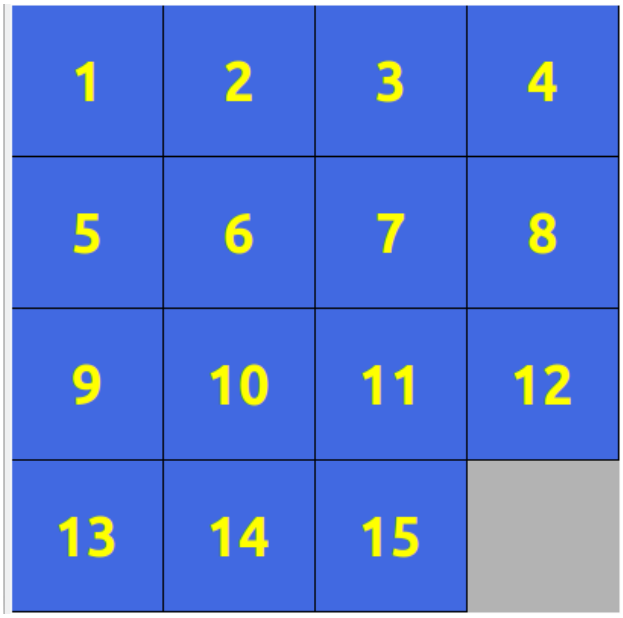
Pascal ORTIZ



Jeu du taquin

Version du 10 novembre 2023

Table des matières

Présentation du jeu

Le jeu de taquin est un puzzle constitué de 15 **tuiles carrées** numérotées de 1 à 15, placées sur un plateau 4x4 et qui se déplacent par glissement :

Au départ, les pièces sont mélangées et le but du jeu est de les réorganiser pour qu'elles se suivent dans l'ordre de 1 à 15 :

On appelle parfois ce jeu le « pousse-pousse » et le jeu est connu en anglais sous le nom de *fifteen-puzzle*.

Le jeu existe en version matérielle (bois, plastique, métal). Il existe aussi en version logicielle. Par exemple, il y avait un taquin sous le nom de Picture Puzzle au moment de la sortie du système d'exploitation Windows 7 et qui était incorporé dans les Desktop Gadgets. Il existe aussi de nombreuses versions de ce jeu sous Android. Sur le web, voici un taquin agréable à jouer et écrit

en javascript (toutefois, à l'usage, je me rends compte qu'il y a un bug dans certains déplacements à la souris).

A regarder plus en détail, il apparaît qu'il existe 4 possibilités de déplacement des tuiles :

— un mode séquentiel : le déplacement n'est pas progressif, il se fait par à-coups, typiquement cette version ;
— un mode progressif : le déplacement est continu, fluide, ici ;
— un mode séquentiel et multi-déplacement : on peut déplacer par à-coups plusieurs tuiles en une fois ;
— un mode progressif et multi-déplacement (présenté dans la démo).

Présentation de l'activité

L'activité consiste à écrire sous Tkinter un taquin pleinement jouable. L'interface finale a été montrée ci-dessus.

Les étapes pour y parvenir seront les suivantes :

— dessiner un taquin statique
— analyser le déplacement d'une tuile seule
— obtenir un « taquin séquentiel »
— créer une fonction de mélange
— créer un bouton pour réinitialiser le jeu
— implémenter le déplacement progressif (glissement) de chaque tuile
— analyser le multi-déplacement de tuiles
— implémenter un taquin supportant le multi-déplacement séquentiel
— implémenter un taquin supportant le multi-glissement.

Afin de simplifier la présentation, je me limiterai à l'implémentation d'un taquin 4 x 4. Chaque tuile (carrée) aura **toujours** une dimension de **100 pixels**. Le code contiendra donc des « constantes magiques » telle que 100, 50, 16, 17, 4, etc. Dans un code soigné, on doit néanmoins éviter le recours à des constantes magiques et on doit utiliser des variables lisibles permettant d'adapter le jeu à d'autres situations.

Si vous arrivez jusqu'à la réalisation du premier taquin (avec bouton mélangeur et annonce de victoire), vous pouvez considérer que vous avez fait l'essentiel de l'activité.

Prérecquis

— un niveau équivalent à mon cours de découverte de Python est **indispensable** et, même, est un **minimum** requis
— une bonne connaissance des listes ;
— avoir manipulé des listes de listes ;
— avoir manipulé des listes en compréhension bien qu'on pourrait systématiquement s'en passer mais au prix d'une certaine lourdeur ;
— avoir une aisance avec les fonctions ;
— connaissance des variables globales : lecture, écriture, voir par exemple le tutoriel sur le jeu Memory ;

— les bases de Tkinter : fenêtre, widget, canevas, items, items rectangle et texte, événement de la souris, déplacement d'items sur le canevas,

— il pourra être utile de savoir comment on gère un plateau de jeu dont les éléments sont placés dans une grille, cf. le tutoriel sur le jeu Memory ;

— pour le déplacement progressif : contour d'un item, méthode after (animation).

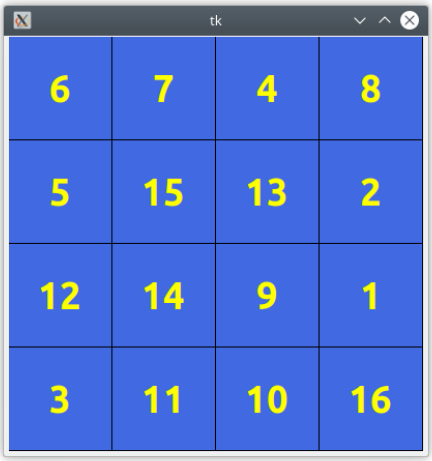
Le code de la dernière partie sur le multi-déplacement progressif des tuiles est nettement plus concentré que celui d'un taquin séquentiel.

Dessiner un taquin statique

Toutes les informations concernant le jeu que Tkinter va dessiner sur le canevas proviendront d'un tableau 2D appelé `board` et de taille 4x4 (le modèle). L'interface graphique (la vue) sera un **miroir** de ce tableau. À la ligne d'indice `i` et à la colonne d'indice `j` de `board`, on trouvera une valeur (par exemple 12) qui est la valeur de la tuile sur le plateau de jeu. La case vide sera marquée par la valeur 16. Dans beaucoup d'exemples, le plateau `board` initial sera le suivant :

```
board=[[6, 7, 4, 8],
       [5, 15, 13, 2],
       [12, 14, 9, 1],
       [3, 11, 10, 16]]
```

Maintenant, on va dessiner un simple plateau de jeu dans un canevas Tkinter :



Pour dessiner une tuile du taquin, on dessine un carré avec la méthode `create_rectangle` du canevas et pour faire apparaître le nombre sur la tuile, on utilise la méthode `create_text`.

Le code qui suit dessine un taquin statique :

`taquin_statique.py`

```
1 from tkinter import *
2
3 board=[[6, 7, 4, 8],
4        [5, 15, 13, 2],
```

```
5         [12, 14, 9, 1],
6         [3, 11, 10, 16]]
7
8  FONT=('Ubuntu', 27, 'bold')
9  master=Tk()
10 cnv=Canvas(master, width=400, height=400, bg='gray70')
11 cnv.pack()
12
13 for i in range(4):
14     for j in range(4):
15         x, y=100*j, 100*i
16         A, B, C=(x, y), (x+100, y+100), (x+50, y+50)
17         cnv.create_rectangle(A, B, fill="royal blue")
18         cnv.create_text(C, text=board[i][j], fill="yellow",
19                                 font=FONT)
20 master.mainloop()
```

— Ligne 8 : `FONT` est la police qui va être utilisée pour afficher les nombres. J'ai choisi la taille en fonction du rendu sur mon système, ça peut-être différent chez vous.

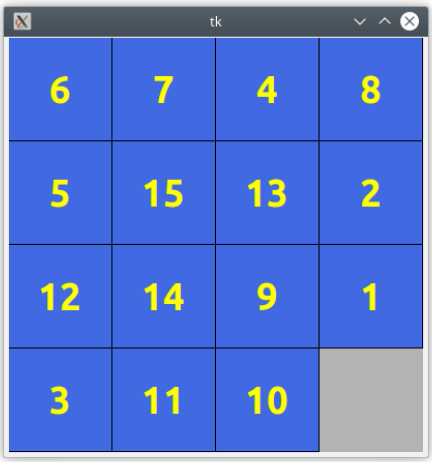
— Lignes 9-11 : on crée une fenêtre (`master`) et un canevas `cnv` que l'on insère dans la fenêtre.

— Lignes 13-19 : à l'aide d'une double boucle, on dessine les tuiles du taquin (lignes 17 et 18). La valeur `board[i][j]` est affichée sur la tuile (ligne 18, option `text`) ; bien que cette valeur soit un entier (type `int`), Tkinter la convertit en chaîne de caractères.

— Ligne 16 : on définit les coins A et B de la tuile courante, son centre C et on dessine le rectangle (ligne 17) puis le texte (ligne 18).

La case vide

On observe sur le dessin qu'il n'y a pas de case vide. Sans surprise, pour faire apparaître la case vide, il suffit d'effacer la dernière tuile créée :



Pour effacer un item du canevas, on dispose de la méthode `delete`. Pour désigner l'item qui doit être effacé, il suffit de donner à la méthode l'id de l'item.

Le code complet est :

taquin_statique_effacer.py

```
1  from tkinter import *
2
3  board=[[6, 7, 4, 8],
4         [5, 15, 13, 2],
5         [12, 14, 9, 1],
6         [3, 11, 10, 16]]
7
8  FONT=('Ubuntu', 27, 'bold')
9  master=Tk()
10 cnv=Canvas(master, width=400, height=400, bg='gray70')
11 cnv.pack()
12
13 for i in range(4):
14     for j in range(4):
15         x, y=100*j, 100*i
16         A, B, C=(x, y), (x+100, y+100), (x+50, y+50)
17         rect = cnv.create_rectangle(A, B, fill="royal blue")
18         txt = cnv.create_text(C, text=board[i][j], fill="yellow",
19                               font=FONT)
20
21 cnv.delete(rect)
22 cnv.delete(txt)
23 master.mainloop()
```

Le code aux lignes 21 et 22 fonctionne car lorsque la double `for` se termine (ligne 19), dans les variables `rect` et `txt` (lignes 17 et 18) on trouve les id de la **dernière** tuile qui a été dessinée, celle numérotée 16 et que l'on veut justement effacer.

Identifier les tuiles

Lorsqu'on va coder le jeu, certaines fonctions auront besoin de savoir quel est le numéro d'une tuile qui a réceptionné un clic. Plusieurs méthodes sont possibles pour arriver à ce résultat. Ici, on va construire une liste `items` de taille 17=16+1 et telle que `items[k]` contienne les id de la tuile numérotée `k`. Plus précisément, et par exemple, `items[12]` contiendra un tuple de la forme `(rect, txt)` où `rect` est l'id du rectangle de la tuile numérotée 12 et `txt` l'id du texte (à savoir le nombre `"12"`) sur cette tuile.

items.py

```
1  from tkinter import *
2
3  board=[[6, 7, 4, 8],
4         [5, 15, 13, 2],
5         [12, 14, 9, 1],
6         [3, 11, 10, 16]]
7
8  FONT=('Ubuntu', 27, 'bold')
```

```
9  master=Tk()
10 cnv=Canvas(master, width=400, height=400, bg='gray70')
11 cnv.pack()
12
13 items=[None for i in range(17)]
14
15 for i in range(4):
16     for j in range(4):
17         x, y=100*j, 100*i
18         A, B, C=(x, y), (x+100, y+100), (x+50, y+50)
19         rect=cnv.create_rectangle(A, B, fill="royal blue")
20         nro=board[i][j]
21         txt=cnv.create_text(C, text=nro, fill="yellow",
22                             font=FONT)
23         items[nro]=(rect, txt)
24 cnv.delete(rect)
25 cnv.delete(txt)
26 master.mainloop()
```

— Ligne 13 : la liste des items à remplir contient 17 éléments et non pas 16 : en effet, l'indice dans la liste `items` doit correspondre au numéro de la tuile et les numéros commencent à 1 et pas à 0.

— Ligne 23 : à chaque construction de tuile, on place à l'indice `k` de `items` un tuple `(tile, txt)` contenant les id de la tuile de numéro `k`.

Lorsqu'une zone sur le plateau de jeu réceptionne un clic, on passera par les étapes suivantes :

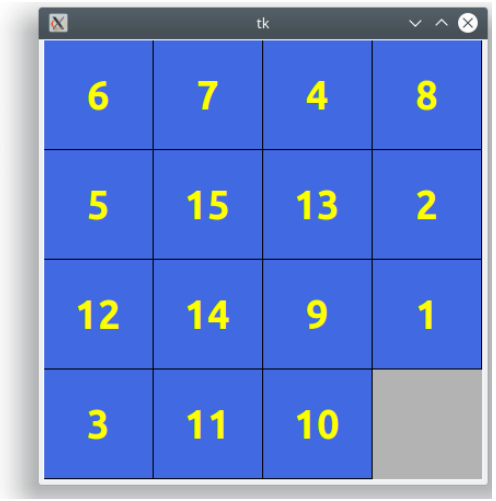
— on calculera la position `(i, j)` ligne x colonne sur le plateau de jeu,

— on ira regarder dans le tableau 2D `board`, à la position ligne `i` et colonne `j`, quel est le numéro de la tuile ;

— grâce à la liste `items`, on aura accès aux id dans le plateau Tkinter de la tuile numérotée `k` ce qui permettra de déplacer la tuile, etc.

Dans le code ci-dessous, on implémente la technique précédente : si l'utilisateur clique sur le plateau de jeu, l'indice de ligne et de colonne est affiché et le numéro porté par la tuile aussi ce qui montre :

```
ligne= 0 colonne= 0 nro= 6
ligne= 0 colonne= 1 nro= 7
ligne= 1 colonne= 1 nro= 15
ligne= 1 colonne= 0 nro= 5
ligne= 3 colonne= 3 nro= 16
ligne= 3 colonne= 2 nro= 10
```



lire_ligne_col.py

```
from tkinter import *

def clic(event):
    i=event.y//100
    j=event.x//100
    print("ligne=", i, "colonne=", j,"nro=", board[i][j])

board=[[6, 7, 4, 8],
       [5, 15, 13, 2],
       [12, 14, 9, 1],
       [3, 11, 10, 16]]

FONT=('Ubuntu', 27, 'bold')
master=Tk()
cnv=Canvas(master, width=400, height=400, bg='gray70')
cnv.pack()

cnv.bind("<Button-1>",clic)

items=[None for i in range(17)]

for i in range(4):
    for j in range(4):
        x, y=100*j, 100*i
        A, B, C=(x, y), (x+100, y+100), (x+50, y+50)
        rect=cnv.create_rectangle(A, B, fill="royal blue")
        nro=board[i][j]
        txt=cnv.create_text(C, text=nro, fill="yellow",
                            font=FONT)
        items[nro]=(rect, txt)
cnv.delete(rect)
cnv.delete(txt)
```

```
33 master.mainloop()
```

— Quand le canevas reçoit un clic (ligne 18), la fonction `clic` est appelé (lignes 3-6).
— Lignes 4-5 :
  — `(event.x, event.y)` sont les coordonnées du clic sur le canevas.
  — comme chaque tuile est un carré de côté 100 pixels, les divisions donnent bien les indices de ligne et de colonne. Noter l'inversion : `event.x` fournit `j` et non pas `i`.
— Ligne 6 : `board[i][j]` contient le numéro de la tuile placé aux indices `i` et `j`.

Enfin, montrons comment on peut agir sur une tuile grâce à la liste `items`. Ici, en guise d'illustration, on décide par exemple de colorier en rouge la tuile sur laquelle on clique :

colorier_tuile.py

```
1  from tkinter import *
2
3  def clic(event):
4      i=event.y//100
5      j=event.x//100
6      nro=board[i][j]
7      rect, txt = items[nro]
8      cnv.itemconfigure(rect, fill="red")
9
10 board=[[6, 7, 4, 8],
11        [5, 15, 13, 2],
12        [12, 14, 9, 1],
13        [3, 11, 10, 16]]
14
15 FONT=('Ubuntu', 27, 'bold')
16 master=Tk()
17 cnv=Canvas(master, width=400, height=400, bg='gray70')
18 cnv.pack()
19
20 cnv.bind("<Button-1>",clic)
```

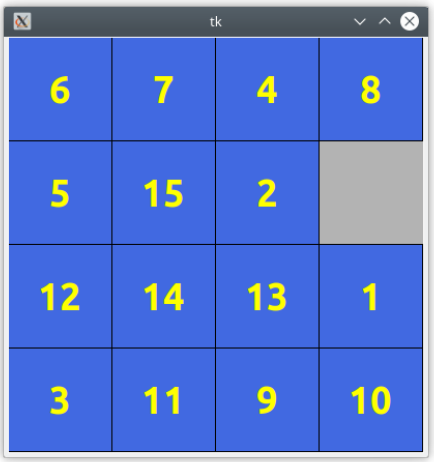
```
21
22 items=[None for i in range(17)]
23
24 for i in range(4):
25     for j in range(4):
26         x, y=100*j, 100*i
27         A, B, C=(x, y), (x+100, y+100), (x+50, y+50)
28         rect=cnv.create_rectangle(A, B, fill="royal blue")
29         board[i][j]
30         nro=board[i][j]
31         txt=cnv.create_text(C, text=nro, fill="yellow",
32                             font=FONT)
33         items[nro]=(rect, txt)
34 cnv.delete(rect)
35 cnv.delete(txt)
36 master.mainloop()
```

— Lignes 6-7 : une fois récupéré le numéro de la tuile cliquée, on dispose par exemple de l'id du rectangle correspondant à la tuile.

— Ligne 8 : on peut alors changer des options du rectangle, comme sa couleur `fill`, grâce à la méthode `itemconfigure`.

Le déplacement de pièce

Si on observe un taquin



on se rend compte qu'à tout instant du jeu si une tuile peut se déplacer, elle ne peut le faire que d'une seule façon : en prenant la place de la case libre (qui est toujours unique). Par exemple, ci-dessus, il y a trois déplacements possibles :

— soit on déplace la tuile n°2, forcément vers la droite,

— soit on déplace la tuile n°1, forcément vers le haut,

— soit on déplace la tuile n°8, forcément vers le bas.

Ainsi, au lieu d'accompagner le déplacement de la tuile dans la direction de la case vide comme on le ferait manuellement, il suffira au joueur de *cliquer sur la tuile* voulue pour provoquer son (seul) déplacement possible.

Du point de vue du code, la case vide est représentée par le numéro 16. Donc déplacer une tuile, disons `k`, revient à échanger `k` avec 16 dans `board`. Par exemple, dans le dessin ci-dessus, le déplacement de la tuile n°2 s'écrit :

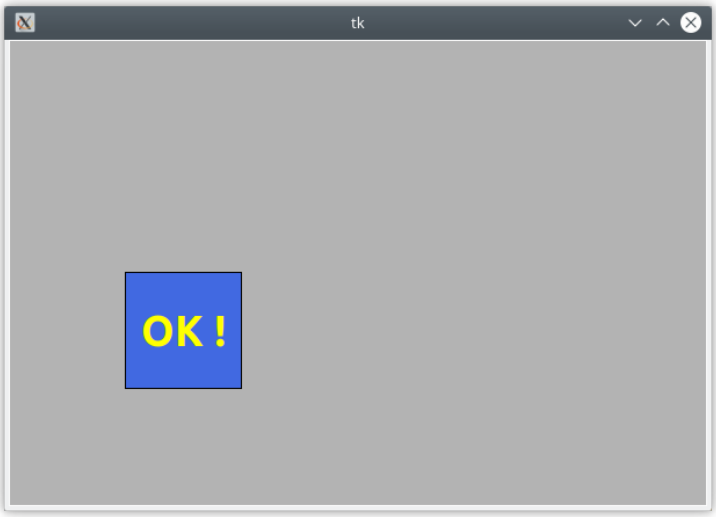
```
board[1][2], board[1][3] = board[1][3], board[1][2]
```

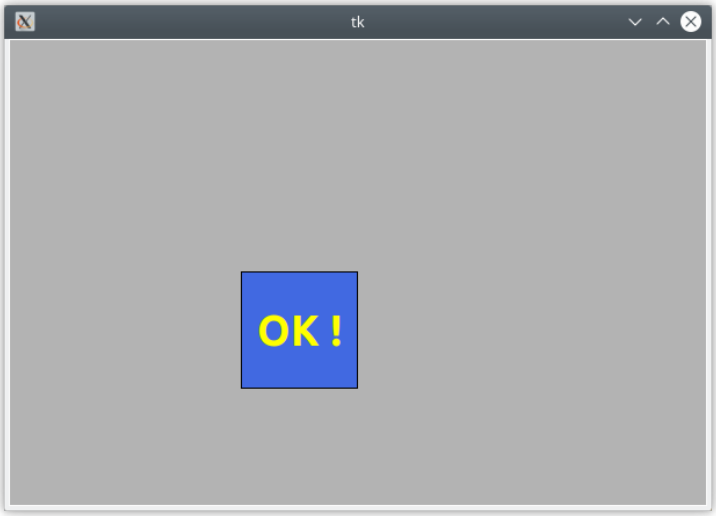
En effet, on rappelle qu'en Python, l'échange de deux références `u` et `v` se fait par `u, v = v, u`.

Du point de vue de Tkinter, un déplacement se fait exactement comme dans le jeu physique : il est tout simplement obtenu en appliquant la méthode `move` à une tuile et donc en fait aux deux items du canevas qui la constituent : le rectangle et le texte.

Déplacer une tuile seule

On va regarder maintenant comment déplacer une tuile à la souris. Pour cela, on va examiner une situation simplifiée d'un plateau avec une seule tuile portant un texte et quand on clique **sur** la tuile, elle se déplace à droite :





Le code correspondant est :

deplacer.py

```
from tkinter import *

def clic(event):
    global j_tile
    i=event.y//100
    j=event.x//100
    if i==i_tile and j==j_tile:
        cnv.move(tile, 100, 0)
        cnv.move(txt, 100, 0)
        j_tile+=1

FONT=('Ubuntu', 27, 'bold')
master=Tk()
cnv=Canvas(master, width=600, height=400, bg='gray70')
cnv.pack()

cnv.bind("<Button-1>",clic)

i=2
j=1

x, y=100*j, 100*i
A, B, C=(x, y), (x+100, y+100), (x+50, y+50)
tile=cnv.create_rectangle(A, B, fill="royal blue")
txt=cnv.create_text(C, text="OK !", fill="yellow", font=FONT)

i_tile=i
j_tile=j

```

```
30 master.mainloop()
```

On définit une tuile carrée marquée `OK` de côté 100 pixels (lignes 22-25). Elle est placée dans notre grille en ligne d'indice 2 et en colonne d'indice 1 (lignes 19-20 et 27-28). Quand on clique sur le canevas (ce qui se traduit par l'événement `"<Button-1>"`, cf. ligne 17), la fonction `clic` est appelée (lignes 3-10). Cette fonction

— identifie la ligne et la colonne du point cliqué (lignes 5-6)

— examine si c'est la tuile qui a été cliquée (ou pas) (ligne 7)

— déplace la tuile d'un vecteur de coordonnées `(100, 0)` autrement dit 100 pixels à droite et 0 pixel vers le bas (Lignes 8-9).

Si on clique ailleurs que sur la tuile, l'exécution, arrivée ligne 7, quitte la fonction `clic` et il ne se passe rien.

Le programme sait où est positionnée la tuile grâce aux variables globales `i_tile` et `j_tile` (cf. lignes 27-28 et ligne 4) qui donnent la ligne et la colonne où se trouve la tuile. D'autre part, une fois la tuile identifiée, il n'y a pas de difficulté à accéder (lignes 8-9) à ses id (rectangle et texte) puisqu'elles sont en variables globales (lignes 24-25).

Une fois que la tuile a bougé, il faut mettre à jour sa nouvelle position. Pour cela, on modifie la variable `j_tile` en incrémentant de 1 (ligne 10) puisque la tuile est ici déplacée uniquement vers la droite (et donc, l'indice de ligne `i_tile` n'a pas être mis-à-jour). Comme `j_tile` est une variable globale, nous sommes obligés pour la modifier (ligne 10) de la déclarer en `global` (ligne 4).

C'est exactement ce principe que l'on va mettre en œuvre pour créer notre premier taquin jouable.

Premier taquin jouable

Le déplacement d'une tuile est la fonctionnalité essentielle d'un taquin. Techniquement, pour déplacer une tuile sur un taquin, il faut :

— déterminer la position ligne x colonne qui réceptionne le clic ;

— ignorer le clic

  — s'il est sur la case vide ;

  — si aucune case voisine de `T` (la tuile choisie) n'est la case vide

— si une des cases voisines est vide, déplacer la tuile `T` dans cette case et mettre à jour la position de la case vide.

Pour savoir si un tuile `T`, placée en position ligne `x` colonne `(i, j)`, peut bouger, il suffit de s'assurer que l'une de ses quatre voisines est la case vide. Dans tous les codes qui suivent, la position ligne `x` colonne de la case vide est placée dans des variables (globales) `i_empty` et `j_empty`.

La méthode la plus directe consiste à tester successivement si :

— la case vide est immédiatement à droite de `T`

— la case vide est immédiatement à gauche de `T`

— la case vide est immédiatement au-dessous de `T`

— la case vide est immédiatement au-dessus de T.

Si on n'est pas dans l'une de ces 4 situations, c'est qu'on a cliqué sur une case qui est immobile ou sur la case vide.

Pour tester, par exemple, si la case vide est immédiatement à droite de T, il suffit d'examiner la valeur de

```
j + 1 == j_empty and i == i_empty
```

Si ce booléen vaut `True`, c'est que la case vide est juste à droite de T et pour déplacer la tuile, il suffit de déplacer avec la méthode `move` les items rectangle et texte suivant le vecteur `(100, 0)` pixels. C'est analogue pour les 3 autres situations.

Noter que ces 4 situations doivent être traitées par une suite de `if/elif` et pas une suite de `if` puisque les conditions s'excluent.

D'où le code suivant qui fournit un premier taquin pleinement jouable :

taquin_sequentiel.py

```
from tkinter import *

def clic(event):
    global i_empty, j_empty
    i=event.y//100
    j=event.x//100
    nro=board[i][j]
    rect, txt=items[nro]
    if j+1 ==j_empty and i==i_empty:
        cnv.move(rect, 100, 0)
        cnv.move(txt, 100, 0)
    elif j-1 ==j_empty and i==i_empty:
        cnv.move(rect, -100, 0)
        cnv.move(txt, -100, 0)
    elif i+1 ==i_empty and j==j_empty:
        cnv.move(rect, 0, 100)
        cnv.move(txt, 0, 100)
    elif i-1 ==i_empty and j==j_empty:
        cnv.move(rect, 0, -100)
        cnv.move(txt, 0, -100)
    else:
        return
    board[i][j],board[i_empty][j_empty]=(
        board[i_empty][j_empty],board[i][j])
    i_empty=i
    j_empty=j

items=[None for i in range(17)]
board=[[6, 7, 4, 8],
       [5, 15, 13, 2],
       [12, 14, 9, 1],
       [3, 11, 10, 16]]

```

```
34 FONT=('Ubuntu', 27, 'bold')
35 master=Tk()
36 cnv=Canvas(master, width=400, height=400, bg='gray70')
37 cnv.pack(side='left')
38
39 i_empty, j_empty=3,3
40
41 for i in range(4):
42     for j in range(4):
43         x, y=100*j, 100*i
44         A, B, C=(x, y), (x+100, y+100), (x+50, y+50)
45         rect=cnv.create_rectangle(A, B, fill="royal blue")
46         nro=board[i][j]
47         txt=cnv.create_text(C, text=nro, fill="yellow",
48                             font=FONT)
49         items[nro]=(rect, txt)
50
51 cnv.delete(txt)
52 cnv.delete(rect)
53
54 cnv.bind("<Button-1>",clic)
55
56 master.mainloop()
```

— Lignes 5-6 : la position ligne x colonne `(i, j)` est là où une tuile a reçu un clic.

— Lignes 7-8 : on récupère le numéro de la tuile et les deux id d'items qui forment une tuile

— On examine successivement si la tuile peut aller

  — à droite (lignes 9-11)

  — à gauche (ligne 12-14)

  — en bas (ligne 15-17)

  — en haut (ligne 18-20).

— Si aucune des conditions précédentes n'est satisfaite (ligne 21), on quitte la fonction `clic` (ligne 22) ;

— Dans tout cas de déplacement, on met à jour le plateau `board` (lignes 23-24) ainsi que la case vide (lignes 25-26)

— Le reste du code est inchangé par rapport à `lire_ligne_col.py`.

Mélanger le plateau de jeu

Bien sûr, avec un vrai taquin, quand on commence à jouer, le plateau a été préalablement mélangé. Dans notre cas, le mélange va être effectué dans le tableau `board` et l'affichage par Tkinter répercutera visuellement ce mélange.

La première méthode qui vient à l'esprit est de mélanger une liste de 16 nombres puis de les placer dans `board` par paquets de 4 dans chaque ligne, comme ci-dessous :

```
from random import shuffle

nros=[k for k in range(1, 17)]
shuffle(nros)

board=[[nros[4*i+j] for j in range(4)] for i in range(4)]

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

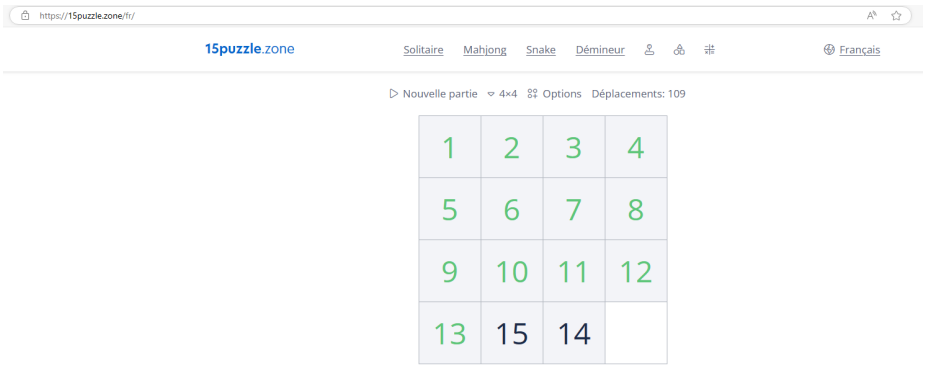
```
[15, 6, 11, 5]
[10, 2, 1, 3]
[12, 16, 9, 13]
[4, 14, 7, 8]
```

— Pour mélanger une liste, on utilise la fonction `shuffle` du module `random` (lignes 1 et 4)

— On regroupe ensuite (ligne 6) le mélange par paquets de 4 que l'on place dans les lignes du tableau 2D.

Toutefois, rien ne garantit a priori que n'importe quel mélange des tuiles obtenu de cette façon pourra être réordonné convenablement par glissements des tuiles. Et justement, il se trouve que certains mélanges ne peuvent être résolus : c'est le fameux problème de Sam Loyd.

D'ailleurs, certaines versions en ligne ont ce bug, par exemple sur le site 15puzzle, le jeu peut se terminer ainsi :



Finalement, pour mélanger le plateau, on va donc procéder comme on le ferait avec un taquin matériel. Partant d'un taquin ordonné, il suffit d'échanger suffisamment de fois la case vide (numéro 16 dans `board`) et une case voisine (dans `board`) :

melanger.py

```
1 from random import randrange
2
3 def voisins(n, i, j):
4     return [(a,b) for (a, b) in
5             [(i, j+1),(i, j-1), (i-1, j), (i+1,j)]
6             if a in range(n) and b in range(n)]
7
```

```
8  def echange(board, empty):
9      i, j=empty
10     V=voisins(4, i, j)
11     ii, jj=V[randrange(len(V))]
12     board[ii][jj], board[i][j]=board[i][j],board[ii][jj]
13     return ii, jj
14
15 def melanger(N):
16     board=[[4*lin+1+col for col in range(4)]
17         for lin in range(4)]
18     print(*board, sep='\n')
19
20     empty=(3,3)
21
22     for i in range(N):
23         empty=echange(board, empty)
24     print('-----------')
25     print(*board, sep='\n')
26
27 melanger(200)
```

```
28 [1, 2, 3, 4]
29 [5, 6, 7, 8]
30 [9, 10, 11, 12]
31 [13, 14, 15, 16]
32 -----------
33 [13, 8, 2, 12]
34 [6, 7, 9, 3]
35 [1, 15, 5, 10]
36 [14, 11, 4, 16]
```

— Lignes 16-17 et 28-31 : avec une liste en compréhension, on crée un plateau rempli avec les entiers de 1 à 16.

— Lignes 22-23 : on effectue `N` échanges successifs de 16 et d'une de ses voisines

— Pour chaque échange, on part de la case vide (on la connait, cf. ligne 20). On construit la liste des voisines de la case vide (lignes 10 et 3-6) et on en choisit une au hasard (ligne 11).

— On déplace la tuile choisie dans la vase vide (ligne 12), ce qui revient à échanger les positions dans `board`.

— On renvoie la nouvelle position de la case vide (ligne 13) ce qui évite de la rechercher quand on fait un nouvel échange (ligne 23).

En pratique, il faut choisir `N` assez grand pour obtenir un bon mélange. Par exemple, si vous examinez le taquin sur ce site pour lequel il semble que `N=200` ait été choisi vous observerez que bien souvent une case n'est jamais bien loin de sa case initiale. Dans la sortie ci-dessus où `N=200` a aussi été choisi, aucune tuile n'est à plus d'une distance de 3 de sa position initiale (la distance est la distance de Manhattan de déplacement sur une grille, autrement dit, c'est le nombre de coups minimal), sur un maximum de 6 possibles.

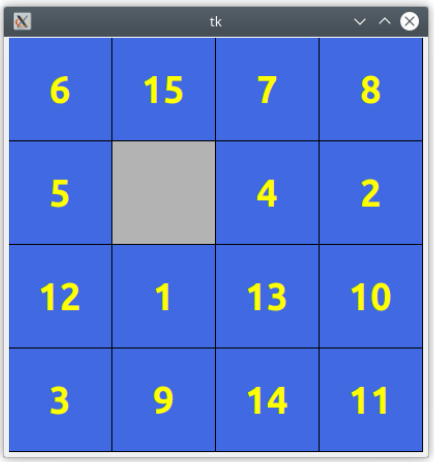
Voici une sortie pour `N=1000` :

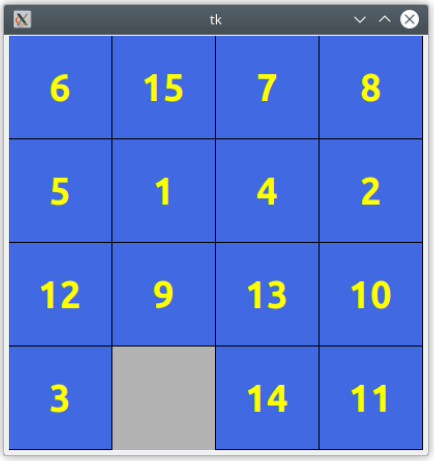
```
[10, 13, 9, 3]
[11, 8, 15, 4]
[16, 2, 5, 6]
[7, 14, 1, 12]
```

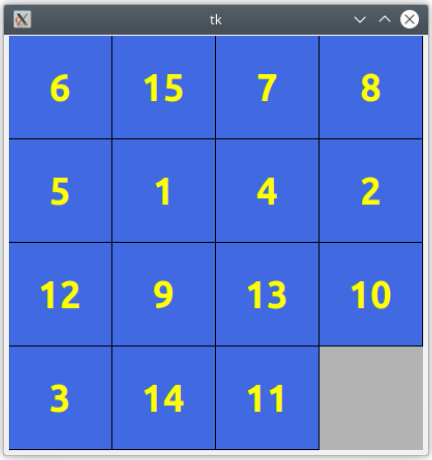
On observe que les quatre tuiles 1, 7, 9 et 13 sont à une distance d'au moins 4 de leur position d'origine, c'est mieux.

Amélioration mineure

Noter que le mélange place la case vide n'importe où sur le plateau alors qu'on pourrait souhaiter que la case vide soit toujours au même endroit, en bas à droite. Pour remédier à cela, il suffit de déplacer la case vide dans le coin en effectuant au plus deux déplacements de blocs de tuiles comme visible ci-dessous :







On va donc écrire une fonction `normal` qui, en deux étapes, place la case vide dans le coin inférieur droit (« normalise »). Pour cela, la case vide est déplacée dans la ligne du bas puis, dans un 2e temps, déplacée dans la dernière colonne :

```
1 def normal(board, empty):
2     i_empty, j_empty = empty
3     for i in range(i_empty, 4):
4         (board[i][j_empty], board[i_empty][j_empty])= (
5             board[i_empty][j_empty], board[i][j_empty])
6         i_empty=i
7     for j in range(j_empty, 4):
8         board[i_empty][j], board[i_empty][j_empty]= (
9             board[i_empty][j_empty],board[i_empty][j])
10        j_empty=j
```

— Lignes 3-5 : déplacement de la case vide dans la dernière ligne par une suite d'échanges sur le plateau `board`.

— Ligne 6 : à chaque étape, la variable pointant vers la case vide est mise à jour

— Ligne 7-10 : on fait de même sauf qu'on place la case vide dans la colonne de droite.

Voici un code effectuant le mélange complet :

`melange_plus.py`

```
1 from random import randrange
2
3 def voisins(n, i, j):
4     return [(a,b) for (a, b) in
5             [(i, j+1),(i, j-1), (i-1, j), (i+1,j)]
6             if a in range(n) and b in range(n)]
7
8 def echange(board, empty):
9     i, j=empty
10    V=voisins(4, i, j)
11    ii, jj=V[randrange(len(V))]
```

```
12      board[ii][jj], board[i][j]=board[i][j],board[ii][jj]
13      return ii, jj
14
15  def normal(board, empty):
16      i_empty, j_empty = empty
17      for i in range(i_empty, 4):
18          (board[i][j_empty], board[i_empty][j_empty])= (
19              board[i_empty][j_empty], board[i][j_empty])
20          i_empty=i
21      for j in range(j_empty, 4):
22          board[i_empty][j], board[i_empty][j_empty]= (
23              board[i_empty][j_empty],board[i_empty][j])
24          j_empty=j
25
26  def melanger(N):
27      board=[[4*lin+1+col for col in range(4)]
28          for lin in range(4)]
29
30      empty=(3,3)
31
32      for i in range(N):
33          empty=echange(board, empty)
34
35      normal(board, empty)
36
37      print(*board, sep='\n')
38
39  melanger(500)
```

```
40  [10, 7, 6, 4]
41  [9, 3, 1, 14]
42  [12, 5, 2, 13]
43  [15, 11, 8, 16]
```

— La sortie : comme on le voit, le plateau est mélangé et la case vide est au coin inférieur droit.

— Ligne 35 : une fois le plateau mélangé, on appelle la fonction `normal` pour placer la case vide dans le coin.

Ajouter un bouton

Pour avoir un jeu rejouable, il reste à ajouter un bouton au canevas contenant notre jeu. Bien que nous ayons déjà tout le code utile, l'ajout va entraîner un nombre important de **refactorisations** de code.

Ajouter un bouton inactif est assez simple. On repart du code `taquin_statique_effacer.py` et on rajoute juste un widget avec la méthode pack (lignes 16-17 ci-dessous) :

taquin_bouton_fake.py

```
1  from tkinter import *
2
3  def f():
4      print("TODO")
5
6  board=[[6, 7, 4, 8],
7         [5, 15, 13, 2],
8         [12, 14, 9, 1],
9         [3, 11, 10, 16]]
10
11 FONT=('Ubuntu', 27, 'bold')
12 master=Tk()
13 cnv=Canvas(master, width=400, height=400, bg='gray70')
14 cnv.pack(side="left")
15
16 btn=Button(text="Mélanger", command=f)
17 btn.pack()
18
19 for i in range(4):
20     for j in range(4):
21         x, y=100*j, 100*i
22         A, B, C=(x, y), (x+100, y+100), (x+50, y+50)
23         rect=cnv.create_rectangle(A, B, fill="royal blue")
24         txt=cnv.create_text(C, text=board[i][j], fill="yellow",
25                             font=FONT)
26 cnv.delete(rect)
27 cnv.delete(txt)
28 master.mainloop()
```

ce qui donne :

Ici, la commande `f` (lignes 16 et 3-4) ne fait rien quand on clique sur le bouton, si ce n'est qu'elle

affiche juste **TODO** dans la console.

La fonction qui sera exécutée à chaque clic sur le bouton **Mélanger** va faire en sorte que le jeu reprenne à zéro. Il s'agit donc en fait d'une fonction d'**initialisation** du jeu et on va l'appeler non pas `f` mais `init`. Cette fonction `init` sera même appelée au premier plateau qui est présenté au joueur.

Donc, à chaque appel de `init`, il va falloir définir ou redéfinir

— la tableau 2D `board`,

— la liste `items`,

— la position ligne et colonne `i_empty` et `j_empty`.

Puis il faudra :

— mélanger `board` avec les fonctions déjà créées,

— afficher le plateau Tkinter,

— récupérer les id des items.

Par ailleurs, suite à un mélange de `board`, plutôt que déplacer les tuiles déjà créées à leur nouvel emplacement, il est beaucoup plus simple de les effacer et de recréer des tuiles aux nouveaux emplacements.

La fonction `init` pourrait apparaître ainsi :

```
1  def init():
2      global i_empty, j_empty, items, board
3      cnv.delete("all")
4      items=[None]
5
6      board=melanger()
7      for i in range(4):
8          for j in range(4):
9              if board[i][j]==16:
10                 i_empty, j_empty=i, j
11
12     items=[None for i in range(17)]
13
14     for i in range(4):
15         for j in range(4):
16             x, y=100*j, 100*i
17             A, B, C=(x, y), (x+100, y+100), (x+50, y+50)
18             tile=cnv.create_rectangle(A, B, fill="royal blue")
19             nro=board[i][j]
20             txt=cnv.create_text(C, text=nro, fill="yellow", font=FONT)
21             items[nro]=(tile, txt)
22     cnv.delete(tile)
23     cnv.delete(txt)
```

— Ligne 3 : le canevas est défini à l'extérieur de la fonction et accessible en lecture sans déclaration particulière.

— Ligne 2 : la fonction `init` définit un certain nombre de variables utiles à d'autres fonctions du programme, voilà pourquoi, il faut les déclarer en global.

— Ligne 6 : création du tableau 2D des numéros.
— Lignes 7-10 : extraction de la case vide.
— Ligne 12 : `items` est une liste pré-construite avec 17 éléments (16 numéros et une valeur factice placée à l'indice 0).
— Ligne 3 : comme on recrée les tuiles à chaque mélange, il faut tout effacer (noter qu'il est possible d'effacer même si rien n'a été construit).
— Ligne 21 : la tuile portant le numéro `nro` est placée à l'indice `nro` de la liste `item`.

En incorporant ce code avec les codes précédents, on obtient un 2e taquin pleinement jouable :

`taquin_sequentiel_complet.py`

```
1  from random import randrange
2  from tkinter import *
3
4  def clic(event):
5      i=event.y//100
6      j=event.x//100
7      global i_empty, j_empty
8      nro=board[i][j]
9      tile, txt=items[nro]
10     if j+1 ==j_empty and i==i_empty:
11         cnv.move(tile, 100, 0)
12         cnv.move(txt, 100, 0)
13     elif j-1 ==j_empty and i==i_empty:
14         cnv.move(tile, -100, 0)
15         cnv.move(txt, -100, 0)
16     elif i+1 ==i_empty and j==j_empty:
17         cnv.move(tile, 0, 100)
18         cnv.move(txt, 0, 100)
19     elif i-1 ==i_empty and j==j_empty:
20         cnv.move(tile, 0, -100)
21         cnv.move(txt, 0, -100)
22     else:
23         return
24     board[i][j],board[i_empty][j_empty]=board[i_empty][j_empty],board[i][j]
25     i_empty=i
26     j_empty=j
27
28 def voisins(n, i, j):
29     return [(a,b) for (a, b) in
30             [(i, j+1),(i, j-1), (i-1, j), (i+1,j)]
31             if a in range(n) and b in range(n)]
32
33 def echange(board, empty):
34     i, j=empty
35     V=voisins(4, i, j)
36     ii, jj=V[randrange(len(V))]
37     board[ii][jj], board[i][j]=board[i][j],board[ii][jj]
38     return ii, jj
```

```
39
40 def normal(board, empty):
41     i_empty, j_empty = empty
42     for i in range(i_empty, 4):
43         (board[i][j_empty], board[i_empty][j_empty])= (
44             board[i_empty][j_empty], board[i][j_empty])
45         i_empty=i
46     for j in range(j_empty, 4):
47         board[i_empty][j], board[i_empty][j_empty]= (
48             board[i_empty][j_empty],board[i_empty][j])
49         j_empty=j
50
51 def melanger(N):
52     board=[[4*lin+1+col for col in range(4)]
53         for lin in range(4)]
54
55     empty=(3,3)
56
57     for i in range(N):
58         empty=echange(board, empty)
59     return board
60
61 def init(N=500):
62     global i_empty, j_empty, items, board
63     cnv.delete("all")
64     items=[None]
65
66     board=melanger(N)
67     for i in range(4):
68         for j in range(4):
69             if board[i][j]==16:
70                 i_empty, j_empty=i, j
71     empty=i_empty, j_empty
72     normal(board, empty)
73     i_empty, j_empty=3,3
74     items=[None for i in range(17)]
75
76     for i in range(4):
77         for j in range(4):
78             x, y=100*j, 100*i
79             A, B, C=(x, y), (x+100, y+100), (x+50, y+50)
80             rect=cnv.create_rectangle(A, B, fill="royal blue")
81             nro=board[i][j]
82             txt=cnv.create_text(C, text=nro, fill="yellow",
83                                 font=FONT)
84             items[nro]=(rect, txt)
85     rect, txt=items[16]
86     cnv.delete(txt)
87     cnv.delete(rect)
```

```
88
89 FONT=('Ubuntu', 27, 'bold')
90 master=Tk()
91 cnv=Canvas(master, width=400, height=400, bg='gray70')
92 cnv.pack(side='left')
93
94 btn=Button(text="Mélanger", command=init)
95 btn.pack()
96
97 cnv.bind("<Button-1>",clic)
98 init()
99
100 master.mainloop()
```

Détection de la position gagnante

On voudrait envoyer un message de félicitation au joueur lorsqu'il a terminé son taquin. Ce message apparaîtra sous la forme suivante :

En outre, le plateau sera bloqué après la victoire car cela n'a plus de sens de le modifier.

Pour afficher le message, on utilise un label Tkinter. Voici ce que cela donne hors-contexte de victoire :

victoire_fake.py

```
1 from tkinter import *
2
3 def f():
4     print("TODO")
5
6 board=[[6, 7, 4, 8],
7         [5, 15, 13, 2],
```

```
8         [12, 14, 9, 1],
9         [3, 11, 10, 16]]
10
11 FONT=('Ubuntu', 27, 'bold')
12 master=Tk()
13 cnv=Canvas(master, width=400, height=400, bg='gray70')
14 cnv.pack(side="left")
15
16 btn=Button(text="Mélanger", command=f)
17 btn.pack()
18
19 lbl=Label(text="Bravo !", font=('Ubuntu', 25, 'bold'),
20           justify=CENTER, width=7)
21 lbl.pack(side="left")
22
23 for i in range(4):
24     for j in range(4):
25         x, y=100*j, 100*i
26         A, B, C=(x, y), (x+100, y+100), (x+50, y+50)
27         rect=cnv.create_rectangle(A, B, fill="royal blue")
28         board[i][j]
29         txt=cnv.create_text(C, text=board[i][j], fill="yellow",
30                             font=FONT)
31 cnv.delete(rect)
32 cnv.delete(txt)
33 master.mainloop()
```

qui affiche :

— Lignes 19-20 : le label contient le message `Bravo !` (option `text`), centré (option `justify`), avec une police de taille adéquate (option `font`). En fait, au début de la partie, le message, bien entendu, est absent et il apparaît en fin de partie. Si on ne prévoit pas l'emplacement au

départ (l'option `width` à 7 caractères ligne 20), cela entraîne un élargissement disgrâcieux de la fenêtre.

Pour que le message apparaisse, il faut surveiller en permanence l'état du plateau `board` pour voir si le taquin est ordonné ou pas. Il suffit donc de définir prélablement un plateau rangé (`win` ci-dessous) et de le comparer à `board`, comme dans cet exemple artificiel :

```
win=[[1, 2, 3, 4],
     [5, 6, 7, 8],
     [9, 10, 11, 12],
     [13, 14, 15, 16]]

board=[[6, 7, 4, 8],
       [5, 15, 13, 2],
       [12, 14, 9, 1],
       [3, 11, 10, 16]]

print(win==board)
```

Cette comparaison doit se faire à chaque modification de `board` (donc après un clic), cf. le code ci-dessous, ligne 27. Par ailleurs, il faut à ce moment-là mettre à jour le label avec la méthode `configure`, cf ligne 28. En outre, il faut un drapeau (`bravo`, ligne 3) qui enregistre si la grille a été résolue ou pas. Voici un code partiel :

```
1  def clic(event):
2      global i_empty, j_empty, bravo
3      if bravo:
4          return
5      i=event.y//100
6      j=event.x//100
7      nro=board[i][j]
8      tile, txt=items[nro]
9      if j+1 ==j_empty and i==i_empty:
10         cnv.move(tile, 100, 0)
11         cnv.move(txt, 100, 0)
12     elif j-1 ==j_empty and i==i_empty:
13         cnv.move(tile, -100, 0)
14         cnv.move(txt, -100, 0)
15     elif i+1 ==i_empty and j==j_empty:
16         cnv.move(tile, 0, 100)
17         cnv.move(txt, 0, 100)
18     elif i-1 ==i_empty and j==j_empty:
19         cnv.move(tile, 0, -100)
20         cnv.move(txt, 0, -100)
21     else:
22         return
23     board[i][j],board[i_empty][j_empty]=(
24         board[i_empty][j_empty],board[i][j])
25     i_empty=i
26     j_empty=j
```

```
27      if board==win:
28          lbl.configure(text="Bravo !")
29          bravo=True
30
31  def init(N=5):
32      global i_empty, j_empty, items, board, bravo
33      cnv.delete("all")
34      items=[None]
35
36      board=melanger(N)
37      for i in range(4):
38          for j in range(4):
39              if board[i][j]==16:
40                  i_empty, j_empty=i, j
41
42      empty=i_empty, j_empty
43      normal(board, empty)
44      i_empty, j_empty=3,3
45      print(board)
46      items=[None for i in range(17)]
47
48      for i in range(4):
49          for j in range(4):
50              x, y=100*j, 100*i
51              A, B, C=(x, y), (x+100, y+100), (x+50, y+50)
52              rect=cnv.create_rectangle(A, B, fill="royal blue")
53              nro=board[i][j]
54              txt=cnv.create_text(C, text=nro, fill="yellow",
55                                  font=FONT)
56              items[nro]=(rect, txt)
57      rect, txt=items[16]
58      cnv.delete(txt)
59      cnv.delete(rect)
60      lbl.configure(text="")
61      bravo=False
62
63
64  win=[[1, 2, 3, 4],
65       [5, 6, 7, 8],
66       [9, 10, 11, 12],
67       [13, 14, 15, 16]]
```

La variable `bravo` est déclarée en variable globale dans `init` pour être lue dans la fonction `clic` et déclarée en global dans `clic` pour y être modifiée. Quand le jeu est mélangé, il faut penser à retirer le message de victoire (cf. ligne 26). Enfin, le drapeau `bravo` sert à empêcher de modifier le jeu (ligne 3) par un clic de déplacement de tuile.

Autre façon de désactiver

Plutôt que d'utiliser le drapeau `bravo` pour bloquer les clics sur le plateau, il est parfois plus commode de désactiver la fonction de rappel avec la méthode `unbind`. Ici, on a activé le déclenchement de la fonction clic par action d'un clic de souris sur le canevas en utilisant l'instruction suivante :

```
cnv.bind("<Button-1>",clic)
```

Pour supprimer tout déclenchement de fonction suite à un évément de clic sur le canevas, il suffit d'écrire l'instruction :

```
cnv.unbind("<Button-1>")
```

Plus précisément, dans le code ci-dessus, on pourrait supprimer toute référence au drapeau `bravo` et écrire, lorsque la victoire est détectée (lignes 28-29), le code suivant :

```
if board==win:
    lbl.configure(text="Bravo !")
    cnv.unbind("<Button-1>")
```

On trouvera un autre exemple dans le cours Tkinter sur les événements.

Application complète

Le code complet et exécutable est :

taquin_sequentiel_complet_win.py

```
1  from random import randrange
2  from tkinter import *
3
4  def clic(event):
5      global i_empty, j_empty, bravo
6      if bravo:
7          return
8      i=event.y//100
9      j=event.x//100
10     nro=board[i][j]
11     tile, txt=items[nro]
12     if j+1 ==j_empty and i==i_empty:
13         cnv.move(tile, 100, 0)
14         cnv.move(txt, 100, 0)
15     elif j-1 ==j_empty and i==i_empty:
16         cnv.move(tile, -100, 0)
17         cnv.move(txt, -100, 0)
18     elif i+1 ==i_empty and j==j_empty:
19         cnv.move(tile, 0, 100)
20         cnv.move(txt, 0, 100)
21     elif i-1 ==i_empty and j==j_empty:
22         cnv.move(tile, 0, -100)
23         cnv.move(txt, 0, -100)
24     else:
```

```
25          return
26      board[i][j],board[i_empty][j_empty]=(
27          board[i_empty][j_empty],board[i][j])
28      i_empty=i
29      j_empty=j
30      if board==win:
31          lbl.configure(text="Bravo !")
32          bravo=True
33
34  def voisins(n, i, j):
35      return [(a,b) for (a, b) in
36              [(i, j+1),(i, j-1), (i-1, j), (i+1,j)]
37              if a in range(n) and b in range(n)]
38
39  def echange(board, empty):
40      i, j=empty
41      V=voisins(4, i, j)
42      ii, jj=V[randrange(len(V))]
43      board[ii][jj], board[i][j]=board[i][j],board[ii][jj]
44      return ii, jj
45
46  def normal(board, empty):
47      i_empty, j_empty = empty
48      for i in range(i_empty, 4):
49          (board[i][j_empty], board[i_empty][j_empty])= (
50              board[i_empty][j_empty], board[i][j_empty])
51          i_empty=i
52      for j in range(j_empty, 4):
53          board[i_empty][j], board[i_empty][j_empty]= (
54              board[i_empty][j_empty],board[i_empty][j])
55          j_empty=j
56
57  def melanger(N):
58      board=[[4*lin+1+col for col in range(4)]
59          for lin in range(4)]
60
61      empty=(3,3)
62
63      for i in range(N):
64          empty=echange(board, empty)
65      return board
66
67  def init(N=5):
68      global i_empty, j_empty, items, board, bravo
69      cnv.delete("all")
70      items=[None]
71
72
```

```
    board=melanger(N)
    for i in range(4):
        for j in range(4):
            if board[i][j]==16:
                i_empty, j_empty=i, j

    empty=i_empty, j_empty
    normal(board, empty)
    i_empty, j_empty=3,3
    print(board)
    items=[None for i in range(17)]

    for i in range(4):
        for j in range(4):
            x, y=100*j, 100*i
            A, B, C=(x, y), (x+100, y+100), (x+50, y+50)
            rect=cnv.create_rectangle(A, B, fill="royal blue")
            nro=board[i][j]
            txt=cnv.create_text(C, text=nro, fill="yellow",
                                font=FONT)
            items[nro]=(rect, txt)
    rect, txt=items[16]
    cnv.delete(txt)
    cnv.delete(rect)
    lbl.configure(text="")
    bravo=False

FONT=('Ubuntu', 27, 'bold')
master=Tk()
cnv=Canvas(master, width=400, height=400, bg='gray70')
cnv.pack(side='left')

btn=Button(text="Mélanger", command=init)
btn.pack()

lbl=Label(text="      ", font=('Ubuntu', 25, 'bold'),
          justify=CENTER, width=7)
lbl.pack(side="left")

cnv.bind("<Button-1>",clic)
init()

win=[[1, 2, 3, 4],
     [5, 6, 7, 8],
     [9, 10, 11, 12],
     [13, 14, 15, 16]]

master.mainloop()
```

— Ligne 67 : la petite valeur `N=5` permet de tester l'apparition du message sans avoir à passer longtemps à résoudre le taquin.

Alternative de mélange

On peut simplifier la génération d'un mélange du plateau de la manière suivante :

— on part de la liste ordonnée de tous les numéros de 1 à 15,
— on effectue un nombre **pair** d'échanges de deux numéros
— on place le mélange obtenu dans une grille, ligne par ligne
— on termine la grille en rajoutant la tuile 16.

Le programme ci-dessous génère une grille valide conformément à la méthode ci-dessus :

```
1  from random import randrange
2
3  def echange(L):
4      i,j = randrange(15),randrange(15)
5      L[i], L[j]=L[j], L[i]
6
7  def melanger(N=10):
8      L=list(range(1,16))
9
10     for i in range(2*N):
11         echange(L)
12     L.append(16)
13     return [[L[4*i+j] for j in range(4)] for i in range(4)]
14
15 board=melanger()
16
17 print(*board, sep='\n')
```

```
18 [10, 4, 9, 14]
19 [5, 1, 2, 11]
20 [8, 7, 6, 12]
21 [15, 13, 3, 16]
```

En fait, si on génère aléatoirement une permutation des entiers de 1 à 15, on a une chance sur 2 que cette permutation soit résoluble ; le caractère résoluble de la permutation peut se déterminer facilement par un calcul de signature de permutation.

Alternative de codage du déplacement de tuile

Dans le code précédent `taquin_sequentiel.py` de codage du déplacement d'une tuile :

```
1  def clic(event):
2      global i_empty, j_empty
3      i=event.y//100
4      j=event.x//100
```

```
5       nro=board[i][j]
6       rect, txt=items[nro]
7       if j+1 ==j_empty and i==i_empty:
8           cnv.move(rect, 100, 0)
9           cnv.move(txt, 100, 0)
10      elif j-1 ==j_empty and i==i_empty:
11          cnv.move(rect, -100, 0)
12          cnv.move(txt, -100, 0)
13      elif i+1 ==i_empty and j==j_empty:
14          cnv.move(rect, 0, 100)
15          cnv.move(txt, 0, 100)
16      elif i-1 ==i_empty and j==j_empty:
17          cnv.move(rect, 0, -100)
18          cnv.move(txt, 0, -100)
19      else:
20          return
21      board[i][j],board[i_empty][j_empty]=(
22          board[i_empty][j_empty],board[i][j])
23      i_empty=i
24      j_empty=j
```

il y a beaucoup de redondance de code parce qu'on examine les 4 cas les uns après les autres et que ces cas se traitent de manière analogue. En réalité, au prix (peut-être) d'une moins grande facilité de compréhension, il est possible de réduire au minimum le traitement individuel de chaque cas. Il suffit essentiellement de produire un vecteur de déplacement valable dans tous les cas. C'est basé sur le principe qu'un vecteur se comprend comme « extrémité moins origine » (lignes 10-11 ci-dessous), comme le montre le code suivant :

```
1 def clic(event):
2       global i_empty, j_empty
3       i=event.y//100
4       j=event.x//100
5       empty_around=(i_empty in [i-1, i+1] and j==j_empty
6                     or j_empty in [j-1, j+1] and i==i_empty)
7       if empty_around:
8           nro=board[i][j]
9           tile, txt=items[nro]
10          move_x=(j_empty-j)*100
11          move_y=(i_empty-i)*100
12          cnv.move(tile, move_x, move_y)
13          cnv.move(txt, move_x, move_y)
14          board[i][j],board[i_empty][j_empty]=(
15              board[i_empty][j_empty],board[i][j])
16          i_empty=i
17          j_empty=j
```

— Lignes 5-6 : la fonction `clic` commence par déterminer si oui ou non, autour du clic se trouve la case vide ; c'est le booléen `empty_around` qui enregistre cet état. Le code commence d'abord par regarder si la case vide serait au-dessus (`i_empty == i-1`) ou au-dessous (`i_empty == i+1`) ; ensuite, il est examiné si la case vide est à gauche ou à droite. C'est la

seule partie du code qui fait du cas par cas.

— Ligne 7 : si `empty_around` vaut `False`, c'est que soit on a cliqué sur la case vide soit, au contraire, qu'on a cliqué sur une tuile « bloquée ». Dans ce cas, aucun mouvement n'a lieu et l'exécution quitte la fonction `clic`.

— Comme dans le code précédent, on peut identifier (lignes 8-9) la tuile, son numéro et ses id.

— Pour rechercher le vecteur de déplacement (`move_x, move_y`) de la tuile, grâce à une astuce de calcul, on peut s'éviter de considérer les quatre cas. L'idée de base est qu'un vecteur se mesure par la formule « extrémité moins origine », ici plutôt « destination moins origine », cf. lignes 10-11. Comme le vecteur est en pixels, il faut multiplier par 100, la taille de chaque tuile.

— Comme dans le code précédent, on échange la case vide et la tuile déplacée (lignes 14-15) et on met à jour la case vide (lignes 16-17).

Voici le code complet :

`taquin_sequentiel_plus.py`

```
1  from random import randrange
2  from tkinter import *
3
4  def clic(event):
5      global i_empty, j_empty
6      i=event.y//100
7      j=event.x//100
8      empty_around=(i_empty in [i-1, i+1] and j==j_empty
9                    or j_empty in [j-1, j+1] and i==i_empty)
10     if empty_around:
11         nro=board[i][j]
12         tile, txt=items[nro]
13         move_x=(j_empty-j)*100
14         move_y=(i_empty-i)*100
15         cnv.move(tile, move_x, move_y)
16         cnv.move(txt, move_x, move_y)
17         board[i][j],board[i_empty][j_empty]=(
18             board[i_empty][j_empty],board[i][j])
19         i_empty=i
20         j_empty=j
21
22 items=[None for i in range(17)]
23 board=[[6, 7, 4, 8],
24        [5, 15, 13, 2],
25        [12, 14, 9, 1],
26        [3, 11, 10, 16]]
27
28 FONT=('Ubuntu', 27, 'bold')
29 master=Tk()
30 cnv=Canvas(master, width=400, height=400, bg='gray70')
31 cnv.pack(side='left')
32
33 i_empty, j_empty=3,3
```

```
34
35 for i in range(4):
36     for j in range(4):
37         x, y=100*j, 100*i
38         A, B, C=(x, y), (x+100, y+100), (x+50, y+50)
39         rect=cnv.create_rectangle(A, B, fill="royal blue")
40         nro=board[i][j]
41         txt=cnv.create_text(C, text=nro, fill="yellow",
42                             font=FONT)
43         items[nro]=(rect, txt)
44
45 cnv.delete(txt)
46 cnv.delete(rect)
47
48 cnv.bind("<Button-1>",clic)
49
50 master.mainloop()
```

Déplacement progressif des tuiles

Dans notre taquin séquentiel actuel, la méthode `move` du canevas fait un déplacement **instantané** d'un item du canevas. Pour obtenir un effet de fluidité, on souhaiterait un déplacement **progressif** de chaque tuile.

La technique employée pour changer une tuile de position est la suivante. On va déplacer la tuile

— d'une certaine distance, petite, par exemple `DIST = 10` pixels,

— à intervalles réguliers, par exemple `DELTA = 20` millisecondes,

ce qui donnera un effet d'animation. La répétition sera assurée par l'utilisation de la méthode `after`.

Lorsqu'on doit déplacer une tuile, on peut assez facilement déterminer la direction de déplacement sous la forme d'un vecteur « unitaire » `(x,y)`, où `x` représente une ligne, `y` représente une colonne et où `x` et `y` valent $-1, 1$ ou `0`; par exemple un vecteur valant `(0, -1)` désignera un déplacement de la tuile vers le haut. On connaît par ailleurs la position initiale et la position finale de la tuile (dans la case vide).

Voici la fonction d'animation :

```
1 DELTA=20
2 DIST=10
3
4 def anim(item, target, drtn):
5     L =cnv.coords(item)
6     a=L[0]
7     b=L[1]
8     x, y=target
9     u, v=drtn
10    if u*(x-a)+v*(y-b)>DIST:
11        cnv.move(item, u*DIST, v*DIST)
12        cnv.after(DELTA, anim, item, target, drtn)
```

```
13      else:
14          cnv.move(item, (x-a), (y-b))
```

On remarquera que cette fonction d'animation traite génériquement d'un déplacement horizontal (droite ou gauche) et d'un déplacement vertical (haut ou bas), aucun cas n'est explicitement distingué. La fonction `anim` reçoit l'id de l'item à déplacer (`item`), la position finale de l'item (`target`) et la direction `drtn` de déplacement sous la forme `(x, y)` où `x` et `y` valent `0`, -1 ou `1`.

On commence par récupérer les coordonnées `(a, b)` d'un élément de l'item grâce à la méthode `coords` du canevas. Pour un rectangle, cet élément sera le coin supérieur gauche. Pour du texte, ce sera le centre du texte. Ensuite on regarde la distance que l'item doit encore à parcourir avant destination ; cette distance vaut `u*(x-a)+v*(y-b)`, ce qui est encore une variante du raccourci « vecteur = extrémité - origine ». Si cette distance est inférieure à un écart DIST (lignes 10) qu'on s'est donné au départ (ligne 2), on déplace l'item avec la méthode `move` de cette distance suivant la direction `drtn` cf. ligne 9). Sinon (lignes 13-14), c'est qu'on est très proche de la position définitive et on place l'item à sa position finale (à nouveau, « vecteur = extrémité - origine »)

La fonction `anim` est appelée à la place des appels que l'on faisait à la méthode `clic` dans la fonction de déplacement de tuile du taquin séquentiel. On connaît les items à déplacer : un rectangle et du texte. Le coin supérieur gauche de la tuile, représenté par `board[i][j]`, a pour coordonnées sur le canevas

```
a = j * 100
b = i * 100
```

puisque 100 est la dimension de chaque tuile. Le centre de la tuile a pour coordonnées

```
j * 100 + 50
i * 100 + 50
```

Pour la suite, on se base sur le code `taquin_sequentiel_plus.py`. Voici le nouveau code de la fonction `clic` :

```
1  def clic(event):
2      global i_empty, j_empty
3      i=event.y//100
4      j=event.x//100
5
6      empty_around=(i_empty in [i-1, i+1] and j==j_empty
7                    or j_empty in [j-1, j+1] and i==i_empty)
8      if empty_around:
9          nro=board[i][j]
10         rect, txt=items[nro]
11
12         target=100*j_empty, 100*i_empty
13         drtn=j_empty -j, i_empty-i
14         target_txt=100*j_empty+50, 100*i_empty+50
15
16         anim(rect, target, drtn)
17         anim(txt, target_txt, drtn)
18
19         board[i][j],board[i_empty][j_empty]=(board[i_empty][j_empty],
20                                             board[i][j])
```

```
21          i_empty=i
22          j_empty=j
```

— On détermine la position finale sur le canevas du coin supérieur gauche du rectangle (`target`, ligne 12) et du texte (`target_txt`, ligne 14)

— On determine la direction unitaire de déplacement, cf. ligne 13. Et on appelle la fonction d'animation `anim` (lignes 16-17).

D'où le code complet suivant :

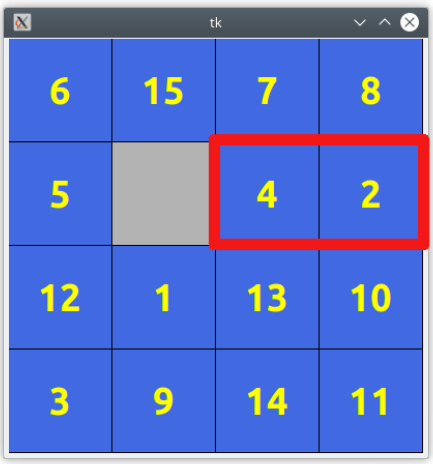
taquin_progressif_simple.py

```
1  from random import randrange
2  from tkinter import *
3
4  DELTA=20
5  DIST=10
6
7  def anim(item, target, drtn):
8      L =cnv.coords(item)
9      a=L[0]
10     b=L[1]
11     x, y=target
12     u, v=drtn
13     if u*(x-a)+v*(y-b)>DIST:
14         cnv.move(item, u*DIST, v*DIST)
15         cnv.after(DELTA, anim, item, target, drtn)
16     else:
17         cnv.move(item, (x-a), (y-b))
18
19 def clic(event):
20     global i_empty, j_empty
21     i=event.y//100
22     j=event.x//100
23
24     empty_around=(i_empty in [i-1, i+1] and j==j_empty
25                   or j_empty in [j-1, j+1] and i==i_empty)
26     if empty_around:
27         nro=board[i][j]
28         rect, txt=items[nro]
29
30         target=100*j_empty, 100*i_empty
31         drtn=j_empty -j, i_empty-i
32         target_txt=100*j_empty+50, 100*i_empty+50
33
34         anim(rect, target, drtn)
35         anim(txt, target_txt, drtn)
36
37         board[i][j],board[i_empty][j_empty]=(board[i_empty][j_empty],
38                                             board[i][j])
39         i_empty=i
```

```
40          j_empty=j
41
42  items=[None for i in range(17)]
43  board=[[6, 7, 4, 8],
44         [5, 15, 13, 2],
45         [12, 14, 9, 1],
46         [3, 11, 10, 16]]
47
48  FONT=('Ubuntu', 27, 'bold')
49  master=Tk()
50  cnv=Canvas(master, width=400, height=400, bg='gray70')
51  cnv.pack()
52
53  i_empty, j_empty=3,3
54
55  for i in range(4):
56      for j in range(4):
57          x, y=100*j, 100*i
58          A, B, C=(x, y), (x+100, y+100), (x+50, y+50)
59          rect=cnv.create_rectangle(A, B, fill="royal blue")
60          nro=board[i][j]
61          txt=cnv.create_text(C, text=nro, fill="yellow",
62                              font=FONT)
63          items[nro]=(rect, txt)
64
65  cnv.delete(txt)
66  cnv.delete(rect)
67
68  cnv.bind("<Button-1>",clic)
69
70  master.mainloop()
```

Multidéplacement de tuiles

Quand on joue avec un taquin physique, il est courant que l'on déplace en une seule action, non pas une unique tuile mais une succession de tuiles voisines. Par exemple, dans le dessin ci-dessous :



on peut déplacer les **deux** tuiles marquées 4 et 2 vers la gauche en un seul mouvement consistant à déplacer 2 vers la gauche ce qui aura pour effet d'entraîner 4. Là encore, inutile dans la version numérique, d'accompagner la tuile dans son déplacement, si on sélectionne la tuile n°2 pour déplacement, le déplacement est unique et entrainera la tuile voisine.

On repart du code `taquin_sequentiel.py` dont on va modifier la fonction de déplacement `clic` pour tenir compte de cette possibilité. La logique est la même sauf qu'il ne faut pas se contenter de regarder si une case voisine de la tuile cliquée, disons `T` est la case vide mais si *une case dans la même ligne ou la même colonne* que `T` est la case vide. Si c'est le cas, il faut déplacer une par une les tuiles placées entre la tuile `T` et la case vide, puis penser à mettre à jour `board`. Par exemple, ci-dessus, si on clique sur la tuile n°2, il faudra déplacer vers la gauche d'une case toutes les tuiles entre la tuile n°2 et avant la case vide.

On obtient alors le code suivant :

```
def clic(event):
    global i_empty, j_empty
    i=event.y//100
    j=event.x//100

    if i==i_empty and j<j_empty:
        for k in range(j, j_empty):
            nro=board[i][k]
            rect, txt=items[nro]
            cnv.move(rect, 100, 0)
            cnv.move(txt, 100, 0)
        for k in range(j_empty, j, -1):
            board[i][k]=board[i][k-1]
    elif i==i_empty and j>j_empty:
        for k in range(j, j_empty,-1):
            nro=board[i][k]
            rect, txt=items[nro]
            cnv.move(rect, -100, 0)
            cnv.move(txt, -100, 0)
        for k in range(j_empty, j):
```

```
21              board[i][k]=board[i][k+1]
22      elif j==j_empty and i<i_empty:
23          for k in range(i, i_empty):
24              nro=board[k][j]
25              rect, txt=items[nro]
26              cnv.move(rect, 0, 100)
27              cnv.move(txt, 0, 100)
28          for k in range(i_empty, i, -1):
29              board[k][j]=board[k-1][j]
30      elif j==j_empty and i>i_empty:
31          for k in range(i, i_empty, -1):
32              nro=board[k][j]
33              rect, txt=items[nro]
34              cnv.move(rect, 0, -100)
35              cnv.move(txt, 0, -100)
36          for k in range(i_empty, i):
37              board[k][j]=board[k+1][j]
38      else:
39          return
40      board[i][j]=16
41      i_empty=i
42      j_empty=j
```

Ainsi lignes 22-29 pour fixer les idées, le code regarde si dans la même colonne que la tuile cliquée et au-dessous, se trouverait la case vide (ligne 22). Si c'est le cas, chaque tuile au-dessous de la tuile `(i, j)` est déplacée vers le bas (cf. `0, 100` lignes 26-27). Ensuite, on met à jour `board` (lignes 28-29) en décalant chaque tuile déplacée, en commençant par la plus proche de la case vide. La mise à jour de la case vide a lieu plus loin, aux lignes 40-42.

Voici le code complet et exécutable :

taquin_sequentiel_multi_simple.py

```
1  from random import randrange
2  from tkinter import *
3
4
5  def clic(event):
6      global i_empty, j_empty
7      i=event.y//100
8      j=event.x//100
9
10     if i==i_empty and j<j_empty:
11         for k in range(j, j_empty):
12             nro=board[i][k]
13             rect, txt=items[nro]
14             cnv.move(rect, 100, 0)
15             cnv.move(txt, 100, 0)
16         for k in range(j_empty, j, -1):
17             board[i][k]=board[i][k-1]
18     elif i==i_empty and j>j_empty:
```

```
        for k in range(j, j_empty,-1):
            nro=board[i][k]
            rect, txt=items[nro]
            cnv.move(rect, -100, 0)
            cnv.move(txt, -100, 0)
        for k in range(j_empty, j):
            board[i][k]=board[i][k+1]
    elif j==j_empty and i<i_empty:
        for k in range(i, i_empty):
            nro=board[k][j]
            rect, txt=items[nro]
            cnv.move(rect, 0, 100)
            cnv.move(txt, 0, 100)
        for k in range(i_empty, i, -1):
            board[k][j]=board[k-1][j]
    elif j==j_empty and i>i_empty:
        for k in range(i, i_empty, -1):
            nro=board[k][j]
            rect, txt=items[nro]
            cnv.move(rect, 0, -100)
            cnv.move(txt, 0, -100)
        for k in range(i_empty, i):
            board[k][j]=board[k+1][j]
    else:
        return
    board[i][j]=16
    i_empty=i
    j_empty=j

items=[None for i in range(17)]
board=[[6, 7, 4, 8],
       [5, 15, 13, 2],
       [12, 14, 9, 1],
       [3, 11, 10, 16]]

FONT=('Ubuntu', 27, 'bold')
master=Tk()
cnv=Canvas(master, width=400, height=400, bg='gray70')
cnv.pack(side='left')

i_empty, j_empty=3,3

for i in range(4):
    for j in range(4):
        x, y=100*j, 100*i
        A, B, C=(x, y), (x+100, y+100), (x+50, y+50)
        rect=cnv.create_rectangle(A, B, fill="royal blue")
        nro=board[i][j]
```

```
67          txt=cnv.create_text(C, text=nro, fill="yellow",
68                              font=FONT)
69          items[nro]=(rect, txt)
70
71  cnv.delete(txt)
72  cnv.delete(rect)
73
74  cnv.bind("<Button-1>",clic)
75
76  master.mainloop()
```

On pourra toutefois regretter que la fonction `clic` comporte autant de redondance de code.

Multidéplacement progressif des tuiles

Maintenant, on veut non seulement pouvoir déplacer des blocs de tuiles contiguës mais aussi que le déplacement soit progressif. On ne va pas procéder comme dans le codage précédent du multi-déplacement séquentiel qui passait en revue tous les cas. Le code de base sera celui de `taquin_sequentiel_plus.py` ; en particulier, on **veut conserver sans changement** la fonction d'animation `anim` de déplacement de tuile, rappelée ci-dessous :

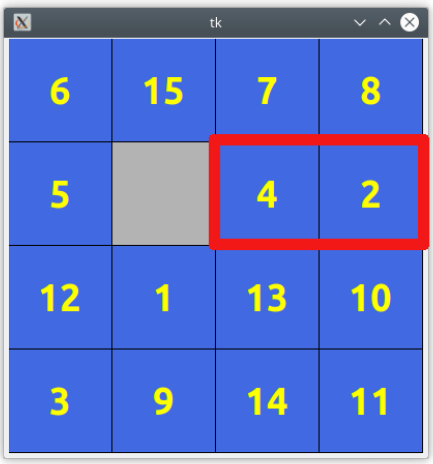
```
1  DELTA=20
2  DIST=10
3
4  def anim(item, target, drtn):
5      L =cnv.coords(item)
6      a=L[0]
7      b=L[1]
8      x, y=target
9      u, v=drtn
10     if u*(x-a)+v*(y-b)>DIST:
11         cnv.move(item, u*DIST, v*DIST)
12         cnv.after(DELTA, anim, item, target, drtn)
13     else:
14         cnv.move(item, (x-a), (y-b))
```

Avant de se lancer dans la partie graphique, on va déjà écrire une fonction qui **analyse** le plateau `board` pour nous renvoyer toutes les informations utiles lorsque le joueur cliquera une tuile et qu'on connait la position de la case vide (ce type de fonction s'appelle souvent *helper function*). Cette fonction appelée `multi` va nous indiquer, par rapport à `board` :

— suivant quel vecteur unitaire le déplacement se fera, par exemple `(0, -1)` pour un déplacement vers la gauche ;

— une liste formée, pour chaque tuile déplacée, de sa position de départ et de sa position de destination.

Par exemple, si le plateau est ci-dessous :



et que l'on clique sur la tuile n°2, l'appel sera `multi((1, 3), (1, 1))` et doit renvoyer une liste `(L, dir)` où `dir` et le vecteur de direction, ici `(0, -1)` et L est la liste

```
[((1, 2),(1, 1)), ((1, 2),(1, 3))]
```

Voici le code de la fonction `multi` :

```
1  def multi(orig, empty):
2      i, j=orig
3      ii, jj=empty
4      delta=(di, dj)=(ii-i, jj-j)
5      if di!=0!=dj or di==dj==0:
6          return None
7      norm=max(abs(di), abs(dj))
8
9      dirx, diry =(di//norm, dj//norm)
10     L=[((ii-dirx, jj-diry), (ii, jj))]
11
12     for k in range(norm-1):
13         (a, b), destn=L[-1]
14         pos=((a-dirx, b-diry), (a, b))
15         L.append(pos)
16     return L, (dirx, diry)
```

— Ligne 1 : pour comprendre le contexte, `orig` représente la tuile sur laquelle le joueur va cliquer.

— Lignes 2-3 : `i` et `ii` représentent des *lignes* du plateau de jeu `board` (pas des pixels).

— Ligne 4 : `delta` correspond au vecteur déplacement dans le tableau `board` (penser encore « extrémité moins origine »).

— Lignes 5-6 : si `orig` (la future case cliquée) est telle que ni dans sa ligne, ni dans sa colonne ne se trouve la case vide, il ne se passe rien et l'exécution quitte la fonction.

— Ligne 7 : `norm` représente le nombre de tuiles qui seront translatées.

— Ligne 9 : le vecteur `(dirx, diry)` représente le vecteur unitaire de déplacement commun à chaque tuile.

— Ligne 10 : on initialise la liste `L` à un couple formée de la tuile la plus proche de la case vide et de sa destination (la case vide est `(ii, jj)`).

— Lignes 12-15 : la boucle sert à faire la même chose pour les tuiles restantes, c'est pour cela que le nombre d'itérations est un de moins que `norm`.

Voici maintenant un code partiel montrant comment le multi-déplacement est organisé (la fonction `anim` a été omise) :

```
1  def multi(orig, empty):
2      i, j=orig
3      ii, jj=empty
4      delta=(di, dj)=(ii-i, jj-j)
5      if di!=0!=dj or di==dj==0:
6          return None
7      norm=max(abs(di), abs(dj))
8
9      dirx, diry =(di//norm, dj//norm)
10     L=[((ii-dirx, jj-diry), (ii, jj))]
11
12     for k in range(norm-1):
13         (a, b), destn=L[-1]
14         pos=((a-dirx, b-diry), (a, b))
15         L.append(pos)
16     return L, (dirx, diry)
17
18 def move_tile(orig, dstn, drtn):
19     i, j=orig
20     nro=board[i][j]
21     rect, txt=items[nro]
22     ii, jj =dstn
23     target=100*jj, 100*ii
24     target_txt=100*jj+50, 100*ii+50
25
26     anim(rect, target, drtn)
27     anim(txt, target_txt, drtn)
28
29     board[i][j],board[ii][jj]=board[ii][jj], board[i][j]
30
31 def clic(event):
32     global i_empty, j_empty
33     i=event.y//100
34     j=event.x//100
35
36     r=multi((i,j), (i_empty, j_empty))
37     if r is None:
38         return
39     L, drtn=r
40     for orig, dstn in L:
41         move_tile(orig, dstn, drtn[::-1])
```

```
42      i_empty=i
43      j_empty=j
```

— Ligne 31 : quand une case `(i, j)` est cliquée, la fonction `clic` est appelée.
— Ligne 36 : immédiatement, la fonction `multi` d'analyse du plateau est appelée.
— Ligne 37-38 : si elle renvoie `None`, c'est qu'aucun déplacement n'est à affectuer et la fonction `clic` se termine.
— Sinon, à partir des informations fournies par `multi`, on peut appeler sur chaque tuile (lignes 40-41), une fonction `move_tile` de déplacement de tuile. Il ne reste plus qu'à mettre à jour les variables globales des positions de la case vide (lignes 42-43).
— La fonction `move_tile` (lignes 18-29) prépare l'appel à la fonction d'animation `anim` (lignes 26-27) et met à jour le plateau (ligne 29) une fois la tuile déplacée (ce qui aura aussi pour effet de placer la case vide de `board` au bon endroit).

Le code complet est donné ci-après.

Code final

Voici un code qui produit un taquin avec multi-déplacement progressif, un bouton pour mélanger et le message de félicitation. En outre, une « protection » a été ajoutée pour éviter des bugs de déplacement lorsqu'on fait des clics très rapprochés. Cette protection consiste à empêcher tout nouveau déplacement de (groupe de) tuiles tant que le ou les tuiles en mouvement ne sont pas arrivées à destination (cela évite de corrompre la structure `board`). Cette protection est implémentée en utilisant un drapeau `moving` qui enregistre le nombre d'items du canevas qui sont actuellement en mouvement.

taquin_progressif_multi_complet.py

```
1  from random import randrange
2  from tkinter import *
3
4  DELTA=20
5  DIST=10
6  moving=0
7
8  def multi(orig, empty):
9      i, j=orig
10     ii, jj=empty
11     delta=(di, dj)=(ii-i, jj-j)
12     if di!=0!=dj or di==dj==0:
13         return None
14     norm=max(abs(di), abs(dj))
15
16     dirx, diry =(di//norm, dj//norm)
17     L=[((ii-dirx, jj-diry), (ii, jj))]
18
19     for k in range(norm-1):
20         (a, b), destn=L[-1]
21         pos=((a-dirx, b-diry), (a, b))
22         L.append(pos)
```

```
23      return L, (dirx, diry)
24
25  def anim(item, target, drtn):
26      global moving
27      L =cnv.coords(item)
28      a=L[0]
29      b=L[1]
30      x, y=target
31      u, v=drtn
32      d=u*(x-a)+v*(y-b)
33      if d>DIST:
34          cnv.move(item, u*DIST, v*DIST)
35          cnv.after(DELTA, anim, item, target, drtn)
36      else:
37          cnv.move(item, (x-a), (y-b))
38          moving-=1
39
40  def move_tile(orig, dstn, drtn):
41      global moving
42      i, j=orig
43      nro=board[i][j]
44      rect, txt=items[nro]
45      ii, jj =dstn
46      target=100*jj, 100*ii
47      target_txt=100*jj+50, 100*ii+50
48
49      anim(rect, target, drtn)
50      anim(txt, target_txt, drtn)
51      moving+=2
52
53      board[i][j],board[ii][jj]=board[ii][jj], board[i][j]
54
55  def congrat():
56      global bravo
57      if not moving:
58          lbl.configure(text="Bravo !")
59          bravo=True
60      else:
61          cnv.after(20, congrat)
62
63
64  def clic(event):
65      global i_empty, j_empty
66      if bravo:
67          return
68      i=event.y//100
69      j=event.x//100
70
```

```
71      r=multi((i,j), (i_empty, j_empty))
72      if (r is None) or moving:
73          return
74      L, drtn=r
75      for orig, dstn in L:
76          move_tile(orig, dstn, drtn[::-1])
77      i_empty=i
78      j_empty=j
79      if board==win:
80          congrat()
81
82  def voisins(n, i, j):
83      return [(a,b) for (a, b) in
84              [(i, j+1),(i, j-1), (i-1, j), (i+1,j)]
85              if a in range(n) and b in range(n)]
86
87  def echange(board, empty):
88      i, j=empty
89      V=voisins(4, i, j)
90      ii, jj=V[randrange(len(V))]
91      board[ii][jj], board[i][j]=board[i][j],board[ii][jj]
92      return ii, jj
93
94  def normal(board, empty):
95      i_empty, j_empty = empty
96      for i in range(i_empty, 4):
97          (board[i][j_empty], board[i_empty][j_empty])= (
98              board[i_empty][j_empty], board[i][j_empty])
99          i_empty=i
100     for j in range(j_empty, 4):
101         board[i_empty][j], board[i_empty][j_empty]= (
102             board[i_empty][j_empty],board[i_empty][j])
103         j_empty=j
104
105 def melanger(N):
106     board=[[4*lin+1+col for col in range(4)]
107         for lin in range(4)]
108
109     empty=(3,3)
110
111     for i in range(N):
112         empty=echange(board, empty)
113     return board
114
115 def init(N=1000):
116     global i_empty, j_empty, items, board, bravo
117     cnv.delete("all")
118     items=[None]
```

```
119
120     board=melanger(N)
121     for i in range(4):
122         for j in range(4):
123             if board[i][j]==16:
124                 i_empty, j_empty=i, j
125     empty=i_empty, j_empty
126     normal(board, empty)
127     i_empty, j_empty=3,3
128     items=[None for i in range(17)]
129
130     for i in range(4):
131         for j in range(4):
132             x, y=100*j, 100*i
133             A, B, C=(x, y), (x+100, y+100), (x+50, y+50)
134             rect=cnv.create_rectangle(A, B, fill="royal blue")
135             nro=board[i][j]
136             txt=cnv.create_text(C, text=nro, fill="yellow",
137                                 font=FONT)
138             items[nro]=(rect, txt)
139     rect, txt=items[16]
140     cnv.delete(txt)
141     cnv.delete(rect)
142     lbl.configure(text="")
143     bravo=False
144
145
146 win=[[1, 2, 3, 4],
147      [5, 6, 7, 8],
148      [9, 10, 11, 12],
149      [13, 14, 15, 16]]
150
151 FONT=('Ubuntu', 27, 'bold')
152 master=Tk()
153 cnv=Canvas(master, width=400, height=400, bg='gray70')
154 cnv.pack(side='left')
155
156 btn=Button(text="Mélanger", command=init)
157 btn.pack()
158
159 lbl=Label(text="      ", font=('Ubuntu', 25, 'bold'),
160           justify=CENTER, width=7)
161 lbl.pack(side="left")
162
163 cnv.bind("<Button-1>",clic)
164 init()
165
166 master.mainloop()
```

— Ligne 6 : la variable globale `moving` comptabilise le nombre d'items du canevas qui sont en

mouvement. Cette variable est incrémentée chaque fois qu'un item est déplacé (ligne 51) et décrémenté chaque qu'une animation est terminée (cf. ligne 38)

— Ligne 79 : l'annonce de la victoire (lignes 58-59) ne peut être placée à cet endroit du code car la victoire serait annoncée légèrement **avant** la fin du déplacement de la dernière tuile (à cause de l'animation). D'où l'usage de la méthode `after` qui permet de temporiser jusqu'à ce qu'il n'y ait plus d'item en mouvement ce qui garantit que le jeu est terminé.

Prolongements

— Ajouter un compteur de déplacements.

— Implémenter un déplacement des tuiles avec les flèches du clavier.

— Modifier les codes précédents pour accepter un taquin de dimension quelconque, par exemple 8x6.

— Modifier les codes précédents pour pouvoir modifier la taille du canevas et des tuiles.

— Reprendre le code précédent en implémentant une classe `Taquin`.

— Implémenter les tuiles avec des coins arrondis. À faire soi-même de A à Z car Tkinter ne supporte pas nativement les rectangles à coins arrondis.

— Pour améliorer l'esthétique, utiliser des images personnalisées créées avec Gimp ou un outil analogue au lieu d'items Tkinter

— Variante classique : utiliser une image découpée en carrés et à recomposer

— Implémenter un solver naïf : on clique sur un bouton, à n'importe quel moment du jeu et une animation ordonne le taquin ; deux algorithmes peuvent envisagés :

  — on arrange les tuiles ligne par ligne ;

  — on ordonne la ligne la plus haute et la colonne la plus à gauche et on recommence avec le carré restant.

— Implémenter un solver optimal : c'est une application classique du `IDA*` (*iterative deepening A star*). Documentation :

  — The Manhattan Pair Distance Heuristic for the 15-Puzzle (pdf).

  — Solving the 15-Puzzle.

  — Efficiently Searching the 15-Puzzle