons pas besoin de connaître tous ces noms. Néanmoins, si le lecteur désire connaître le nom d'une touche régulière, il affichera l'attribut `keysym` d'un évènement capturé grâce à un binding associant "<KeyPress>" à un callback (voir programme donné à la sous-section 39.7.5).

Note. Nous avons abrégé certains noms dans la colonne de gauche : VERROUILLAGE MAJUSCULE (VERR. MAJ), CONTRÔLE (CTRL), PAGE SUIVANTE (PAGE SUIV.), VERROUILLAGE NUMÉRIQUE (VERR. NUM), PAGE PRÉCÉDENTE (PAGE PRÉC.), ARRÊT DÉFILEMENT (ARRÊT DÉFIL.), MAJUSCULE (MAJ) et TABULATION (TAB).

Ci-dessous le nom (servant de détail) pour les touches spéciales du pavé numérique (*key pad*) :

Touche	Nom anglais	Détail		
0	*key pad 0*	`"KP_0"`	90	65438
1	*key pad 1*	`"KP_1"`	87	65436
2	*key pad 2*	`"KP_2"`	88	65433
3	*key pad 3*	`"KP_3"`	89	65435
4	*key pad 4*	`"KP_4"`	83	65430
5	*key pad 5*	`"KP_5"`	84	65437
6	*key pad 6*	`"KP_6"`	85	65432
7	*key pad 7*	`"KP_7"`	79	65429
8	*key pad 8*	`"KP_8"`	80	65431
9	*key pad 9*	`"KP_9"`	81	65434
.		`"KP_Decimal"`	91	65439
+		`"KP_Add"`	86	654351
−		`"KP_Substract"`	82	65453
∗		`"KP_Multiply"`	63	65450
/		`"KP_Divide"`	112	65455
ENTRÉE	*Enter*	`"KP_Enter"`	108	65421
↑		`"KP_Up"`	80	65431
↓		`"KP_Down"`	88	65433
←		`"KP_Left"`	83	65430
→		`"KP_Right"`	85	65432
DÉBUT	*Home*	`"KP_Home"`	79	65429
FIN	*End*	`"KP_End"`	87	65436
PAGE PRÉC.	*Page up*	`"KP_Prior"`	81	65434
PAGE SUIV.	*Page down*	`"KP_Next"`	89	65435
SUPPRIMER	*Delete*	`"KP_Delete"`	91	65439
INSÉRER	*Insert*	`"KP_Insert"`	90	65438
BEGIN	*Begin*	`"KP_Begin"`	84	65437

Nous terminons avec les boutons de la souris :

Nom usuel	Détail (PC)	Détail (Mac)
Bouton gauche	`"1"`	`"1"`
Bouton milieu	`"2"`	—
Bouton droit	`"3"`	`"2"`
Molette haut	`"3"`	`"3"`
Molette bas	`"4"`	`"4"`
Molette	`"5"`	`"5"`

Note. On ne confondra pas le modificateur `"B1"` et le détail `"1"`, même s'il est vrai que ces deux codes concernent le bouton gauche de la souris.

Exemples. 1. La séquence correspondant à « on appuie sur CTRL-TAB » s'écrit `"<Control-KeyPress-Tab>"`. Idem `"<Control-Tab>"`.

2. La séquence « on appuie sur SHIFT-CTRL » s'écrit `"<Shift-KeyPress-Control_L>"` s'il s'agit de la touche CTRL de gauche. Idem `"<Shift-Control_L>"`.

3. La séquence « CTRL clic gauche » s'écrit `"<Control-ButtonPress-1>"`. Idem `"<Control-1>"`.

4. La séquence « SHIFT-CTRL relâchement clic droit » s'écrit `"<Shift-Control-ButtonRelease-3>"` (PC WINDOWS).

5. La séquence « Entrée du clavier » s'écrit `"<KeyPress-Return>"`. Idem `"<Return>"`.

6. La séquence « Entrée du pavé numérique » s'écrit `"<KeyPress-KP_Enter>"`. Idem `"<KP_Enter>"`.

Note. On a vu dans les exemples des séquences abrégées. Il existe de nombreuses abréviations comme par exemple `"a"` ou `"<a>"` à la place de `"<KeyPress-a>"`. Il y a aussi `"<1>"` pour `"<ButtonPress-1>"`, etc. On ne confondra pas avec `"1"` qui signifie `"<KeyPress-1>"`.

5.5 Touche régulière et focus

L'association « on presse ESPACE » avec un cadre ne fonctionne pas :

```
from tkinter import *
def changer(event):
    global i
    n = len(couleurs)
    valeur = couleurs[i]
    frame.configure(bg=valeur)
    i = (i + 1) % n
couleurs = ["red", "green", "blue", "yellow"]
i = 0
racine = Tk()
frame = Frame(racine, width=300, height=300)
frame.grid()
frame.bind("<space>", changer)
racine.mainloop()
```

Le lecteur vérifiera que presser sur la barre d'espace ne change pas la couleur lors de l'exécution.

Cela est dû au fait que le cadre n'a pas le focus pendant que l'on presse la barre d'espace. La preuve, si l'on reprend ce programme en ajoutant « `takefocus = True` » dans `frame` :

```
frame = Frame(racine, width=300, height=300, takefocus=True)
```

et qu'on appuie sur la touche TAB juqu'à ce que le cadre prenne le focus, alors le fait de presser la barre d'espace modifie bien la couleur du cadre.

Le programme peut donner le focus au cadre en appelant sa méthode `focus_set`.

On retiendra ceci : associer un évènement de type « touche pressée » à un widget n'ayant pas le focus peut être vain.

On peut modifier le programme comme ceci : au lieu d'associer le cadre à "<space>" et changer, on associe la fenêtre principale :

```
racine.bind("<space>", changer)
```

6 Séquences virtuelles

Une séquence virtuelle est un nom regroupant plusieurs séquences. Nous préférons le terme de super-séquence, plus proche selon nous de la réalité. Pour définir une super-séquence, on utilise la méthode universelle event_add :

event_add(ch, *args) :

- ch : nom de séquence virtuelle (chaîne) ;
- *args : suite de séquences (chaînes) ;
- effet : regroupe toutes les séquences passées en *args en une seule super-séquence nommée avec ch. Si la super-séquence ch a déjà été définie, la méthode ajoute les séquences passées en *args à la super-séquence.

Attention. Le nom d'une séquence virtuelle doit commencer par "<<" et s'achever par ">>".

Le programme ci-dessous regroupe les séquences « ESCAPE », « ENTRER » et « ESPACE » en une super-séquence "<<spécial>>". De même, il regroupe « A », « SHIFT A », « Z » et « SHIFT Z » en la super-séquence "<<alerte>>" :

```
from tkinter import *
def changer(event):
    global i
    n = len(couleurs)
    valeur = couleurs[i]
    frame.configure(bg=valeur)
    i = (i + 1) % n
def quitter(event): racine.destroy()
couleurs = ["red", "green", "blue", "yellow"]
i = 0
racine = Tk()
racine.event_add("<<spécial>>", "<Escape>", "<Return>", "<space>")
racine.event_add("<<alerte>>", "<a>", "<z>", "<A>", "<Z>")
frame = Frame(racine, width=300, height=300)
frame.grid()
racine.bind("<<spécial>>", changer)
racine.bind("<<alerte>>", quitter)
racine.mainloop()
```

Une séquence virtuelle peut regrouper autant de séquences qu'on veut. Il est par ailleurs possible de retirer une ou plusieurs séquences à une super-séquence, grâce à la méthode universelle `event_delete` :

`event_delete(ch, *args)` :

- `ch` : nom de séquence virtuelle (chaîne) ;
- `*args` : suite de séquences (chaînes) ;
- effet : retire de la super-séquence `ch` les séquences passées en `*args`.

Il est clair que si l'on retire toutes les séquences à une super-séquence, cette dernière devient *vide* et ne correspond plus à aucun évènement (notre super-séquence ne se réalise jamais).

On peut obtenir des informations sur une séquence virtuelle grâce à la méthode `event_info` :

`event_info(ch=None)` :

- `ch` : séquence virtuelle (chaîne) ;
- effet : un appel `event_info()` sans argument retourne la liste des séquences virtuelles. Un appel `event_info(ch)` retourne la liste des séquences composant la super-séquence `ch`. Si `ch` ne correspond à aucune super-séquence, la méthode retourne `None`.

Reprenons le programme précédent pour y ajouter 3 boutons :

```
(...)
def séquences(): print(racine.event_info())
def spécial(): print(racine.event_info("<<spécial>>"))
def alerte(): print(racine.event_info("<<alerte>>"))
(...)
b1 = Button(racine, text="séquences", command=séquences)
b1.grid()
b2 = Button(racine, text="<<spécial>>", command=spécial)
b2.grid()
b3 = Button(racine, text="<<alerte>>", command=alerte)
b3.grid()
```

Voici l'affichage provoqué par le bouton SPÉCIAL :

```
('<Key-Escape>', '<Key-Return>', '<Key-space>')
```

Voici l'affichage provoqué par le bouton ALERTE :

```
('a', 'z', 'A', 'Z')
```

Et le meilleur pour la fin, l'affichage provoqué par SÉQUENCES, c'est-à-dire toutes les séquences traitées par Tkinter pendant l'exécution de ce programme :

```
('<<alerte>>', '<<NextWindow>>', '<<NextLine>>', '<<PrevPara>>',
'<<Cut>>', '<<Paste>>', (...), '<<SelectLineStart>>', '<<Copy>>',
'<<Redo>>', '<<PrevWord>>', '<<NextWord>>')
```

On notera la présence de super-séquences internes facilement reconnaissables comme <<Cut>>, <<Paste>> et <<Copy>>.

7 Objets de type Event

Chaque fois qu’a lieu un évènement correspondant à la séquence d’un binding, Tkinter crée un objet de type Event et appelle le callback associé en faisant passer cet objet en argument. C’est pour cette raison que les callbacks impliqués dans un binding possèdent un paramètre event en signature (c’est le nom d’usage).

L’instance de Event générée est un enregistrement de toute l’information concernant l’évènement ayant déclenché le callback. Si par exemple l’évènement capté est un clic-gauche, l’objet de type Event correspondant possède des attributs x et y contenant les coordonnées du lieu où s’est produit le clic.

De manière générale, les attributs d’un objet de type Event sont : char, delta, focus, height, keycode, keysym, keysym_num, num, serial, state, time, type, widget, width, x, x_root, y et y_root. Cette section décrit le contenu de ces attributs.

7.1 Touche pressée

Attribut char. On a pressé une touche régulière (c’est l’évènement). Dans ce cas, char contient le caractère obtenu grâce à cet évènement. Si par exemple on a pressé A, char contient "a". En revanche, si on a pressé SHIFT-A, char contient "A". On invite le lecteur à tester des combinaisons comme par exemple ALT-A (ce dernier donne "æ").

Attribut keysym. On a pressé une touche spéciale (c’est l’évènement). Dans ce cas, keysym (*key symbol*) contient le nom de cette touche (voir 39.5.4). Si par exemple on a pressé CONTROL-GAUCHE, keysym contient "Control_L".

Attribut keysym_num. Première possibilité, on a pressé une touche régulière. Dans ce cas keysym_num contient le code ASCII du caractère obtenu grâce à cet évènement. Deuxième possibilité, on a pressé une touche spéciale. Dans ce cas, keysym_num contient un nombre associé à cette touche spéciale (*key symbol number*). Dans tous les cas, keysym_num permet d’identifier la touche pressée (la combinaison pressée). Par exemple « on a pressé A » donne 97, tandis que « on a pressé SHIFT-A » donne 65.

Attribut keycode. On a pressé une touche. Dans ce cas, keycode contient un nombre associé à cette touche (*key code*). Ce nombre permet d’identifier la touche pressée mais ne dit rien sur un éventuel modificateur pressé conjointement, de sorte que A et SHIFT-A, par exemple, donnent un keycode égal à 786529 (voir state, sous-section 39.7.4).

7.2 Action de la souris

Attribut num. On a cliqué (c’est l’évènement). Dans ce cas, num contient le numéro du bouton qui a été pressé. Sur PC WINDOWS, bouton à gauche : 1 ; bouton au milieu : 2 ; bouton à droite : 3. Sur MAC OS, bouton à gauche : 1 ; bouton à droite : 2. Sous LINUX, molette qui tourne vers le haut : 4, molette qui tourne vers le bas : 5.

Attribut delta. On a fait tourner la molette de la souris. Dans ce cas, sous MAC OS, delta contient le nombre de crans que l’on a tournés avec un signe + ou −, selon que l’on a tourné positivement ou négativement (*scroll up* ou *scroll down*). Sous PC WINDOWS, delta contient 120 fois le nombre algébrique de crans.

Attributs x et y. On a cliqué (c'est l'évènement). Dans ce cas, x et y contiennent les coordonnées du lieu du clic dans le repère propre au widget concerné.

Attributs x_root et y_root. On a cliqué. Dans ce cas, x_root et y_root contiennent l'abscisse et l'ordonnée du lieu du clic calculés dans le système de coordonnées de l'écran.

7.3 Redimensionnement

Attribut width. On a redimensionné une fenêtre (c'est l'évènement). Dans ce cas, width contient la nouvelle longueur mesurée en pixels.

Attribut height : On a redimensionné une fenêtre (c'est l'évènement). Dans ce cas, height contient la nouvelle hauteur mesurée en pixels.

7.4 Évènement quelconque

Attribut type. Contient le type de l'évènement : "FocusIn", "KeyPress", etc.

Attribut widget. Contient le nom complet Tkinter du widget concerné.

Attribut state. Contient les modificateurs pressés pendant l'évènement.

Attribut time. Contient un entier permettant de repérer l'ordre chronologique des évènements ainsi que l'intervalle de temps séparant deux évènements. Cet entier augmente de 1 chaque milliseconde. Cet entier n'a pas de signification absolue, ce sont ses variations qui nous intéressent.

Attribut serial. Contient un entier incrémenté chaque fois que le serveur répond à une requête. On peut l'utiliser pour connaître l'ordre chronologique des évènements.

7.5 Exemples

Le programme ci-dessous permet d'étudier les objets de type Event générés au clavier ou à la souris sur une zone de saisie ou sur un cadre :

```
from tkinter import *
def action(evt):
    BR = "\n\n"
    TRAIT = "=" * (len(str(evt)) + 6)
    DP = " : "
    ch = "evt = " + str(evt) + BR + "type = "
    ch = ch + str(type(evt)) + BR + TRAIT + BR
    dico = evt.__dict__
    for k in dico:
        ch = ch + k + DP + str(dico[k]) + BR
    label.configure(text=ch)
racine = Tk()
racine.event_add("<<evt>>", "<ButtonPress>", "<KeyPress>")
cadre = Frame(racine, bg="white", height=100, width=500,
              name="cadre", takefocus=1)
```

```
cadre.grid()
champ = Entry(racine, name="champ")
champ.grid(sticky=EW)
cadre.bind("<<evt>>", action)
champ.bind("<<evt>>", action)
label = Label(racine, bg="grey90", font="courier", justify=LEFT)
label.grid(stick=EW)
racine.mainloop()
```

La super-séquence `"<<evt>>"` regroupe `"<ButtonPress>"` et `"<KeyPress>"`. Un binding associe le cadre, la séquence `"<<evt>>"` et le callback `action`. Un autre binding associe le champ de saisie, `"<<evt>>"` et `action`. Le callback affiche l'objet `Event` généré, son type et son dictionnaire (attributs et valeurs). Voici ce qui est affiché lorsqu'on clique sur le cadre :

```
evt = <ButtonPress event num=1 x=113 y=56>
type = <class 'tkinter.Event'>
========================================================================
serial : 67             width : ??              keysym_num : ??
num : 1                 x : 113                 type : ButtonPress
height : ??             y : 56                  widget : .cadre
keycode : ??            char : ??               x_root : 139
state : 0               send_event : False      y_root : 129
time : -1550361192      keysym : ??             delta : 0
```

L'attribut `type` contient `"ButtonPress"` et `num` vaut `1` : il s'agit bien d'un clic gauche. L'attribut `widget` contient `".cadre"` (nom complet du cadre). Le clic a eu lieu au point de coordonnées `(113,56)` dans le système de coordonnées du cadre. Les attributs non définis contiennent la chaîne `"??"`.

Attention. Ne pas confondre le type de l'objet (*Python type*) et le type de l'évènement (*event type attribute*).

Voici ce qu'on obtient lorsqu'on presse la touche A du clavier alors que la zone de saisie possède le focus (PC WINDOWS) :

```
evt = <KeyPress event keysym=a keycode=786529 char='a' delta=786529
       x=-26 y=-173>
type = <class 'tkinter.Event'>
========================================================================
serial : 545            width : ??              keysym_num : 97
num : ??                x : -26                 type : KeyPress
height : ??             y : -173                widget : .champ
keycode : 786529        char : a                x_root : 0
state : 0               send_event : False      y_root : 0
time : -1548933765      keysym : a              delta : 786529
```

Sur MAC OS, pour ce même évènement, `keycode` vaut `97` et `delta` vaut `0`.

Si on presse CTRL-A, le programme capture 2 évènements : celui correspondant à la touche CTRL et celui correspondant à CTRL-A. On prêtera attention à l'attribut `state`. On peut ajouter l'instruction `print(evt)` au callback `action` pour conserver une trace des deux évènements, on obtient alors (dans le shell) :

```
<KeyPress event state=Control keysym=Control_L keycode=262145 x=-34 y=-87>
<KeyPress event state=Control keysym=a keycode=786433 char='\x01' x=-109 y=-87>
```

De même, on obtient 3 évènements quand on tape CTRL-ALT-SHIFT-A, le dernier étant :

```
<KeyPress event state=Shift|Control|Mod2 keysym=A keycode=786433 char='\x01' delta=786433 x=-26 y=-173>
```

On peut écrire un programme un peu plus sophistiqué pour disséquer d'autres types d'évènements :

```
from tkinter import *
def analyse(event):
    x = racine.focus_get()
    txt = patron.format(x)
    label.configure(text=txt)
    dico = event.__dict__
    for k in dico:
        i = attributs.index(k)
        valeur = dico[k]
        if k == "type": valeur = str(valeur)         #
        labels[i].configure(text=valeur)
patron = "Focus sur : {}"
racine = Tk()
evts = ("<ButtonPress>", "<KeyPress>", "<MouseWheel>", "<Enter>",
        "<Leave>", "<FocusIn>", "<FocusOut>")
racine.event_add("<<evt>>", *evts)
attributs = ["type", "widget", "state", "char", "keysym", "keycode",
             "keysym_num", "num", "x", "y", "x_root", "y_root", "delta",
             "focus", "width", "height", "time", "serial", "send_event"]
label = Label(racine, text="Focus sur:")
label.grid(row=0, column=1, columnspan=2)
labels = []
i = 5
for ch in attributs:
    l1 = Label(text=ch, justify=RIGHT, bg="pink", font="courier")
    l1.grid(row=i, column=1, sticky=E)
    l2 = Label(text="...")
    l2.grid(row=i, column=2, sticky=W)
    labels.append(l2)
```

```
    i = i + 1
cadre1 = Frame(racine, width=400,height=100, bg="yellow", name="cadre1")
cadre1.grid(row=1, column=1, columnspan=2)
cadre1.bind("<<evt>>", analyse)
cadre2 = Frame(racine, width=400, height=100, bg="orange",
               name="cadre2", takefocus=1)
cadre2.grid(row=2, column=1, columnspan=2)
cadre2.bind("<FocusOut>", analyse)
entry1 = Entry(racine, bg="yellow2", name="entry1")
entry1.grid(row=3, column=1, columnspan=2, sticky=EW)
entry1.bind("<<evt>>", analyse)
entry2 = Entry(racine, bg="yellow4", name="entry2")
entry2.grid(row=4, column=1, columnspan=2, sticky=EW)
entry2.bind("<<evt>>", analyse)
racine.mainloop()
```

Le programme crée un cadre jaune, un cadre orange et deux champs de saisie jaunes. Tous ces widgets sont associés à la super-séquence `"<<evt>>"`, sauf le cadre orange qui n'est associé qu'à `"<FocusOut>"`. La super-séquence `"<<evt>>"` regroupe presque tous les types d'évènements. On notera que le cadre orange est inclus dans le cycle du focus (le lecteur tapera sur TAB pour changer le focus).

On invite le lecteur à tester toutes sortes d'évènements sur ces widgets et observer la valeur de leurs attributs.

Note. Si jamais on travaille sur un système sur lequel `event_add` ne fonctionne pas correctement, on écrit une boucle `for` sur l'uplet `evts` effectuant les ajouts `event_add` un par un. Au pire, on écrit une boucle `for` activant un par un chaque binding.

On constate une chose curieuse avec l'attribut `type` d'un évènement. Cet attribut contient un objet de type `<enum 'EventType'>` (ce type est lui-même de type `<class 'enum.EnumMeta'>`, revoir les métaclasses dans [?]) et l'affichage d'un tel objet dans une étiquette produit un entier. Voilà pourquoi nous le transformons avec la fonction `str` avant de l'afficher (ligne #). Si on ajoute ces objets de type `<enum 'EventType'>` dans une liste `capture` (le lecteur modifiera la fonction `analyse` pour cela), voici ce que l'on obtient quand on revient au shell :

```
>>> x = capture[0]
>>> type(x)
<enum 'EventType'>
>>> type(type(x))
<class 'enum.EnumMeta'>
>>> x
<EventType.Enter: '7'>
>>> str(x)
'Enter'
>>> repr(x)
```

```
"<EventType.Enter: '7'>"
>>> x.value
'7'
```

(Le lecteur constatera que le type `EventType` dérive de `str` et `Enum`.)

Autre observation. C'est grâce au cadre orange qu'on peut capturer des évènements de type `"FocusOut"` (précisément quand grâce à TAB, le focus quitte `entry2` pour aller sur le cadre orange). Notons par ailleurs que nous n'arrivons jamais à générer un évènement dont l'attribut `focus` est égal à `True`.

8 Priorités

Supposons que `w` est un widget et `func1` et `func2` sont des fonctions. Supposons que l'on a créé deux associations (bindings) :

`(w, "<KeyPress>", func1)` et `(w, "<KeyPress-a>", func2)`

Que fait alors Tkinter lorsque l'utilisateur presse la touche A ? Déclenche-t-il les deux callbacks `func1` et `func2`, ou seulement `func2` ?

Autrement dit, quelles sont les règles de priorité appliquées par Tkinter sur les bindings ?

8.1 Priorité à la séquence la plus précise

Tkinter obéit à la règle suivante : la séquence la plus précise l'emporte. Par exemple, si on associe `"<KeyPress>"` à une fonction `func1` et `"<KeyPress-a>"` à une fonction `func2`, l'évènement « on presse A » déclenche `func2` mais pas `func1`. Si par dessus le marché, on associe `"<Control-KeyPress-a>"` à une fonction `func3`, alors l'évènement « on presse CTRL-A » ne déclechera pas `func2` mais `func3`. En effet, `"<Control-KeyPress-a>"` est plus précise que `"<KeyPress-a>"`. Testons sur un exemple :

```
from tkinter import *
def touche(event): print("Une touche")
def a(event): print("Touche a")
def ctrl_a(event): print("Control a")
racine = Tk()
entry = Entry(racine)
entry.grid()
entry.bind("<KeyPress>", touche)
entry.bind("<KeyPress-a>", a)
entry.bind("<Control-KeyPress-a>", ctrl_a)
racine.mainloop()
```

Résultat : quand l'utilisateur met le focus sur le champ de saisie et appuie sur une touche autre que A, le programme affiche `"Une touche"`. Quand il appuie sur A, il affiche `"Touche a"`. Quand il appuie sur CTRL-A, il affiche `"Une touche"` et `"Control a"` car dans ce cas, Tkinter réagit à deux évènements : « on a pressé sur une touche (la touche CTRL) » et « on a pressé sur CTRL-A ».

Tout cela est conforme à ce qu'on a l'habitude de voir dans les applications courantes. D'ailleurs, quand on appuie sur CTRL-A, la lettre a n'apparaît pas dans le champ de saisie, ce qui est bien normal.

8.2 Priorité au widget le plus proche

Un binding est un triplet (s, w, f) où s est une séquence, w est le niveau du binding (un widget, une classe ou l'application toute entière) et f une fonction. Supposons qu'un widget W contient un widget w et (s, w, f) et (s, W, g) sont des bindings. Dans ce cas, si l'utilisateur met le focus sur w et provoque un évènement correspondant à s, Tkinter déclenche d'abord f et ensuite g. De manière générale, le callback du niveau le plus proche du lieu de l'évènement est appelé en premier.

Le programme ci-dessous dessine un cadre gris (cadre principal) dans lequel sont placés un cadre vert et un cadre jaune. Il y a un binding impliquant la fenêtre principale, un autre impliquant le cadre gris et un autre le cadre vert. Tous ces bindings sont associés au clic gauche :

```
from tkinter import *
def fonc_racine(evt): print("Bind (racine, clic, fonc_racine)\n\n\n")
def fonc_cadre_principal(evt):
    print("Bind (cadre principal, clic, fonc_cadre_principal)\n\n")
def fonc_cadre_vert(evt):
    print("Bind (cadre vert, clic, fonc_cadre_vert\n")
racine = Tk()
racine.bind("<ButtonPress-1>", fonc_racine)
cadre = Frame(racine, bg="grey", width=600, height=400)
cadre.grid()
cadre.grid_propagate(False)
cadre.grid_anchor(N)
cadre.bind("<ButtonPress-1>", fonc_cadre_principal)
lab1 = Label(cadre, text="cadre principal")
lab1.grid(row=1, column=1, columnspan=2, pady=8)
invisible = Frame(cadre, bg="grey", width=60, height=60)
invisible.grid(row=2, column=1)
vert = Frame(cadre, bg="green", width=220, height=200)
vert.grid(row=3, column=1, padx=20)
vert.grid_propagate(False)
vert.bind("<ButtonPress-1>", fonc_cadre_vert)
lab2 = Label(vert, text="cadre vert dans cadre principal")
lab2.grid(pady=4)
jaune = Frame(cadre, bg="yellow", width=220, height=200)
jaune.grid(row=3, column=2, padx=20)
jaune.grid_propagate(False)
```

```
lab3 = Label(jaune, text="cadre jaune dans cadre principal")
lab3.grid(pady=4)
racine.mainloop()
```

Résultat. Le clic sur le cadre principal (gris) affiche ceci :

```
Bind (cadre principal, clic, fonc_cadre_principal)
Bind (racine, clic, fonc_racine)
```

Le clic sur le cadre vert affiche ceci :

```
Bind (cadre vert, clic, fonc_cadre_vert
Bind (racine, clic, fonc_racine)
```

On note que Tkinter considère que l'on n'a pas cliqué sur le cadre gris (bien que le vert soit dans le gris). En revanche, il considère que nous avons cliqué sur la fenêtre principale (`racine`).

Le clic sur cadre jaune affiche ceci :

```
Bind (racine, clic, fonc_racine)
```

La logique est la même : Tkinter considère que nous n'avons pas cliqué sur le cadre gris (même si le jaune est dans le gris).

9 Notion de *tag binding*

Le mécanisme du binding est en fait plus compliqué que ce qui a été décrit aux sections 39.1, 39.2, 39.3 et 39.4. En vérité, un binding associe

- une séquence ;
- un *tag* ;
- et un callback (gestionnaire d'évènement).

Supposons que `bouton` est un bouton placé dans la fenêtre principale de `name` égal à `"coco"` et donc de nom complet `".coco"`. Si on écrit un binding (méthode `bind`) reliant le clic gauche à ce bouton, Tkinter crée un binding entre `"<ButtonPress-1>"` et le *tag* `".coco"`. Autrement dit, le tag pris en compte est le nom complet du bouton.

Si on écrit un binding reliant le clic gauche à tous les boutons (méthode `bind_class`), Tkinter crée un binding entre `"<ButtonPress-1>"` et le tag `"Button"`. Autrement dit, le tag pris en compte est la « classe Tkinter des boutons ».

Si on écrit un binding reliant le clic gauche à toute l'application (méthode `bind_all`), Tkinter crée un binding entre `"<ButtonPress-1>"` et le tag `"all"`.

De manière générale, un *tag* est une chaîne de caractères. Dans les cas courants, cette chaîne est le nom Tkinter d'un widget, d'une classe Tkinter ou le mot `"all"`, mais on peut, si on le désire, créer ses propres tags.

Pour comprendre comment on crée un tag, il faut savoir comment Tkinter gère les bindings.

Chaque widget possède un uplet de tags. Par défaut, les tags d'un bouton de nom complet `".coco"`, par exemple, sont `".coco"`, `"Button"`, `"."` et `"all"` :

```
>>> racine = Tk()
>>> bouton = Button(racine, text="feu", name="coco")
>>> bouton.grid()
>>> bouton.bindtags()
('.coco', 'Button', '.', 'all')
```

Si un évènement correspondant à la séquence `"<ceci>"` a lieu au niveau du widget `bouton`, Tkinter regarde s'il existe un binding liant `"<ceci>"` au premier tag, à savoir `".coco"`. Si un tel binding existe, Tkinter appelle le callback associé.

Ensuite, Tkinter passe au tag suivant et regarde s'il existe un binding reliant `"<ceci>"` à ce tag, en l'occurence ici `"Button"`. Si un tel binding existe, le callback associé est appelé.

Le processus se poursuit jusqu'au dernier tag du widget en appliquant la règle suivante : si l'un des callbacks appelés retourne `"break"`, le processus s'arrête.

On note au passage que l'ordre des tags du widget `bouton` est important. On comprend mieux d'où viennent les règles de priorité énoncées à la section précédente.

Si on désire que le tag `"mon_tag"` concerne le bouton `bouton`, on utilise la méthode `bouton.bindtags` :

```
>>> uplet = (".coco", "Button", ".", "all", "mon_tag")
>>> bouton.bindtags(uplet)
```

Le fait d'avoir mis `"mon_tag"` en dernier fait que Tkinter exécutera d'abord les éventuels callbacks liés à `bouton`, aux boutons en général et à l'application avant d'exécuter celui qui est associé à `"mon_tag"`. On peut bien sûr changer l'ordre.

De manière générale, si on veut ajouter `"mon_tag"` à la fin de la liste des tags d'un widget `widget`, on écrit ceci :

```
uplet = widget.bindtags()
uplet = uplet + ("mon_tag",)
widget.bindtags(uplet)
```

Finalement, tout repose sur la méthode `bindtags`.

`w.bindtags(t=None)` :

- `t` : uplet de tags (un tag est une chaîne de caractères) ;
- effet : un appel `w.bindtags()` sans argument retourne l'uplet des tags du widget `w`. Un appel `w.bindtags(t)` fait de `t` le nouvel uplet des tags de `w`.

Pour créer un binding associant un tag à une séquence et un callback, on utilise la méthode (déjà rencontrée) `bind_class`.

`w.bind_class(tag, ch=None, func=None, add=None)` :

- `tag` : un tag (chaîne) ;
- `ch` : séquence représentant un type d'évènement (chaîne) ;
- `func` : fonction (callback) ;
- `add` : une seule valeur possible, `"+"` ;
- effet : un appel `bind_class(tag, ch, f)` associe le tag spécifié, la séquence `ch` et la fonction `f`. Chaque fois que l'évènement `ch` aura lieu sur un widget possédant ce tag, le callback `f` sera appelé. S'il existe déjà un binding entre `tag` et `ch`, celui-ci vient le remplacer (le nouveau callback écrase l'ancien), sauf si on prend « `add = "+"` », auquel cas, le nouveau binding s'ajoute à l'ancien.

Tkinter aurait pu proposer une méthode `bind_tag`, pour plus de clarté.

10 Évènements générés artificiellement

Le programme peut générer artificiellement un évènement qui n'a pas eu lieu réellement. Par exemple, pour générer l'évènement « on appuie sur la touche A », on appelle la méthode universelle `event_generate` :

```
w.event_generate("<KeyPress-A>")
```

Ici, `w` est un widget quelconque. Résultat, au moment où Python exécute cette ligne, chaque binding associant `"KeyPress-A"` avec un item du *bindtags* de `w` enclenche son callback, comme si l'utilisateur avait vraiment pressé sur la touche A au niveau de `w`.

`w.event_generate(ch, **kwargs)` :

- `ch` : séquence (chaîne) ;
- `**kwargs` : dictionnaire sous la forme « `attribut = valeur` » ;
- effet : génère un évènement correspondant à la séquence `ch`. Les attributs passés dans `**kwargs` sont affectés à l'instance de `Event` ainsi créée.

Note. La valeur par défaut donnée à l'attribut `widget` de l'évènement généré est `w`. L'évènement est alors considéré comme ayant lieu au niveau de `w`.

Ci-dessous, on génère un clic gauche sur le cadre de nom complet `".cadre"`, au point de coordonnées `(40,50)` :

```
w.event_generate("<ButtonPress-1>", widget=".cadre", x=40, y=50)
```

11 Captation forcée des évènements

La documentation parle de *captation forcée* des évènements par un widget `w` mais on pourrait traduire cela par : désactivation de tous les évènements sur tous les widgets sauf `w`. Si on veut rendre tous les widgets sauf `w` insensibles aux évènements, on appelle la méthode universelle `grab_set` (en anglais, *to grab* signifie saisir) :

`w.grab_set()` :

- effet : les évènements concernant les widgets autres que `w` sont ignorés.

Le programme ci-dessous crée un cadre et deux boutons. Si on clique sur le cadre, le programme affiche les coordonnées du lieu du clic. Si on clique sur le bouton TILT, le programme change la couleur du cadre. Si on clique sur le bouton GRAB, ce dernier se met à capter tous les évènements. Si on reclique dessus, tout redevient normal :

```
from tkinter import *
def coords(event):
    txt = "{}, {}".format(event.x, event.y)
    label.configure(text=txt)
def tilt():
    global i
    n = len(couleurs)
    valeur = couleurs[i]
    frame.configure(bg=valeur)
    i = (i + 1) % n
def change():
    global grab
    grab = not grab
    if grab:
        b1.grab_set()
        b1.configure(text="Ungrab")
    else:
        b1.grab_release()
        b1.configure(text="Grab")
couleurs = ["yellow", "blue", "red", "green"]
i = 1
grab = False
racine = Tk()
racine.configure(bg="grey95")
frame = Frame(racine, width=300, height=300, bg="yellow")
frame.grid()
frame.bind("<1>", coords)
b1 = Button(racine, command=change, text="Grab")
b1.grid()
b2 = Button(racine,command=tilt,text="Tilt")
b2.grid()
label = Label(racine, text="...")
label.grid()
racine.mainloop()
```

La méthode permettant de revenir à la normale (annuler `grab_set`) est `grab_release` :

`w.grab_release()` :

- effet : si `w` capte les évènements, on revient à la normale.

Si on veut savoir quel widget force la captation des évènements, on appelle `grab_current` :

`w.grab_current()` :

- effet : si un widget de l'application capte les évènements, l'appel retourne son nom Tkinter complet, sinon, retourne `None`.

Un widget `w` peut capter tous les évènements de manière globale, c'est-à-dire au delà de l'application : dans ce cas, tout évènement concernant n'importe quelle partie de l'écran sera ignorée (sauf si cela converne `w`). Pour que cela ait lieu, on appelle la méthode `w.grab_set_global`.

`w.grab_set_global()` :

- effet : tous les évènements sur l'écran sont ignorés sauf ceux qui concernent `w`.

Cette méthode est un peu brutale et doit être utilisée le moins possible. La captation globale s'annule avec `grab_release`.

La méthode `grab_status` permet de savoir si un widget capte tous les évènements :

`w.grab_status()` :

- effet : retourne `"local"` si `w` force la captation des évènements, retourne `"global"` si `w` force la captation de manière globale, retourne `None` si `w` ne force pas la captation.

Les méthodes `grab_set`, `grab_set_global`, `grab_release`, `grab_current` et `grab_status` sont universelles.

12 Évènements associés à des variables de contrôle

Nous avons vu au chapitre 16 que la méthode `trace_add` permet de créer un binding associant une variable Tkinter, un évènement concernant cette variable et un callback. Les trois évènements possibles sur une variable sont :

- `"read"` : on a lu le contenu de la variable ;
- `"write"` : on a écrit une valeur dans la variable ;
- `"unset"` : la variable est indéfinie.

Le lecteur consultera la section 16.9 pour les détails.

13 Gestionnaire de protocole

La fenêtre principale et les fenêtres secondaires possèdent une méthode nommée `wm_protocol` (alias `protocol`) permettant de contrôler les interactions entre notre application (programmée en Python-Tkinter) et le gestionnaire de fenêtre (*window manager*). Ci-dessous, `fenêtre` est une fenêtre (principale ou secondaire).

`fenêtre.wm_protocol(nom=None, func=None)` :

- `nom` : type d'évènement concernant le *window manager*, c'est-à-dire `"WM_TAKE_FOCUS"`, `"WM_DELETE_WINDOW"` ou `"WM_SAVE_YOURSELF"` ;
- `func` : un appelable (callback) ;

– effet : un appel `wm_protocol(nom, func)` crée un binding entre `fenêtre`, le type d'évènement spécifié par `nom` et la fonction `func`. Un appel `wm_protocol(nom)` sans fonction retourne une chaîne de caractères construite avec le nom de l'éventuel callback associé à la fenêtre et à l'évènement `nom`. Un appel `wm_protocol()` sans argument retourne la liste des évènements pris en charge par le gestionnaire de la fenêtre.

Dès qu'une association `(fenêtre, nom, func)` est établie, chaque fois qu'un évènement du type spécifié par `nom` a lieu sur `fenêtre`, la fonction `func` est appelée sans argument. D'une certaine manière, `wm_protocol` est un gestionnaire d'évènements concernant les fenêtres du système.

Les évènements considérés sont :

– `"WM_TAKE_FOCUS"` : notre application Tkinter reçoit le focus ;
– `"WM_DELETE_WINDOW"` : on clique sur la petite croix de la fenêtre principale ;
– `"WM_SAVE_YOURSELF"` : non documenté.

Le programme ci-dessous, en mode PRUDENT, lie l'évènement `"WM_DELETE_WINDOW"` à la fonction `prudence`. Du coup, quand on clique sur la petite croix de la fenêtre principale, une fenêtre pop-up nous demande de confirmer le désir de fermer la fenêtre principale. En mode RIGIDE, le programme lie `"WM_DELETE_WINDOW"` à la fonction `bloquer`, du coup, quand on clique sur la petite croix, la fenêtre ne se ferme pas et le programme affiche `"Ça bloque..."`. En mode NORMAL, `"WM_DELETE_WINDOW"` est lié à `racine.destroy`, comme c'est le cas par défaut.

```
from tkinter import *
import tkinter.messagebox as tkm
def protocole():
    global i
    n = len(modes)
    mode = modes[i]
    txt = patron.format(mode)
    label.configure(text=txt)
    if mode == "normal":
        racine.protocol("WM_DELETE_WINDOW", racine.destroy)
    elif mode == "prudent":
        racine.protocol("WM_DELETE_WINDOW", prudence)
    else: racine.protocol("WM_DELETE_WINDOW", bloquer)
    i = (i + 1) % n
def prudence():
    titre = "Attention"
    question = "Sûr de vouloir quitter ?"
    if tkm.askokcancel(titre,question): racine.destroy()
def bloquer(): print("Ça bloque...")
def takingfocus(): print("Taking focus")
patron = "Mode : {}"
modes = ["normal", "prudent", "rigide"]
```

```
i = 1
racine = Tk()
racine.configure(bg="grey95")
racine.protocol("WM_TAKE_FOCUS", takingfocus)
frame = Frame(racine, bg="red", width=300, height=300)
frame.grid()
txt = patron.format("normal")
label = Label(racine, bg="grey95", text=txt)
label.grid()
b1 = Button(racine, text="PROTOCOLE", command=protocole)
b1.grid()
b2 = Button(racine, text="Quitter", command=racine.destroy)
b2.grid()
racine.mainloop()
```

Curieusement le binding `racine.protocol("WM_TAKE_FOCUS", takingfocus)` ne produit aucun effet. On aurait espéré que `takingfocus` soit appelée chaque fois que `racine` prend le focus.

On notera aussi que même en mode PRUDENT, le bouton QUITTER détruit la fenêtre. Autrement dit, le fait de lier `"WM_DELETE_WINDOW"` à un callback qui bloque la fermeture n'empêche pas le programme d'utiliser `racine.destroy`.

On retiendra que pour bloquer la destruction d'une fenêtre, il suffit de lier `"WM_DELETE_WINDOW"` à une fonction qui retourne `None`.

De manière générale, le contrôle de l'évènement `"WM_DELETE_WINDOW"` est utile dans une application qui risque d'être fermée alors que l'utilisateur n'a pas sauvegardé toutes les modifications apportées.

Voici quelques manipulations effectuées sur le shell (à condition de supprimer l'appel à `mainloop` dans le programme) :

```
>>> racine.protocol("WM_TAKE_FOCUS")
'140449228741632takingfocus'
>>> racine.protocol("WM_DELETE_WINDOW")
'140449230088000destroy'
>>> racine.protocol()
('WM_TAKE_FOCUS', 'WM_DELETE_WINDOW')
```

14 Introspection

Regardons quelles sont les séquences impliquées dans un *bind all* :

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

```
('<<PrevWindow>>', '<<NextWindow>>')
```

Regardons ce que signifie "<<PrevWindow>>" :

```
>>> racine.event_info("<<PrevWindow>>")
('<Shift-Key-Tab>',)
```

et "<<NextWindow>>" :

```
>>> racine.event_info("<<NextWindow>>")
('<Key-Tab>',)
```

Regardons les bindings de classe associant "Button", par exemple :

```
>>> racine.bind_class("Button")
('<ButtonRelease-1>', '<Button-1>', '<Leave>', '<Enter>', '<<Invoke>>',
'<Key-space>')
```

Le programme ci-dessous génère un fichier texte donnant toutes les séquences pour chaque classe de widget :

```
import tkinter as tk
def list_to_str(x):
    if not x: return ""
    motif = repr(x[0])
    ch = motif + ", "
    n = len(motif) + 2
    for k in x[1:]:
        motif = repr(k)
        if n + len(motif) + 1 > 80:
            ch = ch + "\n" + motif + ", "
            n = len(motif) + 2
        else:
            ch = ch + motif + ", "
            n = n + len(motif) + 2
    return ch[:-1]
WIDGETS = ["Tk", "Toplevel", "Label", "Message", "Entry", "Spinbox",
           "Scale", "Button", "Checkbutton", "Radiobutton", "Listbox",
           "Canvas", "Text", "Frame", "LabelFrame", "PanedWindow",
           "Scrollbar", "Menu", "Menubutton"]
BR, UNDER, EQ, STAR, PT, ESP = "\n", "_", "=", "*", ".", " "
dico = dict()
racine = tk.Tk()
for w in WIDGETS:
```

```
    liste = racine.bind_class(w)
    ch = list_to_str(liste)
    if ch: dico[w] = ch
    else: dico[w] = "Pas de séquence."
fichier = open("info_bind_class.txt", "w")
titre = "* Séquences liées à chaque type de Widget *"
n = len(titre)
fichier.write(STAR*n + BR + titre + BR + STAR*n + BR*3)
for w in WIDGETS:
    titre = STAR + ESP + w + ESP + STAR
    trait = STAR * len(titre)
    fichier.write(trait + BR + titre + BR + trait + BR*2)
    texte = dico[w]
    fichier.write(texte + BR*2)
fichier.close()
```

La fonction `list_to_str` transforme la liste des chaînes retournée par la méthode `bind_class` en un texte dont les lignes ne dépassent pas 80 caracères. La fonction ne coupe aucun item pour aller à la ligne (elle coupe avant si nécessaire).

Exercice 2. Compléter ce programme pour que tout super-évènement, comme par exemple `"<<Select-None>>"`, soit explicité dans la documentation générée (on pourrait écrire un lexique en prenant les éléments dans l'ordre alphabétique, par exemple).