Série XXI : Premiers pas avec tkinter

Exercice XXI.0 : Jouons

Saisir depuis la console :

`from tkinter import *`	importer toutes les classes du module *tkinter*
`fen1 = Tk()`	*instanciation d'un objet à partir d'une classe* : Avec la *classe* **Tk()**, nous en créons une *instance* la fenêtre **fen1**
`tex1 = Label(fen1, text='Bonjour tout le monde !', fg='red')`	objet "*widget*" **tex1** créé par *instanciation* de la classe **Label()**
`tex1.pack()`	activer une *méthode* associée à l'objet **tex1**
`bou1 = Button(fen1, text='Quitter', command = fen1.destroy)`	création d'un autre widget « esclave » : un bouton.
`bou1.pack()`	méthode **pack()** pour adapter la géométrie de la fenêtre au nouvel objet
`fen1.mainloop()`	démarrage du *réceptionnaire d'événements* associé à la fenêtre

… et apprenons un peu de vocabulaire

REMARQUE XXI.0.A : **Programmes pilotés par des événements**

Dans les scripts réalisés jusque-là, les données entrées au clavier l'étaient toujours dans l'ordre prédéterminé correspondant à la séquence d'instructions du programme.

Dans le cas d'un programme qui utilise une interface graphique, par contre, l'organisation interne est différente. On dit d'un tel programme qu'il est *piloté par les événements*. Après sa phase d'initialisation, un programme de ce type se met en quelque sorte « en attente », et passe la main à un autre logiciel, lequel est plus ou moins intimement intégré au système d'exploitation de l'ordinateur et « tourne » en permanence.

Ce *réceptionnaire d'événements* scrute sans cesse tous les périphériques (clavier, souris, horloge, modem, etc.) et, lorsqu'un événement est détecté, envoie un message spécifique au programme, lequel doit être conçu pour réagir à chacun de ses messages, qui arrivent parfois en parallèle. Un tel événement peut être une action quelconque de l'utilisateur : déplacement de la souris, appui sur une touche, etc., mais aussi un événement externe ou un automatisme (top d'horloge, par exemple).

Exercice XXI.1 : Exemple graphique : tracé de lignes dans un canevas

Que fait le script ci-dessous ?

```
# Petit exercice utilisant la bibliothèque graphique tkinter

from tkinter import *
from random import randrange
# --- définition des fonctions gestionnaires d'événements : ---
def drawline():
      "Tracé d'une ligne dans le canevas can1"
      global x1, y1, x2, y2
      can1.create_line(x1,y1,x2,y2,width=2,fill=coul)

      # modification des coordonnées pour la ligne suivante :
      y2, y1 = y2+10, y1-10

def changecolor():
      "Changement aléatoire de la couleur du tracé"
      global coul
      pal=['purple','cyan','maroon','green','red','blue','orange','yellow']
      c = randrange(8) # => génère un nombre aléatoire de 0 à 7
      coul = pal[c]
```

```
#------ Programme principal -------

# les variables suivantes seront utilisées de manière globale :
x1, y1, x2, y2 = 10, 190, 190, 10 # coordonnées de la ligne
coul = 'dark green' # couleur de la ligne

# Création du widget principal ("maître") :
fen1 = Tk()

# Permet de positionner la fenêtre sur l'écran
xpos = 400
ypos = 300
fen1.wm_geometry("+%d+%d" % (xpos, ypos))

fen1.title("Tacer des lignes avec tkinter") # titre de la fenêtre

# création des widgets "esclaves" :
can1 = Canvas(fen1, bg='dark grey', height=200, width=200)
can1.pack(side=LEFT)
bou1 = Button(fen1, text='Quitter', command=fen1.quit)
bou1.pack(side=BOTTOM)
bou2 = Button(fen1, text='Tracer une ligne', command=drawline)
bou2.pack()
bou3 = Button(fen1, text='Autre couleur', command=changecolor)
bou3.pack()

fen1.mainloop() # démarrage du réceptionnaire d'événements
fen1.destroy() # destruction (fermeture) de la fenêtre
```

Exercice XXI.2 : **Plus que trois couleurs**

Comment faut-il modifier le programme pour ne plus avoir que des lignes de couleur "*cyan, maroon"* et "*green*" ?

Exercice XXI.3 : **Pas dans tous les sens !**

Comment modifier le programme pour que toutes les lignes tracées soient horizontales et parallèles ?

Exercice XXI.4 : **Agrandir**

Agrandissez le canevas de manière à lui donner une largeur de 500 unités et une hauteur de 650. Modifiez également la taille des lignes, afin que leurs extrémités se confondent avec les bords du canevas.

Exercice XXI.5 : **Viseur**

Ajoutez une fonction **drawline2** qui tracera deux lignes rouges en croix au centre du canevas : l'une horizontale et l'autre verticale.

Ajoutez également un bouton portant l'indication « viseur ». Un clic sur ce bouton devra provoquer l'affichage de la croix.

Exercice XXI.6 : **De la ligne à la figure plane**

Reprenez le programme initial. Remplacez la méthode **create_line** par **create_rectangle**. Que se passe-t-il ?

De la même façon, essayez aussi **create_arc**, **create_oval**, et **create_polygon**.

Pour chacune de ces méthodes, notez ce qu'indiquent les coordonnées fournies en paramètres.

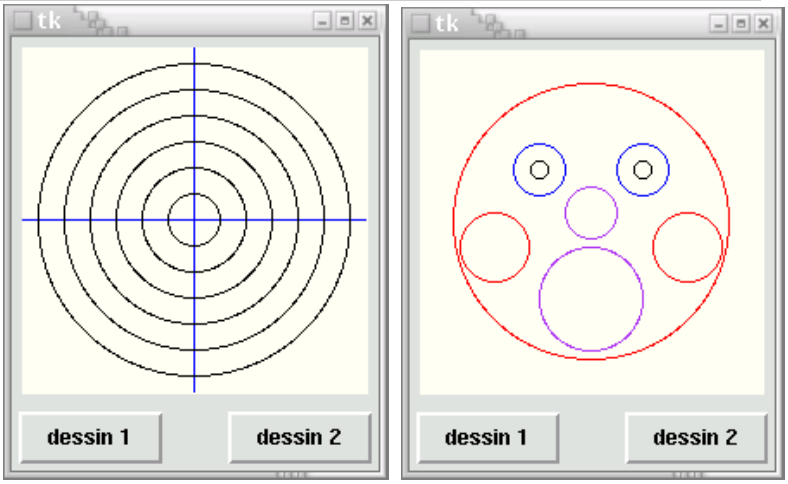
(Remarque : pour le polygone, il est nécessaire de modifier légèrement le programme !)

Exercice XXI.7 : **A quoi sert cette ligne ?**

- Supprimez la ligne **global x1, y1, x2, y2** dans la fonction **drawline** du programme original. Que se passe-t-il ? Pourquoi ?
- Si vous placez plutôt « x1, y1, x2, y2 » entre les parenthèses, dans la ligne de définition de la fonction **drawline**, de manière à transmettre ces variables à la fonction en tant que paramètres, le programme fonctionne-t-il encore ? N'oubliez pas de modifier aussi la ligne du programme qui fait appel à cette fonction !
- Si vous définissez **x1, y1, x2, y2 = 10, 390, 390, 10** à la place de **global x1, y1...**, que se passe-t-il ? Pourquoi ? Quelle conclusion pouvez-vous tirer de tout cela ?

Exercice XXI.8 : **Deux figures**

° Créez un programme qui dessinera l'un ou l'autre des deux dessins reproduits ci-dessous, en fonction du bouton choisi :



Le programme est disponible sous :
http://www.juggling.ch/gisin/python/python_code.html#ex_XXI08a_deux_dessins
Il se trouve aussi dans le corrigé.

Analyse du programme :

Commençons par analyser le programme principal, à la fin du script :

Nous y créons une fenêtre, par instanciation d'un objet de la classe **Tk()** dans la variable **fen**.
Ensuite, nous installons 3 widgets dans cette fenêtre : un canevas et deux boutons. Le canevas est instancié dans la variable **can**, et les deux boutons dans les variables **b1** et **b2**. Comme dans le script précédent, les widgets sont mis en place dans la fenêtre à l'aide de leur méthode **pack()**, mais cette fois nous utilisons celle-ci avec des options :

- l'option **side** peut accepter les valeurs "TOP, BOTTOM, LEFT ou RIGHT", pour « pousser » le widget du côté correspondant dans la fenêtre. Ces noms écrits en majuscules sont en fait ceux d'une série de variables importées avec le module tkinter, et que vous pouvez considérer comme des « pseudo-constantes ».
- les options **padx** et **pady** permettent de réserver un petit espace autour du widget. Cet espace est exprimé en nombre de pixels : **padx** réserve un espace à gauche et à droite du widget, **pady** réserve un espace au-dessus et au-dessous du widget.

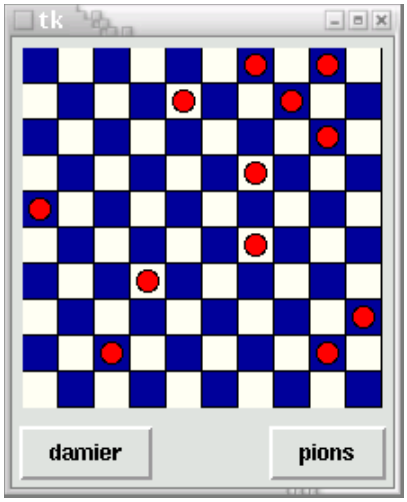
Les boutons commandent l'affichage des deux dessins, en invoquant les fonctions **figure_1()** et **figure_2()**. Considérant que nous aurions à tracer un certain nombre de cercles dans ces dessins, nous avons estimé qu'il serait bien utile de définir d'abord une fonction **cercle()** spécialisée. En effet, vous savez probablement déjà que le canevas tkinter est doté d'une méthode **create_oval()** qui permet de dessiner des ellipses quelconques (et donc aussi des cercles), mais cette méthode doit être invoquée avec quatre arguments qui seront les coordonnées des coins supérieur gauche et inférieur droit d'un rectangle fictif, dans lequel l'ellipse viendra alors s'inscrire. Cela n'est pas très pratique dans le cas particulier du cercle : il nous semblera plus naturel de commander ce tracé en fournissant les coordonnées de son centre ainsi que son rayon. C'est ce que nous obtiendrons avec notre fonction **cercle()**, laquelle invoque la méthode **create_oval()** en effectuant la conversion des coordonnées. Remarquez aussi que cette fonction attend un argument facultatif en ce qui concerne la couleur du cercle à tracer (noir par défaut).
L'efficacité de cette approche apparaît clairement dans la fonction **figure_1()**, ou nous trouvons une simple boucle de répétition pour dessiner toute la série de cercles (de même centre et de rayon croissant). Notez au passage l'utilisation de l'opérateur += qui permet d'incrémenter une variable (dans notre exemple, la variable "**rayon**" voit sa valeur augmenter de 15 unités à chaque itération).
Le second dessin est un peu plus complexe, parce qu'il est composé de cercles de tailles variées centrés sur des points différents. Nous pouvons tout de même tracer tous ces cercles à l'aide d'une seule boucle de répétition, si nous mettons à profit nos connaissances concernant les listes.
En effet, ce qui différencie les cercles que nous voulons tracer tient en quatre caractéristiques : coordonnées **x** et **y** du centre, rayon et couleur. Pour chaque cercle, nous pouvons placer ces quatre caractéristiques dans une petite liste, et rassembler toutes les petites listes ainsi obtenues dans une autre liste plus grande. Nous disposerons ainsi d'une liste de listes, qu'il suffira ensuite de parcourir à l'aide d'une boucle pour effectuer les tracés correspondants.

Exercice XXI.9 : **Olympique**

a) Créez un court programme qui dessinera les 5 anneaux olympiques dans un rectangle de fond blanc (*white*). Un bouton <Quitter> doit permettre de fermer la fenêtre.
b) Modifiez le programme ci-dessus en y ajoutant 5 boutons. Chacun de ces boutons provoquera le tracé de chacun des 5 anneaux.
Le code du programme b peut même être plus court que celui du programme a !

Exercice XXI.10 : **Deux dessins alternés**

Inspirez-vous de l'exemple ci-dessous pour écrire une petite application qui fait apparaître un damier (dessin de cases noires sur fond blanc) lorsque l'on clique sur un bouton :



Exercice XXI.11 : **Avec des pions**

À l'application de l'exercice précédent, ajoutez un bouton qui fera apparaître des pions au hasard sur le damier (chaque pression sur le bouton fera apparaître un nouveau pion).

Exercice XXI.12 : **calculatrice minimaliste**

Bien que très court, le petit script ci-dessous implémente une calculatrice complète, avec laquelle vous pourrez même effectuer des calculs comportant des parenthèses et des fonctions scientifiques. N'y voyez rien d'extraordinaire. Toute cette fonctionnalité n'est qu'une conséquence du fait que vous utilisez un interpréteur plutôt qu'un compilateur pour exécuter vos programmes.
Comme vous le savez, le compilateur n'intervient qu'une seule fois, pour traduire l'ensemble de votre code source en un programme exécutable. Son rôle est donc terminé *avant même* l'exécution du programme. L'interpréteur, quant à lui, est toujours actif *pendant* l'exécution du programme, et donc tout à fait disponible pour traduire un nouveau code source quelconque, comme une expression mathématique entrée au clavier par l'utilisateur.
Les langages interprétés disposent donc toujours de fonctions permettant d'évaluer une chaîne de caractères comme une suite d'instructions du langage lui-même. Il devient alors possible de construire en peu de lignes des structures de programmes très dynamiques. Dans l'exemple ci-dessous, nous utilisons la fonction intégrée **eval()** pour analyser l'expression mathématique entrée par l'utilisateur dans le champ prévu à cet effet, et nous n'avons plus ensuite qu'à afficher le résultat.

```
# Exercice utilisant la bibliothèque graphique tkinter et le module math

from tkinter import *
from math import *

# définition de l'action à effectuer si l'utilisateur actionne
# la touche "enter" alors qu'il édite le champ d'entrée :

def evaluer(event):
     chaine.configure(text = "Résultat = " + str(eval(entree.get())))

# ----- Programme principal : -----

fenetre = Tk()
entree = Entry(fenetre)
entree.bind("<Return>", evaluer)
chaine = Label(fenetre)
entree.pack()
chaine.pack()
fenetre.mainloop()
```

Et quelques explications :
Au début du script, nous commençons par importer les modules **tkinter** et **math**, ce dernier étant nécessaire afin que la dite calculatrice puisse disposer de toutes les fonctions mathématiques et scientifiques usuelles : sinus, cosinus, racine carrée, etc.
Ensuite nous définissons une fonction **evaluer()**, qui sera en fait la commande exécutée par le programme lorsque l'utilisateur actionnera la touche *Return* (ou *Enter*) après avoir entré une expression mathématique quelconque dans le champ d'entrée décrit plus loin.
Cette fonction utilise la méthode **configure()** du widget **chaine**, pour modifier son attribut **text**.
L'attribut en question reçoit donc ici une nouvelle valeur, déterminée par ce que nous avons écrit à la droite du signe égale : il s'agit en l'occurrence d'une chaîne de caractères construite dynamiquement, à l'aide de deux fonctions intégrées dans Python : **eval()** et **str()**, et d'une méthode associée à un widget *tkinter* : la méthode **get()**.
eval() fait appel à l'interpréteur pour évaluer une expression Python qui lui est transmise dans une chaîne de caractères. Le résultat de l'évaluation est fourni en retour. Exemple :

```
chaine = "(25 + 8)/3" # chaîne contenant une expression mathématique
res = eval(chaine) # évaluation de l'expression contenue dans la chaîne
print(res +5) # => le contenu de la variable res est numérique
```

str() transforme une expression numérique en chaîne de caractères. Nous devons faire appel à cette fonction parce que la précédente renvoie une valeur numérique, que nous convertissons à nouveau en chaîne de caractères pour pouvoir l'incorporer au message **Résultat =**.
get() est une méthode associée aux widgets de la classe **Entry**. Dans notre petit programme exemple, nous utilisons un widget de ce type pour permettre à l'utilisateur d'entrer une expression numérique quelconque à l'aide de son clavier. La méthode **get()** permet en quelque sorte « d'extraire » du widget **entree** la chaîne de caractères qui lui a été fournie par l'utilisateur.
Le corps du programme principal contient la phase d'initialisation, qui se termine par la mise en route de l'observateur d'événements (**mainloop**). On y trouve l'instanciation d'une fenêtre **Tk()**, contenant un widget **chaine** instancié à partir de la classe **Label()**, et un widget **entree** instancié à partir de la classe **Entry()**.
Attention : afin que ce dernier widget puisse vraiment faire son travail, c'est-à-dire transmettre au programme l'expression que l'utilisateur y aura encodée, nous lui associons un événement à l'aide de la méthode **bind()** :
entree.bind("<Return>", evaluer)
Cette instruction signifie : « *Lier l'événement <pression sur la touche Return> à l'objet <entree>, le gestionnaire de cet événement étant la fonction <evaluer>* ».
L'événement à prendre en charge est décrit dans une chaîne de caractères spécifique (dans notre exemple, il s'agit de la chaîne **"<Return>"**. Il existe un grand nombre de ces événements (mouvements et clics de la souris, enfoncement des touches du clavier, positionnement et redimensionnement des fenêtres, passage au premier plan, etc.). Vous trouverez la liste des chaînes spécifiques de tous ces événements dans les ouvrages de référence traitant de tkinter.
Remarquez bien qu'il n'y a pas de parenthèses après le nom de la fonction **evaluer**. En effet : dans cette instruction, nous ne souhaitons pas déjà invoquer la fonction elle-même (ce serait prématuré) ; ce que nous voulons, c'est établir un lien entre un type d'événement particulier et cette fonction, de manière à ce qu'elle soit invoquée plus tard, chaque fois que l'événement se produira. Si nous mettions des parenthèses, l'argument qui serait transmis à la méthode **bind()** serait la valeur de retour de cette fonction et non sa référence.
Profitons aussi de l'occasion pour observer encore une fois la syntaxe des instructions destinées à mettre en œuvre une méthode associée à un objet :
objet.méthode(arguments)
On écrit d'abord le nom de l'objet sur lequel on désire intervenir, puis le point (qui fait office d'opérateur), puis le nom de la méthode à mettre en œuvre ; entre les parenthèses associées à cette méthode, on indique enfin les arguments qu'on souhaite lui transmettre.

Exercice XXI.13 : **Détection et positionnement d'un clic de souris**

Dans la définition de la fonction « evaluer » de l'exemple précédent, vous aurez remarqué que nous avons fourni un argument **event** (entre les parenthèses).
Cet argument est obligatoire. Lorsque vous définissez une fonction gestionnaire d'événement qui est associée à un widget quelconque à l'aide de sa méthode **bind()**, vous devez toujours l'utiliser comme premier argument. Cet argument désigne en effet un objet créé automatiquement par tkinter, qui permet de transmettre au gestionnaire d'événement un certain nombre d'attributs de l'événement :

- le type d'événement : déplacement de la souris, enfoncement ou relâchement de l'un de ses boutons, appui sur une touche du clavier, entrée du curseur dans une zone prédéfinie, ouverture ou fermeture d'une fenêtre, etc.
- une série de propriétés de l'événement : l'instant où il s'est produit, ses coordonnées, les caractéristiques du ou des widget(s) concerné(s), etc.

Nous n'allons pas entrer dans trop de détails. Si vous voulez bien encoder et expérimenter le petit script ci-dessous, vous aurez vite compris le principe.

```
# Détection et positionnement d'un clic de souris dans une fenêtre :

from tkinter import *

def pointeur(event):
      chaine.configure( text = "Clic détecté en X =" + str(event.x) \
                        + ", Y =" + str(event.y) + ", num =" + str(event.num))

fen = Tk()
cadre = Frame(fen, width =200, height =150, bg="light yellow")
cadre.bind("<Button-1>", pointeur)
cadre.pack()
chaine = Label(fen)
chaine.pack()

fen.mainloop()
```

Le script fait apparaître une fenêtre contenant un *cadre* (**Frame**) rectangulaire de couleur jaune pâle, dans lequel l'utilisateur est invité à effectuer des clics de souris.
La méthode **bind()** du widget *cadre* associe l'événement <*clic à l'aide du premier bouton de la souris*> au gestionnaire d'événement « pointeur ».
Ce gestionnaire d'événement peut utiliser les attributs **x** et **y** de l'objet **event** généré automatiquement par tkinter, pour construire la chaîne de caractères qui affichera la position de la souris au moment du clic.

Exercice XXI.14 : **Un petit cercle ?**

Modifiez le script ci-dessus de manière à faire apparaître un petit cercle rouge à l'endroit où l'utilisateur a effectué son clic (vous devrez d'abord remplacer le widget **Frame** par un widget Canvas).

REMARQUE XXI.14.A : **Les classes de widgets tkinter**
Il existe 15 classes de base pour les widgets tkinter :

Widget	**Description**
Button	Un bouton classique, à utiliser pour provoquer l'exécution d'une commande quelconque.
Canvas	Un espace pour disposer divers éléments graphiques. Ce widget peut être utilisé pour dessiner, créer des éditeurs graphiques, et aussi pour implémenter des widgets personnalisés.
Checkbutton	Une case à cocher qui peut prendre deux états distincts (la case est cochée ou non). Un clic sur ce widget provoque le changement d'état.
Entry	Un champ d'entrée, dans lequel l'utilisateur du programme pourra insérer un texte quelconque à partir du clavier.
Frame	Une surface rectangulaire dans la fenêtre, où l'on peut disposer d'autres widgets. Cette surface peut être colorée. Elle peut aussi être décorée d'une bordure.
Label	Un texte (ou libellé) quelconque (éventuellement une image).
Listbox	Une liste de choix proposés à l'utilisateur, généralement présentés dans une sorte de boîte. On peut également configurer la Listbox de telle manière qu'elle se comporte comme une série de « boutons radio » ou de cases à cocher.
Menu	Un menu. Ce peut être un menu déroulant attaché à la barre de titre, ou bien un menu « *pop up* » apparaissant n'importe où à la suite d'un clic.
Menubutton	Un bouton-menu, à utiliser pour implémenter des menus déroulants.
Message	Permet d'afficher un texte. Ce widget est une variante du widget Label, qui permet d'adapter automatiquement le texte affiché à une certaine taille ou à un certain rapport largeur/hauteur.
Radiobutton	Représente (par un point noir dans un petit cercle) une des valeurs d'une variable qui peut en posséder plusieurs. Cliquer sur un bouton radio donne la valeur correspondante à la variable, et « vide » tous les autres boutons radio associés à la même variable.
Scale	Vous permet de faire varier de manière très visuelle la valeur d'une variable, en déplaçant un curseur le long d'une règle.
Scrollbar	Ascenseur ou barre de défilement que vous pouvez utiliser en association avec les autres widgets : Canvas, Entry, Listbox, Text.
Text	Affichage de texte formaté. Permet aussi à l'utilisateur d'éditer le texte affiché. Des images peuvent également être insérées.
Toplevel	Une fenêtre affichée séparément, au premier plan.

Ces classes de widgets intègrent chacune un grand nombre de méthodes. On peut aussi leur associer (lier) des événements, comme nous venons de le voir dans les pages précédentes. Vous allez apprendre en outre que tous ces widgets peuvent être positionnés dans les fenêtres à l'aide de trois méthodes différentes : la méthode **grid()**, la méthode **pack()** et la méthode **place()**.
L'utilité de ces méthodes apparaît clairement lorsque l'on s'efforce de réaliser des programmes portables (c'est-à-dire susceptibles de fonctionner de manière identique sur des systèmes d'exploitation aussi différents que *Unix*, *Mac OS* ou *Windows*), et dont les fenêtres soient *redimensionnables.*

Utilisation de la méthode grid pour contrôler la disposition des widgets

Jusqu'à présent, nous avons toujours disposé les widgets dans leur fenêtre à l'aide de la méthode **pack()**. Cette méthode présentait l'avantage d'être extraordinairement simple, mais elle ne nous donnait pas beaucoup de liberté pour disposer les widgets à notre guise.
Il est temps que nous apprenions à utiliser une autre approche du problème. Veuillez donc analyser le script ci-dessous : il contient en effet (presque) la solution :

```
from tkinter import *

fen1 = Tk()
txt1 = Label(fen1, text = 'Premier champ :')
txt2 = Label(fen1, text = 'Second :')
entr1 = Entry(fen1)
entr2 = Entry(fen1)
txt1.grid(row =0)
txt2.grid(row =1)
entr1.grid(row =0, column =1)
entr2.grid(row =1, column =1)
fen1.mainloop()
```

Dans ce script, nous avons donc remplacé la méthode **pack()** par la méthode **grid()**. Comme vous pouvez le constater, l'utilisation de la méthode **grid()** est très simple. Cette méthode considère la fenêtre comme un tableau (ou une grille). Il suffit alors de lui indiquer dans quelle ligne (**row**) et dans quelle colonne (**column**) de ce tableau on souhaite placer les widgets. On peut numéroter les lignes et les colonnes comme on veut, en partant de zéro, ou de un, ou encore d'un nombre quelconque : tkinter ignorera les lignes et colonnes vides. Notez cependant que si vous ne fournissez aucun numéro pour une ligne ou une colonne, la valeur par défaut sera zéro.
Tkinter détermine automatiquement le nombre de lignes et de colonnes nécessaire. Mais ce n'est pas tout : si vous examinez en détail la petite fenêtre produite par le script ci-dessus, vous constaterez que nous n'avons pas encore tout à fait atteint le but poursuivi. Les deux chaînes apparaissant dans la partie gauche de la fenêtre sont *centrées*, alors que nous souhaitions les *aligner* l'une et l'autre par la droite. Pour obtenir ce résultat, il nous suffit d'ajouter un argument dans l'appel de la méthode **grid()** utilisée pour ces widgets. L'option **sticky** peut prendre l'une des quatre valeurs **N, S, W, E** (les quatre points cardinaux en anglais). En fonction de cette valeur, on obtiendra un alignement des widgets par le haut, par le bas, par la gauche ou par la droite. Remplacez donc les deux premières instructions **grid()** du script par :
txt1.grid(row =0, sticky =E)
txt2.grid(row =1, sticky =E)
... et vous atteindrez enfin exactement le but recherché.
Analysons à présent la fenêtre suivante :

Cette fenêtre comporte 3 colonnes : une première avec les 3 chaînes de caractères, une seconde avec les 3 champs d'entrée, et une troisième avec l'image. Les deux premières colonnes comportent chacune 3 lignes, mais l'image située dans la dernière colonne *s'étale* en quelque sorte sur les trois.
Le code correspondant est le suivant :

```
from tkinter import *

fen1 = Tk()

# création de widgets 'Label' et 'Entry' :
txt1 = Label(fen1, text ='Premier champ :')
txt2 = Label(fen1, text ='Second :')
txt3 = Label(fen1, text ='Troisième :')
entr1 = Entry(fen1)
entr2 = Entry(fen1)
entr3 = Entry(fen1)

# création d'un widget 'Canvas' contenant une image bitmap :
can1 = Canvas(fen1, width =160, height =160, bg ='white')
photo = PhotoImage(file ='martin_p.png')
item = can1.create_image(80, 80, image =photo)

# Mise en page à l'aide de la méthode 'grid' :
txt1.grid(row =1, sticky =E)
txt2.grid(row =2, sticky =E)
txt3.grid(row =3, sticky =E)
entr1.grid(row =1, column =2)
entr2.grid(row =2, column =2)
entr3.grid(row =3, column =2)
can1.grid(row =1, column =3, rowspan =3, padx =10, pady =5)

# démarrage :
fen1.mainloop()
```

Pour pouvoir faire fonctionner ce script, il vous faudra probablement remplacer le nom du fichier image (*martin_p.gif*) par le nom d'une image de votre choix. Attention : la bibliothèque tkinter standard n'accepte qu'un petit nombre de formats pour cette image (.png ; .jpg ; .gif). Choisissez de préférence le format PNG.
Nous pouvons remarquer un certain nombre de choses dans ce script :

1. La technique utilisée pour incorporer une image :
   tkinter ne permet pas d'insérer directement une image dans une fenêtre. Il faut d'abord installer un canevas, et ensuite positionner l'image dans celui-ci. Nous avons opté pour un canevas de couleur blanche, afin de pouvoir le distinguer de la fenêtre. Vous pouvez remplacer le paramètre **bg ='white'** par **bg ='gray'** si vous souhaitez que le canevas devienne invisible.
   Étant donné qu'il existe de nombreux types d'images, nous devons en outre déclarer l'objet image comme étant un bitmap PNG, à partir de la classe **PhotoImage()**.
2. La ligne où nous installons l'image dans le canevas est la ligne :
   item = can1.create_image(80, 80, image =photo)
   Pour employer un vocabulaire correct, nous dirons que nous utilisons ici la méthode **create_image()** associée à l'objet **can1** (lequel objet est lui-même une instance de la classe **Canvas**). Les deux premiers arguments transmis (**80**, **80**) indiquent les coordonnées **x** et **y** du canevas où il faut placer le centre de l'image. Les dimensions du canevas étant de 160x160, notre choix aboutira donc à un centrage de l'image au milieu du canevas.

3. La numérotation des lignes et colonnes dans la méthode **grid()** :
   On peut constater que la numérotation des lignes et des colonnes dans la méthode **grid()** utilisée ici commence cette fois à partir de 1 (et non à partir de zéro comme dans le script précédent). Comme nous l'avons déjà signalé plus haut, ce choix de numérotation est tout à fait libre.
   On pourrait tout aussi bien numéroter : 5, 10, 15, 20... puisque tkinter ignore les lignes et les colonnes vides. Numéroter à partir de l augmente probablement la lisibilité de notre code.
4. Les arguments utilisés avec **grid()** pour positionner le canevas :
   can1.grid(row =1, column =3, rowspan =3, padx =10, pady =5)
   Les deux premiers arguments indiquent que le canevas sera placé dans la première ligne de la troisième colonne. Le troisième (**rowspan =3**) indique qu'il pourra « s'étaler » sur trois lignes. Les deux derniers (**padx =10, pady =5**) indiquent la dimension de l'espace qu'il faut réserver autour de ce widget (en largeur et en hauteur).
5. Et tant que nous y sommes, profitons de cet exemple de script, que nous avons déjà bien décortiqué, pour apprendre à simplifier quelque peu notre code…

REMARQUE XXI.14.B : **Composition d'instructions pour écrire un code plus compact**
Python étant un langage de programmation de haut niveau, il est souvent possible (et souhaitable) de retravailler un script afin de le rendre plus compact.
Vous pouvez par exemple assez fréquemment utiliser la composition d'instructions pour appliquer la méthode de mise en page des widgets (**grid()**, **pack()** ou **place()**) au moment même où vous créez ces widgets. Le code correspondant devient alors un peu plus simple, et parfois plus lisible. Vous pouvez par exemple remplacer les deux lignes :

txt1 = Label(fen1, text ='Premier champ :')
txt1.grid(row =1, sticky =E)
du script précédent par une seule, telle que :
Label(fen1, text ='Premier champ :').grid(row =1, sticky =E)
Dans cette nouvelle écriture, vous pouvez constater que nous faisons l'économie de la variable intermédiaire **txt1**. Nous avions utilisé cette variable pour bien dégager les étapes successives de notre démarche, mais elle n'est pas toujours indispensable. Le simple fait d'invoquer la classe **Label()** provoque en effet l'instanciation d'un objet de cette classe, même si l'on ne mémorise pas la référence de cet objet dans une variable (tkinter la conserve de toute façon dans sa représentation interne de la fenêtre). Si l'on procède ainsi, la référence est perdue pour le restant du script, mais elle peut tout de même être transmise à une méthode de mise en page telle que **grid()** au moment même de l'instanciation, en une seule instruction composée. Voyons cela un peu plus en détail.
Jusqu'à présent, nous avons créé des objets divers (par instanciation à partir d'une classe quelconque), en les affectant à chaque fois à des variables. Par exemple, lorsque nous avons écrit :
txt1 = Label(fen1, text ='Premier champ :')
nous avons créé une instance de la classe **Label()**, que nous avons assignée à la variable **txt1.**
La variable **txt1** peut alors être utilisée pour faire référence à cette instance, partout ailleurs dans le script, mais dans les faits nous ne l'utilisons qu'une seule fois pour lui appliquer la méthode **grid()**, le widget dont il est question n'étant rien d'autre qu'une simple étiquette descriptive. Or, créer ainsi une nouvelle variable pour n'y faire référence ensuite qu'une seule fois (et directement après sa création) n'est pas une pratique très recommandable, puisqu'elle consiste à réserver inutilement un certain espace mémoire.
Lorsque ce genre de situation se présente, il est plus judicieux d'utiliser la composition d'instructions. Par exemple, on préférera le plus souvent remplacer les deux instructions :
somme = 45 + 72
print (somme)
par une seule instruction composée, telle que :
print (45 + 72)
on fait ainsi l'économie d'une variable.

De la même manière, lorsque l'on met en place des widgets auxquels on ne souhaite plus revenir par la suite, comme c'est souvent le cas pour les widgets de la classe **Label()**, on peut en général appliquer la méthode de mise en page (**grid()** , **pack()** ou **place()**) directement au moment de la création du widget, en une seule instruction composée.
Cela s'applique seulement aux widgets qui ne sont plus référencés après qu'on les ait créés. *Tous les autres doivent impérativement être assignés à des variables, afin que l'on puisse encore interagir avec eux ailleurs dans le script.*
Et dans ce cas, il faut obligatoirement utiliser deux instructions distinctes, l'une pour instancier le widget, et l'autre pour lui appliquer ensuite la méthode de mise en page. Vous ne pouvez pas, par exemple, construire une instruction composée telle que :
entree = Entry(fen1).pack() # ***faute de programmation !!!***
En apparence, cette instruction devrait instancier un nouveau widget et l'assigner à la variable **entree**, la mise en page s'effectuant dans la même opération à l'aide de la méthode **pack()**.
Dans la réalité, cette instruction produit bel et bien un nouveau widget de la classe **Entry()**, et la méthode **pack()** effectue bel et bien sa mise en page dans la fenêtre, mais la valeur qui est mémorisée dans la variable **entree** n'est pas la référence du widget ! C'est *la valeur de retour de la méthode* **pack()** : vous devez vous rappeler en effet que les méthodes, comme les fonctions, renvoient toujours une valeur au programme qui les appelle. Et vous ne pouvez rien faire de cette valeur de retour : il s'agit en l'occurrence d'un objet vide (**None**).
Pour obtenir une vraie référence du widget, vous devez obligatoirement utiliser deux instructions :
entree = Entry(fen1) # ***instanciation du widget***
entree.pack() # ***application de la mise en page***

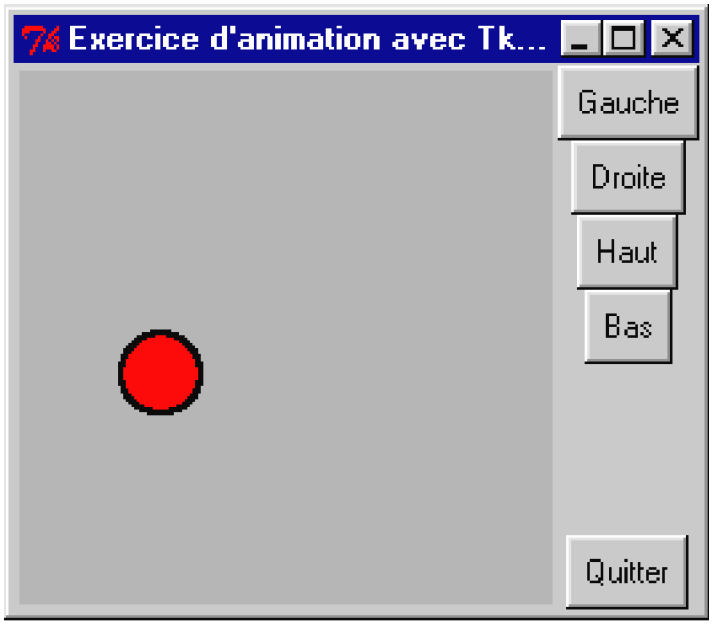
Lorsque vous utilisez la méthode ***grid()****, vous pouvez simplifier encore un peu votre code, en omettant l'indication de nombreux numéros de lignes et de colonnes. À partir du moment où c'est la méthode* ***grid()*** *qui est utilisée pour positionner les widgets, tkinter considère en effet qu'il existe forcément des lignes et des colonnes*[14]*. Si un numéro de ligne ou de colonne n'est pas indiqué, le widget correspondant est placé dans la première case vide disponible.*

Le script ci-dessous intègre les simplifications que nous venons d'expliquer :

```
from tkinter import *
fen1 = Tk()

# création de widgets Label(), Entry(), et Checkbutton() :
Label(fen1, text = 'Premier champ :').grid(sticky =E)
Label(fen1, text = 'Deuxième :').grid(sticky =E)
Label(fen1, text = 'Troisième :').grid(sticky =E)
entr1 = Entry(fen1)
entr2 = Entry(fen1) # ces widgets devront certainement
entr3 = Entry(fen1) # être référencés plus loin :
entr1.grid(row =0, column =1) # il faut donc les assigner chacun
entr2.grid(row =1, column =1) # à une variable distincte
entr3.grid(row =2, column =1)
chek1 = Checkbutton(fen1, text ='Case à cocher, pour voir')
chek1.grid(columnspan =2)

# création d'un widget 'Canvas' contenant une image bitmap :
can1 = Canvas(fen1, width =160, height =160, bg ='white')
photo = PhotoImage(file ='Martin_P.png')
can1.create_image(80,80, image =photo)
can1.grid(row =0, column =2, rowspan =4, padx =10, pady =5)

# démarrage :
fen1.mainloop()
```

Modification des propriétés d'un objet – Animation

À ce stade de votre apprentissage, vous souhaitez probablement pouvoir faire apparaître un petit dessin quelconque dans un canevas, et puis le déplacer à volonté, par exemple à l'aide de boutons.
Veuillez donc écrire, tester, puis analyser le script ci-dessous :




```
from tkinter import *

# procédure générale de déplacement :
def avance(gd, hb):
    global x1, y1
    x1, y1 = x1 +gd, y1 +hb
    can1.coords(oval1, x1, y1,
    x1+30, y1+30)

# gestionnaires d'événements :
def depl_gauche():
    avance(-10, 0)

def depl_droite():
    avance(10, 0)

def depl_haut():
    avance(0, -10)

def depl_bas():
    avance(0, 10)

#------ Programme principal -------

# les variables suivantes seront utilisées de manière globale :
x1, y1 = 10, 10 # coordonnées initiales

# Création du widget principal ("maître") :
fen1 = Tk()
fen1.title("Exercice d'animation avec tkinter")

# création des widgets "esclaves" :
can1 = Canvas(fen1,bg='dark grey',height=300,width=300)
oval1 = can1.create_oval(x1,y1,x1+30,y1+30,width=2,fill='red')
can1.pack(side=LEFT)
Button(fen1,text='Quitter',command=fen1.destroy).pack(side=BOTTOM)
Button(fen1,text='Gauche',command=depl_gauche).pack()
Button(fen1,text='Droite',command=depl_droite).pack()
Button(fen1,text='Haut',command=depl_haut).pack()
Button(fen1,text='Bas',command=depl_bas).pack()

# démarrage du réceptionnaire d'évènements (boucle principale) :
fen1.mainloop()
```

Le corps de ce programme reprend de nombreux éléments connus : nous y créons une fenêtre **fen1**, dans laquelle nous installons un canevas contenant lui-même un cercle coloré, plus cinq boutons de contrôle. Veuillez remarquer au passage que nous n'instancions pas les widgets boutons dans des variables (c'est inutile, puisque nous n'y faisons plus référence par la suite) : nous devons donc appliquer la méthode **pack()** directement au moment de la création de ces objets.
La vraie nouveauté de ce programme réside dans la fonction **avance()** définie au début du script. Chaque fois qu'elle sera appelée, cette fonction redéfinira les coordonnées de l'objet « *oval1* » que nous avons installé dans le canevas, ce qui provoquera l'animation de cet objet.

Cette manière de procéder est tout à fait caractéristique de la programmation « orientée objet » : on commence par créer des objets, puis *on agit sur ces objets en modifiant leurs propriétés, par l'intermédiaire de méthodes*.
En programmation impérative « à l'ancienne » (c'est-à-dire sans utilisation d'objets), on anime des figures en les effaçant à un endroit pour les redessiner ensuite un petit peu plus loin. En programmation « orientée objet », par contre, ces tâches sont prises en charge automatiquement par les classes dont les objets dérivent, et il ne faut donc pas perdre son temps à les reprogrammer.

Exercice XXI.15 : **Deux astres**
Écrivez un programme qui fait apparaître une fenêtre avec un canevas. Dans ce canevas on verra deux cercles (de tailles et de couleurs différentes), qui sont censés représenter deux astres. Des boutons doivent permettre de les déplacer à volonté tous les deux dans toutes les directions. Sous le canevas, le programme doit afficher en permanence :
a) la distance séparant les deux astres;
b) la force gravitationnelle qu'ils exercent l'un sur l'autre (penser à afficher en haut de fenêtre les masses choisies pour chacun d'eux, ainsi que l'échelle des distances). Dans cet exercice, vous utiliserez évidemment la loi de la gravitation universelle de Newton : $F_G = 6{,}67 \cdot 10^{-11} \cdot \frac{m_1 \cdot m_2}{d^2}$

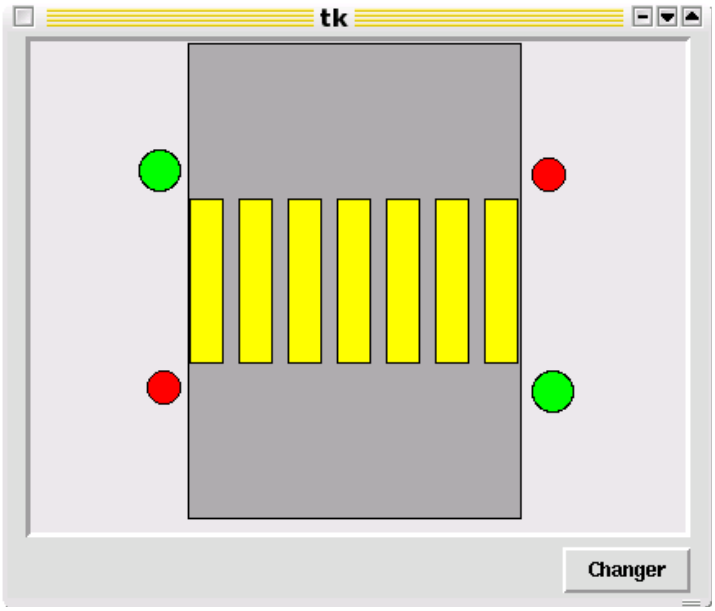
Exercice XXI.16 : **Moins de boutons**
En vous inspirant du programme qui détecte les clics de souris dans un canevas, modifiez le programme ci-dessus pour y réduire le nombre de boutons : pour déplacer un astre, il suffira de le choisir avec un bouton, et ensuite de cliquer sur le canevas pour que cet astre se positionne à l'endroit où l'on a cliqué.
Exercice XXI.17 : **Trois astres**
Extension du programme ci-dessus. Faire apparaître un troisième astre, et afficher en permanence la force résultante agissant sur chacun des trois (en effet : chacun subit en permanence l'attraction gravitationnelle exercée par les deux autres !).

Exercice XXI.18 : **Des charges électriques**
Même exercice avec des charges électriques (loi de Coulomb). Donner cette fois une possibilité de choisir le signe des charges.
Exercice XXI.19 : **Celsius "de" et "vers" Fahrenheit**
Écrivez un petit programme qui fait apparaître une fenêtre avec deux champs : l'un indique une température en degrés *Celsius*, et l'autre la même température exprimée en degrés *Fahrenheit*. Chaque fois que l'on change une quelconque des deux températures, l'autre est corrigée en conséquence. Pour convertir les degrés *Fahrenheit* en *Celsius* et vice-versa, on utilise la formule :
Fahrenheit = 1.8 * *Celsius* + 32.

Exercice XXI.20 : **Une balle**
Écrivez un programme qui fait apparaître une fenêtre avec un canevas. Dans ce canevas, placez un petit cercle censé représenter une balle. Sous le canevas, placez un bouton. Chaque fois que l'on clique sur le bouton, la balle doit avancer d'une petite distance vers la droite, jusqu'à ce qu'elle atteigne l'extrémité du canevas. Si l'on continue à cliquer, la balle doit alors revenir en arrière jusqu'à l'autre extrémité, et ainsi de suite.
Exercice XXI.21 : **Une balle au mouvement circulaire**
Améliorez le programme ci-dessus pour que la balle décrive cette fois une trajectoire circulaire ou elliptique dans le canevas (lorsque l'on clique continuellement). Note : pour arriver au résultat escompté, vous devrez nécessairement définir une variable qui représentera l'angle décrit, et utiliser les fonctions *sinus* et *cosinus* pour positionner la balle en fonction de cet angle.
Exercice XXI.22 : **Avec une jolie trace**
Modifiez le programme ci-dessus de telle manière que la balle, en se déplaçant, laisse derrière elle une trace de la trajectoire décrite.

Exercice XXI.23 : **Feux de circulation**

Écrivez un programme qui fait apparaître une fenêtre avec un canevas et un bouton. Dans le canevas, tracez un rectangle gris foncé, lequel représentera une route, et par-dessus, une série de rectangles jaunes censés représenter un passage pour piétons. Ajoutez quatre cercles colorés pour figurer les feux de circulation concernant les piétons et les véhicules. Chaque utilisation du bouton devra provoquer le changement de couleur des feux :



Exercice XXI.24 : **Circuit électrique**

Écrivez un programme qui montre un canevas dans lequel est dessiné un circuit électrique simple (générateur + interrupteur + résistance). La fenêtre doit être pourvue de champs d'entrée qui permettront de paramétrer chaque élément (c'est-à-dire choisir les valeurs des résistances et tensions). L'interrupteur doit être fonctionnel (prévoyez un bouton <Marche/arrêt> pour cela). Des « étiquettes » doivent afficher en permanence les tensions et intensités résultant des choix effectués par l'utilisateur.

Exercice XXI.25 : **Animation automatique**

Pour conclure cette première prise de contact avec l'interface graphique tkinter, voici un dernier exemple d'animation, qui fonctionne cette fois de manière autonome dès qu'on l'a mise en marche.

```
from tkinter import *

# définition des gestionnaires d'événements :

def move():
    "déplacement de la balle"
    global x1, y1, dx, dy, flag
    x1, y1 = x1 +dx, y1 + dy
    if x1 >210:
        x1, dx, dy = 210, 0, 15
    if y1 >210:
        y1, dx, dy = 210, -15, 0
    if x1 <10:
        x1, dx, dy = 10, 0, -15
    if y1 <10:
        y1, dx, dy = 10, 15, 0
    can1.coords(oval1,x1,y1,x1+30,y1+30)
    if flag >0:
        fen1.after(50, move) # Demande au système d'appeler la fonction "move" après 50 millisecondes

def stop_it():
    "arrêt de l'animation"
    global flag
    flag =0

def start_it():
    "démarrage de l'animation"
    global flag
    if flag ==0: # pour ne lancer qu'une seule boucle
        flag =1
        move()  # que se passe-t-il si "move()" est désindenté ?
```

```
#========== Programme principal =============

# les variables suivantes seront utilisées de manière globale :
x1, y1 = 10, 10 # coordonnées initiales
dx, dy = 15, 0 # 'pas' du déplacement
flag =0 # commutateur

# Création du widget principal ("parent") :
fen1 = Tk()
fen1.title("Exercice d'animation avec tkinter")
# création des widgets "enfants" :
can1 = Canvas(fen1,bg='dark grey',height=250, width=250)
can1.pack(side=LEFT, padx =5, pady =5)
oval1 = can1.create_oval(x1, y1, x1+30, y1+30, width=2, fill='red')
bou1 = Button(fen1,text='Quitter', width =8, command=fen1.destroy)
bou1.pack(side=BOTTOM)
bou2 = Button(fen1, text='Démarrer', width =8, command=start_it)
bou2.pack()
bou3 = Button(fen1, text='Arrêter', width =8, command=stop_it)
bou3.pack()
# démarrage du réceptionnaire d'événements (boucle principale) :
fen1.mainloop()
```

La seule nouveauté mise en œuvre dans ce script se trouve tout à la fin de la définition de la fonction **move()** : vous y noterez l'utilisation de la méthode **after()**. Cette méthode peut s'appliquer à un widget quelconque. Elle déclenche l'appel d'une fonction *après qu'un certain laps de temps se soit écoulé*. Ainsi par exemple, **window.after(200, qqc)** déclenche pour le widget **window** un appel de la fonction **qqc()** après une pause de 200 millisecondes.
Dans notre script, la fonction qui est appelée par la méthode **after()** est la fonction **move()** elle-même. La fonction **move()** est invoquée une première fois lorsque l'on clique sur le bouton <Démarrer>. Elle effectue son travail (c'est-à-dire positionner la balle). Par l'intermédiaire de la méthode **after()**, elle indique au système de l'appeler elle-même après une pause de 50 millisecondes. Elle repart donc pour un second tour. Elle sera donc appelée ainsi régulièrement par le système.
C'est du moins ce qui se passerait si nous n'avions pas pris la précaution de placer quelque part dans la boucle une instruction d'arrêt d'appel. En l'occurrence, il s'agit d'un simple test conditionnel : à chaque itération de la boucle, nous examinons le contenu de la variable **flag** à l'aide d'une instruction **if**. Si le contenu de la variable **flag** est zéro, alors la demande au système d'appeler la fonction "move" n'est plus faite et l'animation s'arrête. Remarquez que nous avons pris la précaution de définir **flag** comme une variable globale. Ainsi nous pouvons aisément changer sa valeur à l'aide d'autres fonctions, en l'occurrence celles que nous avons associées aux boutons <Démarrer> et <Arrêter>.
Nous obtenons ainsi un mécanisme simple pour lancer ou arrêter notre animation : un premier clic sur le bouton <Démarrer> assigne une valeur non-nulle à la variable **flag**, puis provoque immédiatement un premier appel de la fonction **move()**. Celle-ci s'exécute, puis continue à s'appeler elle-même toutes les 50 millisecondes, tant que **flag** ne revient pas à zéro.
Si l'on continue à cliquer sur le bouton <Démarrer>, la fonction **move()** ne peut plus être appelée, parce que la valeur de **flag** vaut désormais 1. On évite ainsi le démarrage de plusieurs boucles concurrentes.
Le bouton <Arrêter> remet **flag** à zéro, et la boucle s'interrompt.

Exercice XXI.26 : **Start_it()**
Dans la fonction **start_it()**, supprimez l'instruction **if flag == 0:** (et l'indentation des deux lignes suivantes). Que se passe-t-il ? (Cliquez plusieurs fois sur le bouton <Démarrer>.)
Tâchez d'exprimer le plus clairement possible votre explication des faits observés.

Exercice XXI.27 : **Changement de couleur**
Modifiez le programme de telle façon que la balle change de couleur à chaque « virage ».

Exercice XXI.28 : **Zig-zag**
Modifiez le programme de telle façon que la balle effectue des mouvements obliques comme une bille de billard qui rebondit sur les bandes (« en zig-zag »).

Exercice XXI.29 : **Circulaire**
Modifiez le programme de manière à obtenir d'autres mouvements. Tâchez par exemple d'obtenir un mouvement circulaire.

Exercice XXI.30 : **Changements de vitesse**
Modifiez ce programme, ou bien écrivez-en un autre similaire, de manière à simuler le mouvement d'une balle qui tombe (sous l'effet de la pesanteur), et rebondit sur le sol. Attention : il s'agit cette fois de mouvements accélérés !

Exercice XXI.31 : **Un jeu de balle**
À partir des scripts précédents, vous pouvez à présent écrire un programme de jeu fonctionnant de la manière suivante : une balle se déplace au hasard sur un canevas, à vitesse faible. Le joueur doit essayer de cliquer sur cette balle à l'aide de la souris. S'il y arrive, il gagne un point, mais la balle se déplace désormais un peu plus vite, et ainsi de suite. Arrêter le jeu après un certain nombre de clics et afficher le score atteint.

Exercice XXI.32 : **La balle rétrécis**
Variante du jeu précédent : chaque fois que le joueur parvient à « l'attraper », la balle devient plus petite (elle peut également changer de couleur).

Exercice XXI.33 : **Plusieurs balles**
Écrivez un programme dans lequel évoluent plusieurs balles de couleurs différentes, qui rebondissent les unes sur les autres ainsi que sur les parois.

Exercice XXI.34 : **Une seule balle à clic**
Perfectionnez le jeu des précédents exercices en y intégrant l'algorithme ci-dessus. Il s'agit à présent pour le joueur de cliquer seulement sur la balle rouge. Un clic erroné (sur une balle d'une autre couleur) lui fait perdre des points.

Exercice XXI.35 : **Deux planètes**
Écrivez un programme qui simule le mouvement de deux planètes tournant autour du soleil sur des orbites circulaires différentes (ou deux électrons tournant autour d'un noyau d'atome...).

Exercice XXI.36 : **Jeu du serpent**
Écrivez un programme pour le jeu du serpent : un « serpent » (constitué en fait d'une courte ligne de carrés) se déplace sur le canevas dans l'une des 4 directions : droite, gauche, haut, bas. Le joueur peut à tout moment changer la direction suivie par le serpent à l'aide des touches fléchées du clavier. Sur le canevas se trouvent également des « proies » (des petits cercles fixes disposés au hasard). Il faut diriger le serpent de manière à ce qu'il « mange » les proies sans arriver en contact avec les bords du canevas. À chaque fois qu'une proie est mangée, le serpent s'allonge d'un carré, le joueur gagne un point, et une nouvelle proie apparaît ailleurs. La partie s'arrête lorsque le serpent touche l'une des parois, ou lorsqu'il a atteint une certaine taille.

Exercice XXI.37 : **Le serpent ne peut pas se croiser**
Perfectionnement du jeu précédent : la partie s'arrête également si le serpent « se recoupe ».

Corrigés :

Exercice XXI.1 : **Exemple graphique : tracé de lignes dans un canevas**
Explication du code du script de l'exercice XXI.1

Le script crée une fenêtre comportant trois boutons et un *canevas*. Suivant la terminologie de *tkinter*, un *canevas* est une surface rectangulaire délimitée, dans laquelle on peut installer ensuite divers dessins et images à l'aide de méthodes spécifiques
Lorsque l'on clique sur le bouton <Tracer une ligne>, une nouvelle ligne colorée apparaît sur le canevas, avec à chaque fois une inclinaison différente de la précédente.
Si l'on actionne le bouton <Autre couleur>, une nouvelle couleur est tirée au hasard dans une série limitée. Cette couleur est celle qui s'appliquera aux tracés suivants.
Le bouton <Quitter> sert bien évidemment à terminer l'application en refermant la fenêtre.

Ou encore pour aller plus loin :

La fonctionnalité de ce programme est essentiellement assurée par les deux fonctions **drawline()** et **changecolor()**, qui seront activées par des événements, ceux-ci étant eux-mêmes définis dans la phase d'initialisation.
Dans cette phase d'initialisation, on commence par importer l'intégralité du module tkinter ainsi qu'une fonction du module *random* qui permet de tirer des nombres au hasard. On crée ensuite les différents widgets par instanciation à partir des classes **Tk()**, **Canvas()** et **Button()**. Remarquons au passage que la même classe **Button()** sert à instancier plusieurs boutons, qui sont des objets similaires pour l'essentiel, mais néanmoins individualisés grâce aux options de création et qui pourront fonctionner indépendamment l'un de l'autre.
L'initialisation se termine avec l'instruction **fen1.mainloop()** qui démarre le réceptionnaire d'événements. Les instructions qui suivent ne seront exécutées qu'à la sortie de cette boucle, sortie elle-même déclenchée par la méthode **fen1.quit()** (voir ci-après).
L'option **command** utilisée dans l'instruction d'instanciation des boutons permet de désigner la fonction qui devra être appelée lorsqu'un événement « *clic gauche de la souris sur le widget* » se produira. Il s'agit en fait d'un raccourci pour cet événement particulier, qui vous est proposé par *tkinter* pour votre facilité parce que cet événement est celui que l'on associe naturellement à un widget de type bouton. Nous verrons plus loin qu'il existe d'autres techniques plus générales pour associer n'importe quel type d'événement à n'importe quel widget.
Les fonctions de ce script peuvent modifier les valeurs de certaines variables qui ont été définies au niveau principal du programme. Cela est rendu possible grâce à l'instruction **global** utilisée dans la définition de ces fonctions. Nous nous permettrons de procéder ainsi pendant quelque temps encore (ne serait-ce que pour vous habituer à distinguer les comportements des variables locales et globales), mais comme vous le comprendrez plus loin, *cette pratique n'est pas vraiment recommandable,* surtout lorsqu'il s'agit d'écrire de grands programmes. Nous apprendrons une meilleure technique lorsque nous aborderons l'étude des classes (à partir de la page 163).
Dans notre fonction **changecolor()**, une couleur est choisie au hasard dans une liste. Nous utilisons pour ce faire la fonction **randrange()** importée du module *random*. Appelée avec un argument **N**, cette fonction renvoie un nombre entier, tiré au hasard entre **0** et **N-1**.
La commande liée au bouton <Quitter> appelle la méthode **quit()** de la fenêtre **fen1**. Cette méthode sert à fermer (quitter) le réceptionnaire d'événements (**mainloop**) associé à cette fenêtre. Lorsque cette méthode est activée, l'exécution du programme se poursuit avec les instructions qui suivent l'appel de **mainloop**. Dans notre exemple, cela consiste donc à effacer (**destroy**) la fenêtre.

Exercice XXI.3 : **Trace de lignes dans un canevas**

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
'''
ex_XXI03a_trace_de_lignes_dans_un_canevas.py
Utilisation de tkinter pour tracer des lignes dans un canevas
Les lignes sont horizontales et verticales
'''

from tkinter import *  # importe toutes les classes du module tkinter
from random import randrange

# --- définition des fonctions gestionnaires d'événements : ---
def drawline():
#==============
    "Tracé d'une ligne dans le canevas can1"
    global x1, y1, x2, y2
    can1.create_line(x1,  3,  x1, 203,     width=2, fill=coul)
    can1.create_line( 3, y1, 203, y1, width=2, fill=coul)

    # modification des coordonnées pour les lignes suivantes :
    x1 = x1+10
    y1 = y1+10

def changecolor():
#=================
    "Changement aléatoire de la couleur du tracé"
    global coul
    pal=['purple','cyan','maroon','green','red','blue','orange','yellow']
    c = randrange(8) # => génère un nombre aléatoire de 0 à 7
    coul = pal[c]

#------ Programme principal -------

# les variables suivantes seront utilisées de manière globale :
x1, y1 = 3, 3 # coordonnées de la ligne
coul = 'dark green' # couleur de la ligne

fen1 = Tk()
# Création du widget principal ("maître") :
# instanciation d'un objet à partir d'une classe :
# Avec la classe Tk(), nous en créons une instance la fenêtre fen1

# Permet de positionner la fenêtre sur l'écran
xpos = 400
ypos = 300
fen1.wm_geometry("+%d+%d" % (xpos, ypos))

fen1.title("Tacer des lignes avec tkinter") # titre de la fenêtre

# création des widgets "esclaves" :
can1 = Canvas(fen1, bg='dark grey', height=208, width=208)
can1.pack(side=LEFT)
bou1 = Button(fen1, text='Quitter', command=fen1.quit)
bou1.pack(side=BOTTOM)
bou2 = Button(fen1, text='Tracer une ligne', command=drawline)
bou2.pack()
bou3 = Button(fen1, text='Autre couleur', command=changecolor)
bou3.pack()

fen1.mainloop() # démarrage du réceptionnaire d'événements associé à la fenêtre
fen1.destroy() # destruction (fermeture) de la fenêtre
```

Exercice XXI.8 : **Deux figures**

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
'''
ex_XXI08a_deux_dessins.py
Utilisation de tkinter pour tracer des deux types de dessins.
'''

from tkinter import *

def cercle(x, y, r, coul ='black'):
#==================================
    "tracé d'un cercle de centre (x,y) et de rayon r"
    can.create_oval(x-r, y-r, x+r, y+r, outline=coul)

def figure_1():
#==============
    "dessiner une cible"
    # Effacer d'abord tout dessin préexistant :
    can.delete(ALL)

    # tracer les deux lignes (vert. Et horiz.) :
    can.create_line(100, 0, 100, 200, fill ='blue')
    can.create_line(0, 100, 200, 100, fill ='blue')

    # tracer plusieurs cercles concentriques :
    rayon = 15
    while rayon < 100:
        cercle(100, 100, rayon)
        rayon += 15

def figure_2():
#==============
    "dessiner un visage simplifié"
    # Effacer d'abord tout dessin préexistant :
    can.delete(ALL)

    # Les caractéristiques de chaque cercle sont
    # placées dans une liste de listes :
    cc = [[100, 100, 80, 'red'], # visage
          [70, 70, 15, 'blue'], # yeux
          [130, 70, 15, 'blue'],
          [70, 70, 5, 'black'],
          [130, 70, 5, 'black'],
          [44, 115, 20, 'red'], # joues
          [156, 115, 20, 'red'],
          [100, 95, 15, 'purple'], # nez
          [100, 145, 30, 'purple']] # bouche

    # on trace tous les cercles à l'aide d'une boucle :
    ii =0
    while ii < len(cc): # parcours de la liste
        el = cc[ii] # chaque élément est lui-même une liste
        cercle(el[0], el[1], el[2], el[3])
        ii += 1

##### Programme principal : ############
fen = Tk()
can = Canvas(fen, width=200, height=200, bg='ivory')
can.pack(side=TOP, padx=5, pady=5)
b1 = Button(fen, text='dessin 1', command=figure_1)
b1.pack(side=LEFT, padx=3, pady=3)
b2 = Button(fen, text='dessin 2', command=figure_2)
b2.pack(side=RIGHT, padx=3, pady=3)
fen.mainloop()
```

Exercice XXI.9 : **Olympique**

```
#!/usr/bin/python3
# -*- coding: utf-8 -*-
'''
ex_XXI09a_olympique.py
Utilisation de tkinter pour tracer des 5 anneaux olympiques
'''

from tkinter import *

def cercle(x, y, r, coul ='black'):
#==================================
    "tracé d'un cercle de centre (x,y) et de rayon r"
    can.create_oval(x-r, y-r, x+r, y+r, outline=coul, width=5)
    # c.f. http://infohost.nmt.edu/tcc/help/pubs/tkinter/web/create_oval.html

def figure_1():
#==============
    "dessin des 5 anneaux olympique"
    # Effacer d'abord tout dessin préexistant :
    can.delete(ALL)

    # Les caractéristiques de chaque cercle sont
    # placées dans une liste de listes :
    cc = [[ 50,  80, 40, 'cyan'],   # 'cyan' = '#00FFFF'
          [100,  80, 40, 'black'],  # 'black' = '#000000'
          [150,  80, 40, 'red'],    # 'red' = '#FF0000'
          [ 75, 110, 40, 'yellow'], # 'cyan' = '#00FFFF'
          [125, 110, 40, 'green']]  # 'green' = '#00FF00'

    # on trace tous les cercles à l'aide d'une boucle :
    ii =0
    while ii < len(cc): # parcours de la liste
        el = cc[ii] # chaque élément est lui-même une liste
        cercle(el[0], el[1], el[2], el[3])
        ii += 1

##### Programme principal : ############
fen = Tk()
can = Canvas(fen, width=200, height=200, bg='#808080') # '#FFFFFF' = 'white'
can.pack(side=TOP, padx=5, pady=5)
b1 = Button(fen, text='Olympique', command=figure_1)
b1.pack(side=LEFT, padx=3, pady=3)
b2 = Button(fen, text='Quitter', command=fen.destroy)
b2.pack(side=RIGHT, padx=3, pady=3)
fen.mainloop()
```

Série XXII : En savoir plus sur les listes et chaînes de caractères

REMARQUE XXII.0.A : **Les chaînes de caractères**

ch="Elements de programmation en Python"

- Les caractères d'une chaîne sont indicés à partir de 0 (print(ch[0]) ✉ E)
- Les espaces sont des caractères (print(ch[8])✉) !
- On peut partir depuis la fin (print(ch[-1])✉ n)
- On peut aussi extraire des parties de la chaîne (*"Slicing" ; slice = tranche*) (print(ch[12 :24])✉ programmatio) **Attention :** l'indice de fin n'est pas inclus !!
- Valeur par défaut du premier indice est 0 (print(ch[:8]✉Elements)
- Et du deuxième est la longueur de la chaîne (print(ch[9 :])✉ de programmation en Python) est équivalent à print(ch[9 : len(ch)]
- Les chaînes peuvent être *concaténées* avec l'opérateur **"+"** et *répétées* avec l'opérateur **"*"**

On peut parcourir tous les éléments d'une chaîne de caractères à l'aide de l'instruction **for ... in**. C'est une *instruction composée* qui remplace avantageusement l'instruction *while* :

```
>>> nom = 'Joséphine'
>>> for c in nom :
        print(c + '*', end=' ')


J* o* s* é* p* h* i* n* e*
```

L'équivalent avec une boucle *while* serait :

```
>>> nom = 'Joséphine'
>>> i = 0
>>> while i < len(nom) :
        print(nom[i] + '*', end = ' ')
        i+=1


J* o* s* é* p* h* i* n* e*
```

L'instruction **in** peut être utilisée seule, indépendamment de **for**, *pour vérifier si un élément donné fait partie ou non d'une séquence*.

Exemple :

```
>>> voyelles = 'aeiouyAEIOUYàâäéèêëîïùû'
>>> def carVoyelle(ch) :
        if ch in voyelles :
                print(ch, "est une voyelle")
        else :
                print(ch, "n'est pas une voyelle")

>>> carVoyelle('e')
e est une voyelle
>>> carVoyelle('k')
k n'est pas une voyelle
```

Les chaînes de caractères ne sont pas modifiables à l'aide de l'instruction d'affectation :

```
>>> salut = 'bonjour à tous'
>>> salut[0] = 'B'
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    salut[0] = 'B'
TypeError: 'str' object does not support item assignment
```

On peut contourner le problème ainsi :

```
>>> salut = 'bonjour à tous'
>>> salut = 'B' + salut[1:]
>>> print(salut)
Bonjour à tous
```

Les chaînes de caractères sont des *objets*. On peut agir sur elles à l'aide de *méthodes* associées à cette *classe d'objets*. Exemples parmi les plus courants :

- **split() :** convertit une chaîne en une liste de sous-chaînes ; on peut choisir le séparateur des sous-chaînes en le fournissant comme argument, sinon par défaut c'est un espace.

```
>>> d = "Cet exemple, parmi d'autres, peut encore servir"
>>> d.split()
['Cet', 'exemple,', 'parmi', "d'autres,", 'peut', 'encore', 'servir']
>>> d.split(',')
['Cet exemple', " parmi d'autres", ' peut encore servir']
```

- **join(liste) :** rassemble une liste de chaînes en une seule.

```
>>> b = ["Bête", "à", "manger", "du", "foin"]
>>> " ".join(b)
'Bête à manger du foin'
>>> print(" ".join(b))
Bête à manger du foin
>>> print("---".join(b))
Bête---à---manger---du---foin
>>> print("**".join(b))
Bête**à**manger**du**foin
```

- **find(sch) :** cherche la position d'une sous-chaîne **sch** dans la chaîne.

```
>>> ch1 = "Cette leçon vaut bien un fromage"
>>> ch2 = "fromage"
>>> ch1.find(ch2)
25
```

- **count(sch) :** compte le nombre de sous-chaînes **sch** dans la chaîne.

```
>>> ch1 = "Le héron au long bec emmanché d'un long cou"
>>> ch2 = "long"
>>> ch1.count(ch2)
2
```

- **lower() :** convertit une chaîne en minuscules.

```
>>> ch = "CÉLIMÈNE est un prénom ancien"
>>> ch.lower()
'célimène est un prénom ancien'
>>> print(ch.lower())
célimène est un prénom ancien
```

- **upper() :** convertit une chaîne en majuscules.

```
>>> ch = "Maître Jean-Noël Mamère"
>>> ch.upper()
'MAÎTRE JEAN-NOËL MAMÈRE'
>>> print(ch.upper())
MAÎTRE JEAN-NOËL MAMÈRE
```

- **title() :** convertit la lettre initiale de chaque mot en majuscule.

```
>>> ch = "jean albert gaston renaud"
>>> print(ch.title())
Jean Albert Gaston Renaud
```

- **capitalize() :** convertit la première lettre de la chaîne en majuscule.

```
>>> ch = "jean albert gaston renaud"
>>> print(ch.capitalize())
Jean albert gaston renaud
```

- **swapcase() :** convertit toutes les minuscules en majuscules et vice-versa.

```
>>> ch = "Harold Et Maude"
>>> print(ch.swapcase())
hAROLD eT mAUDE
```

- **strip() :** enlève les espaces éventuels au début et à la fin de la chaîne.

```
>>> ch = "    Il y a des espaces en trop dans cette chaîne   "
>>> ch.strip()
'Il y a des espaces en trop dans cette chaîne'
```

- **replace(c1, c2) :** remplace tous les caractères **c1** par des caractères **c2** dans la chaîne.

```
>>> ch = "turlututu"
>>> print(ch.replace('u', 'a'))
tarlatata
```

- **index(car) :** retrouve l'indice de la première occurrence du caractère **car** dans la chaîne.

```
>>> ch = "Il n'y a pas de fumée sans feu, selon le dicton"
>>> ch.index('s')
11
```

On peut indiquer à partir de quel indice on veut commencer la recherche :

```
>>> ch = "Il n'y a pas de fumée sans feu, selon le dicton"
>>> ch.index('s', 12)
22
>>> ch.index('s', 29)
32
```

Exercice XXI.1 : fonction ou méthode …

Écrire une fonction `maFonctionIndex( ??? )` qui retourne la position de la première occurrence d'un caractère `car` dans une chaîne `ch`, sans utiliser la méthode `index()` …

REMARQUE XXII.0.B : **Les listes**

On peut définir une liste comme une collection ordonnée d'objets. Les éléments d'une liste peuvent de différents types et ils sont accessibles au moyen de leur indice, celui du premier élément étant le 0. Le "slicing" existe aussi, comme pour les chaînes de caractères.

```
>>> nombres = [34, 7, 45, 97]
>>> mots = ["vélo", "moto", "trottinette", "voiture"]
>>> stuff = [5000, "Gaston", 3.1416, ["Hubert", "Arthur", 1934]]
>>> print(nombres[2])
45
>>> print(nombres[2:3])
[45]
>>> print(nombres[1:3])
[7, 45]
>>> print(nombres[2:])
[45, 97]
>>> print(nombres[:2])
[34, 7]
>>> print(nombres[-1])
97
>>> print(nombres[-2])
45
>>> print(nombres[-3:-1])
[7, 45]
>>> print(nombres[-3:])
[7, 45, 97]
>>> print(mots[1])
```

On peut accéder à un élément d'une liste, elle-même située à l'intérieur d'une autre liste :

```
>>> stuff[3][1]
'Arthur'
```

On peut modifier les éléments d'une liste, contrairement à ce qui se passe avec les chaînes de caractères :

```
>>> print(mots)
['vélo', 'moto', 'trottinette', 'voiture']
>>> mots[2]='vespa'
>>> print(mots)
['vélo', 'moto', 'vespa', 'voiture']
```

On peut même modifier les éléments d'une liste située à l'intérieur d'une autre :

```
>>> stuff[3][1]="Constantin"
>>> print(stuff)
[5000, 'Gaston', 3.1416, ['Hubert', 'Constantin', 1934]]
```

Les listes étant des objets, on peut leur appliquer un certain nombre de méthodes associées à cette classe d'objets. Exemples :

- **sort() :** trie les éléments d'une liste

```
>>> nombres=[17, 38, 10, 25, 72]
>>> nombres.sort()
>>> nombres
[10, 17, 25, 38, 72]
>>> mots=['lampion', 'cornichon', 'alambic', 'torchon']
>>> mots.sort()
>>> mots
['alambic', 'cornichon', 'lampion', 'torchon']
```

- **append() :** ajoute un élément à la fin de la liste

```
>>> nombres.append(12)
>>> nombres
[10, 17, 25, 38, 72, 12]
```

- **reverse() :** inverse l'ordre des éléments

```
>>> nombres.reverse()
>>> nombres
[12, 72, 38, 25, 17, 10]
>>>
```

- **index() :** retrouve l'indice d'un élément

```
>>> nombres.index(17)
4
```

- **remove() :** enlève un élément de la liste

```
>>> nombres.remove(38)
>>> nombres
[12, 72, 25, 17, 10]
```

Il existe aussi l'instruction **del** (qui n'est pas une méthode), qui permet d'effacer un ou plusieurs éléments à partir de leur(s) indice(s) :

```
>>> del nombres[2]
>>> nombres
[12, 72, 17, 10]
>>> del nombres[1:3]
>>> nombres
[12, 10]
```

Notez la différence entre **del** et **remove()** : **del** travaille avec l'*indice* ou une *tranche d'indices*, alors que **remove()** recherche une *valeur* (si plusieurs éléments de la liste possèdent la même valeur, seul le premier sera effacé).

Exercice XXII.1 : **Slicing : remplacer des tranches par d'autres tranches**

Testons :

```
>>> mots = ['jambon', 'fromage', 'confiture', 'chocolat']
>>> mots[2:2] = ['miel']
```

Que contient à présent mots ? Que s'est-il passé ?

Remplacer (!?!) en fin de liste :

```
>>> mots[5:5] = ['saucisson', 'ketchup']
```

Et avec une chaîne vide :

```
>>> mots[2:5] = []
```

Sans indice de fin :

```
>>> mots[1:] = ['mayonnaise', 'poulet', 'tomate']
```

Exercice XXII.2 : **La fonction range()**

Trouver les arguments de la fonction range() à l'aide de la fonction help().
Générer la liste [2, 5, 8, 11, 14, 17, 20] à l'aide de range() et de list().
Peut-on créer [12, 5, -2, -9, -16] de la même manière ?

Exercice XXII.3 : **Quelle « méthode » ?**

A l'aide d'une boucle for et de ch='Fais de ta vie un reve et de tes reves une realite', comment obtenir :

```
0 Fais
1 de
2 ta
3 vie
4 un
5 reve
6 et
7 de
8 tes
9 reves
10 une
11 realite
```

Indication : utiliser l'une des méthodes présentées ci-dessus

Exercice XXII.4 : **Copie d'une liste**

Avec :

```
>>> l1=['Je','plie','mais','ne','romps','point']
>>> l2=l1
```

Que vaut l2 ? Et l1 ?

Si l'on modifie l1,

```
>>> l1[4]='casse'
```

que valent alors l1 et l2 ?

Si l'on modifie l2,

```
>>> l2[4]='me brise'
```

que valent alors l1 et l2 ?

Et si on modifie la longueur de l1,

```
>>> l1=l1+['!']
```

que valent alors l1 et l2 ?

Comment créer l3 proprement une copie de l2 ?

Exercice XXII.5 : **Listes de nombres aléatoires**

A l'aide de la fonction randrange(m,n,p) du module **random**, générer de façon élégante une liste de 15 nombres aléatoires multiples de 3 et compris entre -10 et 50.

REMARQUE XXII.5.A : **Les tuples**

Le "tuple" est comparable à une liste, c'est une collection d'objets séparés par des virgules, encadrés par des parenthèses (au lieu de crochets).

Les tuples, tout comme les chaînes, ne sont pas modifiables. A ce titre, ils sont préférables aux listes partout où l'on veut être certain que les données transmises ne soient pas modifiées par erreur au sein d'un programme. En outre, les tuples sont moins « gourmands » en ressources système (ils occupent moins de place en mémoire, et peuvent être traités plus rapidement par l'interpréteur).

Du point de vue de la syntaxe, un tuple est une collection d'éléments séparés par des virgules :

```
>>> tup = ('a', 'b', 'c', 'd', 'e') #parenthèses vivement conseillées (pas
obligatoires)
>>> print(tup)
('a', 'b', 'c', 'd', 'e')
>>> tup = (1,) #tuple à 1 caractère, virgule obligatoire
```

Opérations sur les tuples

Les opérations que l'on peut effectuer sur des tuples sont syntaxiquement similaires à celles que l'on effectue sur les listes : **len()** pour sa longueur, une boucle **for**, pour le parcourir, l'instruction **in** pour savoir si un élément donné en fait partie, les opérateurs de concaténation et de multiplication, etc

Les tuples ne sont pas modifiables, donc pas de **del** ni de **remove()**.

Corrigé En savoir plus sur les listes et chaînes de caractères

Corrigé Slicing : remplacer des tranches par d'autres tranches Exercice XXII.1 :

```
>>> mots = ['jambon', 'fromage', 'confiture', 'chocolat']
>>> mots[2:2] = ['miel']
>>> mots
['jambon', 'fromage', 'miel', 'confiture', 'chocolat']
>>> mots[5:5] = ['saucisson', 'ketchup']
>>> mots
['jambon', 'fromage', 'miel', 'confiture', 'chocolat', 'saucisson',
'ketchup']
>>> mots[2:5] = []
>>> mots
['jambon', 'fromage', 'saucisson', 'ketchup']
>>> mots[1:3] = ['salade']
>>> mots
['jambon', 'salade', 'ketchup']
>>> mots[1:] = ['mayonnaise', 'poulet', 'tomate']
>>> mots
['jambon', 'mayonnaise', 'poulet', 'tomate']
```

Remarque : même lorsqu'on ne remplace qu'un seul élément, celui-ci est à donner sous forme de liste !

Corrigé La fonction range() Exercice XXII.2 :

- help(range) renseigne sur l'utilisation de range(), notamment :
  1. range(n) fournit les nombres entiers de **0 à n-1**
  2. range(m,n) fournit les nombres entiers de m à n-1
  3. range(m,n,p) fournir les nombres entiers de m à n(exclu) avec un incrément p
- Par exemple list(range(2,21,3)) permet de générer [2, 5, 8, 11, 14, 17, 20]
- Oui, par exemple list(range(12,-21,-7)) donne [12, 5, -2, -9, -16] (avec des arguments négatifs)

Corrigé Quelle « méthode » ? Exercice XXII.3 :

```
>>> ch='Fais de ta vie un reve et de tes reves une realite'
>>> devise=ch.split()
>>> for index in range(len(devise)):
      print(index, devise[index])
```

Corrigé Copie d'une liste Exercice XXII.4 :
On constate que, lorsque l'on modifie le contenu de l1, alors l2 se modifie aussi et réciproquement ! l1 et l2 sont deux variables qui désignent le même emplacement dans la mémoire de l'ordinateur. Le nom l2 est ce que l'on appelle un **alias** du nom l1.
Par contre, dès qu'on change la longueur de l1 ou de l2, ces variables deviennent indépendantes. Pour créer l3, copie de l2, on peut procéder comme suit :

```
>>> l3=[]
>>> for i in range(0,len(l2)):
      l3.append(l2[i])
```

Corrigé Listes de nombres aléatoires Exercice XXII.5 :

```
>>> from random import *
>>> s=[0]*15 # plus élégant que "append()", on crée une liste de 15 zéros
>>> for i in range(len(s)):
      s[i]=randrange(-9, 50,3 ) #on part de -9, multiple de 3
```

Série XXIII : Les dictionnaires

Les *chaînes de caractères*, les *listes* et les *tuples* sont tous des exemples de *séquences*. Celles-ci font partie de ce que l'on appelle les *types* composites.
Les ***dictionnaires*** sont un autre exemple de *type composite* mais ce ne sont pas des séquences. L'ordre des éléments qui les composent n'est pas immuable ; par contre, on pourra accéder à chacun de ces éléments à l'aide d'un indice spécifique que l'on appellera ***clé***, laquelle pourra être alphabétique, numérique ou même d'un autre type.
Comme dans une liste, les éléments mémorisés dans un dictionnaire peuvent être de n'importe quel type.

Le dictionnaire étant un type **modifiable**, on peut le créer comme les listes, à savoir en commençant par un dictionnaire vide que l'on remplira petit à petit. Syntaxiquement, on reconnaît un dictionnaire au fait que ses éléments sont disposés entre des *accolades*.

Exercice XXIII.0 : **Un premier dictionnaire**

Essayons (attention aux ', ", virgules et points et deux-points) :

```
>>> dico = {'screen' : 'ecran', 'operating system' : "systeme d'exploitation",
'price' : 1592.35}
>>> print(dico)
?
>>> dico['computer'] = 'ordinateur'
>>> dico['mouse'] ='souris'
>>> dico['keyboard'] ='clavier'
>>> print(dico)
?
>>> print(dico['mouse'])
?
>>> print(dico['souris'])
?
>>> dico.get('souris', 'y a pas')
>>> dico.get('mouse', 'y a pas')
?
>>> print(dico.keys())
?
>>> 'screen' in dico
?
>>> 'souris' in dico
?
>>> for i in dico.keys():
      print("avec la clé :", i, " on a la valeur :", dico[i])
?
>>> dico.items()
?
>>> dico.values()
?
>>> len(dico)
?
>>> dico2=dico.copy()
>>> del dico['mouse']
>>> print(dico)
>>> print(dico2)
?
```

On remarque donc que les opérateurs usuels s'appliquent aux dictionnaires :
len() , **del** , **in** (permet de savoir si un dictionnaire contient une clé bien déterminée)
Et quelques nouveautés, des méthodes associées aux objets dictionnaire :

- **values() :** renvoie la séquence des valeurs mémorisées dans le dictionnaire :
- **items() :** renvoie les paires du dictionnaire sous forme de tuples :
- **copy() :** permet d'effectuer une **vraie copie** d'un dictionnaire (et non pas un *alias*, cf. explication pour les listes)

Attention : comme les dictionnaires ne sont pas des séquences, on ne peut ni les concaténer ni utiliser la méthode de "slicing"

REMARQUE XXIII.0.A : ***paires "clé - valeur" et quelques opérateurs***

Un dictionnaire est donc constitué de *paires "clé - valeur"*. Dans l'exercice ci-dessus, les noms en anglais sont les clés et les noms en français les valeurs, tous deux de type chaîne de caractères. Pour accéder à un élément particulier, on utilise sa clé.

Attention : dico.keys() n'est pas une liste, c'est une *classe*

ATTENTION : l'ordre dans lequel les éléments sont extraits est imprévisible, car un dictionnaire n'est pas une séquence.

Exercice XXIII.1 : **Opération sur le dictionnaire**

Dans l'exercice précédent, nous avons parcouru le dictionnaire à l'aide de la boucle :

```
for i in dico.keys():
      print("avec la clé :", i, " on a la valeur :", dico[i])
```

Ré-écrire cette boucle for utilisant la méthode items() au lieu de keys().

Exercice XXIII.2 : **Base de donnée**

Écrivez un script qui crée un mini-système de base de données fonctionnant à l'aide d'un dictionnaire, dans lequel vous mémoriserez les noms d'une série de copains, leur âge et leur taille. Votre script devra comporter deux fonctions : la première pour le remplissage du dictionnaire, et la seconde pour sa consultation.

- Dans la fonction de remplissage, utilisez une boucle pour accepter les données entrées par l'utilisateur. Dans le dictionnaire, le nom de l'élève servira de clé d'accès, et les valeurs seront constituées de tuples (âge, taille), dans lesquels l'âge sera exprimé en années (donnée de type entier), et la taille en mètres (donnée de type réel).
- La fonction de consultation comportera elle aussi une boucle, dans laquelle l'utilisateur pourra fournir un nom quelconque pour obtenir en retour le couple « âge, taille » correspondant. Le résultat de la requête devra être une ligne de texte bien formatée, telle par exemple : « Nom : Jean Dhoute - âge : 15 ans - taille : 1.74 m ». Pour obtenir ce résultat, servez-vous du formatage des chaînes de caractères.

Exercice XXIII.3 : **Dictionnaire bilingue**

Écrivez une fonction qui échange les clés et les valeurs d'un dictionnaire (ce qui permettra par exemple de transformer un dictionnaire anglais/français en un dictionnaire français/anglais). On suppose que le dictionnaire ne contient pas plusieurs valeurs identiques.

Exercice XXIII.4 : **Construction d'un histogramme***

A l'aide d'un dictionnaire, un outil très élégant pour construire des *histogrammes*, donner dans l'ordre alphabétique la fréquence d'apparition des lettres du texte:

texte ="les saucisses et saucissons secs sont dans le saloir"

Indications :

- Créer un dictionnaire « lettres » ayant pour clé les caractères de l'alphabets présents dans « texte » liés à la fréquence du caractère.
- Utiliser la méthode get(), qui doit retourner zéro si le caractère n'est pas encore connu.
- La méthode sort() ne s'appliquant qu'aux listes, créer une liste à partir du dictionnaire.

Exercice XXIII.5 : **Histogramme à partir d'un fichier**

Prenez un fichier texte quelconque (pas trop gros). À partir du script de l'exercice précédent, écrivez un script qui compte les occurrences de chacune des lettres de l'alphabet dans ce texte.

Traiter les lettres comme si elles n'étaient ni accentuées ni majuscules.

Exercice XXIII.6 : **Histogrammes de mots**
Modifiez le script ci-dessus afin qu'il établisse une table des occurrences de chaque *mot* dans le texte.
Conseil : dans un texte quelconque, les mots ne sont pas seulement séparés par des espaces, mais également par divers signes de ponctuation. Pour simplifier le problème, vous pouvez commencer par remplacer tous les caractères non-alphabétiques par des espaces, et convertir la chaîne résultante en une liste de mots à l'aide de la méthode **split()**.

Exercice XXIII.7 : **Dictionnaire de listes**
Prenez un fichier texte quelconque (pas trop gros). Écrivez un script qui analyse ce texte, et mémorise dans un dictionnaire l'emplacement exact de chacun des mots (compté en nombre de caractères à partir du début). Lorsqu'un même mot apparaît plusieurs fois, tous ses emplacements doivent être mémorisés : chaque valeur de votre dictionnaire doit donc être une liste d'emplacements.

Exercice XXIII.8 : **Contrôle du flux d'exécution à l'aide d'un dictionnaire**
Les séries d'instructions **if - elif - else** peuvent devenir assez lourd et inélégant si vous avez affaire à un grand nombre de possibilités.
Choix du matériau :

```
materiau = input("Choisissez le matériau : ")

if materiau == 'fer':
    fonctionA()
  elif materiau == 'bois':
    fonctionC()
  elif materiau == 'cuivre':
    fonctionB()
  elif materiau == 'pierre':
    fonctionD()
  elif ... etc ...
```

Les langages de programmation proposent souvent des instructions spécifiques pour traiter ce genre de problème, telles les instructions *switch* ou *case* du *C* ou du *Pascal*. Python n'en propose aucune, on propose de se tirer d'affaire à l'aide d'un dictionnaire.

Proposer une démarche qui, à l'aide d'un dictionnaire, permet le choix d'une fonction selon le matériau sans utiliser de if-elif-else.

Exercice XXIII.9 : **Dictionnaire de et vers un fichier texte**
Complétez Exercice XXIII.2 : (Base de donnée) en lui ajoutant deux fonctions : l'une pour enregistrer le dictionnaire résultant dans un fichier texte, et l'autre pour reconstituer ce dictionnaire à partir du fichier correspondant.
Chaque ligne de votre fichier texte correspondra à un élément du dictionnaire. Elle sera formatée de manière à bien séparer :
- la clé et la valeur (c'est-à-dire le nom de la personne, d'une part, et l'ensemble : « âge + taille », d'autre part ;
- dans l'ensemble « âge + taille », ces deux données numériques.
Vous utiliserez donc deux caractères séparateurs différents, par exemple « @ » pour séparer la clé et la valeur, et « # » pour séparer les données constituant cette valeur :
Juliette@18#1.67
Jean-Pierre@17#1.78
Delphine@19#1.71
Anne-Marie@17#1.63 etc.

Exercice XXIII.10 : **Dictionnaire pour contrôler le flux**
Améliorez encore le script de l'exercice précédent, en utilisant un dictionnaire pour diriger le flux d'exécution du programme au niveau du menu principal.
Votre programme affichera par exemple :

Choisissez :
(R)écupérer un dictionnaire préexistant sauvegardé dans un fichier
(A)jouter des données au dictionnaire courant
(C)onsulter le dictionnaire courant
(S)auvegarder le dictionnaire courant dans un fichier
(T)erminer :
Suivant le choix opéré par l'utilisateur, vous effectuerez alors l'appel de la fonction correspondante en la sélectionnant dans un dictionnaire de fonctions.

Corrigé Les dictionnaires Série XXIII :

Corrigé Opération sur le dictionnaire Exercice XXIII.1 :

```
>>> for (k, val) in dico.items():
        print("avec la clé :", k, " on a la valeur :", val)
```

avec la clé : screen on a la valeur : ecran
avec la clé : operating system on a la valeur : systeme d'exploitation
avec la clé : price on a la valeur : 1592.35
avec la clé : computer on a la valeur : ordinateur
avec la clé : mouse on a la valeur : souris
avec la clé : keyboard on a la valeur : clavier

Corrigé

```
texte ="les saucisses et saucissons secs sont dans le saloir"
lettres ={}
for c in texte:
    lettres[c] =lettres.get(c, 0) + 1
listeLettres = list(lettres.items())
listeLettres.sort()
print(listeLettres)
```

Réponse :

```
[(' ', 8), ('a', 4), ('c', 3), ('d', 1), ('e', 5), ('i', 3), ('l', 3), ('n', 3),
('o', 3), ('r', 1), ('s', 14), ('t', 2), ('u', 2)]
```

Corrigé

```
materiau = input("Choisissez le matériau : ")

dico = {'fer':fonctionA,
'bois':fonctionC,
'cuivre':fonctionB,
'pierre':fonctionD,
... etc ...}

dico.get(materiau,fonctionAutre)()
```

Lorsque la valeur de la variable **materiau** ne correspond à aucune clé du dictionnaire, c'est la fonction **fonctAutre()** qui est invoquée.

Attention, on n'indique dans le dictionnaire seulement des *noms* de ces fonctions, *qu'il ne faut surtout pas faire suivre de parenthèses dans ce cas* (sinon Python exécuterait chacune de ces fonctions au moment de la création du dictionnaire). Les () sont placées lors de la recherche dans le dictionnaire : dico.get(materiau,fonctionAutre)**()**

Sources :

- A disposition au CEDOC : Gérard Swinnen, ***Apprendre à programmer avec Python 3***, ouvrage distribué suivant les termes de la Licence Creative Commons « Paternité-Pas d'Utilisation Commerciale-Partage des Conditions Initiales à l'Identique - 2.0 France ».
- www.mathex.net (2016)
- Eric Von Aarburg, mon collègue de Calvin
- F. Marchino, Programmation Python 16-17, Collège et Ecole de Commerce Emilie-Gourd