FICHE PYTHON 15 :
INTERFACE GRAPHIQUE 4 - ANIMATION (4 pages)

Vocabulaire

Une **variable** est l'association :	• d'un ensemble d'octets encodés • dans un identifiant mémoire (qu'on peut connaître avec `id(ma_variable)`. • avec un encodage dépendant du type de contenu à stocker (qu'on peut connaître avec `type(ma_variable)`.
Un **objet** est une entité complexe qui renferme à la fois :	• Des variables propres (internes) qu'on nomme **attributs.** Ex : `monObjet.attribut` permet d'y accéder dans certains cas. • Des fonctions propres (internes) qu'on nomme **méthodes**. Ex: `monObjet.methode()` permet de l'utiliser. Comme toutes les autres entités informatiques, un objet possède un **identifiant** qui permet de retrouver l'objet et d'agir sur lui. On peut le connaître avec `id(mon_objet)`. Pour connaître le type exact de l'objet, il faut utiliser `type(mon_objet)` .
Instanciation d'objet à partir d'une **Classe**	L'**instanciation** est le nom donné à la création d'un objet à partir d'un moule qu'on nomme une **Classe**. On génère cet objet à l'aide d'une méthode particulière, qu'on nomme un constructeur : cette méthode va créer l'objet et va renvoyer l'adresse de ce nouvel objet. Exemples d'instanciation : `fen_princ = Tk()` crée un objet-fenêtre Tk et stocke son adresse dans fen_princ. `monCanvas = Canvas(...)` crée un objet-Canvas et stocke son adresse dans monCanvas.

Item du Canvas	Dans le langage de Tkinter, les items sont des objets non indépendants : ils font partie intégrante de leur objet-conteneur à savoir le Canvas.
	`monCanvas.create_arc(50,50,150,150,fill="yellow",start=15,extent=330)`
	Cette ligne crée un arc mais sans stocker son adresse dans un identifiant.
	`arc1= monCanvas.create_arc(50,50,150,150,fill="yellow",start=15,extent=330)`
	Cette ligne crée le même item arc mais stocke son adresse dans un identifiant.
	On pourra donc utiliser le nom « arc1 » pour modifier ou supprimer l'arc pour la suite.

Item : Modification d'un item de Canvas à l'aide de itemconfig()

Méthode **itemconfig()**	Cette méthode est une méthode de la classe Canvas.
	Elle permet de modifier les items d'un Canvas après leurs créations.
	Son appel se fait donc à partir du nom du Canvas : `monCanvas.itemconfig(…)`
	On précise ensuite l'item sur lequel on veut agir : `monCanvas.itemconfig(pacman_1, ...)`
	On rajoute ensuite le ou les attributs à modifier ainsi que les nouvelles valeurs à appliquer.
	1- Création du Canvas :
	`monCanvas = Canvas(fen_princ, width=500, height=500 . . .)`
	2 – Création de l'item :
	`pacman_1 = monCanvas.create_arc(50,50,150,150,fill="yellow",start=15)`
	3 – Utilisation d'itemconfig :
	`monCanvas.itemconfig(pacman_1, fill='red')`

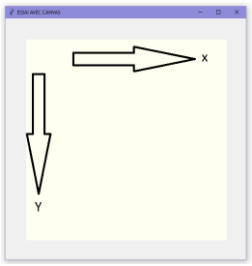
Item : Faire apparaître et disparaître un item avec itemconfig() et l'attribut state

Attribut **state**	Cet attribut des items peut être réglé sur :
	`monCanvas.itemconfig(pacman_1, state=NORMAL)` Valeur par défaut
	`monCanvas.itemconfig(pacman_1, state=HIDDEN)`
	`monCanvas.itemconfig(pacman_1, state=DISABLED)`

Item : Récupération de la valeur d'un attribut d'un item avec la méthode itemcget()

Méthode **itemcget()**	On peut récupérer les valeurs d'un item à l'aide d'une méthode de la classe Canvas. `couleur = monCanvas.itemcget(pacman_1,'fill')` Veut dire : • Applique la méthode **itemcget** sur l'objet-canvas **monCanvas**. • Cherche la valeur de **fill** pour l'item pacman_1. • Place le résultat dans la variable **couleur**.

Item : Déplacer un item

Méthode **move()** 	Cette méthode de Canvas permet de déplacer un objet par rapport à sa position actuelle dans le canvas. `monCanvas.move(item, dx, dy)` • monCanvas désigne le canvas sur lequel l'item est dessiné. • dx désigne le mouvement en pixels le long de l'axe x (positif à droite, négatif à gauche) • dy désigne le mouvement en pixels le long de l'axe y (positif en bas, négatif en haut).

Méthode **coords()**	Cette méthode de Canvas possède deux utilisations.
	Première utilisation : récupération des coordonnées dans le Canvas :
	`liste_coord = monCanvas.coords(pacman_1)`
	On obtient alors une liste contenant dans l'ordre les coordonnées de l'item.
	Si l'item est un rectangle, défini par x0, y0, x1, y1, on aurait :
	`liste_coord[0]` contient x0, `liste_coord[1]` contient y0,
	`liste_coord[2]` contient x1, `liste_coord[3]` contient y1.
	Deuxième utilisation : modifications des coordonnées dans le Canvas :
	`monCanvas.coords(pacman_1, 10,10,50,50)`
	Ces lignes vont imposer un changement de coordonnées pour l'item pacman_1 : x0 = 10, y0 = 10, x1=50 et y1=50.

Objet widget : Modification d'un widget à l'aide de config()

Méthode **config()**	Cette méthode est une méthode des classes widget.
	Elle permet de modifier les objets-widget après leurs créations.
	
	Son appel se fait donc à partir du nom du widget :
	`monButton.config(…)`
	On précise ensuite l'attribut sur lequel on veut agir en précisant la nouvelle valeur :
	`monButton.config(bg='blue')`

Objet widget : lire les attributs d'un widget à l'aide de cget()

Méthode **cget()**	C'est l'inverse de config : on ne fixe pas la valeur, on la récupère pour la lire.
	ATTENTION : la méthode renvoie un STRING. Pensez à convertir si vous voulez faire des calculs.
	
	Son appel se fait donc à partir du nom du widget :
	`monButton.config(…)`
	On précise ensuite l'attribut à lire en envoyant le nom de l'attribut dans un STRING :
	`monButton.config('bg')`

Gestion des collisions entre les items d'un Canvas

Méthode **find_overlapping()**	On peut trouver facilement la documentation de cette méthode de Canvas :
	
	find_overlapping(x1, y1, x2, y2) [#]
	Finds all items that overlap the given rectangle, or that are completely enclosed by it.
	x1 :Left edge.
	y1 : Upper edge.
	x2 : Right edge.
	y2 : Lower edge.
	Returns:
	A tuple containing all matching items.
	
	On doit donc fournir le nom du Canvas et donner les coordonnées du rectangle dans lequel on cherche à obtenir la liste des items présents.
	Si on donne les coordonnées d'un item lors de l'appel de la méthode, on obtient donc au moins l'item lui-même. Si on obtient deux retours, c'est qu'un deuxième item est présent : il a donc collision !
	Voici le code pour changer la couleur du pac-man en cas de collision :
	`# On place les coordonnées du pacman_1 dans la liste`
	`liste_coord = monCanvas.coords(pacman_1)`
	
	`# On place dans liste_items le tuple des items trouvés`
	`liste_items = monCanvas.find_overlapping( liste_coord[0], liste_coord[1], liste_coord[2], liste_coord[3])`
	
	`# On compte les items avec len pour détecter les collisions`
	`if len(liste_items) > 1:`
	`monCanvas.itemconfig(pacman_1, fill="red")`
	`else:`
	`monCanvas.itemconfig(pacman_1, fill="yellow")`

Méthode after() pour temporiser une action

Méthode **after()**	C'est une méthode de la classe Tk. Elle s'applique donc à un objet-fenêtre. `fen_princ.after(500,changement)` Veut dire qu'on attends 500 ms dans la fenêtre fen_princ et qu'on lance ensuite la fonction changement. On notera que la fonction changement() doit être définie si on ne veut pas créer d'erreur.

Récursivité	Cela consiste à lancer l'appel d'une fonction à l'intérieur de la fonction elle-même. La fonction tourne alors en boucle.
	`def modification():`
	`  if (monCanvas.itemcget(pacman_1, 'state')== HIDDEN):`
	`  monCanvas.itemconfig(pacman_1, state=NORMAL)`
	`  monCanvas.itemconfig(pacman_2, state=HIDDEN)`
	`  else:`
	`  monCanvas.itemconfig(pacman_2, state=NORMAL)`
	`  monCanvas.itemconfig(pacman_1, state=HIDDEN)`
	`  fen_princ.after(300,modification)`
	Cette fonction va donc faire clignoter le pac-man en alternant entre son allure 1 et son allure 2.

Variables globales et animation

Le problème des boutons qui activent une fonction récursive de déplacement est simple : si vous cliquez deux fois sur un bouton AVANCE_DE_10, le curseur va avancer de 20 pixels. On ne pourra jamais l'annuler. Il faut alors s'arranger pour annuler les mouvements avec, par exemple, des variables globales. Résumé d'une solution utilisant un **after()** :

On définit les variables suivantes :

- **vitesse** (initialement à 0, elle contiendra le nombre de pixels de déplacement)
- **stop_animation**, un booléen (True ou False) qui indique s'il faut utiliser **after()** pour continuer l'animation.
- **animation_active**, un booléen qui permet de savoir si on utilise déjà la fonction **after()**.

On utilise principalement deux fonctions :

- Une fonction **avance()** liée à un bouton qui fixe la valeur de **vitesse** et lance la fonction **modification().**
- Une fonction **modification()** dans laquelle on déplace l'item. On y trouve un **after()** récursif.
- Une fonction **arret()** qui permet d'arrêter l'animation indirectement en fixant **stop_animation** à True.
- RAPPEL : Pour que les fonctions puissent modifier les variables définies dans le programme principal, il faut les importer dans les fonctions à l'aide du mot-clé global.

Schéma global :

```
vitesse = 0
animation_stop = False
animation_active = False

...

Button(…command=avance)
Button(…command=arret)
```

```
def avance():
    global vitesse
    global stop_animation
    stop_animation = False
    vitesse =10
    if animation_active == False:
        modification()
```

```
def modification():
    global vitesse
    global animation_active
    global stop_animation
    animation_active = True
    monCanvas.move(pacman_1,vitesse,0)
    if stop_animation == False:
        fen_princ.after(100, modification)
    else:
        stop_animation = False
        animation_active = False
```

```
def arret():
    global stop_animation
    stop_animation = True
```