Python FAQ FR Documentation

Release 0.1

Daniel Gillet

Oct 24, 2021

PYTHON EN GÉNÉRAL

Cette documentation regroupe les questions les plus fréquemment posées (FAQ) par les débutants en Python.

Cette documentation est un effort collectif. Tout pull request est le bienvenue afin d'ajouter un nouveau sujet ou pour proposer une modification.

CHAPTER

ONE

EXÉCUTER SES SCRIPTS PYTHON

1.1 Windows

Lorsque Python est installé sous Windows, il est déroutant au début de trouver comment exécuter un script. Il existe bien entendu la possibilité d'utiliser **IDLE**. A l'installation, une option par défaut est d'associer les fichier *.py avec le *Python Launcher*. Ceci a pour effet qu'en faisant un double clic sur le fichier *.py, le script s'exécute.

L'exécution d'un script Python a pour effet d'ouvrir une console. C'est la fenêtre avec le fond noir.

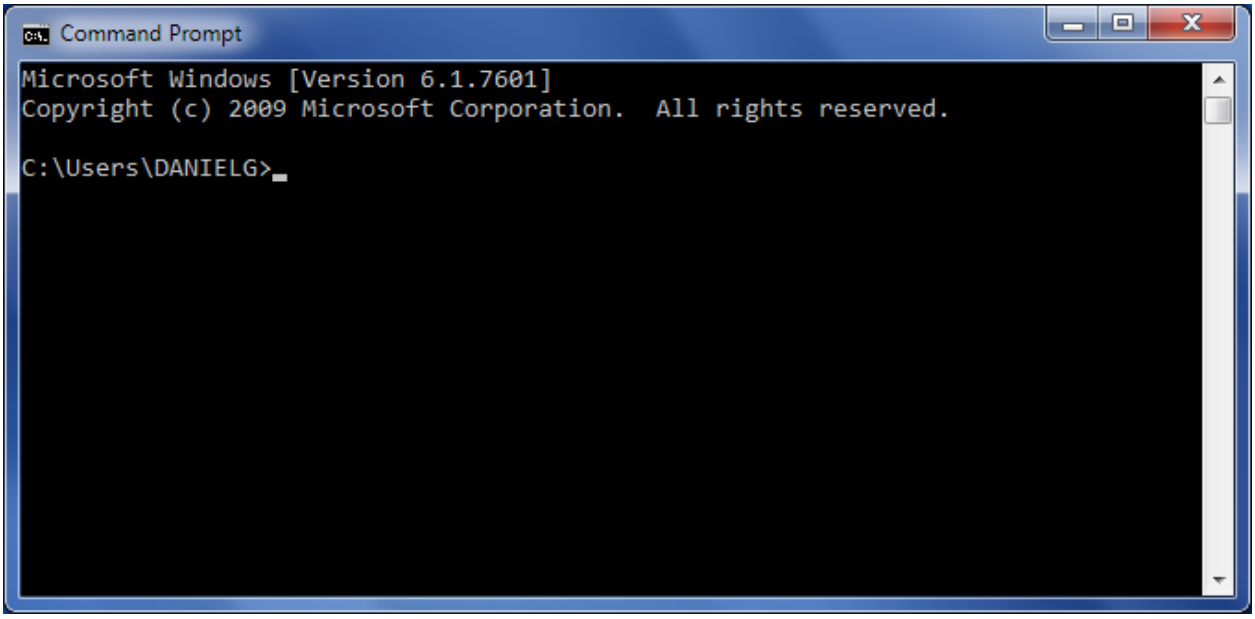


Fig. 1: la console de commande

Le problème est que lorsque le script est terminé, Python n'ayant plus rien à faire se ferme aussi ce qui a pour conséquence de fermer la console de commande qui y était associée. Donc lorsqu'on découvre Python, on nous demande généralement de commencer en tapant ce code :

Listing 1: premier_script.py

```
print("Hello World!")
```

Mais un double clic sur notre fichier `premier_script.py` va ouvrir la console de commande et la refermer aussi vite, ne nous permettant pas de voir notre message. Malheureusement un bien mauvais conseil est alors fourni. On nous dit d'ajouter les lignes suivantes à notre programme :

Listing 2: premier_script.py

```
import os
print("Hello World!")
os.system("pause")
```

Et en effet, en ouvrant notre console, on voit notre message et un deuxième message nous demande d'appuyer sur une touche pour continuer. Fantastique ! Sauf que ...

Warning: L'appel à `os.system("pause")` va exécuter un programme qui est normalement est interne à la console (`cmd.exe`). Cependant si un programme s'appelle `pause.exe` il sera appelé à la place de la commande interne. Ce programme pourrait être malicieux et exécuter du code qui aurait pour but de vous nuire. Cette pratique est fortement découragée, pas seulement pour les risques de sécurité, mais aussi pour la simple raison qu'elle donne une fausse idée de la manière dont un programme fonctionne.

On s'expose au risque décrit ci-dessus. De plus si on fait une erreur dans notre code et que Python lance une exception, la ligne contenant la pause ne sera pas exécutée et on revient au problème initial: la console de commande se referme toute seule, sans nous donner la possibilité de voir où se trouve le problème car le message d'erreur a disparu immédiatement.

1.1.1 Exécuter un script Python sous Windows : la bonne manière

Au lieu de lutter contre le système d'exploitation, il serait préférable de l'utiliser de la manière adéquate. Si on veut exécuter un programme destiné à la console de commande, on ouvre une console de commande et on exécute le script. Pour l'utilisateur lambda de notre programme qui ne connaît rien à la programmation, il pourra double cliquer sur notre script. Mais durant le développement, on veut **voir** ce qui se passe dans cette console. Alors faisons connaissance avec la console de commande.

Comment ouvrir une console de commande ?

La console de commande est un programme comme un autre qui se nomme `cmd.exe`. Il faut donc cliquer sur le bouton Windows et en tapant les lettres *cmd*, Windows nous propose d'ouvrir la console de commande.

On est accueilli par une console au fond noir. On peut configurer quelques propriétés utiles. Pour ce faire, il faut cliquer sur l'icône au haut à gauche de la barre de titre de notre console.

Dans le premier onglet *Options*, je conseille de sélectionner l'option **Mode d'édition rapide** qui permet de sélectionner le texte de la console avec sa souris, pratique pour faire des *copier-coller* des messages de sa console vers un autre programme. En cliquant et déplaçant la souris sur la console, on peut voir qu'une partie du texte se surligne. Pour copier le texte, Il ne faut **pas** appuyer sur *Ctrl-C* mais sur *Return* (la touche retour à la ligne). Le texte sera copier dans le *presse-papier* et peut être coller où on le veut avec un *Ctrl-V*, ou un clic droit et **coller**.

Fig. 2: Ouvrir les propriétés de la console de commande

Comment lancer l'interpréteur Python ?

Il faut d'abord s'assurer que le *Python launcher* est bien visible depuis la console. Il suffit simplement d'entrer la commande `py`. Soit vous vous retrouvez dans l'interpréteur Python :

```
C:\Users\DANIELG>py
Python 3.6.4 (v3.6.4:d48eceb, Dec 19 2017, 06:54:40) [MSC v.1900 64 bit (AMD64)]
 on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> _
```

Fig. 3: Ouvrir l'interpréteur Python depuis la console

Note: On quitte l'interpréteur Python en tapant `exit()` ou en appuyant sur *CTRL-Z*.

Ou alors vous obtenez un message d'erreur disant :

```
C:\Users\DANIELG>py
'py' n'est pas reconnu en tant que commande interne
ou externe, un programme exécutable ou un fichier de commandes.
```

Lorsqu'on entre la commande `py` dans la console, Windows tente de trouver un programme `py.exe` dans le répertoire courant (là où on se trouve). Comme il ne le trouve pas, il va voir dans d'autres répertoires fournis dans la variable d'environnement `%PATH%`. Mais le dossier du *Python launcher* n'a pas été ajouté à la liste des répertoires dans lesquelles chercher.

Pour remédier au problème, il suffit de cliquer sur le bouton Windows, et de sélectionner **Ajouter ou supprimer des programmes**. Dans la liste des programmes, on sélectionne notre installation de Python et on clique sur le bouton **Modifier**. Une nouvelle fenêtre apparaît :

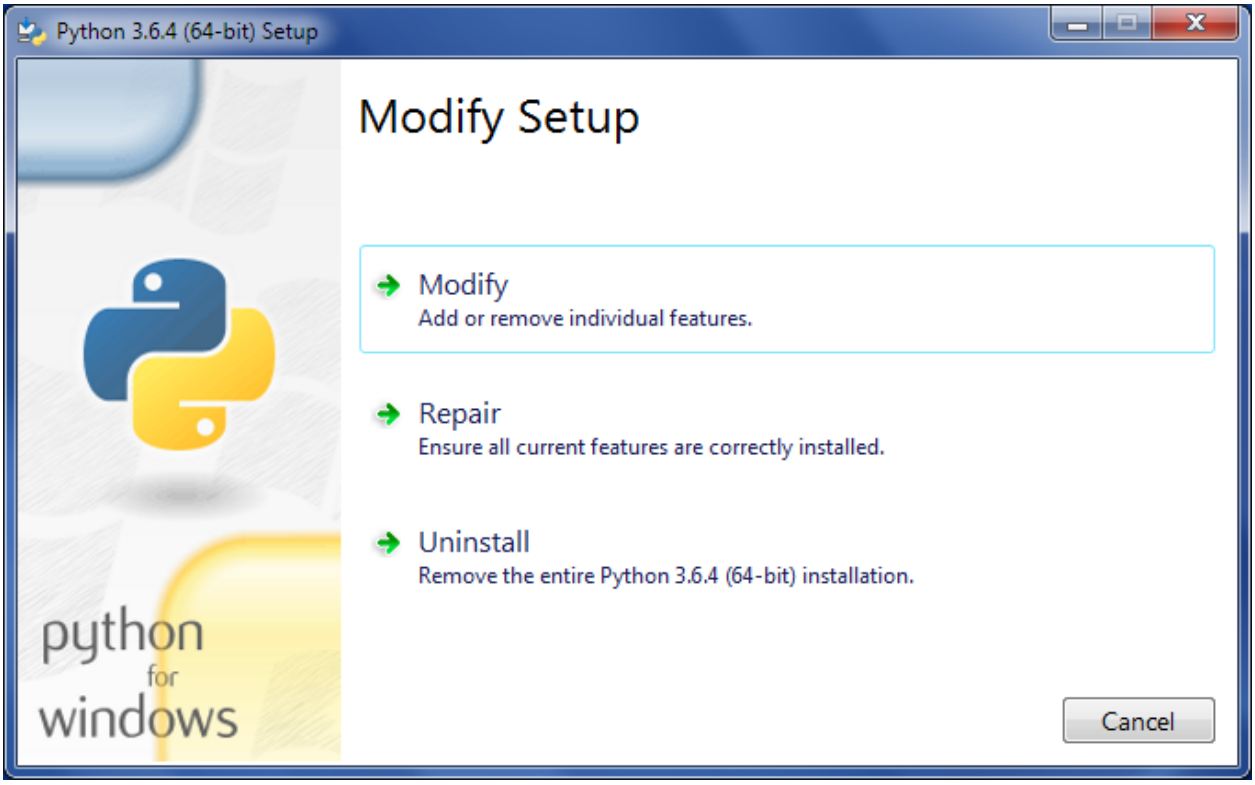


Fig. 4: Modifier l'installation de Python

Il faut choisir **Modifier**. Sur le prochain écran, il n'y a rien à changer.

Finalement c'est le dernier écran qui nous intéresse.

On veut ajouter Python aux variables d'environnement. Une fois cette case cochée et la modification de l'installation terminée, on doit fermer la console et ouvrir une nouvelle afin que les changements soient pris en compte. A présent la commande `py` est bien reconnue et on peut lancer l'interpréteur Python.

Comment exécuter un script Python ?

A présent voyons comment exécuter notre script depuis la console. Il faut tout d'abord *naviguer* vers le dossier où se trouve notre script. La commande `cd` (**C**hange **D**irectory) permet de changer le répertoire dans lequel on se trouve. Mais où se trouve-t-on en fait ? Analysons un instant l'invite fournie par Windows. Dans mon cas c'est :

```
C:\Users\DANIELG>
```

La lettre `C:` est associée à mon disque dur principal, là où est installé Windows. C'est ce qu'on appelle la *racine* de mon disque. Le symbole \ est le symbole séparateur de dossiers. Le tout premier symbole veut dire qu'on part de la racine, puis vient le dossier `Users` qui contient le dossier `DANIELG`. C'est le **répertoire courant**. Pour se rendre dans le répertoire `Desktop` contenu dans le répertoire courant `\Users\DANIELG`, on utilise la commande `cd` suivie du nom du dossier `Desktop`. Comme les programmeurs sont des gros fainéants, on utilise l'*auto-completion*. On ne tape que quelques lettres du dossier et on appuie sur la touche de tabulation `cd Des`*TAB*. Windows nous propose un nom de dossier qui commence par ces lettres. Si par malchance un autre dossier commençait par ces 3 lettres aussi, des appuis successifs sur *TAB* permettent de passer à la prochaine suggestion.

Tip: L'utilisation de la flèche ↑ permet de rappeler une commande précédemment exécutée dans la console de com-

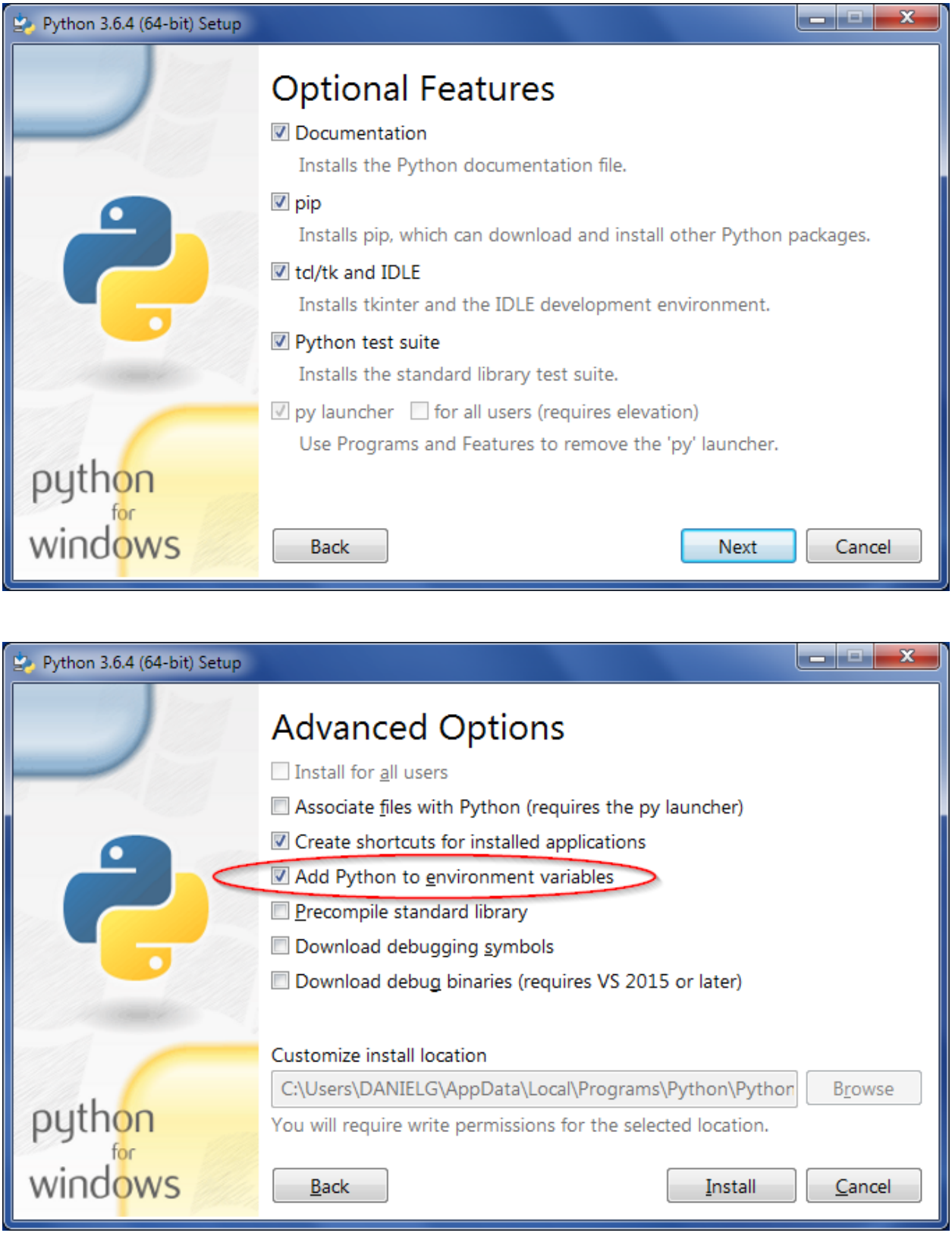


Fig. 5: Les options avancées

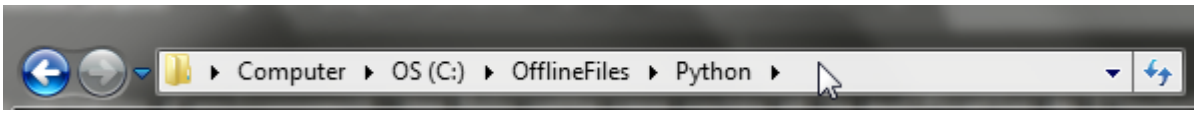
mande.

Si on désire se rendre dans le dossier parent, on utilise la commande `cd ..`. Si on veut retourner à la racine, on utilise la commande `cd \`. Il est possible de mettre immédiatement tout le chemin en une fois et en utilisant à chaque fois *TAB* pour nous éviter de tout taper. Donc si je dois me rendre dans le dossier `C:\OfflineFiles\Python` alors que je suis actuellement dans le dossier `\Users\DANIELG`, je dois retourner depuis la racine et me rendre successivement dans `OfflineFiles` et `Python`. En une seule commande, ça donne `cd \OfflineFiles\Python`. Et bien entendu je n'ai tapé que `of` suivi de *TAB* puis `\p` suivi de *TAB* et Windows a complété le chemin pour moi :

```
C:\Users\DANIELG>cd \OfflineFiles\Python

C:\OfflineFiles\Python>
```

Vous devez donc à présent vous rendre dans le dossier où se trouve votre script Python. Il est aussi possible d'ouvrir l'explorateur Windows, de vous rendre là où se trouve votre script Python et cliquer sur la barre de navigation.



En cliquant dessus, le chemin apparaît. Pour moi c'est `C:\OfflineFiles\Python`. On peut copier ce texte avec *Ctrl-C* et le placer dans la console en tapant d'abord la commande `cd` suivie d'un espace puis un clic droit de la souris dans la console va coller le chemin.

Une fois dans le dossier où se trouve notre script, il ne nous reste plus qu'à l'exécuter en tapant :

```
C:\OfflineFiles\Python>py premier_script.py
```

On peut à présent voir en toute quiétude l'exécution de notre script ou ses messages d'erreurs. Notre console reste ouverte.

CHAPTER

TWO

GÉRER LES COLLISIONS AVEC PYGAME

Voici un petit tuto qui explique comment gérer les collisions avec pygame pour un jeu de type _plateforme_.

Pygame offre quelques fonctions de collisions avec les objets *Rect*. Les difficultés apparaissent dès qu'on veut réagir à une collision. Ce petit tuto va tenter d'expliquer comment détecter une collision et comment y réagir.

2.1 Détection des collisions

Nous ne traiterons que le cas simple de rectangles qui entrent en collision avec d'autres rectangles. Ces rectangles sont orientés selon les axes. Autrement dit ils ne sont pas _penchés_. Pour savoir s'il y a collision, on peut soit utiliser la méthode *colliderect* d'un objet *Rect* en lui passant en argument un autre objet *Rect*. La méthode retourne *True* s'il y a collision, *False* sinon.

Il est aussi possible de facilement faire cette détection par nous-même en utilisant le théorème de la séparation des axes. Le principe est très simple. Considérons 2 rectangles A et B.

![Séparation des axes][SAT]

Si le rectangle B est à gauche du rectangle A, on voit que peu importe sa hauteur, il n'entrera jamais en intersection avec A. De même si B est au-dessus de A, peu importe s'il est à gauche ou à droite, il n'entrera jamais en intersection avec A, Le même raisonnement s'applique si B est en-dessous ou à droite.

On détermine le premier cas où B est à gauche de A en vérifiant que *B.right < A.left*. On fera une opération similaire pour les autres cas.

La chose étonnante est que si on n'est dans aucun des 4 cas précédents, alors **forcément** il y a intersection entre A et B. Ainsi on pourrait écrire une fonction de détection de collision entre 2 *Rect* de cette manière :

```
def collision(rectA, rectB):
    if rectB.right < rectA.left:  # rectB est à gauche return False
    if rectB.bottom < rectA.top:  # rectB est au-dessus return False
    if rectB.left > rectA.right:  # rectB est à droite return False
    if rectB.top > rectA.bottom:  # rectB est en-dessous return False
    # Dans tous les autres cas il y a collision return True
```

2.2 Squelette de notre jeu de plateforme

C'est déjà bien de savoir comment détecter des collisions. Mais ça ne nous dit pas quoi faire lorsqu'il y a collision ! Dans le cas de notre jeu de plateforme, notre personnage peut bouger à gauche, à droite, il peut sauter et il est soumis à la pesanteur. Ainsi il tente de bouger constament vers le bas. C'est le sol sous ses pieds qui l'empêche de tomber.

Voici tout d'abord le squelette de notre jeu sans s'occuper des collisions. Le personnage peut aller à gauche et à droite. Et on peut sauter. Il tombe aussi mais on limite sa chute à une hauteur prédéfinie, sans quoi il disparaîtrait rapidement de l'écran.

``` #!/usr/bin/env python3 # -- *coding: utf-8* --

import pygame from pygame.locals import *

# On initialise pygame pygame.init() taille_fenetre = (600, 400) fenetre_rect = pygame.Rect((0, 0), taille_fenetre) screen_surface = pygame.display.set_mode(taille_fenetre)

BLEU_NUIT = ( 5, 5, 30) VERT = ( 0, 255, 0) JAUNE = (255, 255, 0)

timer = pygame.time.Clock()

joueur = pygame.Surface((25, 25)) joueur.fill(JAUNE) # Position du joueur x, y = 25, 100 # Vitesse du joueur vx, vy = 0, 0 # Gravité vers le bas donc positive GRAVITE = 2

mur = pygame.Surface((25, 25)) mur.fill(VERT)

**niveau = [** [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1], [1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1], [1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1], [1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1], [1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1], [1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1], [1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], ]

**def dessiner_niveau(surface, niveau):** “””Dessine le niveau sur la surface donnée.

Utilise la surface *mur* pour dessiner les cases de valeur 1 “”” for j, ligne in enumerate(niveau):

**for i, case in enumerate(ligne):**

**if case == 1:** surface.blit(mur, (i*25, j*25))

# Boucle événementielle continuer = True while continuer:

**for event in pygame.event.get():**

**if event.type == QUIT:** continuer = False

**elif event.type == KEYDOWN:**

**if event.key == K_SPACE:** vy = -20

timer.tick(30) keys_pressed = pygame.key.get_pressed() vx = (keys_pressed[K_RIGHT] - keys_pressed[K_LEFT]) * 5 vy += GRAVITE vy = min(20, vy) # On limite la vitesse de la chute à un max de 20 # Chaque frame (1/30 sec) on avance de la vitesse vx et vy. C'est comme # si on admettait que le temps unitaire était de 1/30 de sec. x += vx y += vy # Limitons pour le moment jusqu'où le personnage peut tomber y = min(300, y)

screen_surface.fill(BLEU_NUIT) dessiner_niveau(screen_surface, niveau) screen_surface.blit(joueur, (x, y)) pygame.display.flip()

pygame.quit() ```

On voit notre personnage tomber et s'arrêter à la hauteur de 300 px. On peut aller à gauche, à droite et on peut sauter. A chaque frame on calcule sa vitesse *vx* en regardant si les flèches gauche et droite sont enfoncées. Si oui *keys_pressed[K_RIGHT]* vaudra 1 et on le multiplie par 5, ce qui donne un déplacement de 5 px par frame. Si c'est *keys_pressed[K_LEFT]* qui est enfoncée, la valeur sera aussi de 1, mais on retire cette valeur, donc ce sera -1 si on
```

enfonce que cette touche, et on multiplie aussi par 5. Si on enfonce les deux flèches, les vitesses s'annulent et on ne va nul part.

Pour la gravité, c'est très simple. On ajoute à la vitesse *vy* une accélération constante de 2 px par frame. Donc à la première frame, *vy* vaut 0 et on ajoute 2. Donc vy vaut 2. Ensuite à la frame suivante on ajoute encore 2, et *vy* devient 4. Et ainsi de suite. On limite toutefois la vitesse à 20 px / frame, juste pour que la chute n'atteigne pas des vitesses vertigineuses.

2.3 Détecter les collisions avec le décor

Pour détecter si notre personnage entre en collision avec le décor, on doit vérifier si notre niveau contient des valeurs de 1. Là où se trouve notre personnage. La position de notre personne x, y définit en réalité son coin supérieur gauche. Voici un exemple de sa position à un moment donné.

![Collision avec un mur][pos in grid]

On voit que puisque la taille du joueur est la même que la taille des cases de notre niveau, le joueur ne peut à tout moment entrer en contact qu'avec 4 cases : la case contenant son coin supérieur gauche, la case à droite, la case en dessous et la case en-dessous et à droite. Il nous faut donc une fonction qui pour une position *x, y* nous dise dans quel case *i, j* on se trouve. Il nous faut aussi une autre fonction qui nous retourne une liste de *Rect* représentant les murs de notre niveau dans les 4 cases en partant de la position *i, j*. C'est donc la case contenant le coin supérieur gauche de notre personnage.

Pour la première fonction, on obtient ceci :

``` def from_coord_to_grid(pos):

    """Retourne la position dans le niveau en indice (i, j)

    *pos* est un tuple contenant la position (x, y) du coin supérieur gauche. On limite i et j à être positif. """ x, y = pos i = max(0, int(x / 25)) j = max(0, int(y / 25)) return i, j

```

Cette fonction ne prend qu'un seul argument qui sera par exemple un tuple avec la position *x, y*. On récupère les deux éléments de notre argument dans les variables *x* et *y*. Voir [l'unpacking](http://sametmax.com/quest-ce-que-lunpacking-en-python-et-a-quoi-ca-sert/). Puisque nos bloques de décor ont une largeur / hauteur de 25, il suffit de diviser la position par 25. On s'assure ensuite de ne garder que la partie entière. Et on limite le résulat à un nombre positif. On ne veut pas retourner d'indices négatifs, car ça aurait des effets indésirables quand on les utilisera dans des slices.

Pour la deuxième fonction :

``` def get_neighbour_blocks(niveau, i_start, j_start):

    """Retourne la liste des rectangles autour de la position (i_start, j_start).

    Vu que le personnage est dans le carré (i_start, j_start), il ne peut entrer en collision qu'avec des blocks dans sa case, la case en-dessous, la case à droite ou celle en bas et à droite. On ne prend en compte que les cases du niveau avec une valeur de 1. """ blocks = list() for j in range(j_start, j_start+2):

        **for i in range(i_start, i_start+2):**

            **if niveau[j][i] == 1:** topleft = i*25, j*25 blocks.append(pygame.Rect((topleft), (25, 25)))

    return blocks

```

On se créé d'abord une liste viste qui va contenir nos *Rect*. On parcourt ensuite notre liste de listes en visitant les 4 cases partant de *i_start, j_start*. Si la valeur trouvée dans cette case est un 1, il y a un mur et on créé un *Rect* avec

les coordonnées adéquates, à savoir un coin supérieur gauche égal à la position *i, j* multiplié par 25, et une largeur et hauteur de 25. On retourne cette liste de rectangles.

A présent on peut détecter si on entre en collision avec notre décor. Mais que faire dans ce cas ?

Une solution naïve est de garder un copie de notre position lors de la frame précédente, de faire avancer notre personnage selon sa vitesse, de vérifier s'il y a collision avec le décor, et si oui, de le replacer à son ancienne position.

Il suffit de modifier la boucle principale comme ceci, en ajoutant une fonction pour détecter s'il y a collision :

```
def collision(niveau, pos):
    rect = pygame.Rect(pos, (25, 25)) i, j = from_coord_to_grid(pos) for block in get_neighbour_blocks(niveau, i, j):
        if rect.colliderect(block): return True
    return False
# Boucle événementielle continuer = True while continuer:
    for event in pygame.event.get():
        if event.type == QUIT: continuer = False
        elif event.type == KEYDOWN:
            if event.key == K_SPACE: vy = -20
    timer.tick(30) keys_pressed = pygame.key.get_pressed() # Sauvegarde de l'ancienne position old_x, old_y = x, y vx = (keys_pressed[K_RIGHT] - keys_pressed[K_LEFT]) * 5 vy += GRAVITE vy = min(20, vy) # vy ne peut pas dépasser 25 sinon effet tunnel... x += vx y += vy if collision(niveau, (x, y)):
        x, y = old_x, old_y
    screen_surface.fill(BLEU_NUIT) dessiner_niveau(screen_surface, niveau) screen_surface.blit(joueur, (x, y)) pygame.display.flip()
pygame.quit()
```

La fonction *collision* prend en argument notre niveau et la position du personnage. Elle crée un *Rect* à la position du personnage, détermine les coordonnées *i, j* où se trouve notre personnage et itère sur tous les *Rect* retournés par notre fonction *get_neighbour_blocks*. Elle vérifie s'il y a collision et retourne True ou False selon.

Dans notre boucle principale, on garde une copie de notre ancienne position dans *old_x, old_y*. On vérifie s'il y a collision et si oui, on replace notre personnage à l'ancienne position.

On voit notre personnage tomber, puis il semble bloqué. Si on le fait sauter, il se débloque mais se retrouve de nouveau coincé. Aussi il ne tombe jamais _contre_ le bloque en-dessous de lui.

La raison de ce comportement est due au fait qu'à chaque frame, notre personnage est en chute libre. *vy* vaut probablement 20 et donc la nouvelle position du personnage entre en collision avec le bloque sous ses pieds. En cas de collision on l'empêche de bouger, donc il ne sait rien faire d'autre.

2.4 Distance de pénétration

Pour résoudre notre problème, il faudrait réagir différement à la collision. Si le personnage entre en collision, il devrait pouvoir se déplacer jusqu'à arriver contre l'obstacle. Pour ce faire, il va falloir déterminer de combien il a pénétré l'obstacle pour le faire reculer d'autant. Facile à dire ! Mais comment faire ce calcul ?

Prenons le cas du notre personnage qui tombe. A la frame *$ N $*, il était encore au-dessus du bloque sous ses pieds. A la frame *$ N+1 $*, il entre en collision avec.

![Pénétration][penetration]

On voit que le joueur devrait être reculé d'une distance *dy_correction* pour se retrouver en contact avec le bloque. Cette distance est simplement egale à *dy_correction = block.top - perso_rect.bottom*.

On ne doit faire ce calcul que lorsqu'il y a collision. Une manière de procéder est de comparer la position de *perso_rect.bottom* avant le déplacement et après. Si avant le déplacement il était au-dessus de *block.top* et qu'après déplacement il est en-dessous de *block.top* ça veut dire que c'est son côté bas qui vient d'entrer en collision. Et dans ce cas, on peut calculer le *dy_correction*. On peut faire de même pour les 4 faces.

Voici la fonction qui nous retourne les distances de pénétration sur les 2 axes :

``` def compute_penetration(block, old_rect, new_rect):

    """Calcule la distance de pénétration du *new_rect* dans le *block* donné.

    *block*, *old_rect* et *new_rect* sont des pygame.Rect. Retourne les distances *dx_correction* et *dy_correction*.
    """ dx_correction = dy_correction = 0.0 if old_rect.bottom <= block.top < new_rect.bottom:

        dy_correction = block.top - new_rect.bottom

    **elif old_rect.top >= block.bottom > new_rect.top:** dy_correction = block.bottom - new_rect.top

    **if old_rect.right <= block.left < new_rect.right:** dx_correction = block.left - new_rect.right

    **elif old_rect.left >= block.right > new_rect.left:** dx_correction = block.right - new_rect.left

    return dx_correction, dy_correction

```

Elle prend en argument notre *Rect* du mur, et deux autres *Rect* représentant la position du personnage avant et après son déplacement. On reçoit en retour les corrections sur l'axe des X et Y.

On peut à présent se créer un fonction qui gère les collisions de notre personnage et nous retourne sa position finale, après résolution des collisions.

``` def bloque_sur_collision(niveau, old_pos, new_pos, vx, vy):

    """Tente de déplacer old_pos vers new_pos dans le niveau.

    S'il y a collision avec les éléments du niveau, new_pos sera ajusté pour être adjacents aux éléments avec lesquels il entre en collision.

    La fonction retourne la position modifiée pour new_pos. """ old_rect = pygame.Rect(old_pos, (25, 25)) new_rect = pygame.Rect(new_pos, (25, 25)) i, j = from_coord_to_grid(new_pos) blocks = get_neighbour_blocks(niveau, i, j) for block in blocks:

        **if not new_rect.colliderect(block):** continue

        dx_correction, dy_correction = compute_penetration(block, old_rect, new_rect) new_rect.top += dy_correction new_rect.left += dx_correction

    x, y = new_rect.topleft return x, y

# Boucle événementielle continuer = True while continuer:
```

for event in pygame.event.get():

if event.type == QUIT: continuer = False

elif event.type == KEYDOWN:

if event.key == K_SPACE: vy = -20

timer.tick(30) keys_pressed = pygame.key.get_pressed() # Sauvegarde de l'ancienne position old_x, old_y = x, y vx = (keys_pressed[K_RIGHT] - keys_pressed[K_LEFT]) * 5 vy += GRAVITE vy = min(20, vy) # vy ne peut pas dépasser 25 sinon effet tunnel… x += vx y += vy x, y = bloque_sur_collision(niveau, (old_x, old_y), (x, y))

screen_surface.fill(BLEU_NUIT) dessiner_niveau(screen_surface, niveau) screen_surface.blit(joueur, (x, y)) pygame.display.flip()

pygame.quit() ```

On voit le personnage tomber jusqu'à toucher le sol. Hourra ! On se déplace à droite puis à gauche, et le personnage est bloqué ! Arrghh !!!

2.5 Pourquoi le personnage bloque ?

Pour comprendre pourquoi le personnage bloque, il faut comprendre ce qu'il se passe lorsqu'on tente d'aller à gauche.

![Correction minimum][min correction]

Le personnage possède une vitesse *vx* et *vy*. Sa nouvelle position est donc en collision avec les deux bloques sous lui. On teste les collisions dans l'ordre donné sur le diagramme : de gauche à droite et du haut vers le bas. Donc on vérifie les collisions dans la case 3 avant la case 4. Pour la case 3, il y a collision à la fois avec le bord inférieur et avec le bord gauche de notre personnage. La fonction *compute_penetration* nous retournera un *dx_correction* et un *dy_correction* que nous appliquons à la position du personnage et il se retrouve dans sa position initiale. Donc il est bloqué !

En réalité il ne faut corriger que par la plus petite distance pour sortir hors de l'obstacle. Malgré tout, dans cet exemple on voit bien que ça ne fonctionne pas, car la plus petite distance est *dx_correction*. L'autre problème est **l'ordre** dans lequel on considère les bloques. Si on considérait la case 4 avant la 3, on verrait que notre personnage n'entre en collision qu'avec le bas. Et il n'y a qu'une correction *dy_correction*. Ensuite en regardant avec la case 3, il n'y a plus de collision puisque la case 4 nous a repoussé vers le haut.

Mais quel ordre devons-nous choisir ? Il faut commencer par résoudre les cases dont la correction ne s'applique que sur un seul axe, soit X ou Y. On garde les autres bloques pour plus tard. Une fois qu'on a résolu toutes ces collisions, on reprend les bloques qu'on avait gardés, et on résout les collision en n'appliquant que la plus petite correction entre *dx_correction* et *dy_correction*. On entend bien sûr la plus petite valeur **absolue**. Ainsi pour une correction *dx_correction = -8, dy_correction = 2*, on n'appliquerait que la correction *dy_correction*.

Notre nouvelle fonction *bloque_sur_collision* va également prendre en argument les vitesses *vx* et *vy* et elle retournera les vitesses corrigées en cas de collision. Ainsi si notre personnage tombe sur le sol, sa *vy* deviendra 0, jusqu'à la prochaine frame bien entendu…

``` def bloque_sur_collision(niveau, old_pos, new_pos, vx, vy):

"""Tente de déplacer old_pos vers new_pos dans le niveau.

S'il y a collision avec les éléments du niveau, new_pos sera ajusté pour être adjacent aux éléments avec lesquels il entre en collision. On passe également en argument les vitesses *vx* et *vy*.

La fonction retourne la position modifiée pour new_pos ainsi que les vitesses modifiées selon les éventuelles collisions. """ old_rect = pygame.Rect(old_pos, (25, 25)) new_rect = pygame.Rect(new_pos, (25, 25)) i, j = from_coord_to_grid(new_pos) collide_later = list() blocks = get_neighbour_blocks(niveau, i, j) for block in blocks:
```

if not new_rect.colliderect(block): continue

dx_correction, dy_correction = compute_penetration(block, old_rect, new_rect) # Dans cette première phase, on n'ajuste que les pénétrations sur un # seul axe. if dx_correction == 0.0:

new_rect.top += dy_correction vy = 0.0

elif dy_correction == 0.0: new_rect.left += dx_correction vx = 0.0

else: collide_later.append(block)

Deuxième phase. On teste à présent les distances de pénétrations pour # les blocks qui en possédaient sur les 2 axes. for block in collide_later:

dx_correction, dy_correction = compute_penetration(block, old_rect, new_rect) if dx_correction == dy_correction == 0.0:

Finalement plus de pénétration. Le new_rect a bougé précédemment # lors d'une résolution de collision continue

if abs(dx_correction) < abs(dy_correction): # Faire la correction que sur l'axe X (plus bas) dy_correction = 0.0

elif abs(dy_correction) < abs(dx_correction): # Faire la correction que sur l'axe Y (plus bas) dx_correction = 0.0

if dy_correction != 0.0: new_rect.top += dy_correction vy = 0.0

elif dx_correction != 0.0: new_rect.left += dx_correction vx = 0.0

x, y = new_rect.topleft return x, y, vx, vy

```

Comme avant on se créé les *Rect* de notre position avant et après déplacement et on calcule les coordonnées *i, j*. On se prépare une liste *collide_later* dans laquelle on placera les *Rect* pour lesquels la fonction *compute_penetration* a retourné une correction sur les 2 axes.

Pour chaque bloque dans les alentours, on vérifie s'il y a collision. Si oui on calcule les corrections *dx_correction, dy_correction*. Si la correction est nulle sur un axe, on l'applique sur l'autre axe. On en profite pour mettre la vitesse à 0, puisqu'on a heurté qqch dans cette direction. Sinon on place le *Rect* dans notre liste de rectangles qu'on considérera plus tard.

Une fois la boucle finie, on itère cette fois sur les *Rect* qui nous donnaient une correction sur les 2 axes. On re-caclule la correction qui maintenant a pu changer, puisque d'autres bloques auraient pu interagir avec le personnage. On regarde sur quel axe est la plus petite correction, et on annule l'autre correction. Finalement on applique cette correction. On retourne notre nouvelle position et la nouvelle vitesse.

Nous y sommes ! Notre personnage peut se déplacer dans le décor et s'arrêter lorsqu'il entre en contact avec un obstacle. Voici le code final :

``` #!/usr/bin/env python3 # -- *coding: utf-8* --

import pygame from pygame.locals import *

On initialise pygame pygame.init() taille_fenetre = (600, 400) fenetre_rect = pygame.Rect((0, 0), taille_fenetre) screen_surface = pygame.display.set_mode(taille_fenetre)

BLEU_NUIT = (5, 5, 30) VERT = (0, 255, 0) JAUNE = (255, 255, 0)

timer = pygame.time.Clock()

joueur = pygame.Surface((25, 25)) joueur.fill(JAUNE) # Position du joueur x, y = 25, 100 # Vitesse du joueur vx, vy = 0, 0 # Gravité vers le bas donc positive GRAVITE = 2

mur = pygame.Surface((25, 25)) mur.fill(VERT)

niveau = [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0], [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1], [1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1], [1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1], [1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1], [1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1], [1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1], [1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],]

def dessiner_niveau(surface, niveau): “””Dessine le niveau sur la surface donnée.

Utilise la surface *mur* pour dessiner les cases de valeur 1 “”” for j, ligne in enumerate(niveau):

for i, case in enumerate(ligne):

if case == 1: surface.blit(mur, (i*25, j*25))

def from_coord_to_grid(pos): “””Retourne la position dans le niveau en indice (i, j)

pos est un tuple contenant la position (x, y) du coin supérieur gauche. On limite i et j à être positif. “”” x, y = pos i = max(0, int(x // 25)) j = max(0, int(y // 25)) return i, j

def get_neighbour_blocks(niveau, i_start, j_start): “””Retourne la liste des rectangles autour de la position (i_start, j_start).

Vu que le personnage est dans le carré (i_start, j_start), il ne peut entrer en collision qu’avec des blocks dans sa case, la case en-dessous, la case à droite ou celle en bas et à droite. On ne prend en compte que les cases du niveau avec une valeur de 1. “”” blocks = list() for j in range(j_start, j_start+2):

for i in range(i_start, i_start+2):

if niveau[j][i] == 1: topleft = i*25, j*25 blocks.append(pygame.Rect((topleft), (25, 25)))

return blocks

def bloque_sur_collision(niveau, old_pos, new_pos, vx, vy): “””Tente de déplacer old_pos vers new_pos dans le niveau.

S’il y a collision avec les éléments du niveau, new_pos sera ajusté pour être adjacents aux éléments avec lesquels il entre en collision. On passe également en argument les vitesses *vx* et *vy*.

La fonction retourne la position modifiée pour new_pos ainsi que les vitesses modifiées selon les éventuelles collisions. “”” old_rect = pygame.Rect(old_pos, (25, 25)) new_rect = pygame.Rect(new_pos, (25, 25)) i, j = from_coord_to_grid(new_pos) collide_later = list() blocks = get_neighbour_blocks(niveau, i, j) for block in blocks:

if not new_rect.colliderect(block): continue

dx_correction, dy_correction = compute_penetration(block, old_rect, new_rect) # Dans cette première phase, on n’ajuste que les pénétrations sur un # seul axe. if dx_correction == 0.0:

new_rect.top += dy_correction vy = 0.0

elif dy_correction == 0.0: new_rect.left += dx_correction vx = 0.0

else: collide_later.append(block)

Deuxième phase. On teste à présent les distances de pénétrations pour # les blocks qui en possédaient sur les 2 axes. for block in collide_later:

dx_correction, dy_correction = compute_penetration(block, old_rect, new_rect) if dx_correction == dy_correction == 0.0:

Finalement plus de pénétration. Le new_rect a bougé précédemment # lors d’une résolution de collision continue

if abs(dx_correction) < abs(dy_correction): # Faire la correction que sur l’axe X (plus bas) dy_correction = 0.0

elif abs(dy_correction) < abs(dx_correction): # Faire la correction que sur l'axe Y (plus bas) dx_correction = 0.0

if dy_correction != 0.0: new_rect.top += dy_correction vy = 0.0

elif dx_correction != 0.0: new_rect.left += dx_correction vx = 0.0

x, y = new_rect.topleft return x, y, vx, vy

def compute_penetration(block, old_rect, new_rect): """Calcul la distance de pénétration du *new_rect* dans le *block* donné.

block, *old_rect* et *new_rect* sont des pygame.Rect. Retourne les distances *dx_correction* et *dy_correction*. """ dx_correction = dy_correction = 0.0 if old_rect.bottom <= block.top < new_rect.bottom:

dy_correction = block.top - new_rect.bottom

elif old_rect.top >= block.bottom > new_rect.top: dy_correction = block.bottom - new_rect.top

if old_rect.right <= block.left < new_rect.right: dx_correction = block.left - new_rect.right

elif old_rect.left >= block.right > new_rect.left: dx_correction = block.right - new_rect.left

return dx_correction, dy_correction

Boucle événementielle continuer = True while continuer:

for event in pygame.event.get():

if event.type == QUIT: continuer = False

elif event.type == KEYDOWN:

if event.key == K_SPACE: vy = -20

timer.tick(30) keys_pressed = pygame.key.get_pressed() # Sauvegarde de l'ancienne position old_x, old_y = x, y vx = (keys_pressed[K_RIGHT] - keys_pressed[K_LEFT]) * 5 vy += GRAVITE vy = min(20, vy) # vy ne peut pas dépasser 25 sinon effet tunnel... x += vx y += vy x, y, vx, vy = bloque_sur_collision(niveau, (old_x, old_y), (x, y), vx, vy)

screen_surface.fill(BLEU_NUIT) dessiner_niveau(screen_surface, niveau) screen_surface.blit(joueur, (x, y)) pygame.display.flip()

pygame.quit() ```

[SAT]: pygame_collisions_img/SAT.png "Théorème de la séparation des axes" [pos in grid]: pygame_collisions_img/pos_in_grid.png "Collision du joueur avec un mur" [penetration]: pygame_collisions_img/penetration.png "Pénétration des rectangles." [min correction]: pygame_collisions_img/minimum_correction.png "Correction minimum"

CHAPTER

THREE

LES BONNES PRATIQUES AVEC TKINTER

Tkinter est une interface graphique (IHM). Elle permet de pouvoir communiquer graphiquement entre l'homme et la machine afin d'exécuter votre programme python.

Nous allons voir ici certaines bonnes pratiques utilisées afin d'éviter des déboires tant dans la conception de votre programme que dans la syntaxe.

Pour cela nous allons parler ici de :

- l'importation du module
- la compatibilité de python 2.x et 3.x
- l'initialisation de la fenêtre principale
- la création d'un bouton avec sa commande
- le mot clé global

3.1 Importation du module

1. À éviter:

Il faut éviter d'importer son module Tkinter, ni même d'autres modules d'ailleurs sous la forme ci-dessous

*from tkinter import **

La raison est assez simple, lorsque vous importez tkinter de cette manière, vous importez l'ensemble des variables liées au module. Mais connaissez-vous ces noms de variables ? Non ! Il est donc fort possible que vous écrasiez une de ces variables lors de l'écriture de votre code et ainsi rendre cette variable inutilisable par le module lui-même.

2. La bonne manière:

Il est donc important d'importer concrètement le nom de cette variable ou mieux, d'importer tkinter et appeler une de ces variables de cette manière:

```
`python import tkinter # super! ou # peut devenir vite pénible quand beaucoup de
variables à importer from tkinter import variable1, ... `
```

Alors vous me direz, ouais, pas grand chose me convient, importer tkinter est assez verbeux, et l'autre solution semble plus pénible. Pas de problème ! On peut raccourcir cette importation à l'aide du mot clé *as*.

import tkinter as tk et là on met tout le monde d'accord !!! C'est plus court, on peut appeler nos variables à volonté très simplement.

3.2 La compatibilité entre la version 2.x et 3.x

Il arrive parfois que l'on souhaite que notre programme soit utilisable tant pour les utilisateurs de la version 3.x que pour les utilisateurs de la version 2.x (même si on ne conseille plus cette version depuis un certains moment).

Pour cela on va tester une importation avec une certaine version de python, et indiquer qu'en cas d'erreur d'importation, on importe sur l'autre version. On codera cela de la manière ci-dessous…

```python try:

    # pour python 3.x import tkinter as tk

**except ImportError:** # pour python 2.x import Tkinter as tk

```

3.3 Initialisation de la fenêtre principale

Tkinter est une interface graphique, son initialisation est représentée par une fenêtre principale vide attendant des évènements de l'utilisateur.

Niveau code, c'est en une ligne *window = tk.Tk()*

Tk (avec un T majuscule) est le nom de la classe appelée (la classe est l'usine créant l'objet) *window* est le nom de l'instance de classe (l'instance est l'objet créé)

Il est très fortement conseillé d'appeler cette classe une seule fois, surtout quand on débute… Dans le cas où vous souhaitez de multiples fenêtres, il faudra utiliser une autre classe nommée *Toplevel*.

3.4 Création d'un bouton avec sa commande

Pour créer un bouton, il faut appeler (roulement de tambour), la classe Button.

Nous allons ici ne donner qu'un petit exemple parmi beaucoup d'autres sur l'utilisation d'un objet Button.

```python try:

    # pour python 3.x import tkinter as tk

**except ImportError:** # pour python 2.x import Tkinter as tk

root = tk.Tk()

button = tk.Button(root, text='quitter', command=root.quit) button.pack()

root.mainloop() ```

Le bouton *button* créé dans notre fenêtre principale *root* avec comme texte d'information 'quitter', permettra de détruire la fenêtre principale et stopper l'exécution du programme à l'aide de la commande *root.quit()*.

```eval_rst .. note:

Remarquez l'absence de parenthèses après `root.quit`, lié au fait que
le paramètre `command` demande l'objet fonction et non l'objet retourné par la fonction.

```

pack est une méthode pour placer l'objet *button* créé dans la fenêtre *root*.

mainloop indique la création de la boucle événementielle et attend les évènements utilisateurs
```

3.5 Le mot clé global

Ce mot clé permet entre autre, la modification d'une variable à partir d'une fonction.

```python def test():

    global x x = 5

x = 2 print(x) # affiche 2

test() print(x) # affiche 5 ```

Sans le terme *global*, *x* aurait toujours comme valeur 2 !

Ça peut avoir un intérêt, mais lorsqu'on est expérimenté et que l'on sait exactement pourquoi on le fait… Souvent ça permet de rendre fonctionnel (un genre de rustine) un programme où l'on avait besoin d'ajouter une variable dont celle-ci serait modifiée dans une fonction.

Par contre, le mot clé *global* est en règle générale une très mauvaise pratique, car souvent très nombreux dans les codes et rendant le code difficilement débuggable par la suite et maintenable. Si on souhaite modifier une partie du code, c'est souvent ce mot clé qui fait planter le code par la suite, car on ne maîtrise plus les variables qui sont ou ne sont pas modifiées.

```eval_rst .. tip::  Ne l'utilisez jamais, surtout en tant que débutant ! ```

CHAPTER

FOUR

# PROGRAMMATION ÉVÉNEMENTIELLE AVEC TKINTER

Programmer un GUI (*Graphic User Interface*) est différent de la programmation sous console. En effet pour un programme console, on affiche du texte avec des `print` et on attend l'entrée de l'utilisateur avec des `input`. Le code *bloque* tant que l'utilisateur n'a pas enfoncé la touche *Entrée*.

Dans une application GUI, l'interface graphique qui comprend divers *widgets* (tels que boutons, menus, entrées de texte, ...) apparaît à l'utilisateur et le programme *attend* que l'utilisateur interagisse au travers d'une action. On appelle cette action un *événement*. Parmi les événements les plus communs, on peut citer :

- Un clic sur un bouton de la souris
- Le déplacement de la souris
- L'appui sur une touche du clavier
- Le relâchement d'une touche de clavier
- Un clic sur la croix de fermeture de la fenêtre principale
- Un compte à rebours (*timer*) est écoulé (cet événement n'est pas déclenché par l'utilisateur à proprement parler)

Mais comment tkinter fait-il pour *attendre* un événement ? Tout simplement avec une boucle infinie. tkinter ne sortira de cette boucle que lorsque l'utilisateur quittera le programme, le plus souvent en cliquant sur la croix de fermeture de la fenêtre. Voyons comment tout ça fonctionne.

## 4.1 Boucle événementielle

Un exemple minimaliste pour un programme tkinter est le suivant :

```
import tkinter as tk

app = tk.Tk()
app.mainloop()
print("On quitte le programe.") # On teste quand on sort du mainloop
```

Après avoir créé une instance de `Tk`, nous appelons la méthode `mainloop`. On remarquera que tant que la fenêtre tkinter est affichée, la console ne montre pas encore le texte *On quitte le programe*. La raison est bien entendu que la méthode `mainloop` contient notre boucle infinie dont on ne sortira que lorsqu'on fermera la fenêtre. Ensuite on voit notre texte s'afficher dans la console.

Cette méthode `mainloop` est souvent source de confusion au début. On ne comprends pas trop où la placer, on veut en placer à divers endroits du code, et son fonctionnement reste souvent mystérieux. Pour utiliser un outil, il est impératif de le comprendre afin de raisonner correctement. Voyons ce que cache la *boucle principale* `mainloop`.

La boucle principale est donc une boucle infinie avec laquelle on ne peut interagir que de deux manières :

- En définissant une fonction (*callback*) que tkinter devra appeler lorsqu'un événement donné se produira. Exemple: lorsque la souris se déplace, appelle la fonction qui va afficher la position de la souris dans un `Label`. On *lie* un callback à un événement avec la méthode bind. On verra plus en détail son utilisation dans la prochaine section **Réagir à un événement**.
- En créant un compte à rebours (*timer*) qui exécutera une fonction après un temps donné. On nomme communément cette fonction un **callback**, qui signifie *rappel*.

  Exemple: dans 30 secondes, appelle la fonction qui vérifie si j'ai un nouvel email. On crée un nouveau timer avec la méthode after. On verra aussi plus en détail cette méthode dans la section **Gérer le temps**.

Cette boucle événementielle pourrait se schématiser comme ceci :

```
def mainloop(app):
    """Boucle événementielle schématisée (pseudo code) de tkinter"""
    continuer = True
    while continuer:
        # Appel des callbacks enregistrés avec *after* si le temps est écoulé
        # Pour chaque événement (clavier, souris, ...):
            #Si on a lié l'événement avec *bind*, on appelle le callback
        # Si on clique sur la croix pour fermer la fenêtre:
            continuer = False
```

A chaque tour de boucle, tkinter exécute ces opérations :

- Pour chaque événement détecté depuis le dernier tour de boucle (comme l'appui d'une touche de clavier, le déplacement de la souris, ...) tkinter exécute tout *callback* lié à cet événement.
- Si le temps qui s'est écoulé est supérieur au temps du compte à rebours, le *callback* est exécuté.

Les *Widgets* de tkinter utilisent aussi en interne la méthode bind pour produire les comportements attendus. Ainsi un widget `Button` va utiliser un événement de clic gauche de souris pour appeler son propre *callback* (fonction). Ce *callback* va déterminer si la position de la souris était sur le bouton et le cas échéant, va appeler la fonction fournie par l'utilisateur lors de la création du bouton.

Les deux prochaines sections vont illustrer comment lier une fonction à un événement et comment utiliser un timer.

4.2 Réagir à un événement

Comme on vient de le voir, il est possible de *lier* une fonction à un événement grâce à la méthode bind. Nous allons à présent voir un peu plus en détail comment utiliser cette méthode.

4.2.1 Format des événements

Le premier argument de la méthode `bind` est un événement. On le définit par une chaîne de caractères de la forme :

`"<modificateur-type-détail>"`

L'élément le plus important est celui du milieu `type`. C'est le type d'événement qui nous intéresse comme `Button` pour un clic de souris ou encore `Key` pour l'appui d'une touche du clavier. `modificateur` et `détail` permettent d'ajouter des informations supplémentaires sur le type d'événement qui nous intéresse. Par exemple la partie `modificateur` peut renseigner qu'on ne s'intéresse qu'à un *double-clic* (au lieu d'un simple clic) ou encore si c'est la combinaison des touches *Alt-X* qui est enfoncée (contrairement à la touche *X* seule). La partie `détail` nous permettra de renseigner si le clic de bouton doit être le droit ou le gauche par exemple.

Les parties `modificateur` et `détail` peuvent être omises. Pour l'appui d'une touche de clavier, on simplifie même encore plus les choses, puisqu'on ne renseigne que le nom de la touche qui nous intéresse avec ou sans les crochets

"<>". Ainsi pour l'appui de la touche *X* on aura tout simplement la chaîne de caractères "x". Par contre si on veut la combinaison de touches *Alt-X* on écrira "<Alt-x>" entre crochets, puisqu'on renseigne un *modificateur*.

Les types d'événements pour débuter :

Type	Description
Button	Un clic de souris. La partie modificateur peut renseigner si c'est un clic gauche (1), un clic avec la molette (2) ou un clic droit(3)
ButtonRelease	Le bouton de la souris a été relâché
KeyPress (ou Key plus simplement)	L'utilisateur a enfoncé une touche. C'est le type utilisé par défaut quand on ne renseigne que le nom d'une touche
KeyRelease	L'utilisateur a relâché une touche
Enter	L'utilisateur a déplacé la souris dans le Widget sur lequel on *lie* l'événement
Leave	L'utilisateur a déplacé la souris hors du Widget sur lequel on *lie* l'événement

Il existe une longue liste de types d'événements. Il est préférable de lire la documentation pour trouver votre bonheur. On peut trouver quelques exemples ici. Sinon on aura un peu plus de détails avec ces différents liens :

- types d'événements
- modificateurs
- Pour les details, voici la liste des noms des touches spéciales

Quelques exemples d'événements

- "<KeyPress-s>" : Touche *S* enfoncée
- "<Return>" : Touche *Entrée* enfoncée. (Attention! C'est différent de "<Enter>" qui est un type d'événement)
- "<Button-1>" : Clic gauche de souris
- "<Double-Button-1>" : Double clic gauche de souris
- "<Any-KeyPress>" : N'importe quelle touche est enfoncée

4.2.2 Le callback

Le *callback* est la fonction qui sera appelée par tkinter lorsque l'événement se produit. Cette fonction doit accepter un argument, qui sera un objet *event*. Donc sa signature devra être quelque chose comme :

```
def mon_callback(event):
    pass
```

L'objet *event* permet à tkinter de nous donner des informations sur l'événement qui a été déclenché dans différents *attributs*. On accède à ces attributs comme pour n'importe quel objet, avec la notation event.nom_attribut. Voici les différents attributs disponibles :

- widget : c'est l'instance du widget qui a déclenché cet événement.
- x, y : la position de la souris par rapport à la fenêtre
- x_root, y_root : la position de la souris par rapport à l'écran
- char : le caractère (seulement pour un événement clavier) sous forme de chaîne de caractères

- `keysym` : la *représentation* du caractère (seulement pour un événement clavier)
- `keycode` : un code unique pour une touche du clavier
- `num` : le numéro du bouton (seulement pour un événement souris). 1 pour bouton gauche, 2 pour molette et 3 pour bouton droit.
- `width`, `height` : la nouvelle largeur et hauteur du widget (seulement pour un événement `Configure`)
- `type` : le type de l'événement, représenté par un nombre. Tableau de correspondance entre le numéro et le type d'événement

Comme on le voit, certains types d'événements ajoutent des informations supplémentaires. La différence entre `keysym` et `char` se voit seulement avec les touches *spéciales* comme par exemple *F4*. En appuyant sur cette touche, `char` vaut `""` alors que `keysym` vaut `"F4"`.

4.2.3 Mise en pratique

Afin de voir en action comment tout ça fonctionne, on va écrire un simple programme qui va nous indiquer où se trouve la souris lorsqu'on fait un double clic gauche. Il nous faudra donc réagir à l'événement `"<Double-Button-1>"`, lire les attributs `x` et `y` de l'objet `event` et afficher le résultat dans la console :

```
import tkinter as tk

def on_double_click(event):
    print("Position de la souris:", event.x, event.y)

app = tk.Tk()
app.bind("<Double-Button-1>", on_double_click)
app.mainloop()
```

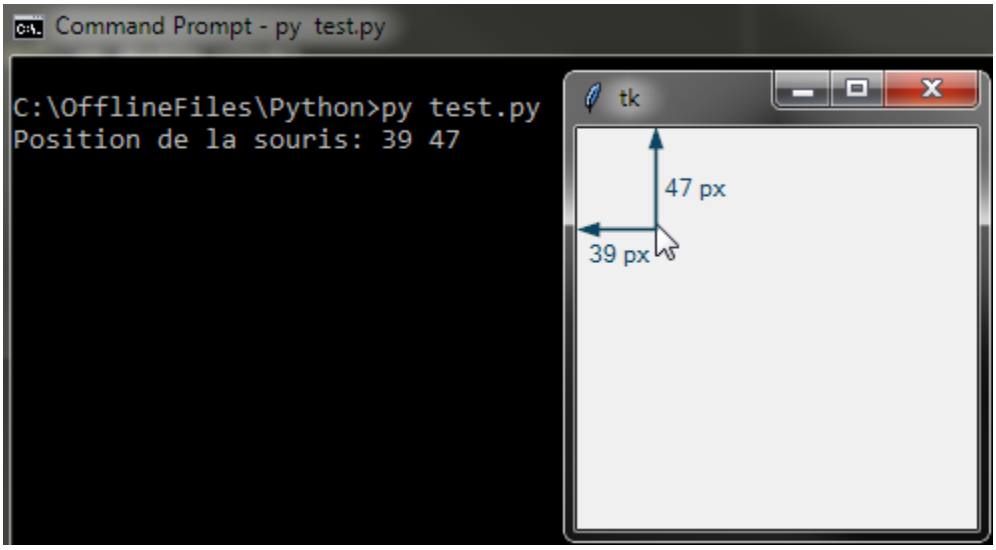


Fig. 1: Position de la souris sur un double clic

4.3 Gérer le temps

Parfois on veut écrire un code qui ne s'exécutera qu'après un délai. L'erreur classique est d'utiliser des fonctions comme `sleep` qui suspendent l'exécution de notre programme pendant un temps donné. Avec notre *boucle principale*, on ne peut pas suspendre le programme, sans quoi tkinter ne gèrerait plus les événements et aux yeux de l'utilisateur, l'application aurait l'air bloquée (*frozen*). En effet, si l'utilisateur cliquait avec sa souris sur un Widget, il n'y aurait plus aucune réaction visible.

Afin de résoudre ce problème, tkinter offre la méthode de widgets `after(delay, function, *args, **kwargs)` qui met dans une *file d'attente* la fonction renseignée. Une fois que le temps `delay` exprimé en millisecondes est écoulé, tkinter exécutera la fonction `function` en lui passant les arguments `args` et `kwargs` fournis.

Prenons un exemple tout simple. On veut faire une petite application avec un compteur qui s'incrémente à chaque seconde. Il nous suffit donc de créer un `Label` qui contiendra le texte qu'on veut montrer (le compteur), et une fonction qu'on va appeler toutes les secondes pour incrémenter notre valeur de 1 :

```
import tkinter as tk

def incremente():
    "Incrémente le compteur à chaque seconde"
    global compteur
    compteur += 1
    compteur_lbl['text'] = str(compteur)

app = tk.Tk()
compteur = 0
compteur_lbl = tk.Label(app, text=str(compteur), font=("", 16))
compteur_lbl.grid(padx=8, pady=8)

app.after(1000, incremente)
app.mainloop()
```

Attention: J'ai utilisé le mot clé `global` afin de changer la valeur de `compteur`. Il y a une manière beaucoup plus propre d'arriver à ce résultat en utilisant un objet `IntVar`. Ceci sera couvert dans une future partie du tutoriel.

On observe que la valeur du `Label` passe de 0 à 1 en 1 seconde, mais ensuite plus rien ne se passe. Pourquoi ?

La raison est que `after` place notre fonction dans la file d'attente. Après 1 seconde, tkinter exécute la fonction et **enlève la fonction de la file d'attente**. Afin d'exécuter la fonction de manière répétitive, il faut qu'ensuite la fonction elle-même se remette dans la file d'attente en utilisant … la méthode `after`. Ce qui donne ce code modifié :

```
import tkinter as tk

def incremente():
    "Incrémente le compteur à chaque seconde"
    global compteur
    compteur += 1
    compteur_lbl['text'] = str(compteur)
    app.after(1000, incremente)

app = tk.Tk()
compteur = 0
compteur_lbl = tk.Label(app, text=str(compteur), font=("", 16))
```

(continues on next page)

(continued from previous page)

```
compteur_lbl.grid(padx=8, pady=8)

app.after(1000, incremente)
app.mainloop()
```

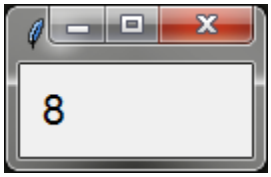


Fig. 2: Chrono en tkinter

A présent le compteur s'incrémente indéfiniment. Si on ne voulait incrémenter le compteur que 10 fois par exemple, on aurait pu ajouter en fin de fonction `incremente` une condition sur la valeur de `compteur`. Si elle est plus petite que 10, on exécute la ligne `app.after(1000, incremente)`. Sinon on ne fait rien, et la fonction ne sera plus dans la file d'attente.

Ajoutons une deuxième action répétitive à notre programme. On ne va rien faire de très compliqué. On va juste ajouter un autre compteur mais qui va un peu plus vite que le premier. Il s'incrémentera de 1 toutes les 0.8 secondes :

```
import tkinter as tk

def incremente():
    "Incrémente le compteur à chaque seconde"
    global compteur
    compteur += 1
    compteur_lbl['text'] = str(compteur)
    app.after(1000, incremente)

def incremente_rapide():
    "Incrémente le compteur toutes les 0.8 secondes"
    global compteur_rapide
    compteur_rapide += 1
    compteur_rapide_lbl['text'] = str(compteur_rapide)
    app.after(800, incremente_rapide)

app = tk.Tk()
compteur = 0
compteur_rapide = 0
compteur_lbl = tk.Label(app, text=str(compteur), font=("", 16))
compteur_lbl.grid(padx=8, pady=8)
compteur_rapide_lbl = tk.Label(app, text=str(compteur_rapide), font=("", 16))
compteur_rapide_lbl.grid(padx=8, pady=8)

app.after(1000, incremente)
app.after(800, incremente_rapide)
app.mainloop()
```

Les deux compteurs s'incrémentent chacun à leur rythme. Il est important de réaliser que la *boucle principale* n'arrête pas de *tourner* en attendant un événement ou que le délai de la fonction dans la liste d'attente soit écoulé. C'est parce que cette boucle tourne continuellement qu'elle donne l'illusion que des actions se produisent simultanément. On a l'impression que chaque compteur est un programme indépendant. Hors il ne s'agit pas de *Threads* ou autres

mécanismes de concurrence. Il s'agit simplement d'une boucle qui tourne rapidement et qui tantôt exécute le code `incremente` et tantôt le code de `incremente_rapide`.

Il est à présent plus clair pourquoi l'une de ces fonctions ne peut pas contenir de fonction `sleep`. Si c'était le cas, le programme s'arrêterait et tkinter n'aurait plus l'opportunité de continuer sa boucle principale. Il n'aurait donc plus l'occasion d'observer le temps qui s'écoule et ne pourrait plus exécuter les éventuelles fonctions placées dans la file d'attente grâce à `after`.

4.4 Raisonner avec une boucle événementielle

Pour bien comprendre les challenges qu'induisent une boucle événementielle, prenons un exemple. Imaginons qu'on veuille écrire un petit jeu **devine le nombre auquel je pense**.

Dans un programme console, ce jeu simplisme pourrait s'écrire ainsi :

```
from random import randint

print("Devine le nombre auquel je pense.")
nombre_secret = randint(0, 100) + 1

gagne = False
while not gagne:
    reponse = int(input("Choisi un nombre entre 1 et 100 inclus: "))
    if nombre_secret > reponse:
        print("Le nombre est plus grand")
    elif nombre_secret < reponse:
        print("Le nombre est plus petit")
    else:
        gagne = True

print("Tu as trouvé le nombre. Bravo!")
```

Il n'y a pas de gestion des mauvaises entrées faites par l'utilisateur. Pour l'exemple ce n'est pas important car ça alourdirait le code.

Si on veut transposer ce jeu à tkinter, on voit tout de suite plusieurs *différences* :

- Pour afficher un message, on doit utiliser un Widget de tkinter. Probablement qu'un Label fera l'affaire.
- Pour demander une entrée à l'utilisateur, on doit aussi utiliser un Widget. Un Entry fera aussi l'affaire.
- Mais comment savoir quand l'utilisateur a fini d'entrer son nombre ? On pourrait ajouter un bouton *Valider* à côté du Widget `Entry`, ou alors on pourrait *lier* l'événement <appui de la touche *Entrée*> avec la fonction qui validerait l'entrée de l'utilisateur.
- Les messages *"Le nombre est plus grand"*, *"Le nombre est plus petit"* et *"Tu as trouvé le nombre. Bravo!"* devraient apparaître dans un autre `Label`.

Mais comment coordonner tout ça ? On va d'abord mettre en place les éléments que l'utilisateur voit au début du jeu :

```
from random import randint
import tkinter as tk

app = tk.Tk()
titre = tk.Label(app, text="Devine le nombre auquel je pense", font=("", 16))
titre.grid(row=0, columnspan=2, pady=8)
```

(continues on next page)

(continued from previous page)

```
nombre_secret = randint(0, 100) + 1

lbl_reponse = tk.Label(app, text="Choisi un nombre entre 1 et 100 inclus:")
lbl_reponse.grid(row=1, column=0, pady=5, padx=5)

reponse = tk.Entry(app)
reponse.grid(row=1, column=1, pady=5, padx=5)

app.mainloop()
```

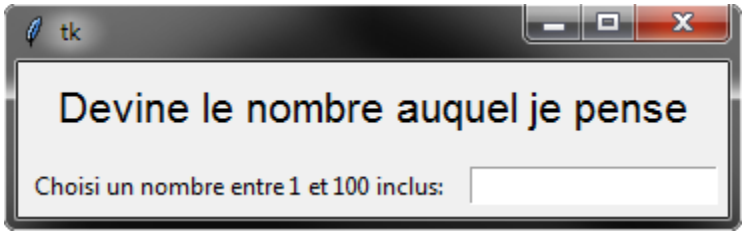


Fig. 3: Devine le nombre auquel je pense

La dernière ligne nous fait entrer dans la boucle événementielle. Mais comme on n'a rien lié comme événement, il ne se passera rien. En gros, on va rester sur cette interface jusqu'à ce que l'utilisateur ferme la fenêtre. On doit maintenant décider de l'élément déclencheur qui nous permettra de poursuivre l'exécution de notre code. Cet élément est l'appui de la touche *Entrée*. Comme dit plus haut, ça aurait pu être l'appui sur un bouton *Valider*. Mais nous choisissons dans cette version de réagir à la touche *Entrée*.

Pour ce faire il suffit d'ajouter en ligne 15 cette ligne de code

```
reponse.bind("<Return>", nombre_choisi)
```

Tip: Ce serait un bon exercice d'ajouter un bouton à droite du *Entry* et en cas de clic, on appellerait aussi la fonction `nombre_choisi`.

On lie donc l'appui de la touche *Entrée* à l'exécution d'une fonction `nombre_choisi` que nous n'avons pas encore définie. Dans cette fonction, on pourra comparer le nombre entré par l'utilisateur au nombre secret et ajouter un `Label` avec le message adéquat.

```
def nombre_choisi(event):
    "Callback quand le joueur a entré un nombre."
    nbre_choisi = int(reponse.get())
    if nombre_secret > nbre_choisi:
        # Faut-il créer un nouveau Label ici ??
    elif nombre_secret < nbre_choisi:
        # Même question
    else:
        # Gagné. Nouveau Label?? Et puis on fait quoi??
```

Mais comme l'indiquent les commentaires du code, si on créé un nouveau `Label`, à chaque exécution de la fonction `nombre_choisi`, un nouveau `Label` viendra se placer sur le précédent. Au final on aura une pile de Label inutile, car seul le dernier Label sera visible. Ce n'est pas ce que l'on veut. On veut pouvoir afficher ce message, et le remplacer par un nouveau message lorsque nécessaire. Autrement dit on veut juste *changer* le texte du *Label*. Il nous faut donc

un `Label` créé dès le départ mais ne contenant rien comme texte. Dans la fonction `nombre_choisi` on pourra juste changer le texte du Label pour afficher ce que l'on souhaite. On peut placer ce texte où bon nous semble. Je choisis de le mettre juste en dessous du `Entry`. Le code devient alors :

```
def nombre_choisi(event):
    "Callback quand le joueur a entré un nombre."
    nbre_choisi = int(reponse.get())
    if nombre_secret > nbre_choisi:
        resultat["text"] = "Le nombre est plus grand"
    elif nombre_secret < nbre_choisi:
        resultat["text"] = "Le nombre est plus petit"
    else:
        resultat["text"] = "Tu as trouvé le nombre. Bravo!"
```

A ajouter dans le corps du script

```
resultat = tk.Label(app, text="")
resultat.grid(row=2, column=1, pady=5, padx=5)
```

Note: Changer le texte d'un Label revient à changer une de ses *options*. Je vous recommande de lire dans la documentation les différencesentes manières de lire et écrire les options d'un Widget: http://effbot.org/tkinterbook/tkinter-widget-configuration.htm

J'utilise ici la méthode `widget["option"] = value`. Plus loin j'utiliserai `widget.config(opt1=val1, opt2=val2, ...)` lorsque je devrai configurer plus d'une option à la fois.

Ce n'est pas encore parfait, mais on a un début de programme utilisable. Il reste quelques *problèmes*. Le premier est que lorsqu'on a trouvé le bon nombre, le programme continue à nous demander un nouveau nombre. Et c'est normal, puisqu'il n'y a rien décrivant ce qu'il faut faire en cas de victoire, hormis afficher un message.

Pour résoudre ce problème, il faudrait déjà décider de ce qu'il doit se passer lorsque le nombre secret est découvert. Si on quitte directement le programme, le joueur n'aura pas le temps de lire le message de victoire et ne comprendra pas pourquoi la fenêtre s'est fermée. Si on laisse le `Entry`, le joueur pourra continuer à entrer de nouveaux nombres, et en appuyant sur *Entrée*, la fonction `nombre_choisi` continuera a être appelée.

L'idéal serait d'*enlever* les éléments qui ne nous sont plus nécessaires et de laisser un Label avec le message de victoire. L'utilisateur n'aura d'autres choix que de fermer lui-même la fenêtre après avoir lu le message. Pour accomplir cette tâche, on peut *détruire* les Widgets inutiles, et replacer notre `resultat` en dessous du titre. Il servira à afficher notre message de victoire :

```
def nombre_choisi(event):
    "Callback quand le joueur a entré un nombre."
    nbre_choisi = int(reponse.get())
    if nombre_secret > nbre_choisi:
        resultat["text"] = "Le nombre est plus grand"
    elif nombre_secret < nbre_choisi:
        resultat["text"] = "Le nombre est plus petit"
    else:
        # On enlève les éléments dont on n'a plus besoin
        lbl_reponse.destroy()
        reponse.destroy()
        # On replace le Label `resultat` dans la ligne en dessous du titre
        resultat.grid_forget()
        resultat.grid(row=1, columnspan=2)
```

(continues on next page)

(continued from previous page)

```
        # On configure le label avec le texte voulu, dans la font voulue et
        # dans la couleur désirée.
        resultat.config(text="Tu as trouvé le nombre. Bravo!",
                        font=("", 12),
                        fg="green")
```

Un deuxième *problème* est que l'utilisateur doit effacer le nombre qu'il a fourni en entrée avant de faire une autre proposition. Une solution serait d'écrire le nombre qu'il a choisi dans un `Label` et d'effacer le contenu du `Entry`. Tout comme pour le *Label* `resultat`, on va se créer un *Label* vide qu'on remplira une fois que le joueur aura fait une proposition. On veillera aussi à garder la proposition en cas de victoire. Le code finale donne ainsi :

```
from random import randint
import tkinter as tk

def nombre_choisi(event):
    "Callback quand le joueur a entré un nombre."
    nbre_choisi = int(reponse.get())
    reponse.delete(0, tk.END)
    proposition["text"] = nbre_choisi
    if nombre_secret > nbre_choisi:
        resultat["text"] = "Le nombre est plus grand"
    elif nombre_secret < nbre_choisi:
        resultat["text"] = "Le nombre est plus petit"
    else:
        # On enlève les éléments dont on n'a plus besoin
        lbl_reponse.destroy()
        reponse.destroy()
        # On replace les Labels `proposition` et `resultat` dans la ligne
        # en dessous du titre
        proposition.grid_forget()
        proposition.grid(row=1, column=0)
        resultat.grid_forget()
        resultat.grid(row=1, column=1)
        # On configure le label avec le texte voulu, dans la font voulue et
        # dans la couleur désirée.
        resultat.config(text="Tu as trouvé le nombre. Bravo!",
                        font=("", 12),
                        fg="green")


app = tk.Tk()
titre = tk.Label(app, text="Devine le nombre auquel je pense", font=("", 16))
titre.grid(row=0, columnspan=2, pady=8)

nombre_secret = randint(0, 100) + 1

lbl_reponse = tk.Label(app, text="Choisi un nombre entre 1 et 100 inclus:")
lbl_reponse.grid(row=1, column=0, pady=5, padx=5)

reponse = tk.Entry(app)
reponse.grid(row=1, column=1, pady=5, padx=5)
reponse.bind("<Return>", nombre_choisi)
```

(continues on next page)

(continued from previous page)

```
proposition = tk.Label(app, text="")
proposition.grid(row=2, column=0, pady=5, padx=5)

resultat = tk.Label(app, text="")
resultat.grid(row=2, column=1, pady=5, padx=5)

app.mainloop()
```

4.5 Conclusion

Nous avons à présent vu les principales différences entre la structure d'un programme séquentiel et événementiel. Si vous parvenez à garder à l'esprit qu'il y a une boucle principale qui tourne et que tout ce que vous pouvez faire pour interagir avec votre programme c'est de répondre à des événements ou utiliser des compte à rebours, vous pourrez produire des programmes cohérents utilisant des interfaces graphiques, rendant le programme plus conviviale pour l'utilisateur.

La programmation orienté objet (POO) pourra aider également à mieux structurer le code.

CHAPTER

FIVE

COMMENT CONTRIBUER À PYTHON FAQ FR ?

On peut contribuer de toute sorte de manière à Python FAQ FR. Ça peut aller de la création d'un nouvel article à l'ajout d'une section ou à la correction d'un texte.

5.1 Cloner le dépôt

La première chose à faire est de cloner le dépôt. Il vous faut un compte sur Github si ce n'est pas déjà fait. Rendez-vous ensuite sur le dépôt de Python FAQ FR à l'adresse [https://github.com/dangillet/PythonFaqFr{]}(https://github.com/dangillet/PythonFaqFr) et cliquer en haut à droite sur l'icône **Fork**.

Depuis le _fork_ du dépôt, vous allez maintenant pouvoir le cloner. Vous trouverez le lien correct à utiliser en cliquant sur le bouton vert **Clone or download**. La commande à entrer dans votre terminal est :

```
` git clone https://github.com/username/PythonFaqFr.git `
```

en mettant l'adresse récupérée dans le bouton vert **Clone or download**.

Ceci créera dans le répertoire courant un dossier *PythonFaqFr*. Dedans se trouve un répertoire *doc* qui contient les articles.

5.2 Création d'un environnement virtuel

Afin d'avoir les bibliothèques requises pour visualiser la documentation offline, il est recommandé de créer un environnement virtuel. La méthode décrite ci-dessous utilise pipenv. Tout d'abord il faut l'installer :

```
` pip install pipenv `
```

Ensuite, depuis le dossier *PythonFaqFr* il suffit de faire :

```
` pipenv install `
```

pipenv va trouver le fichier *Pipfile* et installer les dépendances nécessaires dans un nouvel environement virtuel. Il faut à présent activer cet environnement virtuel en faisant :

```
` pipenv shell `
```

Vous êtes à présent dans l'environnement virtuel.

Pour construire la documentation, vous vous rendez dans le sous-dossier *doc* et vous faites :

```
` make html `
```

Un sous-dossier *_build* est créer contenant deux dossiers : *doctrees* et *html*. Dans le dossier *html* se trouve le fichier *index.html*. En ouvrant ce dernier avec votre browser préféré, vous pourrez voir la documentation offline.

Pour quitter l'environnement virtuel, il suffit de faire :

```
exit
```

5.3 Structure du projet

La documentation est générée grâce à [sphinx](http://www.sphinx-doc.org/en/master/). Toutes les pages de la documentation se trouvent dans le dossier *doc*. La page d'accueil se trouve dans le fichier *index.rst*. Ce dernier contient la table des matières.

Pour ajouter une nouvelle page, il suffit d'ajouter un nouveau fichier avec l'extension *.rst* pour utiliser le format [reStructuredText](http://docutils.sourceforge.net/rst.html) ou l'extension *.md* pour utiliser le format [Commonmark](http://commonmark.org/) (un type de Markdown).

Ensuite il faut référencer ce nouveau fichier depuis la table des matières contenue dans *index.rst*. Il suffit d'ajouter le nom du fichier, sans son extension, dans la table des matières appropriée. Il existe une table des matières par catégorie. Il est aisé d'ajouter une nouvelle catégorie si nécessaire.

5.4 Créer un _pull request_

Une fois le travail de rédaction ou de correction effectué, il faut créer un _pull **request**_ afin de l'ajouter au projet. Toutes vos modifications doivent d'abord être ajoutées à votre dépôt. Il ne s'agit pas ici de faire un tutoriel complet sur l'utilisation de Git, mais ces quelques commandes devraient suffire.

```
git add nom_fichier1 nom_fichier2
```

git add permet d'ajouter des fichiers modifiés. Il suffit de donner les noms des fichiers séparés d'un espace à cette commande. Ensuite il faut faire un _commit_, c'est à dire sauvegarder l'état des fichiers.

```
git commit -m "Votre message"
```

Vous écrivez un message décrivant succinctement les modifications apportées. Finalement vous devez poussez ces changement vers votre dépôt Github :

```
git push origin master
```

Vous devez à présent vous rendre sur la page de votre dépôt Github où se trouve le _fork_ de PythonFaqFr. Cliquez dans le menu en haut sur **Pull requests**. Sur cette page, cliquez sur _create a pull **request**_. Le système va vous présenter les différences entre votre dépôt et le dépôt d'origine. Vous pouvez changer le texte si nécessaire. Il suffit de cliquer sur le bouton _Create pull **request**_.