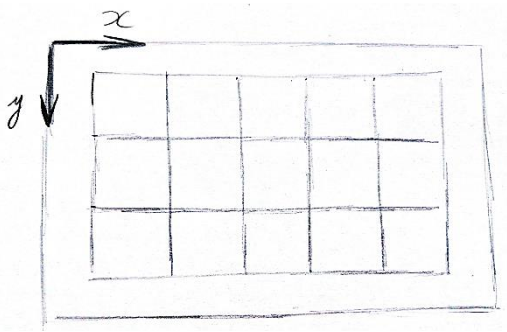
Info 15-A - Tkinter – Tableau dans Canvas

OBJECTIFS : Pour certaines configurations de Canvas, il est intéressant d'insérer les éléments graphiques dans un tableau. On voit cela dans ce Tp.

L'évaluation de ce travail est basée sur le rendu du fichier *.py* qui sera constitué. **On ne demande pas** de constituer de fichier texte contenant des copies d'écran.



1. DEMARRAGE ET IMPORTATION DES BIBLIOTHEQUES :

⇨ Pour débuter, télécharger le dossier *tp15.zip* contenant les fichiers des images qui seront utilisés. Le dézipper dans votre espace personnel sur ***U:*** . Ouvrir un nouveau fichier *.py* dans le dossier contenant ces images et le sauvegarder sous le nom ***tp15A.py*** .

⇨ Copié-collé le code donné ci-dessous, dans votre fichier *tp15A.py* . Il permet d'importer les librairies et d'organiser votre fichier.

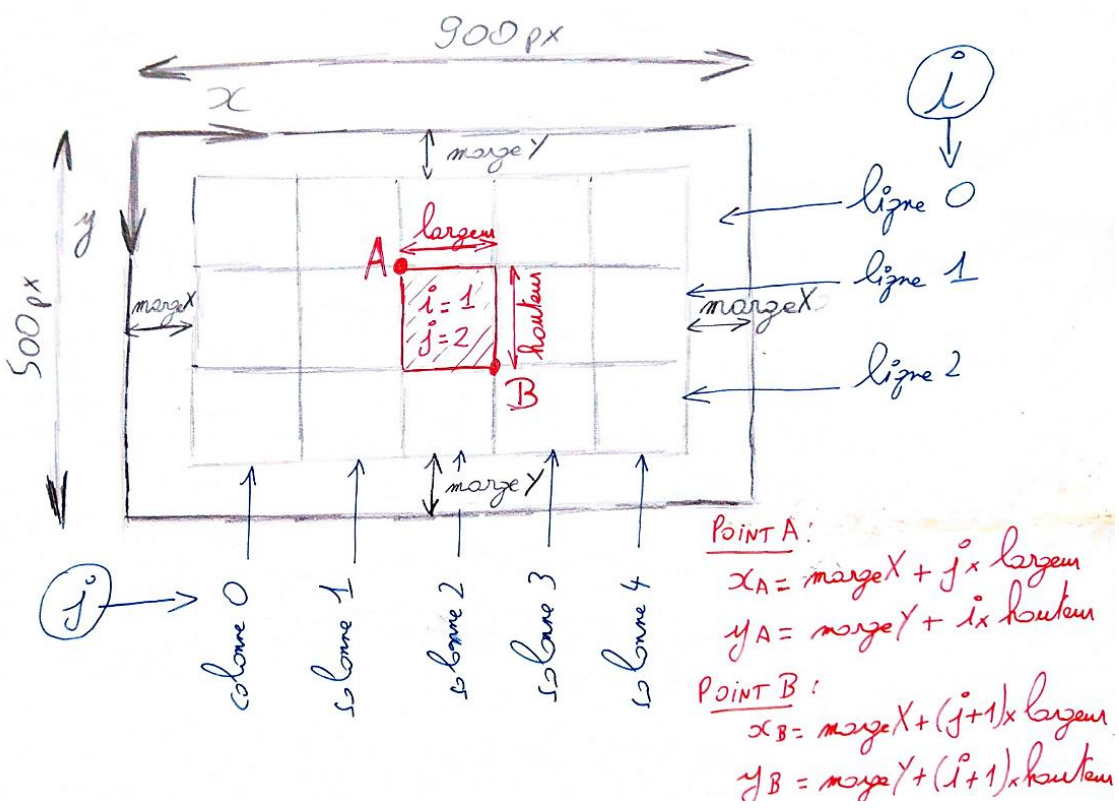
```
# Modules --------------------------------------------------------------
from tkinter import Tk , Menu , Canvas , Label , Entry , StringVar ,Button, Text
from PIL import Image, ImageTk # pip install pillow
from random import randint
# Fonctions ------------------------------------------------------------

# Variables globales ---------------------------------------------------

# Main -----------------------------------------------------------------
```

2. CREATION DE LA PARTIE GRAPHIQUE :

On souhaite insérer dans le Canvas, sur un nombre *nLignes* de lignes, un nombre *nColonnes* de **rectangles**.

Le Canvas a comme dimensions 900 x 500 px.

La marge par rapport aux bords du Canvas, est d'environ 50px.



Pour plus de clarté, on compartimente le code lié à la création de la partie graphique, dans les fonctions suivantes :

- La fenêtre *tkinter* sera créée dans la fonction *creer_fenetre(),*
- Le widget *Canvas* dans la fonction *creer_widgets(),*
- Les dimensions des rectangles à insérer sont calculées dans la fonction *dimensions(),*
- Les rectangles sont créés dans la fonction *creer_graphisme().*

⇨ Créer la fonction *creer_fenetre()* , initialiser les variables globales et appliquer *mainloop()* et tester :

```
# Fonctions ---------------------------------------------------
def creer_fenetre() :
    fenetre = Tk()
    fenetre.title("Tableaux")
    return fenetre

# Variables globales ------------------------------------------
W = 900
H = 500
nLignes = 5
nColonnes = 9

# Main --------------------
fenetre = creer_fenetre()

fenetre.mainloop()
```

On définit la taille de la fenêtre dans le programme principal : 900 x 500

On définit les nombres de lignes et de colonnes que l'on souhaite avoir

On exécute la fonction *creer_fenetre()*

⇨ Créer la fonction *creer_widgets()* et tester :

```
def creer_widgets() :
    zone_graphique = Canvas(fenetre, width=W, height=H , bg = 'black')
    zone_graphique.grid(row = 0 , column = 0)
    return zone_graphique
```

```
# Main ------------------------------------------------------------
fenetre = creer_fenetre()
zone_graphique = creer_widgets()

fenetre.mainloop()
```

⇨ Créer la fonction *dimensions()* et tester *:*

```
def dimensions() :
    margeX = 50
    margeY = 50
    largeur = (W-2*margeX) // nColonnes
    hauteur = (H-2*margeY) // nLignes

    return largeur,hauteur, margeX, margeY
```

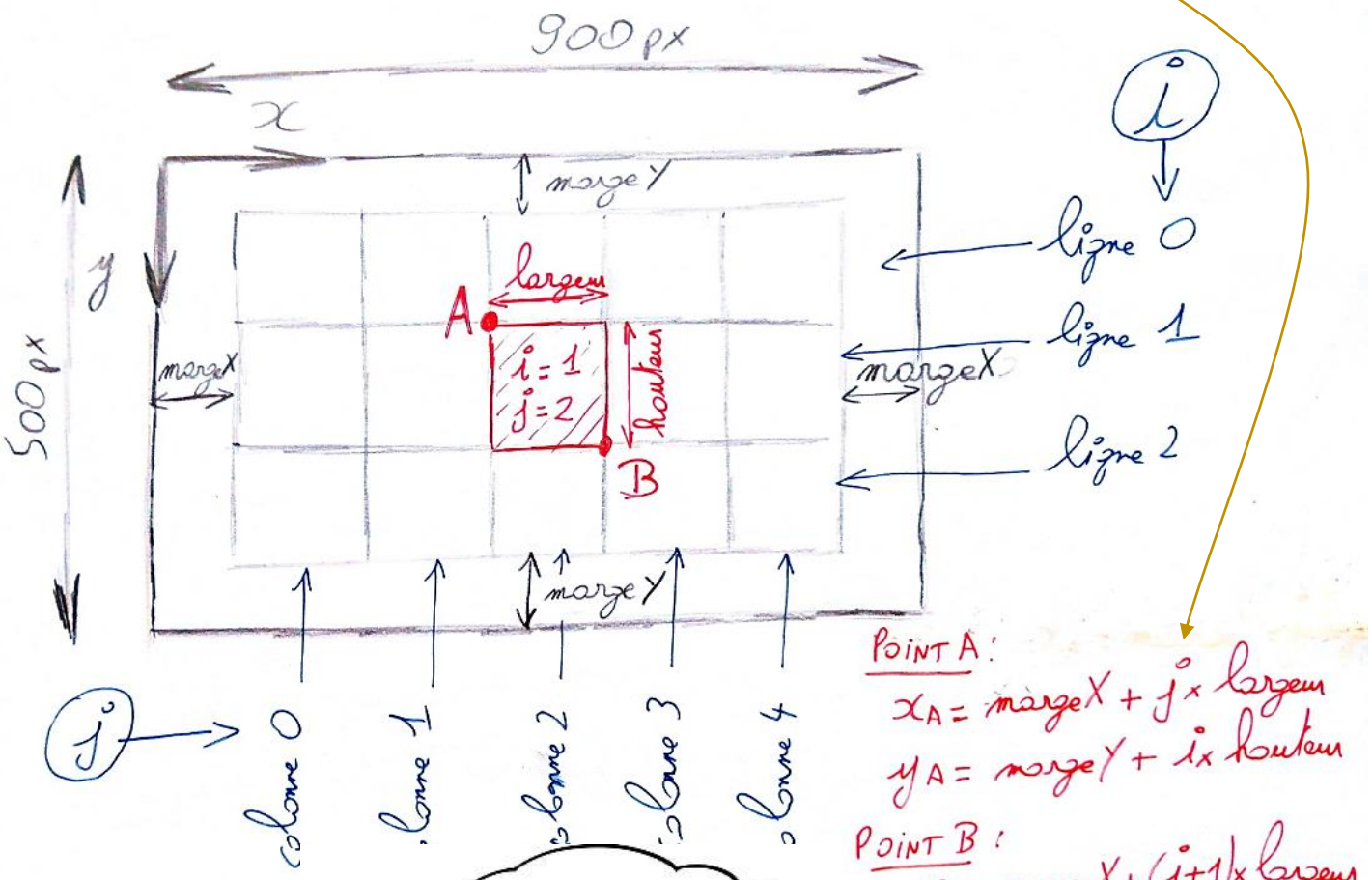
// est l'opérateur de la division euclidienne. Elle retourne un nombre **entier**. Par exemple : 8/5 = 1,6 mais 8//5=1

```
# Main ------------------------------------------------------------
fenetre = creer_fenetre()
zone_graphique = creer_widgets()
largeur,hauteur, margeX, margeY = dimensions()

fenetre.mainloop()
```

⇨ Créer la fonction *creer_graphisme()* et tester *:*

```
def creer_graphisme() :
    tab = []
    for i in range(nLignes) :
        ligne = []
        for j in range(nColonnes) :
            xA = margeX + j*largeur
            yA = margeY + i*hauteur
            xB = margeX + (j+1)*largeur
            yB = margeY + (i+1)*hauteur
            num = zone_graphique.create_rectangle(xA,yA,xB,yB,fill = "white")
            ligne.append(num)
        tab.append(ligne)
    return  tab
```

```
# Main ----------------------------------------------------------------
fenetre = creer_fenetre()
zone_graphique = creer_widgets()
largeur,hauteur, margeX, margeY = dimensions()
tab = creer_graphisme()
```



POINT A :

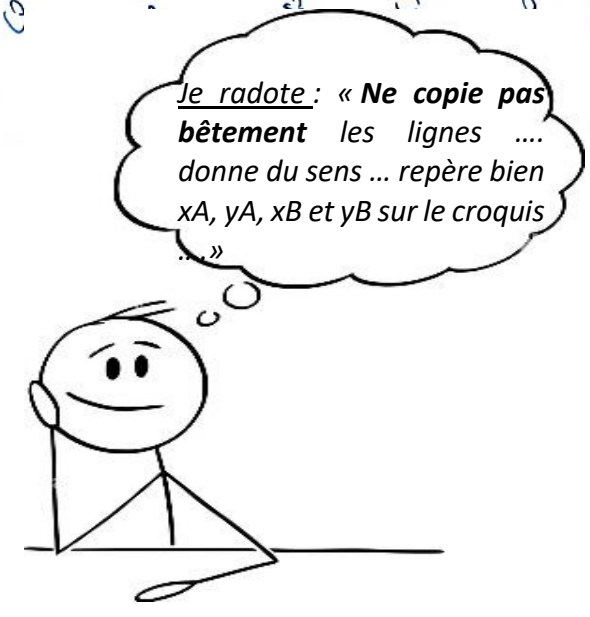
$x_A = margeX + j \times largeur$

$y_A = margeY + i \times hauteur$

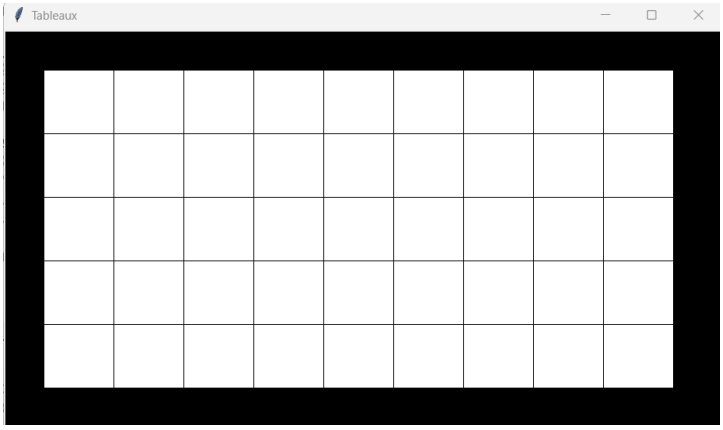
POINT B :

$x_B = margeX + (j+1) \times largeur$

$y_B = margeY + (i+1) \times hauteur$



A l'exécution du script, on obtient bien le tableau de rectangles souhaité :



⇨ Dans la console, afficher le contenu de la liste *tab* :

```
>>> tab
```

…… cette liste contient bien les numéros d'identification de chaque rectangle.

⇨ Exécuter avec

```
nLignes = 50
nColonnes = 90
```

….. normalement, à l'exécution, on obtient bien un tableau de rectangles … mais **mal centré**.

Pour corriger ce problème on retourne dans la fonction *dimension()*. Les variables *largeur* et *hauteur* qui définissent la taille des rectangles ont été obtenus par division euclidienne, car en informatique, la dimension minimale repérable est le pixel. Les distances en pixels sont nécessairement repérées par des nombres entiers. La largeur calculée sera ainsi légèrement plus petite que celle qu'il aurait fallu avoir pour obtenir une marge de 50 px sur les 2 cotés. Pour corriger ce défaut, il est nécessaire de recalculer les marges afin que le centrage soit respecté.

Cela donne le code modifié suivant :

On recalcule les valeurs des marges

```
def dimensions() :
    margeX = 50
    margeY = 50
    largeur = (W-2*margeX) // nColonnes
    hauteur = (H-2*margeY) // nLignes
    margeX = (W-nColonnes*largeur) //2
    margeY = (H-nLignes*hauteur) //2

    return largeur,hauteur, margeX, margeY
```

3. EVENEMENT DE SURVOL SOURIS :

On se propose de recréer l'évènement de survol *hover*, vu en Css dans la partie Web.

⇨ Créer un évènement souri de type « *survol* », dans la partie programme principal. La fonction callback est appelée *survol()* :

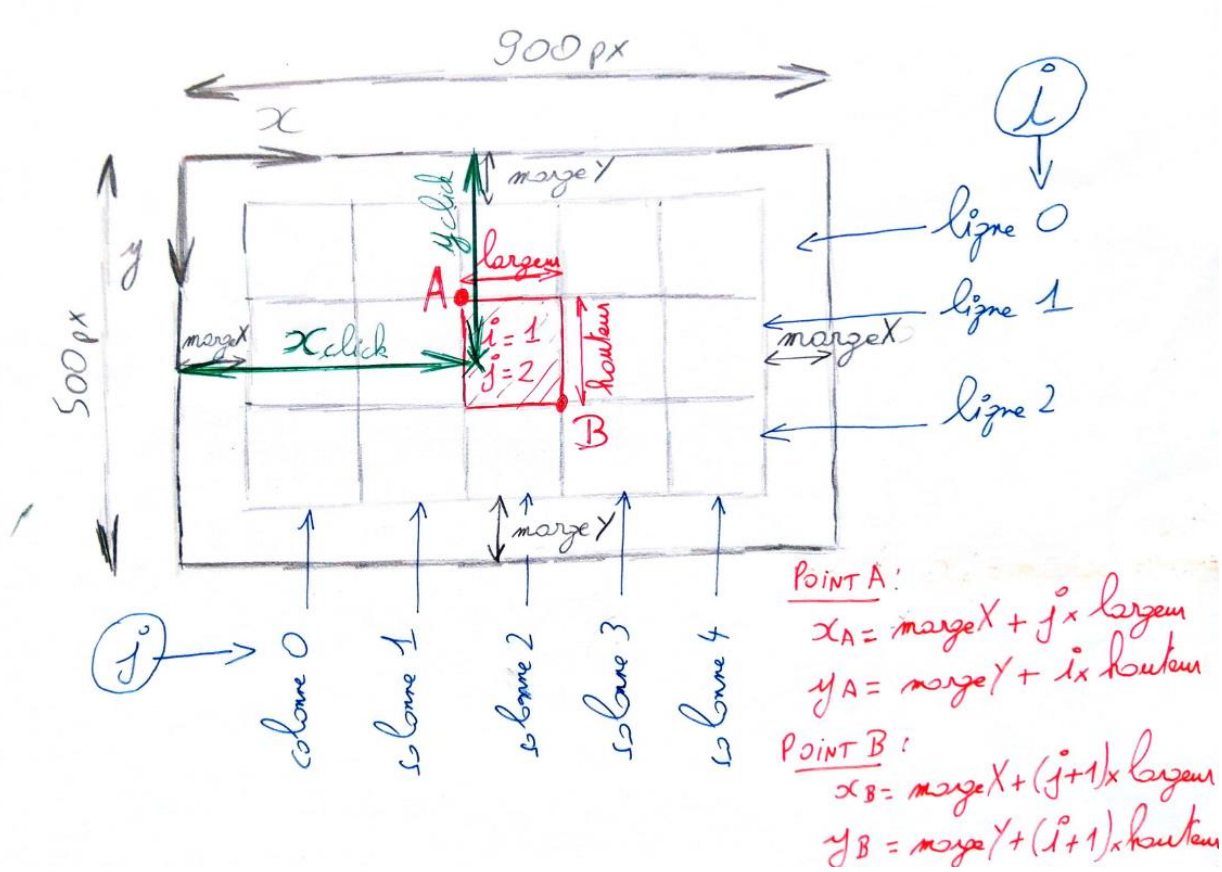
```
# Main ----------------------------------------------------------
fenetre = creer_fenetre()
zone_graphique = creer_widgets()
largeur,hauteur, margeX, margeY = dimensions()
tab = creer_graphisme()
zone_graphique.bind("<Motion>",survol)
```

⇨ Créer la fonction callback associée :

```
def survol(event):
    x_click = event.x
    y_click = event.y
    print(x_click,y_click)
```

⇨ Exécuter et tester l'ensemble pour vérifier l'exactitude des coordonnées affichées dans la console.

On souhaite à présent créer un script qui modifie la couleur du rectangle qui est survolé.



Une solution simple consiste à rechercher dans la liste *tab* les rectangles pour lesquels on a $x_A < x_{click} < x_B$ et $y_A < y_{click} < y_B$. Cela donne le script suivant :

```
def survol(event):
    x_click = event.x
    y_click = event.y
    for i in range(nLignes) :
        for j in range(nColonnes) :
            num = tab[i][j]
            xA,yA,xB,yB = zone_graphique.coords(num)
            if xA < x_click < xB and yA < y_click < yB :
                zone_graphique.itemconfigure(num , fill = "red")
```

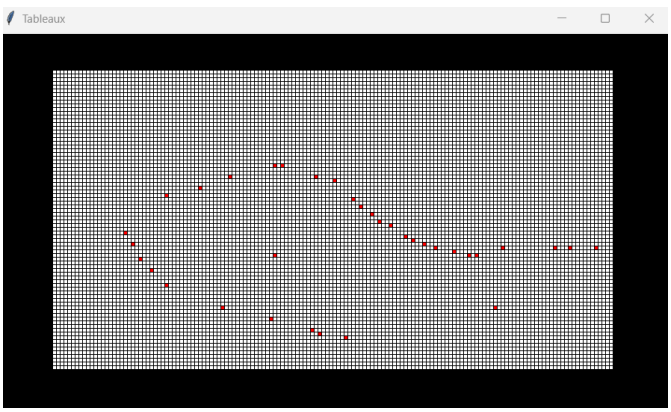
On récupère les coordonnées des points A et B du rectangle

On récupère le numéro d'identification du rectangle

On modifie la couleur du rectangle

⇨ Tester le script avec
```
nLignes = 5
nColonnes = 9
```
… normalement cela fonctionne correctement.

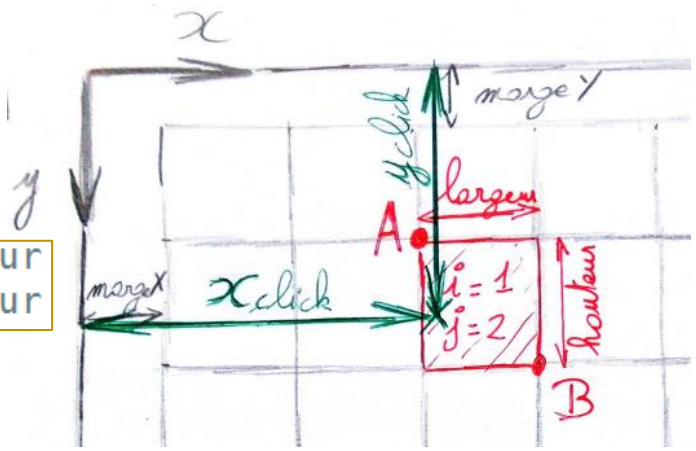
⇨ Tester à présent avec
```
nLignes = 80
nColonnes = 150
```
… on s'aperçoit qu'en survolant rapidement les rectangles, le processeur n'a plus le temps d'exécuter toutes les actions qu'on lui demande. Cela peut se comprendre : l'évènement se produit à chaque fois que la souris se déplace d'un pixel et à chaque fois un double-bouclage est réalisé … la charge de travail devient trop importante.



On propose de procéder autrement.

Comme les dimensions des rectangles sont toutes les mêmes, on peut facilement retrouver les index i et j du rectangle qui est cliqué :

```
i = (y_click - margeY) // hauteur
j = (x_click - margeX) // largeur
```



Mettre le précédent script de survol() en commentaire et écrire le code suivant :

```
def survol(event):
    global numPrecedent
    x_click = event.x
    y_click = event.y
    i = (y_click - margeY) // hauteur
    j = (x_click - margeX) // largeur
    if 0<=i and i<nLignes and 0<=j and j<nColonnes :
        num = tab[i][j]
        zone_graphique.itemconfigure(num , fill = "red")
    else :
        num = None
```

Il faut que $0 \leq i < nLignes$ et $0 \leq j < nColonnes$, sinon cela provoque une erreur. On n'est pas dans ces conditions quand la souris survole la marge.

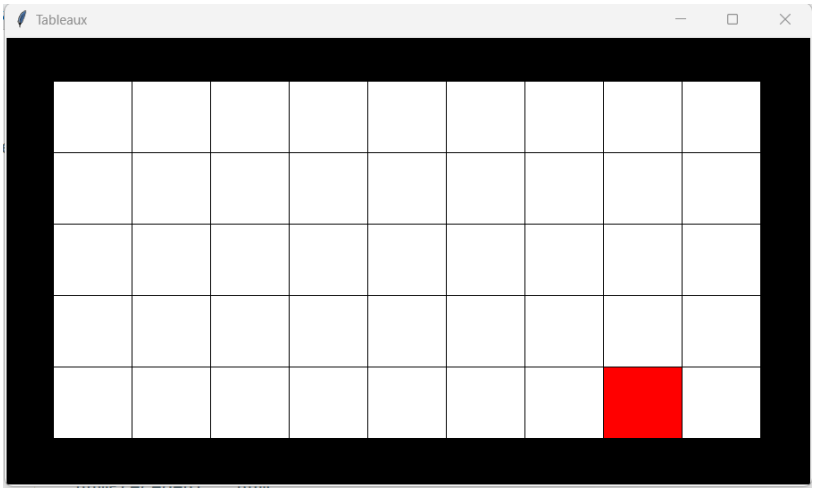
Ligne inutile à ce stade, mais utile dans la suite.

⇨ Tester à présent avec
```
nLignes = 80
nColonnes = 150
```
… on s'aperçoit qu'en survolant rapidement les rectangles, le changement de couleur se fait « mieux qu'avant ».

⇨ Revenir à présent aux valeurs
```
nLignes = 5
nColonnes = 9
```
. On se propose de compléter à présent la fonction survol() afin d'obtenir vraiment l'équivalent d'un « *hover* » en Css. Il s'agit donc de redonner la couleur blanche au rectangle qui est quitté.

Pour cela, il est nécessaire de créer une variable qui mémorise le rectangle précédent qui a été visité. Comme les fonctions callback liées aux évènements n'acceptent pas d'arguments et de retour, on crée la variable globale *numPrecedent* que l'on initialise à la valeur *None* :

```
# Variables globales ----------------------------------------
W = 900
H = 500
nLignes = 5
nColonnes = 9
numPrecedent = None
```



Pour l'instant, dans la fonction *survol(),* la variable *num* contient le numéro d'identification du rectangle qui est survolé. Si aucun rectangle n'est survolé, *num* prend la valeur *None*.

Il s'agit là de copier cette valeur dans cette autre variable *mumPrecedent*.

Lorsque l'on quitte un rectangle, *num* prend une nouvelle valeur qui sera donc différente de celle de *numPrecedent*, qui n'a pas changé. Il s'agit là de détecter cette différence pour ensuite changer la couleur du rectangle repéré par *numPrecedent*. Par contre si *num* ou *numPrecedent* ont la valeur *None*, on ne fait rien.

Cette façon de faire est reprise dans le script ci-dessous. La partie détaillée ci-dessus est bien séparée du reste, pour offrir une meilleur lisibilité :

```
def survol(event):
    global numPrecedent
    x_click = event.x
    y_click = event.y
    i = (y_click - margeY) // hauteur
    j = (x_click - margeX) // largeur
    if 0<=i and i<nLignes and 0<=j and j<nColonnes :
        num = tab[i][j]
        zone_graphique.itemconfigure(num , fill = "red")
    else :
        num = None

    # On remet le rectangle blanc quand on le quitte
    if num != numPrecedent  :
        if numPrecedent != None or num == None :
                zone_graphique.itemconfigure(numPrecedent , fill = "white")
        numPrecedent = num
```

⇨ Tester l'ensemble du script.

⇨ Uploader le fichier *tp15A.py* sur nsibranly.fr avec le code ***tp13***.