



## 11

# AUTRES GRAPHISMES AVEC LA TORTUE

Nous allons examiner un peu plus en profondeur le module `turtle` abordé au chapitre 4. Nous verrons qu'en Python, les tortues peuvent dessiner bien plus que quelques lignes noires. Nous pouvons notamment dessiner des formes géométriques évoluées, de couleurs variées et même remplir ces formes de couleurs.

## Dessiner un carré, pour commencer

Nous avons déjà abordé le dessin de formes simples. Pour rappel, avant d'utiliser la tortue, nous devons importer le module `turtle` et créer l'objet `Pen` :

---

```
>>> import turtle  
>>> t = turtle.Pen()
```

---

Au chapitre 4, nous avons utilisé le code suivant pour dessiner un carré :

---

```
>>> t.forward(50)  
>>> t.left(90)  
>>> t.forward(50)  
>>> t.left(90)  
>>> t.forward(50)  
>>> t.left(90)  
>>> t.forward(50)
```

---

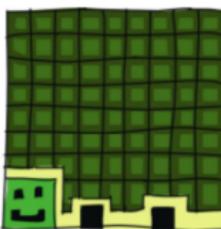
Comme nous avons étudié au chapitre 6 l'utilisation de boucles `for`, nous pouvons réduire ce code fastidieux avec une boucle comme celle-ci :

---

```
>>> t.reset()  
>>> for x in range(1, 5):  
    t.forward(50)  
    t.left(90)
```

---

La première ligne réinitialise l'objet `Pen`. La deuxième ligne débute une boucle `for` qui compte de 1 (inclus) jusqu'à 5 (exclu) avec `range(1,5)`. Les deux lignes suivantes sont exécutées à chaque passage dans la boucle : l'une avance de 50 pixels et l'autre « tourne » à gauche de 90 degrés. Utiliser une boucle `for` donne un code un peu plus court que celui de la version précédente. Le `reset` mis à part, nous sommes passés de six lignes à trois seulement.



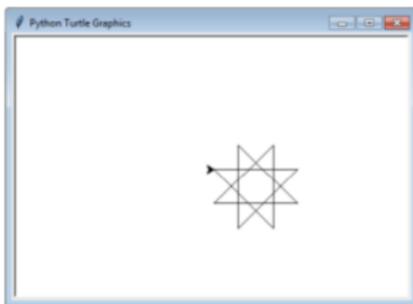
Avec quelques modifications simples à cette boucle `for`, nous pouvons dessiner quelque chose de plus rigolo. Tape le code suivant :

---

```
>>> t.reset()  
>>> for x in range(1, 9):  
    t.forward(100)  
    t.left(225)
```

---

Ce code génère une étoile à huit branches :



Le code ressemble fort à celui utilisé pour dessiner un carré.  
Voici les différences :

- au lieu de boucler quatre fois, nous bouclons huit fois, avec `range(1,9)` ;
- au lieu d'avancer de 50 pixels, nous avançons de 100 pixels ;
- au lieu de tourner de 90 degrés, nous tournons de 225 degrés à gauche.

Voyons ce que cela donne si nous allons un peu plus loin. Avec un angle de 175 degrés et une boucle de 37 fois, nous obtenons une étoile ayant bien plus de branches :

---

```
>>> t.reset()
>>> for x in range(1, 38):
...     t.forward(100)
...     t.left(175)
```

---

Ce code produit la forme suivante :



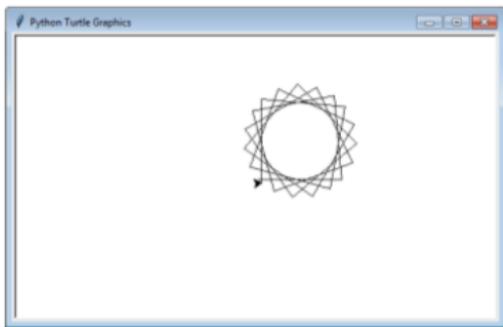
Et pendant que nous jouons avec les étoiles, voici le code pour créer une étoile en spirale :

---

```
>>> t.reset()
>>> for x in range(1, 20):
    t.forward(100)
    t.left(95)
```

---

En modifiant seulement les nombres de degrés du changement de direction (ou de « rotation ») et de boucles, la tortue dessine un type d'étoile très différent :



Ainsi, un code de ce genre permet de créer toute une variété de formes, du simple carré à l'étoile en spirale. Les boucles `for` simplifient donc réellement l'écriture de code et le dessin de telles formes car, sans ces boucles, le code serait fastidieux à entrer au clavier.



Examinons à présent une variante d'étoile, avec l'utilisation d'une instruction `if` pour contrôler la rotation de la tortue. Dans l'exemple qui suit, nous voulons que la tortue tourne d'un certain angle la première fois, puis d'un autre angle la fois suivante.

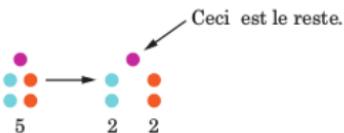
---

```
>>> t.reset()
>>> for x in range(1, 19):
    t.forward(100)
    if x % 2 == 0:
        t.left(175)
    else:
        t.left(225)
```

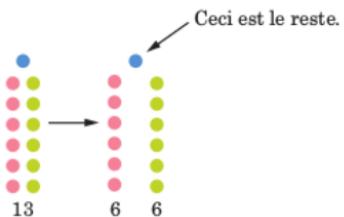
---

La boucle effectue cette fois 18 passages (`range(1,19)`) et indique à la tortue d'avancer de 100 pixels (`t.forward(100)`). Les nouveautés se situent dans l'instruction `if`. Cette dernière vérifie si la variable `x` contient un nombre pair à l'aide de l'opérateur modulo (d'autres langages utilisent *mod*, en abrégé), le `%` dans l'expression `x % 2 == 0`, qui revient à dire « `x` modulo 2 » est-il égal à 0 ?

L'expression `x % 2` signifie exactement : « Quel est le reste de la division entière du nombre dans la variable `x` en deux parties égales ? » Pour bien comprendre de quoi il s'agit, prenons l'exemple de cinq billes à diviser en deux tas. Nous aurions deux tas de deux billes, pour un total de quatre. Le reste de la division entière (autrement dit, du modulo) serait la bille supplémentaire, en sachant que nous ne pouvons pas la couper en deux :



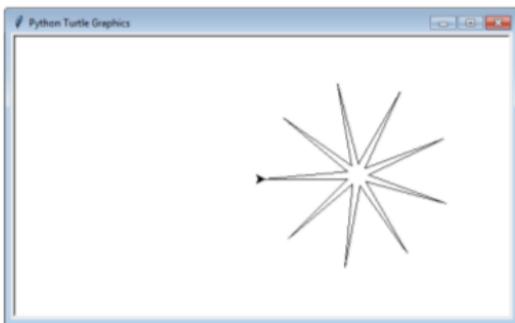
De même, si nous divisons par exemple 13 billes en deux tas, nous aurions deux tas de 6 billes et une bille qui reste :



Autrement dit, lorsque nous vérifions si le reste est égal à zéro lors de la division de `x` par 2, nous vérifions si `x` est pair, puisque la division de nombres pairs en deux « tas » ne donne aucun reste.

La cinquième ligne indique à la tortue de tourner de 175 degrés à gauche (`t.left(175)`), si le nombre dans `x` est pair. Dans le cas contraire (`else:`), à la ligne finale, nous lui indiquons de tourner de 225 degrés à gauche (`t.left(225)`).

Le résultat est le suivant :



## Dessiner une voiture

Une tortue peut dessiner bien d'autres choses que des formes géométriques simples. Nos exemples suivants vont tracer une voiture, plutôt simpliste, mais une voiture tout de même. Commençons par dessiner le corps de l'auto. Dans IDLE, clique sur le menu **File>New File**, puis entre le code suivant dans cette nouvelle fenêtre :

---

```
t.reset()
t.color(1,0,0)
t.begin_fill()
t.forward(100)
t.left(90)
t.forward(20)
t.left(90)
t.forward(20)
t.right(90)
t.forward(20)
t.left(90)
t.forward(60)
t.left(90)
t.forward(20)
t.right(90)
t.forward(20)
t.left(90)
t.forward(20)
t.end_fill()
```

---

Ensuite, dessinons la première roue :

---

```
t.color(0,0,0)
t.up()
t.forward(10)
t.down()
t.begin_fill()
t.circle(10)
t.end_fill()
```

---

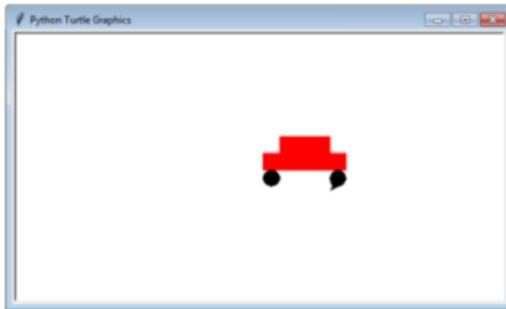
Et enfin, dessinons la seconde roue :

---

```
t.setheading(0)
t.up()
t.forward(90)
t.right(90)
t.forward(10)
t.setheading(0)
t.begin_fill()
t.down()
t.circle(10)
t.end_fill()
```

---

Clique sur le menu **File>Save As**, entre le nom de fichier **voiture.py** et appuie sur **Entrée**. Clique ensuite sur le menu **Run>Run Module** pour exécuter ce code. Et voici la voiture :



Tu as sans doute remarqué que ce code contient quelques nouvelles fonctions de la tortue :

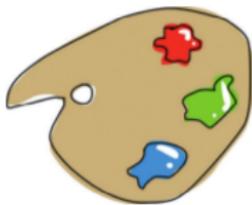
- la fonction **color** change la couleur du stylo ;
- les fonctions **begin\_fill** (début remplissage) et **end\_fill** (fin remplissage) servent à remplir une zone fermée du canevas avec une couleur ;

- la fonction `circle` dessine un cercle d'une taille donnée ;
- la commande `setheading` fait tourner la tortue pour qu'elle se dirige dans une direction donnée.

Voyons comment utiliser ces fonctions pour ajouter des couleurs à nos dessins.

## Voir la vie en couleurs

La fonction `color` accepte trois paramètres. Le premier indique la quantité de rouge, le deuxième la quantité de vert et le troisième la quantité de bleu. Pour obtenir la couleur rouge vif de l'auto, nous avons écrit `color(1,0,0)`, ce qui signifie que la tortue doit utiliser un stylo avec 100 % de rouge, mais 0 % de vert et de bleu.



La recette de la combinaison de couleurs rouge, vert, bleu pour obtenir une couleur spécifique s'appelle RVB (*RGB* en anglais). C'est la manière dont l'ordinateur représente les couleurs à l'écran. Le mélange relatif de ces couleurs primaires permet de générer bien d'autres couleurs, comme lorsque tu fais de la peinture et que tu mélanges du rouge et du jaune pour obtenir de l'orange, ou du rouge et du bleu pour obtenir du violet. Les couleurs rouge, vert, bleu sont appelées « couleurs primaires », parce qu'il n'est pas possible de mélanger deux de ces couleurs pour obtenir la troisième.

Même si nous utilisons de la lumière, ici, et si les combinaisons des teintes ne sont pas les mêmes qu'en peinture, nous allons comparer cette recette du RVB à celle des pots de peinture : un rouge, un vert et un bleu. Chacun des pots est plein et, dans ce cas-là, nous attribuons la valeur `1` (ce qui correspond à 100 %) à chaque couleur. Quand nous mélangeons tout le pot de peinture rouge avec tout le pot de peinture verte dans un pot plus grand, c'est-à-dire `1` de rouge, `1` de vert, ou 100 % de ces deux couleurs, nous obtenons de la peinture jaune.

Revenons à l'univers de la programmation. Pour dessiner un cercle jaune avec la tortue, nous devons utiliser 100 % de rouge et de vert, mais 0 % de bleu, comme ceci :

---

```
>>> t.color(1,1,0)
>>> t.begin_fill()
>>> t.circle(50)
>>> t.end_fill()
```

---

Donc le (1,1,0) de la première ligne représente 100 % de rouge, 100 % de vert et 0 % de bleu. La ligne suivante indique à la tortue de remplir avec cette couleur RVB (`t.begin_fill()`) les prochaines formes qu'elle dessine. La troisième ligne dit à la tortue de dessiner un cercle (`t.circle()`), tandis qu'à la dernière ligne, `end_fill()` ordonne à la tortue de terminer (`end`) le remplissage (`fill`) du cercle avec la couleur RVB choisie.

## Une fonction pour dessiner un cercle plein

Pour simplifier nos prochaines expériences avec des couleurs différentes, créons une fonction à partir du code que nous venons d'utiliser, pour tracer un cercle plein.

---

```
>>> def moncercle(rouge, vert, bleu):
    t.color(rouge, vert, bleu)
    t.begin_fill()
    t.circle(50)
    t.end_fill()
```

---

Pour dessiner un cercle vert brillant, il suffit alors d'appeler la fonction `moncercle` avec 100 % de vert et 0 % des autres couleurs :

---

```
>>> moncercle(0, 1, 0)
```

---

Pour un vert un peu plus foncé, la moitié (0.5) de vert suffit :

---

```
>>> moncercle(0, 0.5, 0)
```

---

Pour essayer quelques couleurs RVB à l'écran, essaie de dessiner des cercles, d'abord avec le rouge complet, puis la moitié de rouge, puis le bleu complet et enfin la moitié de bleu, comme ceci :

---

```
>>> moncercle(1, 0, 0)
>>> moncercle(0.5, 0, 0)
>>> moncercle(0, 0, 1)
>>> moncercle(0, 0, 0.5)
```

---

### NOTE

*Si le canevas commence à être saturé sous un peu trop de formes de toutes sortes, tape la commande `t.reset()` pour le nettoyer de tes précédents dessins. Pour rappel aussi, tu peux déplacer la souris sans dessiner à l'aide de `t.up()`, juste avant des déplacements, puis `t.down()` pour recommencer à dessiner.*

Des combinaisons diverses de rouge, vert et bleu produisent une grande variété de couleurs, par exemple la couleur or :

---

```
>>> moncercle(0.9, 0.75, 0)
```

---

Ou du rose clair :

---

```
>>> moncercle(1, 0.7, 0.75)
```

---

Ou encore deux variantes d'orange :

---

```
>>> moncercle(1, 0.5, 0)  
>>> moncercle(0.9, 0.5, 0.15)
```

---

N'hésite pas à essayer d'autres combinaisons.

## Dessiner en noir et blanc

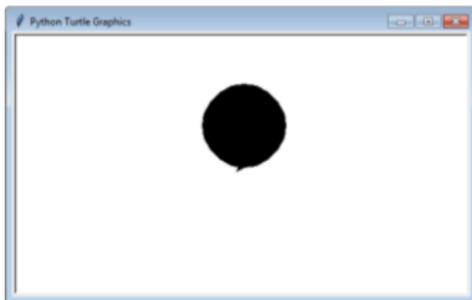
Mais alors, que se passe-t-il si tu éteins toutes les lumières ? C'est simple : tout devient noir. L'ordinateur réagit exactement de la même manière : pas de lumière signifie aucune couleur. Donc un cercle dessiné avec `0` pour les trois couleurs donne un cercle noir :

---

```
>>> moncercle(0, 0, 0)
```

---

Voici le résultat :



À l'inverse, si tu donnes 100 % aux trois couleurs, tu allumes tout et tu obtiens du blanc. La ligne suivante efface complètement le cercle noir mais également la tortue :

---

```
>>> moncercle(1, 1, 1)
```

---

## Fonction de dessin de carré

À ce stade, nous savons que, pour remplir des formes, nous devons d'abord choisir une couleur avec `color`, indiquer à la tortue de commencer à remplir, avec `begin_fill`, puis forcer le remplissage après tous les tracés, avec `end_fill`. Livrons-nous à quelques expériences de formes et de remplissages, en créant une fonction de dessin de carré, comme celle du début de ce chapitre, mais dont la taille du carré est transmise en paramètre.

---

```
>>> def moncarre(taille):
    for x in range(1, 5):
        t.forward(taille)
        t.left(90)
```

---

Pour essayer la fonction, appelle-la avec une taille de `50`, comme ceci :

---

```
>>> moncarre(50)
```

---

La tortue dessine un petit carré :



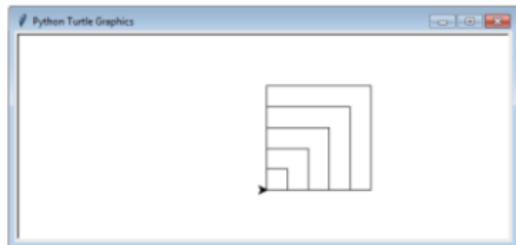
Essayons à présent avec des tailles différentes. Le code suivant dessine cinq carrés successifs de tailles `25`, `50`, `75`, `100` et `125` :

---

```
>>> t.reset()
>>> moncarre(25)
>>> moncarre(50)
>>> moncarre(75)
>>> moncarre(100)
>>> moncarre(125)
```

---

Voici comment apparaissent les carrés :



## Dessiner des carrés pleins

Pour ensuite dessiner des carrés pleins, nous réinitialisons d'abord le canevas (`reset`), démarrons le remplissage (`begin_fill`), puis appelons la fonction de dessin de carré, comme suit :

---

```
>>> t.reset()  
>>> t.begin_fill()  
>>> moncarre(50)
```

---

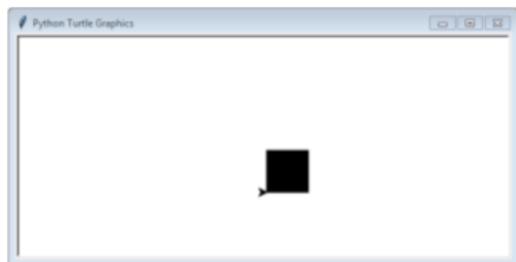
Le carré apparaît vide, jusqu'au moment du `end_fill` :

---

```
>>> t.end_fill()
```

---

Là, le carré s'affiche rempli de noir.



Modifions la fonction pour qu'elle affiche un carré, non seulement de taille donnée, mais également soit plein, soit vide. Pour cela, nous ajoutons un second paramètre et compliquons légèrement le code.

---

```
>>> def moncarre(taille, plein):
    if plein == True:
        t.begin_fill()
    for x in range(1, 5):
        t.forward(taille)
        t.left(90)
    if plein == True:
        t.end_fill()
```

---

La première ligne change la définition de la fonction pour prendre deux paramètres : `taille` et `plein`. Ensuite, nous testons la valeur de `plein` avec un `if` : si c'est `True`, nous appelons `begin_fill` pour indiquer à la tortue de remplir la forme que nous allons dessiner. Quatre passages par la boucle `for` dessinent les quatre côtés du carré avec des déplacements en avant, puis des rotations à gauche. Nous testons encore une fois `plein` avec un nouveau `if` et, si c'est `True`, un `t.end_fill` remplit le carré de couleur.

Il nous suffit ensuite de demander le dessin d'un carré plein avec cette ligne :

---

```
>>> moncarre(50, True)
```

---

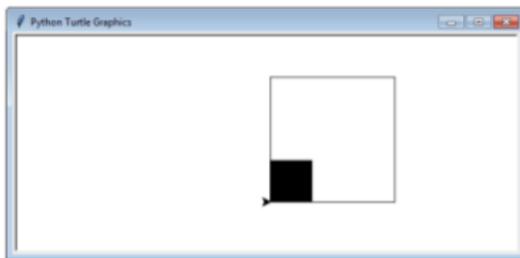
Pour afficher un carré vide, l'appel devient :

---

```
>>> moncarre(150, False)
```

---

Après ces deux appels à la fonction `moncarre`, nous obtenons l'image suivante :



Tout cela est bien joli et, finalement, il n'y a pas de raison de s'arrêter là. Il est possible de dessiner toutes sortes de formes et de les remplir de couleurs. Voyons cela.

## Dessiner des étoiles pleines

Pour notre dernier exemple, reprenons une des étoiles que nous avons dessinées plus haut et apportons-lui un peu de couleur. Le code initial de l'étoile à 18 branches était celui-ci :

---

```
>>> for x in range(1, 19):
    t.forward(100)
    if x % 2 == 0:
        t.left(175)
    else:
        t.left(225)
```

---

À partir de ce code, nous définissons une fonction `monetoile`, nous reprenons les instructions `if` de la fonction `moncarre` et nous ajoutons les paramètres `taille` et `plein`.

---

```
>>> def monetoile(taille, plein):
    if plein == True:
        t.begin_fill()
    for x in range(1, 19):
        t.forward(taille)
        if x % 2 == 0:
            t.left(175)
        else:
            t.left(225)
    if plein == True:
        t.end_fill()
```

---

Aux deux premières lignes du corps de la fonction, nous vérifions si `plein` est vrai et, s'il l'est, nous démarrons le remplissage. De même, aux deux dernières lignes, nous vérifions si `plein` est vrai, auquel cas, nous achérons le remplissage. Comme dans la fonction `moncarre`, nous utilisons le paramètre `taille` pour le passer à `t.forward` et imposer la taille de l'étoile.

Voyons ce que cela donne mais, auparavant, nous imposons la couleur or (90 % de rouge, 75 % de vert et 0 % de bleu). Il reste à appeler la fonction, par exemple avec la taille 120 et le remplissage :

---

```
>>> t.color(0.9, 0.75, 0)
>>> monetoile(120, True)
```

---

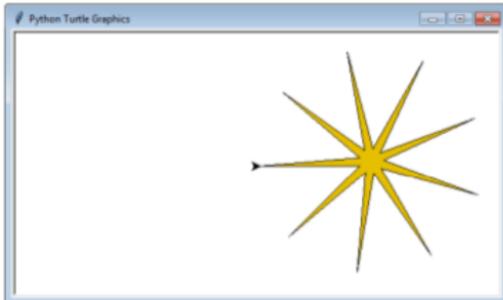
La tortue dessine une étoile remplie de couleur or :



Pour ajouter les traits noirs qui délimitent l'étoile, nous changeons la couleur en noir et nous redessinons l'étoile mais cette fois sans remplissage :

```
>>> t.color(0, 0, 0)  
>>> monetoile(120, False)
```

L'étoile achevée a des traits noirs et un remplissage or :



## Ce que tu as appris

Dans ce chapitre, tu as appris à utiliser le module `turtle` pour dessiner des formes géométriques de base à l'aide de boucles `for` et d'instructions `if`, qui permettent de contrôler les actions de la tortue à l'écran. Tu as changé la couleur du stylo de la tortue et rempli



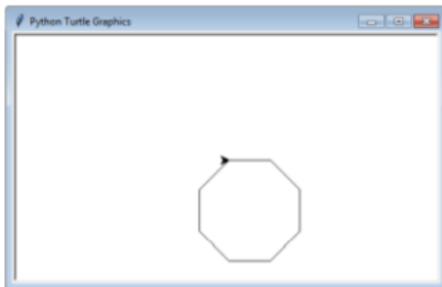
de couleur les formes qu'elle dessine. Tu as aussi réutilisé le code d'autres fonctions pour faciliter le dessin de formes avec des couleurs et des tailles différentes, le tout en un simple appel à une fonction.

## Puzzles de programmation

Voici des exercices pour te livrer à quelques expériences sur le dessin de formes particulières avec la tortue. Les réponses sont disponibles sur le site web d'accompagnement du livre.

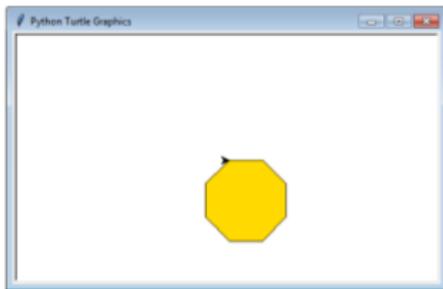
### 1. Dessiner un octogone

Tu as dessiné des étoiles, des carrés et des cercles dans ce chapitre, mais que penserais-tu de créer une fonction qui dessine une forme à huit côtés, c'est-à-dire un octogone ? Conseil : fais tourner la tortue de 45 degrés.



### 2. Dessiner un octogone plein

À présent que ta fonction dessine un octogone vide, modifie-la pour qu'elle puisse aussi dessiner un octogone plein. Inspire-toi de notre dernier exemple d'étoile remplie et avec des traits noirs pour dessiner un équivalent à ceci :



### 3. Autre fonction de dessin d'étoile

Plus fort encore, crée une fonction pour dessiner une étoile qui attend deux paramètres : la taille et le nombre de branches. La ligne de définition de la fonction sera du genre :

---

```
def dessine_etoile(taille, branches):
```

---





12

# DE MEILLEURS GRAPHISMES AVEC TKINTER

Le problème de la tortue pour dessiner, c'est qu'elle est, comment dire, très... lente ! Même lorsqu'elle est réglée à sa vitesse maximale, elle n'avance décidément pas vite. Pour une tortue, ce n'est pas grave mais, lorsqu'il s'agit de réaliser des graphismes informatiques, la lenteur devient vraiment un problème.

Les graphismes informatiques, en particulier dans les jeux, demandent de la vitesse. Si tu as une console ou si tu joues à des jeux sur ordinateur, pense aux graphismes que tu vois à l'écran. Ceux en deux dimensions (2D) sont à plat : les personnages se déplacent généralement seulement vers le haut, le bas, la gauche ou la droite. C'est ainsi dans la plupart des jeux pour Nintendo DS, les consoles PSP (*PlayStation Portable*) et les téléphones mobiles.

Dans les jeux en pseudo-trois dimensions (3D), c'est-à-dire presque en trois dimensions, les images ont l'air un peu plus vraies mais les personnages se déplacent généralement en relation avec un sol plat. C'est ce que l'on appelle les « graphismes isométriques ». Enfin, dans les jeux en trois dimensions, les images dessinées à l'écran tentent d'imiter la réalité. Que des jeux utilisent des graphismes en 2D, en pseudo-3D ou en 3D, tous ont une chose en commun : la nécessité de dessiner très vite à l'écran de l'ordinateur ou de la console.

Si tu n'as jamais créé d'animation, essaie ce petit projet pratique.

1. Prends un bloc de papier et, dans le coin inférieur droit de la première page, dessine quelque chose, par exemple un personnage en fil de fer (« filiforme »).
2. Dans le même coin de la feuille suivante, dessine le même personnage mais avec une jambe légèrement déplacée par rapport à l'image précédente.
3. À la page suivante, reproduis le même dessin mais avec la jambe légèrement encore plus déplacée.
4. Progressivement, de page en page, dessine le personnage filiforme avec des mouvements successifs.



Lorsque tu as terminé, fais défiler très vite les pages. Tu peux voir ton personnage littéralement bouger. C'est là le procédé à la base de toute animation, qu'il s'agisse de dessins animés (c'est exactement cela : des dessins animés) à la télé ou de jeux sur console et sur ordinateur. Le dessin d'une image, puis d'une suivante avec une légère modification, puis d'une suivante et ainsi de suite, donne l'illusion du mouvement. Pour qu'un dessin donne l'impression de bouger, il faut afficher chaque **image** (*frame*), c'est-à-dire chaque élément d'animation d'une manière très rapide.

Python propose plusieurs manières de créer des graphismes. En plus du module **turtle**, il est possible d'utiliser des modules externes, à installer à part, mais également le module **tkinter**, en principe déjà installé en même temps que Python. Ce module permet de créer des applications complètes, comme un traitement de texte simple, ainsi que des dessins relativement simples. Dans ce chapitre, nous étudions la création de graphismes avec **tkinter**.

## Créer un bouton à cliquer

En guise de premier exemple, utilisons `tkinter` pour créer une application très simple, avec un bouton. Entre ce code dans le shell de Python :



---

```
>>> from tkinter import *
>>> tk = Tk()
>>> btn = Button(tk, text="Clique sur moi")
>>> btn.pack()
```

---

La première ligne importe le contenu du module `tkinter`. La formule `from nom-module import *` permet d'utiliser tout le contenu d'un module `nom-module` sans utiliser son nom dans le code, au contraire de ce que nous faisions jusqu'ici, car `import turtle`, par exemple, nous obligeait à préciser le nom du module pour accéder à son contenu :

---

```
import turtle
t = turtle.Pen()
```

---

Grâce à `import *`, il n'est plus nécessaire de préciser `turtle.Pen`, comme aux chapitres 4 et 11. Dans le cas du module `turtle`, ce n'est pas vraiment indispensable mais, lorsque tu importes des modules avec de nombreuses classes et fonctions, cela permet de réduire les frappes au clavier.

---

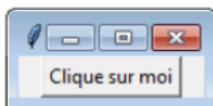
```
from turtle import *
t = Pen()
```

---

La deuxième ligne de l'exemple avec le bouton, `tk=Tk()`, crée une variable pour contenir un objet de la classe `Tk`, de la même manière que dans le cas de la création d'un objet de la classe `Pen` pour la tortue. L'objet `tk` crée une fenêtre simple dans laquelle nous pouvons ensuite ajouter d'autres choses, comme des boutons, des zones d'entrée de texte et des canevas pour y dessiner. Il s'agit de la principale classe fournie par le module `tkinter` et, sans la création d'un objet de la classe `Tk`, il n'est pas possible de réaliser le moindre dessin ou la moindre animation.

À la troisième ligne, nous créons un bouton avec `btn = Button`. Nous lui passons en paramètres la variable `tk` et le texte qu'il doit afficher, à l'aide de `(tk, text="Clique sur moi")`. Si nous avons bien ajouté un bouton à la fenêtre, c'est la ligne `btn.pack()` qui lui indique de s'afficher. Cette fonction aligne aussi tout correctement à l'écran,

dans le cas où d'autres boutons et d'autres objets doivent s'afficher. Le résultat prend l'allure suivante :



Le bouton **Clique sur moi** ne fait pas grand-chose. Tu peux cliquer dessus pendant un an, il ne ferait toujours rien. Nous devons modifier le code pour qu'il réagisse. Assure-toi de fermer cette fenêtre avant d'aller plus loin.

D'abord, créons une fonction qui affiche du texte :

---

```
>>> def bonjour():
    print('Bonjour la compagnie')
```

---

Ensuite, modifions l'exemple pour qu'il utilise cette fonction :

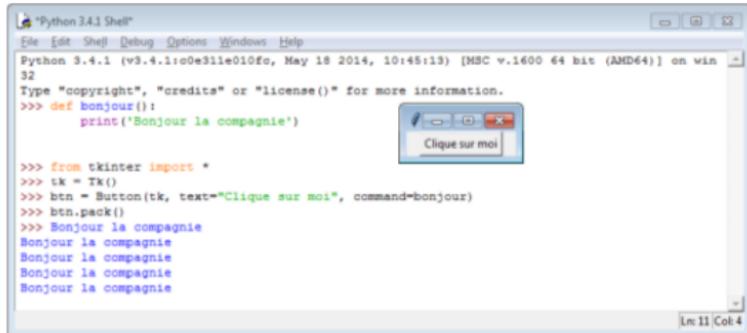
---

```
>>> from tkinter import *
>>> tk = Tk()
>>> btn = Button(tk, text="Clique sur moi", command=bonjour)
>>> btn.pack()
```

---

Note la petite modification apportée à la précédente version de ce code : nous avons ajouté le paramètre **command** à **Button**, qui indique à Python d'utiliser la fonction **bonjour** lors d'un clic sur le bouton.

À présent, tu vois s'afficher **Bonjour la compagnie** dans le shell à chaque clic sur le bouton. Dans la figure suivante, nous avons cliqué cinq fois de suite.



Parmi tous nos exemples de code écrits jusqu'ici, c'est la première fois que nous utilisons des paramètres nommés. Arrêtons-nous un instant sur ceux-ci avant d'aller plus loin dans les dessins.

## Utiliser des paramètres nommés

Jusque-là, quand nous transmettions des valeurs à une fonction, cette dernière associait la première valeur au premier paramètre et ainsi de suite, dans l'ordre de ce qu'elle trouvait entre les parenthèses. Les paramètres nommés sont des paramètres normaux mais nous utilisons explicitement leur nom auquel nous donnons la valeur. Ainsi, nous pouvons fournir les valeurs dans n'importe quel ordre.

Parfois, les fonctions possèdent un grand nombre de paramètres et il n'est pas toujours possible de donner à tous une valeur. Les paramètres nommés permettent de fournir des valeurs aux seuls paramètres qui le nécessitent.

Pour prendre un exemple, supposons que nous ayons une fonction nommée `personne` qui attend deux paramètres : une `largeur` et une `hauteur`.

---

```
>>> def personne(largeur, hauteur):
    print('Je mesure %s cm de large et %s cm de haut.' %
        (largeur, hauteur))
```

---

Habituellement, nous appelons cette fonction comme suit :

---

```
>>> personne(50, 160)
Je mesure 50 cm de large et 160 cm de haut.
```

---

En nommant les paramètres, nous pouvons modifier leur ordre dans l'appel de la fonction et donner à chacun d'eux la valeur correspondante :

---

```
>>> personne(hauteur=160, largeur=50)
Je mesure 50 cm de large et 160 cm de haut.
```

---

Les paramètres nommés deviendront de plus en plus importants à mesure que nous avancerons dans l'utilisation du module `tkinter`.

## Créer le canevas de dessin

Les boutons sont bien jolis, mais ils ne s'avèrent pas très utiles pour dessiner à l'écran. Lorsqu'il s'agit de réellement représenter quelque chose, nous devons faire appel à une zone de dessin (ou « canevas ») : un objet `canvas`. Un tel objet appartient à la classe `Canvas`, fournie par le module `tkinter`.

Lors de la création d'une zone de dessin, nous passons la largeur (`width`) et la hauteur (`height`) en pixels du canevas à Python. À part cela, le code ressemble à celui du bouton :

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=500, height=500)
>>> canvas.pack()
```

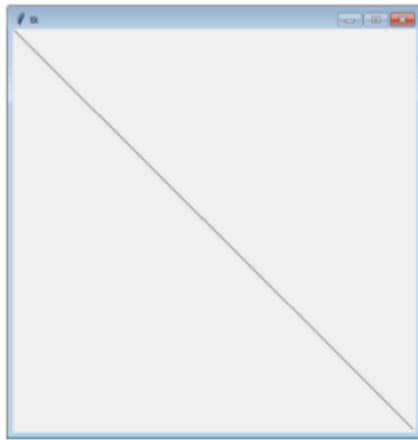
Comme dans l'exemple du bouton, une fenêtre apparaît lors de l'entrée de `tk = Tk()`. La dernière ligne compacte le canevas par `canvas.pack()`, ce qui en modifie la taille à une largeur de 500 pixels et une hauteur de 500 pixels, comme indiqué à la troisième ligne de code. Comme pour le bouton également, la fonction `pack` indique au canevas de s'afficher à l'emplacement correct dans la fenêtre. Si nous n'appelons pas cette fonction, rien ne s'affiche comme il faut.



## Dessiner des lignes

Pour tracer une ligne sur le canevas, nous utilisons des  **coordonnées** en pixels. Les coordonnées déterminent une position en pixels sur une surface. Sur un canevas de `tkinter`, les coordonnées décrivent à combien de pixels en largeur (du bord gauche vers la droite) et à combien de pixels en hauteur (du bord haut vers le bas), le pixel voulu doit venir se placer.

Ainsi, comme notre canevas fait 500 pixels de large sur 500 pixels de haut, les coordonnées du coin inférieur droit de la zone de dessin sont `(500, 500)`. Pour dessiner le trait de l'image suivante, nous commençons aux coordonnées `(0, 0)` du coin supérieur gauche et terminons aux coordonnées `(500, 500)`.



Pour indiquer les coordonnées, nous utilisons la fonction `create_line` comme suit :

---

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=500, height=500)
>>> canvas.pack()
>>> canvas.create_line(0, 0, 500, 500)
1
```

---

La fonction `create_line` renvoie 1, qui représente un identifiant. Nous reviendrons sur ce sujet plus loin. Si nous avions essayé de faire la même chose avec le module `turtle`, nous aurions dû entrer le code suivant :

---

```
>>> import turtle
>>> turtle.setup(width=500, height=500)
>>> t = turtle.Pen()
>>> t.up()
>>> t.goto(-250, 250)
>>> t.down()
>>> t.goto(500, -500)
```

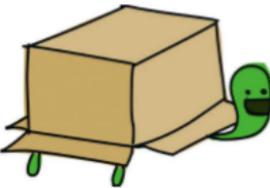
---

Donc, le code `tkinter` représente une véritable amélioration, puisqu'il est un peu plus court et plus simple.

Voyons ensuite quelques fonctions disponibles dans l'objet `canvas`, que nous pourrions utiliser pour des dessins plus intéressants.

## Dessiner des rectangles et des carrés

Pour dessiner un rectangle avec le module `turtle`, nous avons dû avancer, tourner, avancer, tourner et ainsi de suite. Finalement, nous avons pu dessiner un carré de taille variable en changeant la longueur du déplacement.



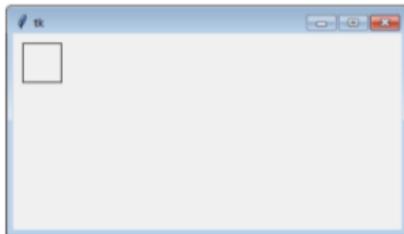
Le module `tkinter` facilite fortement le dessin d'un carré ou d'un rectangle car, tout ce qu'il faut connaître, ce sont les coordonnées des coins. Voici un exemple (ferme les autres fenêtres, à présent) :

---

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=200)
>>> canvas.pack()
>>> canvas.create_rectangle(10, 10, 50, 50)
```

---

Avec ce code, nous créons un canevas de 400 pixels de large et de 200 pixels de haut, puis nous dessinons un carré dans son coin supérieur gauche :



Les paramètres passés à `canvas.create_rectangle` à la dernière ligne de code sont les coordonnées des coins supérieur gauche et inférieur droit du carré. Ces coordonnées sont comptées comme les distances au côté gauche et au côté haut du canevas. Les deux premières coordonnées (du coin supérieur gauche du carré) sont donc à 10 pixels du bord gauche et à 10 pixels du bord haut du canevas ; elles sont données par les deux nombres `10`. Les deux dernières coordonnées (du coin inférieur droit du carré) sont à 50 pixels du bord gauche et 50 pixels du bord haut du canevas ; elles sont données par les deux `50`.

Nous appellerons ces deux jeux de coordonnées  $(x_1, y_1)$  et  $(x_2, y_2)$ . Pour dessiner un rectangle, nous augmentons la distance du deu-

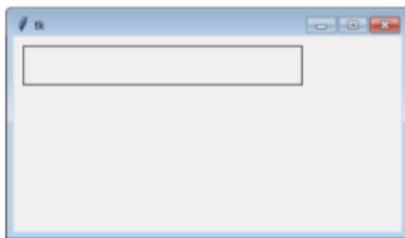
xième coin au bord du canevas, ce qui revient à augmenter la valeur du paramètre *x2*, comme ceci :

---

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=200)
>>> canvas.pack()
>>> canvas.create_rectangle(10, 10, 300, 50)
```

---

Dans cet exemple, les coordonnées de la position du coin supérieur gauche du rectangle sont (10, 10), tandis que celles du coin inférieur droit sont (300, 50). Le résultat donne un rectangle de même hauteur que le carré initial mais beaucoup large.



Nous pouvons aussi dessiner un rectangle en augmentant la distance du second coin au bord haut du canevas, ce qui revient à augmenter la valeur du paramètre *y2*. Auparavant, nous portons la hauteur du canevas à 400 pixels, pour accepter le rectangle de 300 pixels de haut :

---

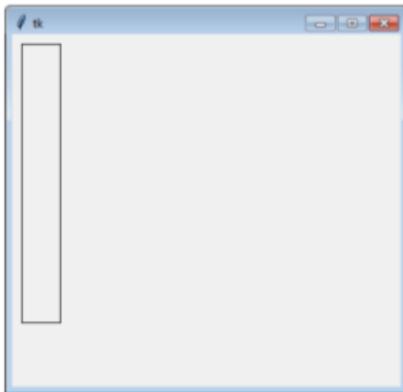
```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_rectangle(10, 10, 50, 300)
```

---

Dans cet appel de la fonction `create_rectangle`, nous disons dans l'ordre :

- aller à 10 pixels à droite, du bord gauche du canevas ;
- aller à 10 pixels vers le bas, du bord haut du canevas. C'est là le coin de départ du rectangle ;
- dessiner le rectangle à 50 pixels vers la droite ;
- dessiner le rectangle à 300 pixels vers le bas.

Le résultat final donne à peu près ceci :



### Dessiner de nombreux rectangles

Tiens ! Et si nous remplissions le canevas avec des rectangles de tailles différentes ? Nous pouvons importer le module `random` et créer une fonction qui utilise un nombre aléatoire pour les coordonnées des coins supérieur gauche et inférieur droit d'un rectangle.

Nous allons utiliser une fonction fournie par le module `random`, nommée `randrange`. Lorsque nous donnons un nombre à cette fonction, elle renvoie un entier compris entre 0 et le nombre donné (exclu). Par exemple, `randrange(10)` retourne un entier compris entre 0 et 9, tandis que `randrange(100)` retourne un nombre compris entre 0 et 99, et ainsi de suite.

Crée une nouvelle fenêtre d'IDLE avec **File>New File**, puis entre le code suivant :

---

```
from tkinter import *
import random
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()

def rectangle_aleatoire(largeur, hauteur): ❶
    x1 = random.randrange(largeur) ❷
    y1 = random.randrange(hauteur) ❸
    x2 = x1 + random.randrange(largeur) ❹
    y2 = y1 + random.randrange(hauteur) ❺
    canvas.create_rectangle(x1, y1, x2, y2) ❻
```

---

Nous définissons d'abord notre fonction `rectangle_aleatoire` ❶ qui prend deux paramètres `largeur` et `hauteur`. Ensuite, nous créons les coordonnées du coin supérieur gauche du rectangle ❷ à l'aide de la fonction `randrange` à laquelle nous passons d'abord la largeur, puis la hauteur en paramètre, avec `x1 = random.randrange(largeur)` et `y1 = random.randrange(hauteur)`, respectivement. En pratique, dans la deuxième ligne de cette fonction, nous disons « crée une variable nommée `x1` et règle sa valeur à un nombre aléatoire compris entre 0 et la valeur du paramètre `largeur` ».

Les deux lignes suivantes créent des variables pour le coin inférieur droit du rectangle ❸, en tenant compte des coordonnées du coin supérieur gauche (`x1` ou `y1`) et en ajoutant un nombre aléatoire à ces valeurs. La troisième ligne de la fonction dit en pratique « crée la variable `x2` avec comme valeur la somme d'un nombre aléatoire et de la valeur déjà calculée pour `x1` ».

Enfin, avec `canvas.create_rectangle`, nous utilisons les variables `x1`, `x2`, `y1` et `y2` pour dessiner le rectangle sur le canevas ❹.

Pour essayer la fonction `rectangle_aleatoire`, nous devons lui transmettre la largeur et la hauteur du canevas. Ajoute la ligne de code suivante sous la fonction déjà présente :

---

```
rectangle_aleatoire(400, 400)
```

---

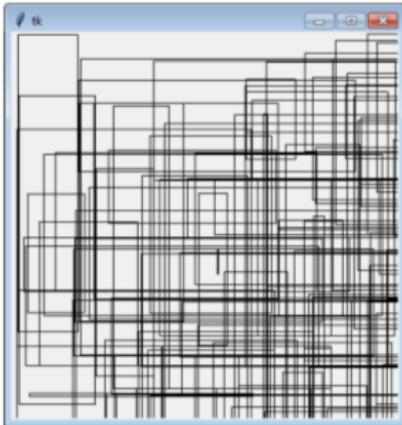
Enregistre le code de ce programme (menu **File>Save As**) sous le nom `rectangles_aleatoires.py` dans ton dossier personnel ou sur ta clé mémoire USB, puis clique sur le menu **Run>Run Module** pour exécuter le programme. Dès que cette partie du code fonctionne et affiche un rectangle à l'écran, nous passons à l'étape suivante, qui consiste à remplir l'écran de rectangles aléatoires. Pour cela, nous ajoutons une boucle qui appelle `rectangles_aleatoires` un certain nombre de fois. Essayons avec une boucle `for` de 100 rectangles au hasard. Ajoute le code suivant à la fin du programme, enregistre le fichier, puis exécute-le à nouveau :

---

```
for x in range(0, 100):
    rectangle_aleatoire(400, 400)
```

---

Le résultat est un peu confus, mais il s'agit presque d'une peinture d'art moderne.



## Définir la couleur

Bien entendu, nous devons ajouter un peu de couleur à nos graphismes. Modifions la fonction `rectangle_aleatoire` pour lui passer en paramètre supplémentaire une couleur à appliquer au rectangle (`coul_remp1`). Entre ce code dans un nouveau fichier et enregistre-le sous le nom `rectangles_en_couleurs.py`.

---

```
from tkinter import *
import random
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()

def rectangle_aleatoire(largeur, hauteur, coul_remp1):
    x1 = random.randrange(largeur)
    y1 = random.randrange(hauteur)
    x2 = x1 + random.randrange(largeur)
    y2 = y1 + random.randrange(hauteur)
    canvas.create_rectangle(x1, y1, x2, y2, fill=coul_remp1)
```

---

La fonction `create_rectangle` attend à présent le paramètre `fill`, auquel nous passons la valeur du paramètre `coul_remp1`, qui indique la couleur à utiliser pour dessiner le rectangle. Le paramètre `fill` de `create_rectangle` accepte des couleurs nommées (en anglais), comme le montre la suite du code, ce qui permet de créer facilement une série de rectangles de couleurs différentes.

Pour entrer les lignes du code suivant, n'hésite pas à utiliser le copier-coller (comme expliqué en détail au chapitre 2). Entre ce code dans le fichier `rectangles_en_couleurs.py`, après la fonction :



---

```
rectangle_aleatoire(400, 400, 'green')      # vert
rectangle_aleatoire(400, 400, 'red')          # rouge
rectangle_aleatoire(400, 400, 'blue')         # bleu
rectangle_aleatoire(400, 400, 'orange')
rectangle_aleatoire(400, 400, 'yellow')       # jaune
rectangle_aleatoire(400, 400, 'pink')          # rose
rectangle_aleatoire(400, 400, 'purple')        # pourpre
rectangle_aleatoire(400, 400, 'violet')
rectangle_aleatoire(400, 400, 'magenta')
rectangle_aleatoire(400, 400, 'cyan')          # bleu turquoise
rectangle_aleatoire(400, 400, '#ffd800')      # or
```

---

Comme ces noms de couleurs sont en anglais, nous avons mis les traductions françaises, en commentaires, c'est-à-dire précédés du symbole dièse (#). Les commentaires contiennent du texte ignoré par Python, placé là pour faciliter la lecture du code. IDLE les colore en rouge.

Ces couleurs nommées affichent généralement les couleurs attendues, mais certaines autres peuvent provoquer des messages d'erreur (selon que tu utilises Windows, OS X ou Linux).

Que faire si tu cherches une teinte qui ne correspond pas exactement à une couleur nommée ? Si tu te le rappelles, au chapitre 11, nous avons défini la couleur du stylo de la tortue à l'aide de pourcentages des couleurs rouge, vert et bleu. Le réglage de la quantité de chacune de ces couleurs primaires pour obtenir une combinaison est légèrement plus compliqué sous `tkinter`, mais tu vas rapidement comprendre comment il fonctionne.

Par exemple, pour obtenir la couleur or avec le module `turtle`, nous avons mélangé 90 % de rouge, 75 % de vert et 0 % de bleu. Avec `tkinter`, nous obtenons exactement la même couleur à l'aide de la ligne suivante :

---

```
random_rectangle(400, 400, '#ffd800')
```

---

Le symbole dièse (#) avant la valeur `ffd800` indique à Python que nous fournissons un nombre **hexadécimal**. Le système hexadécimal est une manière très habituelle de représenter des nombres dans le

monde de la programmation. Il utilise une base 16 (avec les chiffres 0 à 9 et les lettres A à F) au lieu de la base 10 (chiffres de 0 à 9) du système décimal. Si tu n'as jamais vu la notion de bases en mathématiques, sache juste que tu peux convertir un nombre décimal en hexadécimal à l'aide de l'espace réservé de format `%x` dans une chaîne (voir la section « Insérer des valeurs dans des chaînes » du chapitre 3). Ainsi, pour convertir le nombre décimal 15 en hexadécimal, écris :

---

```
>>> print('%x' % 15)
f
```

---

Pour imposer au moins deux chiffres au nombre hexadécimal généré, nous pouvons modifier légèrement l'espace réservé de format, comme ceci :

---

```
>>> print('%02x' % 15)
0f
```

---

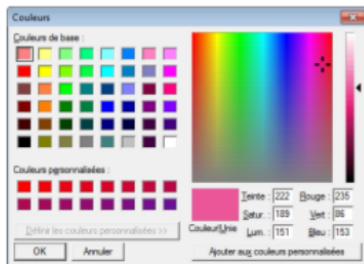
Le module `tkinter` offre un moyen bien pratique d'obtenir une valeur hexadécimale de couleur, sous la forme d'une fonction de sélection de couleur, `colorchooser`. Dans le shell de Python, entre les lignes suivantes :

---

```
>>> from tkinter import *
>>> colorchooser.askcolor()
```

---

Ceci fait apparaître le sélecteur de couleur (dont la présentation dépend de Windows, OS X ou Linux) :



Sélecteur de couleur, sous Windows 7



Sélecteur de couleur, sous Ubuntu

Lorsque tu sélectionnes une couleur et cliques sur `OK`, un tuple apparaît dans le shell, qui contient un autre tuple avec trois nombres et une chaîne :

---

```
>>> colorchooser.askcolor()
((235.91796875, 86.3359375, 153.59765625), '#eb5699')
```

---

Les trois nombres représentent les quantités de rouge, de vert et de bleu. Dans `tkinter`, la quantité de chaque couleur primaire dans une combinaison est représentée par un nombre entre 0 et 255 (ce qui diffère du pourcentage utilisé dans le module `turtle`). La chaîne du tuple contient la version en hexadécimal de ces trois nombres.

Tu peux ensuite copier et coller la chaîne pour l'utiliser directement ou attribuer la totalité du tuple à une variable, puis utiliser l'indice de position de la valeur hexadécimale.

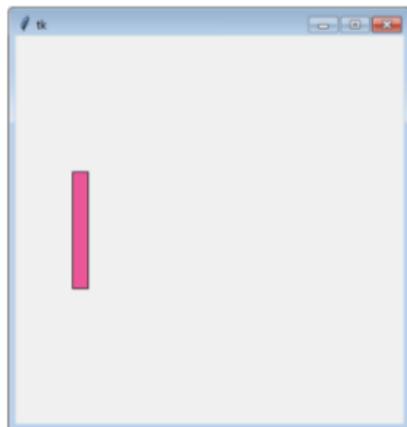
Reprends le contenu du fichier `rectangles_en_couleurs.py` et enregistre-le (menu `File>Save As`) sous le nom `rectangle_colorchooser.py` (tu peux éventuellement supprimer les autres appels à la fonction `rectangle_aleatoire`), puis ajoute ces lignes de code à la fin :

---

```
c = colorchooser.askcolor()
rectangle_aleatoire(400, 400, c[1])
```

---

Enregistre le programme et exécute-le. Tu obtiens quelque chose du genre :



## Dessiner des arcs

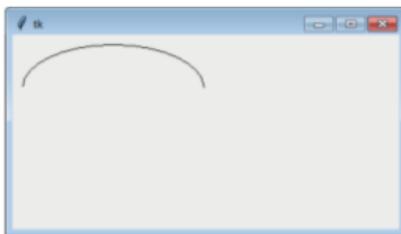
Un arc est un segment (une portion) de la circonférence d'un cercle ou d'un autre type de courbe. Pour en dessiner un avec `tkinter`, il est

nécessaire de le faire à l'intérieur d'un rectangle, avec la fonction `create_arc`, dans une ligne du genre :

---

```
canvas.create_arc(10, 10, 200, 100, extent=180, style=ARC)
```

---



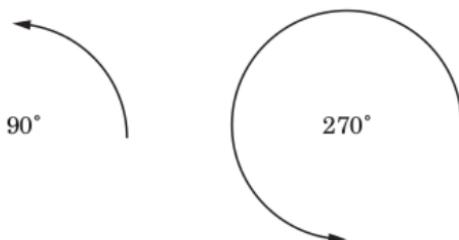
Si tu as fermé toutes les fenêtres de `tkinter` ou redémarré IDLE, assure-toi de réimporter d'abord `tkinter`, puis de recréer le canevas, avec tout le code suivant :

---

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_arc(10, 10, 200, 100, extent=180, style=ARC)
```

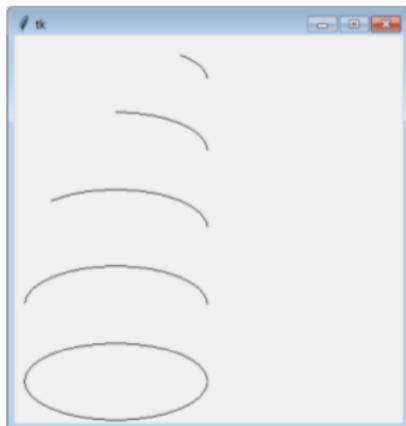
---

La dernière ligne place le coin supérieur gauche du rectangle aux coordonnées `(10, 10)` – à 10 pixels du bord gauche et à 10 pixels du bord supérieur du canevas –, et son coin inférieur droit aux coordonnées `(200, 100)` – soit à 200 pixels du bord gauche et 100 pixels du bord supérieur du canevas. Le paramètre suivant, `extent`, sert à indiquer le nombre de degrés d'angle de l'arc. Au chapitre 4, nous avons vu que les degrés mesurent la distance à parcourir autour d'un cercle. Voici deux exemples d'arcs, où nous voyageons respectivement sur 45 degrés et 270 degrés autour d'un cercle :



L'exemple suivant dessine plusieurs arcs de haut en bas de la fenêtre, pour que tu puisses voir ce qui se produit avec plusieurs valeurs différentes en degrés pour la fonction `create_arc` :

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_arc(10, 10, 200, 80, extent=45, style=ARC)
>>> canvas.create_arc(10, 80, 200, 160, extent=90, style=ARC)
>>> canvas.create_arc(10, 160, 200, 240, extent=135, style=ARC)
>>> canvas.create_arc(10, 240, 200, 320, extent=180, style=ARC)
>>> canvas.create_arc(10, 320, 200, 400, extent=359, style=ARC)
```



#### NOTE

La raison d'utiliser 359 degrés pour le « cercle » final, au lieu de 360, vient de ce que `tkinter` considère que 360 degrés sont identiques à 0 degré, ce qui signifie qu'il ne dessineraient rien du tout. Nous sommes donc obligés de tricher un peu dans ce cas-là.

## Dessiner des polygones

Par définition, un polygone est une forme qui possède plusieurs angles, donc plusieurs côtés. Tu connais les polygones réguliers que sont les triangles, les rectangles, les carrés, les pentagones, les hexagones, et ainsi de suite, mais il en existe d'autres aux côtés inégaux, avec un nombre plus important de côtés, aux formes bizarres.

Lorsque nous dessinons un polygone avec `tkinter`, nous devons fournir les coordonnées de chaque point (angle) de la forme. Voici par exemple comment tracer un triangle :

---

```
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
canvas.create_polygon(10, 10, 100, 10, 100, 110, fill="", outline="black")
```

---

Cet exemple dessine un triangle aux traits noirs (`outline="black"`), d'abord avec les coordonnées  $x$  et  $y$  (`10, 10`), puis il va jusqu'en (`100, 10`), puis en (`100, 110`) et il referme la forme jusqu'au point de départ. Voici le résultat :



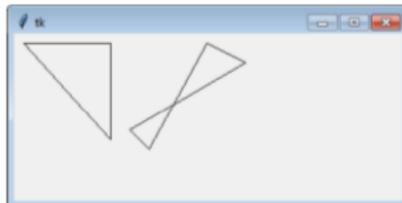
Nous pouvons ajouter un polygone irrégulier, une forme aux angles et côtés irréguliers, avec le code suivant :

---

```
canvas.create_polygon(200, 10, 240, 30, 120, 100, 140, 120, fill="", outline="black")
```

---

Cette ligne de code débute le tracé aux coordonnées (`200, 10`), va en (`240, 30`), puis en (`120, 100`) et finalement en (`140, 120`). `tkinter` rejoint le point de départ et referme automatiquement la forme. Le résultat des deux tracés est le suivant :



## Afficher du texte

S'il est possible de dessiner des formes, la fonction `create_text` permet aussi d'écrire du texte dans le canevas. Cette fonction accepte deux coordonnées, les positions en *x* et *y* du texte, ainsi qu'un paramètre nommé pour le texte à afficher. Dans le code suivant, nous créons un canevas, puis nous affichons une phrase placée aux coordonnées (150, 100). Enregistre ce code sous le nom de fichier `dessiner_texte.py`.

---

```
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=500, height=400)
canvas.pack()
canvas.create_text(150, 100, text='Soyez plutôt maçon')
```

---

La fonction `create_text` accepte d'autres paramètres intéressants, comme la couleur de remplissage du texte. La ligne suivante appelle la fonction `create_text` avec les coordonnées (170, 120), le texte à afficher, puis la couleur de remplissage rouge (`fill='red'`) :

---

```
canvas.create_text(170, 120, text='si c\'est votre talent.', fill='red')
```

---

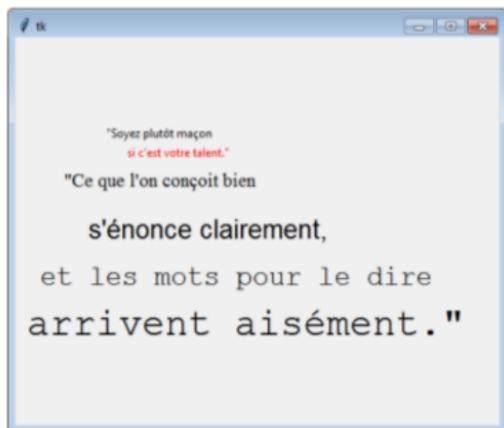
Tu peux également indiquer la police (le type de forme de caractère, *font* en anglais) du texte affiché sous forme d'un tuple contenant le nom de la police et la taille du texte. Par exemple, pour la police Times et la taille 20, le tuple devient (`'Times', 20`). Les exemples de code suivants montrent du texte en police Times et taille 15, en police Helvetica de taille 20 et en Courier de tailles 22, puis 30.

---

```
canvas.create_text(150, 150, text='Ce que l'on conçoit bien', font=('Times', 15))
canvas.create_text(200, 200, text='s\'énonce clairement,', font=('Helvetica', 20))
canvas.create_text(230, 250, text='et les mots pour le dire', font=('Courier', 22))
canvas.create_text(240, 300, text='arrivent aisément.', font=('Courier', 30))
```

---

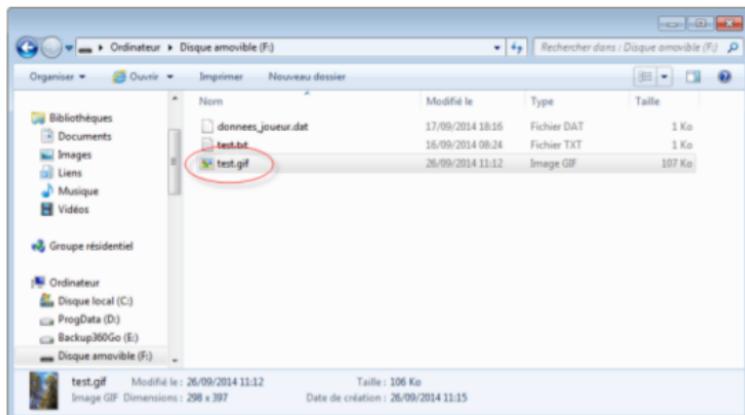
Et voici le résultat (citation de Nicolas Boileau) :



## Afficher des images

Pour afficher une image sur un canevas avec `tkinter`, charge-la d'abord, puis utilise la fonction `create_image` sur l'objet canevas.

Toute image que tu charges doit se situer dans un dossier accessible à Python. Dans l'exemple qui suit, sous Windows, nous avons placé notre image `test.gif` dans le dossier `F:\`, qui correspond au dossier racine d'une clé mémoire USB, mais tu peux la placer où tu veux, sauf en `C:\`, pour des raisons de sécurité.



Sur un Mac ou sous Linux, place l'image dans ton dossier utilisateur.

**NOTE**

Avec `tkinter`, il n'est possible de charger que des images GIF, c'est-à-dire des images dont le fichier porte l'extension `.gif`. Pour afficher d'autres types d'images, comme des images PNG (`.png`) ou JPEG (`.jpg`), il est nécessaire d'utiliser un autre module, par exemple Python imaging library, ou PIL (<http://www.pythongui.com/products/pil/>).

Pour afficher l'image `test.gif`, nous écrivons :

---

```
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
mon_image = PhotoImage(file='f:\\test.gif')
canvas.create_image(0, 0, anchor=NW, image=mon_image)
```

---

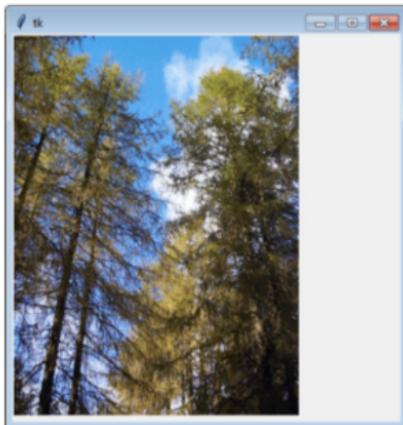
Les quatre premières lignes préparent le canevas comme dans les exemples précédents. La cinquième charge l'image dans la variable `mon_image`. La sixième crée un objet `PhotoImage` à partir du fichier `test.gif`, situé dans `f:\`. Si tu as enregistré l'image sur ton bureau, tu devras plutôt pointer le `PhotoImage` vers celui-ci, comme ceci :

---

```
mon_image = PhotoImage(file='c:\\Users\\nom_utilisateur\\Desktop\\test.gif')
```

---

Lorsque l'image est chargée dans la variable, la dernière ligne l'affiche à l'aide de la fonction `create_image`. Le paramètre `anchor=NW` indique à la fonction d'utiliser le coin supérieur gauche (`NW` signifie *northwest*, ou nord-ouest) de l'image comme point de départ de l'affichage et les coordonnées `(0, 0)` précisent où doit être placé ce point. Si `anchor` n'est pas précisé, la fonction prend par défaut le centre de l'image comme point de départ de l'affichage. Le dernier paramètre, `image`, pointe vers la variable de l'image chargée. Voici le résultat :



(© image : [www.freeimages.co.uk](http://www.freeimages.co.uk))

## Créer une animation de base

Jusqu'ici, nous n'avons parlé que de création de dessins statiques, c'est-à-dire d'images qui ne bougent pas. Voyons comment créer une animation.

L'animation n'est pas vraiment la spécialité du module `tkinter`, mais il en gère les bases. Par exemple, nous pouvons créer un triangle plein et le déplacer en travers de l'écran. Enregistre le code suivant sous le nom de fichier `animation_base1.py` :

---

```
import time
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=200)
canvas.pack()
canvas.create_polygon(10, 10, 10, 60, 50, 35)
for x in range(0, 60):
    canvas.move(1, 5, 0)
    tk.update()
    time.sleep(0.05)
```

---

À l'exécution, le triangle part de la gauche et traverse l'écran de proche en proche, jusqu'à la fin de son chemin :



Voyons comment cela fonctionne. Déjà, nous retrouvons les quatre lignes qui importent `tkinter`, définissent le canevas et l'affichent. Le triangle est créé avec la fonction suivante :

---

```
canvas.create_polygon(10, 10, 10, 60, 50, 35)
```

---

**NOTE**

*Quand tu entres cette ligne directement dans le shell de Python, un nombre s'affiche dans celui-ci. Il s'agit de l'identifiant du polygone. Il nous permettra par la suite d'y faire référence, comme dans l'exemple qui suit.*

Ensuite, nous créons une boucle `for` pour compter de 0 à 59 : `for x in range(0, 60):`. Le bloc de code à l'intérieur de la boucle déplace le triangle horizontalement à l'écran. La fonction `canvas.move` permet en effet de déplacer n'importe quel objet dessiné dans le canevas par l'ajout de valeurs à ses coordonnées `x` et(ou) `y`. Dans l'exemple, `canvas.move(1, 5, 0)` déplace l'objet d'identifiant 1 (celui du triangle) de 5 pixels vers la droite et de 0 pixel vers le bas. Pour reculer au contraire, c'est simple, nous devrions utiliser la fonction `canvas.move(1, -5, 0)`.



La fonction `tk.update` oblige `tkinter` à mettre l'écran à jour, c'est-à-dire à tout redessiner. Si nous ne l'avions pas utilisée, `tkinter` aurait attendu que la boucle se termine avant de déplacer le triangle, ce qui aurait eu pour effet de ne l'afficher qu'à son emplacement final et il aurait donné l'impression de sauter à celui-ci, au lieu de se déplacer lentement sur le canevas. La dernière ligne de la boucle, `time(0.05)` indique à Python de se mettre en veille pendant un vingtième de seconde (0,05 seconde), avant de poursuivre. C'est la raison pour laquelle la toute première ligne de code importe le module `time`.

Pour que le triangle se déplace en diagonale à l'écran, il suffit de modifier l'appel à `move` avec les paramètres `(1, 5, 5)`. Pour essayer, ouvre une nouvelle fenêtre d'IDLE (menu `File>New File`) et enregistre-la sous le nom de fichier `animation_base2.py`, puis entre le code suivant :

---

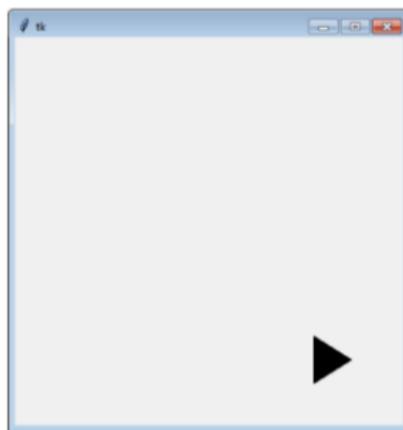
```
import time
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
canvas.create_polygon(10, 10, 10, 60, 50, 35)
for x in range(0, 60):
    canvas.move(1, 5, 5)
    tk.update()
    time.sleep(0.05)
```

---

Ce code diffère de l'original de deux manières :

- il porte la hauteur du canevas à 400 au lieu de 200 ;
- il ajoute 5 à la coordonnée *y* dans `canvas.move(1, 5, 5)`.

Une fois ce code enregistré et exécuté, la position finale du triangle devient la suivante :



Pour ramener le triangle en diagonale à son emplacement de départ, il suffit ensuite d'ajouter la boucle suivante à la fin du fichier :

---

```
for x in range(0, 60):
    canvas.move(1, -5, -5)
    tk.update()
    time.sleep(0.05)
```

---

## Réagir à un événement

Nous pouvons faire réagir le triangle au fait que l'utilisateur appuie sur une touche, grâce à la liaison d'une action à un événement. Les **événements** (objet `event`) sont des choses qui se produisent pendant l'exécution d'un programme, comme l'utilisateur qui déplace la souris, presse une touche ou ferme une fenêtre. Il est possible d'indiquer à `tkinter` de surveiller ces événements et d'agir ensuite en réponse.

Pour gérer les événements, c'est-à-dire faire en sorte que Python fasse quelque chose quand un événement se produit, il nous faut d'abord créer une fonction. La partie liaison apparaît quand nous indiquons à `tkinter` quelle fonction spécifique il faut lier (ou associer) à un événement déterminé. Autrement dit, elle sera automatiquement appelée par `tkinter` pour gérer cet événement.

Par exemple, pour faire bouger le triangle quand l'utilisateur presse la touche `Entrée`, nous pouvons définir la fonction suivante :

---

```
def bouger_triangle(événement):
    canvas.move(1, 5, 0)
```

---

Cette fonction ne prend qu'un seul paramètre, `événement (event)`, que `tkinter` utilise pour envoyer à la fonction des informations à propos de l'événement qui s'est produit. Nous devons ensuite dire à `tkinter` d'utiliser cette fonction pour un événement particulier, avec la fonction `bind_all` du canevas. Le code complet devient :

---

```
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
canvas.create_polygon(10, 10, 10, 60, 50, 35)
def bouger_triangle(événement):
    canvas.move(1, 5, 0)
canvas.bind_all('<KeyPress-Return>', bouger_triangle)
```

---

À la dernière ligne, le premier paramètre de cette fonction décrit l'événement que `tkinter` doit surveiller. Ici, il s'agit d'une pression sur la touche `Entrée` ou `Retour` du clavier normal ou du pavé numérique,

qui correspond à `<KeyPress-Return>`. Nous indiquons donc à `tkinter` d'appeler la fonction `bouger_triangle`, chaque fois que cet événement de pression de touche se produit. Exécute ce programme, clique de la souris dans le canevas, puis appuie plusieurs fois sur la touche `Entrée` du clavier, pour voir le triangle avancer.

Nous pouvons aussi changer la direction de déplacement du triangle selon des pressions de touches différentes, par exemple les touches des flèches du curseur. Il suffit de modifier la fonction `bouger_triangle` comme suit :



---

```
def bouger_triangle(événement):
    if événement.keysym == 'Up':
        canvas.move(1, 0, -3)
    elif événement.keysym == 'Down':
        canvas.move(1, 0, 3)
    elif événement.keysym == 'Left':
        canvas.move(-3, 0, 0)
    else:
        canvas.move(1, 3, 0)
```

---

Il est préférable d'entrer ce code dans une nouvelle fenêtre d'`IDLE` et de l'enregistrer lorsqu'il est complet, sous le nom de fichier `événements2.py`.

L'objet événement passé à `bouger_triangle` contient plusieurs variables, dont une, nommée `keysym` (qui signifie « symbole de touche »), est une chaîne contenant la valeur de la touche qui vient d'être pressée. La ligne `if événement.keysym == 'Up':` dit simplement que, si la variable `keysym` de l'événement contient la chaîne '`Up`' (qui correspond à la touche de flèche `Haut`), alors il faut déplacer l'objet avec les paramètres `(1, 0, -3)`, comme l'indique la ligne suivante. Si la chaîne de `keysym` contient '`Down`', qui correspond à la touche de flèche `Bas`, nous appelons `canvas.move` avec les paramètres `(1, 0, 3)`, et ainsi de suite.

Pour rappel, le premier paramètre de `move` est l'identifiant de la forme dessinée sur le canevas, le deuxième, la valeur à ajouter à la coordonnée en *x* (horizontale) et le troisième, la valeur à ajouter à la coordonnée en *y* (verticale).

Il nous reste à indiquer à `tkinter` qu'il doit appeler la fonction `bouger_triangle` pour gérer les événements de nos quatre touches différentes (`Haut`, `Bas`, `Gauche`, `Droite`). Le code suivant reprend le programme complet, tel qu'il apparaît à ce stade.

---

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

canvas = Canvas(tk, width=400, height=400)
canvas.pack()
canvas.create_polygon(10, 10, 10, 60, 50, 35)

def bouger_triangle(événement):
    ❶    if événement.keysym == 'Up':
    ❷        canvas.move(1, 0, -3)
    ❸    elif événement.keysym == 'Down':
    ❹        canvas.move(1, 0, 3)
    ❺    elif o.keysym == 'Left':
    ❻        canvas.move(1, -3, 0)
    ❼    else:
    ❽        canvas.move(1, 3, 0)

canvas.bind_all('<KeyPress-Up>', bouger_triangle)
canvas.bind_all('<KeyPress-Down>', bouger_triangle)
canvas.bind_all('<KeyPress-Left>', bouger_triangle)
canvas.bind_all('<KeyPress-Right>', bouger_triangle)
```

---

La première ligne du corps de la fonction `bouger_triangle` vérifie si la variable `keysym` contient '`Up`', en ❶. Si c'est le cas, en ❷, nous déplaçons le triangle vers le haut à l'aide de la fonction `move`, à laquelle nous passons les paramètres `(1, 0, -3)`. Le premier paramètre est l'identifiant du triangle, le deuxième la quantité de déplacement horizontal, or, comme nous ne déplaçons pas le triangle horizontalement, ce paramètre a la valeur `0`. Le troisième paramètre correspond à la valeur du déplacement vertical, or, comme nous montons, il faut une valeur négative, de `-3` pixels.

La ligne en ❸ vérifie que `keysym` contient '`Down`', qui correspond à la flèche `Bas`, et si c'est le cas, nous déplaçons, en ❹, le triangle de `3` pixels vers le bas (donc `+3`). La dernière vérification a lieu en ❺, pour la valeur de `keysym 'Left'` (flèche `Gauche`), et si c'est le cas, nous déplaçons le triangle de `-3` pixels ❻, donc vers la gauche. Si `keysym` ne correspond à aucune des valeurs précédentes, alors le `else` final en ❼ déplace le triangle vers la droite en ⽿.

## Autres façons d'utiliser l'identifiant

Chaque fois que nous utilisons une fonction de la série `create_` du canevas, comme `create_polygon` ou `create_rectangle`, elle renvoie un identifiant. Nous pouvons utiliser ce numéro d'identifiant avec

d'autres fonctions du canevas, comme nous l'avons fait avec la fonction `move` :

---

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_polygon(10, 10, 10, 60, 50, 35)
1
>>> canvas.move(1, 5, 0)
```

---

Le problème de cet exemple réside dans le fait que `create_polygon` ne renvoie pas toujours 1. Si nous créons d'autres formes dans le même canevas, le nombre renvoyé devient alors 2, 3 ou même 100 s'il y a lieu, selon le nombre de formes que nous avons dessinées. Si nous modifions légèrement le code pour stocker ce nombre dans une variable, au lieu d'écrire simplement 1 pour l'identifiant dans les fonctions de déplacement, alors le code peut fonctionner, quel que soit l'identifiant renvoyé :

---

```
>>> mon_triangle = canvas.create_polygon(10, 10, 10, 60, 50, 35)
>>> canvas.move(mon_triangle, 5, 0)
```

---

La fonction `move` déplace des objets différents à l'écran en utilisant leur identifiant. Toutefois, d'autres fonctions de canevas existent qui modifient aussi les objets dessinés. Ainsi, la fonction `itemconfig` du canevas sert à changer certains paramètres d'une forme, comme ses couleurs de remplissage et de trait.

Supposons que nous créions un triangle rouge :

---

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> mon_triangle = canvas.create_polygon(10, 10, 10, 60, 50, 35,
fill='red')
```

---

Nous pouvons ensuite changer la couleur du remplissage (`fill`) du triangle avec `itemconfig`, en lui passant l'identifiant du triangle en premier paramètre. Le code suivant dit « changer en bleu (`blue`) la couleur de remplissage de l'objet identifié par le nombre présent dans la variable `mon_triangle` » :

---

```
>>> canvas.itemconfig(mon_triangle, fill='blue')
```

---

Nous pouvons aussi changer de couleur de trait (`outline`), en utilisant de nouveau l'identifiant du triangle en premier paramètre :

```
>>> canvas.itemconfig(mon_triangle, outline='red')
```

Plus loin, nous découvrirons encore d'autres modifications applicables à un dessin, comme la possibilité de le masquer et de le rendre à nouveau visible. Ces modifications de dessins s'avéreront utiles lorsque nous commencerons à écrire des jeux au chapitre suivant.



## Ce que tu as appris

Dans ce chapitre, tu as utilisé le module `tkinter` pour dessiner des formes géométriques simples sur un canevas, pour afficher une image et effectuer des animations de base. Tu as appris à lier des événements pour qu'un dessin réagisse aux pressions de touches d'un utilisateur, ce qui sera bien utile lorsque nous entamerons la programmation d'un jeu. Tu as découvert que les fonctions `create` de `tkinter` renvoient un numéro d'identification, qui peut servir ensuite à modifier les formes après leur création, mais aussi à les déplacer à l'écran ou à changer leurs couleurs.

## Puzzles de programmation

Essaie de réaliser les exercices suivants, qui jouent avec le module `tkinter` et les animations simples. Les réponses sont disponibles dans le site web d'accompagnement du livre.

### 1. Remplir l'écran de triangles

Crée un programme à partir de `tkinter` pour remplir l'écran de triangles. Modifie ensuite le code pour remplir l'écran de triangles de tailles et de couleurs de remplissage différentes.

### 2. Le triangle mobile

Modifie le code du triangle mobile (voir section « Crée une animation de base » de ce chapitre) pour qu'il se déplace à l'écran vers la droite, puis vers le bas, puis vers la gauche, puis vers le haut jusqu'à la position de départ.

### 3. La photo mobile

Essaie d'afficher une photo de toi-même sur un canevas à l'aide de [tkinter](#). Assure-toi que c'est bien une image GIF ! Peux-tu la faire se déplacer de gauche à droite à l'écran ?

PARTIE 2

# REBONDIR !







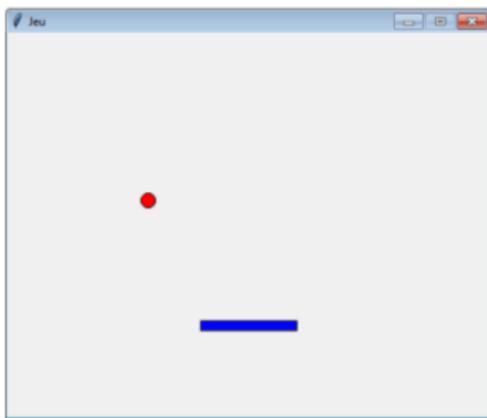
13

# DÉBUTER TON PREMIER JEU : REBONDIR !

Jusqu'ici, nous avons vu les bases de la programmation. Tu as appris à utiliser des variables pour stocker des informations, les instructions `if` pour établir des conditions et les boucles `for` pour répéter du code. Tu sais désormais comment créer des fonctions pour réutiliser du code et exploiter des classes et des objets pour découper ton code en de petits extraits plus faciles à comprendre. Tu as appris aussi à dessiner des graphismes à l'écran avec les modules `turtle` et `tkinter`. Il est temps, à présent, de tirer parti de toutes ces connaissances pour créer ton premier jeu.

## Frapper la balle magique

Nous allons développer un jeu avec une balle magique et une raquette. La balle vole à l'écran et le joueur doit la faire rebondir sur la raquette. Si, en retombant, la balle dépasse la raquette et touche le bas de l'écran, la partie est terminée. Voici un aperçu du jeu achevé :



Ce jeu peut paraître un peu simple, mais le code renferme quelques astuces et se distingue de tout ce que nous avons réalisé jusqu'ici ; il y a en effet de nombreuses choses à gérer. Ainsi, il faut animer la raquette et la balle, puis détecter si celle-ci touche les murs ou la raquette.

Dans ce chapitre, nous entamons la création du jeu, avec la génération du **canevas** et de la balle magique. Au chapitre suivant, nous le terminerons en lui ajoutant la raquette.

## Créer le canevas du jeu

Pour créer ce jeu, commence par ouvrir une nouvelle fenêtre IDLE au départ du shell Python (menu **File>New File**). Dans cette fenêtre, importe **tkinter** et crée un **canevas** sur lequel tu pourras dessiner :

---

```
from tkinter import *
import random
import time
tk = Tk()
```

```
tk.title("Jeu")
tk.resizable(0, 0)
tk.wm_attributes("-topmost", 1)
canvas = Canvas(tk, width=500, height=400, bd=0, highlightthickness=0)
canvas.pack()
tk.update()
```

---

Ceci diffère quelque peu des exemples précédents. Nous importons tout d'abord les modules `random` et `time` pour pouvoir les utiliser par la suite dans le code.

La ligne `tk.title("Jeu")` se sert de la fonction `title` de l'objet `tk` créé avec `tk = Tk()` pour donner un titre à la fenêtre du canevas. La ligne suivante fait appel à la fonction `resizable` pour imposer une taille fixe à la fenêtre : les paramètres `0, 0` indiquent que la taille de la fenêtre ne peut être modifiée ni horizontalement, ni verticalement. Ensuite, nous appelons `wm_attributes` pour préciser à `tkinter` de placer la fenêtre du canevas au-dessus de toutes les autres fenêtres, c'est-à-dire à l'avant-plan (`"-topmost"`).

Note que, lors de la création de l'objet canevas avec `canvas =`, nous utilisons des paramètres supplémentaires par rapport aux exemples précédents. Ainsi, `bd=0` et `highlightthickness=0` garantissent qu'il n'y aura aucune bordure autour du canevas, ce qui donne à notre fenêtre un aspect plus adapté à notre écran de jeu.

La ligne `canvas.pack()` oblige le canevas à se redimensionner en fonction des paramètres de largeur et de hauteur fournis à la ligne précédente. Enfin, `tk.update()` indique à `tkinter` de s'initialiser lui-même en vue de l'animation du jeu. Sans cette dernière ligne, rien ne fonctionnerait comme prévu.

Enregistre le code à mesure que tu avances, avec un nom de fichier explicite, par exemple `rebondir.py`.



## Créer la classe de la balle

Passons à la création de la classe de la balle. Pour débuter le code, il faut que celle-ci se dessine d'elle-même sur le canevas. Voici les étapes à suivre.

1. Crée une classe appelée `Balle`, qui prend des paramètres pour le canevas et la couleur de la balle à dessiner.

- Enregistre le canevas dans une variable d'objet, parce que nous devons dessiner la balle dessus.
- Trace un cercle plein sur le canevas à l'aide de la valeur du paramètre de couleur en tant que couleur de remplissage.
- Mémorise l'identifiant que renvoie `tkinter` lorsqu'il dessine le cercle (un ovale, en fait), parce que nous en avons besoin pour déplacer la balle à l'écran.
- Déplace l'ovale au milieu du canevas.

Ce code vient s'ajouter directement après les trois premières lignes du fichier (juste après `import time`).

---

```

from tkinter import *
import random
import time

❶ class Balle:
❷     def __init__(self, canvas, couleur):
❸         self.canvas = canvas
❹         self.id = canvas.create_oval(10, 10, 25, 25, fill=couleur)
❺         self.canvas.move(self.id, 245, 100)

❻     def dessiner(self):
❼         pass

```

---

Nous nommons notre classe `Balle` (❶) et créons ensuite (❷) la fonction d'initialisation (comme expliqué au chapitre 8), qui prend les paramètres `canvas` et `couleur`. Puis nous définissons la variable d'objet `canvas` avec la valeur du paramètre `canvas` (❸). Nous appelons la fonction `create_oval` avec cinq paramètres (❹) : les coordonnées  $x$  et  $y$  du coin supérieur gauche (`10, 10`), les coordonnées  $x$  et  $y$  du coin inférieur droit (`25, 25`) et, enfin, la couleur de remplissage de l'ovale.

La fonction `create_oval` renvoie un identifiant de la forme dessinée, que nous mémorisons dans la variable d'objet `id`. Ensuite, nous déplaçons (❺) l'ovale vers le milieu du canevas (`245, 100`). Ce dernier sait quel objet déplacer, parce que nous lui indiquons l'identifiant de la forme (la variable d'objet `id`).

Les deux dernières lignes de la classe `Balle` créent une fonction `dessiner(self)` (❻), dont le corps ne contient encore que le mot-clé `pass`. Pour l'instant, cette fonction ne fait encore rien, mais nous allons bientôt la compléter.



À ce stade, la classe `Balle` est définie et nous devons en créer une instance, un objet. Pour rappel, une classe décrit ce qu'elle est capable de réaliser, mais c'est l'objet qui le fait réellement. Ajoute le code suivant au bas du programme pour créer l'objet `balle` rouge :

---

```
balle = Balle(canvas, 'red')
```

---

La chaîne '`red`' force la couleur de remplissage rouge (`red` en anglais). Si tu exécutes ce programme dans certaines versions antérieures de Python, le canevas apparaît pendant une fraction de seconde, puis disparaît. Pour empêcher la fenêtre de se fermer immédiatement, il faut ajouter une boucle d'animation appelée « boucle principale » de notre jeu. Même si la fenêtre ne se ferme pas de suite, nous avons besoin de cette boucle pour la suite des opérations.

La boucle principale constitue la partie centrale d'un programme et elle contrôle généralement l'essentiel de ce qu'il fait. Pour l'instant, elle s'exécute à l'infini (tant que tu ne fermes pas la fenêtre) en indiquant en permanence à `tkinter` de redessiner l'écran, puis en disant au programme de se mettre en sommeil pendant un centième de seconde.

Ajoute le code suivant à la fin du programme :

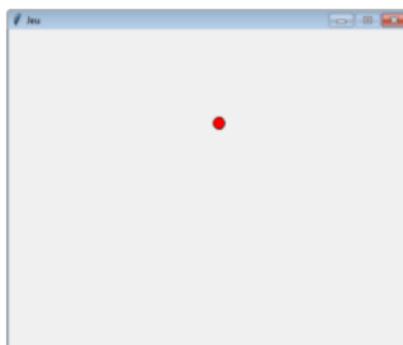
---

```
balle = Balle(canvas, 'red')

while 1:
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

---

Maintenant, quand tu exécutes le programme, la balle apparaît près du centre de l'écran. Remarque que la barre de titre contient **Jeu** et non plus **tk** :



## Ajouter de l'action

À ce stade, la classe `Balle` est définie et nous avons un objet `balle`. Il nous reste à animer cette dernière, en la faisant avancer, rebondir et changer de direction.



### Déplacer la balle

Pour déplacer la balle, il faut agir dans la fonction `dessiner`, comme suit :

---

```
class Balle:  
    def __init__(self, canvas, couleur):  
        self.canvas = canvas  
        self.id = canvas.create_oval(10, 10, 25, 25, fill=couleur)  
        self.canvas.move(self.id, 245, 100)  
  
    def dessiner(self):  
        self.canvas.move(self.id, 0, -1)
```

---

Comme `__init__` a mémorisé le paramètre `canvas` dans la variable d'objet du même nom, nous pouvons faire référence à cette variable sous la forme `self.canvas` et appeler la fonction `move` indirectement.

Nous passons trois paramètres à `move` : l'`id` de l'ovale ainsi que les nombres `0` et `-1`. Le `0` indique de ne pas bouger horizontalement, tandis que le `-1` qu'il faut se déplacer verticalement d'un pixel vers le haut de l'écran.

Cette petite modification est intéressante, car elle nous permet d'essayer des choses au fur et à mesure. Si nous devions écrire le code complet du jeu en une seule fois, que se passerait-il si quelque chose ne devait pas fonctionner comme prévu ? Il nous serait beaucoup plus difficile de trouver d'où vient le problème. En avançant comme ceci, petit à petit, il nous sera beaucoup plus facile ensuite de découvrir ce qui provoque des soucis et pourquoi.

L'autre modification vient se placer dans la boucle principale du bas du programme. Dans le bloc de la boucle `while` de notre boucle principale, nous ajoutons un appel à la fonction `dessiner` de l'objet `balle` :

---

```
while 1:  
    balle.dessiner()  
    tk.update_idletasks()  
    tk.update()  
    time.sleep(0.01)
```

---

Maintenant, quand tu exécutes le programme, la balle avance vers le haut de l'écran pour disparaître, parce que le code oblige `tkinter` à redessiner l'écran très rapidement : les fonctions `update_idletasks` et `update` forcent `tkinter` à se dépêcher pour redessiner le contenu du canevas.

La fonction `sleep` (à savoir, « dormir ») du module `time` demande à Python de placer le programme en sommeil pendant un centième de seconde (`0.01`). Si tu ne mets pas une telle ligne, Python et `tkinter` vont tellement vite que tu n'as pas le temps de voir la balle qu'elle a déjà disparu de l'écran !

Fondamentalement donc, la boucle dit « déplacer la balle d'un peu, redessiner l'écran avec le nouvel emplacement, hiberner un instant, puis recommencer ».

#### NOTE

*Lorsque le programme s'exécute et que tu fermes la fenêtre, le shell Python affiche un message d'erreur. Ceci est dû au fait que, quand tu fermes la fenêtre, le code est interrompu alors qu'il est dans la boucle principale et Python s'en plaint.*

Jusque-là, le code a l'allure suivante :

---

```
from tkinter import *
import random
import time

class Balle:
    def __init__(self, canvas, couleur):
        self.canvas = canvas
        self.id = canvas.create_oval(10, 10, 25, 25, fill=couleur)
        self.canvas.move(self.id, 245, 100)

    def dessiner(self):
        self.canvas.move(self.id, 0, -1)

tk = Tk()
tk.title("Jeu")
tk.resizable(0, 0)
tk.wm_attributes("-topmost", 1)
canvas = Canvas(tk, width=500, height=400, bd=0, highlightthickness=0)
canvas.pack()
tk.update()

balle = Balle(canvas, 'red')

while 1:
    balle.dessiner()
```

```
tk.update_idletasks()  
tk.update()  
time.sleep(0.01)
```

---

## Faire rebondir la balle

Une balle qui disparaît en haut de l'écran ne présente pas vraiment d'intérêt pour un jeu ; ce serait mieux si elle rebondissait sur le mur. Pour cela, nous ajoutons quelques variables d'objet à la fonction d'initialisation de la classe `Balle`, comme suit :

---

```
class Balle:  
    def __init__(self, canvas, couleur):  
        self.canvas = canvas  
        self.id = canvas.create_oval(10, 10, 25, 25, fill=couleur)  
        self.canvas.move(self.id, 245, 100)  
        self.x = 0  
        self.y = -1  
        self.hauteur_canevas = self.canvas.winfo_height()
```

---

Nous avons ajouté trois lignes supplémentaires au programme. Avec `self.x = 0`, nous définissons une variable d'objet `x` avec la valeur `0`, puis, avec `self.y = -1`, nous définissons la variable d'objet `y` avec la valeur initiale `-1`. Enfin, nous définissons la variable d'objet `hauteur_canevas` avec, comme valeur, le résultat de l'appel de la fonction `winfo_height` (qui renvoie la hauteur actuelle du canevas).

Ensuite, nous modifions de nouveau la fonction `dessiner` :

---

```
def dessiner(self):  
    ❶    self.canvas.move(self.id, self.x, self.y)  
    ❷    pos = self.canvas.coords(self.id)  
    ❸    if pos[1] <= 0:  
        self.y = 1  
    ❹    if pos[3] >= self.hauteur_canevas:  
        self.y = -1
```

---

En ❶, nous modifions l'appel à la fonction `move` pour lui passer les variables d'objet `x` et `y` au lieu de valeurs numériques brutes. Ensuite, nous créons une variable `pos` (position) avec comme valeur le résultat de l'appel de la fonction `coords` du canevas (❷). Cette fonction renvoie les coordonnées `x` et `y` de tout objet dessiné à l'écran, du moment que nous en connaissons le numéro d'identifiant. Ici, nous passons à `coords` la variable d'objet `id`, qui contient l'identifiant de l'ovale.

La fonction `coords` renvoie les coordonnées sous la forme d'une liste de quatre nombres. Si nous affichons les résultats de l'appel de cette fonction, nous voyons quelque chose du type :

---

```
print(self.canvas.coords(self.id))
[255.0, 29.0, 270.0, 44.0]
```

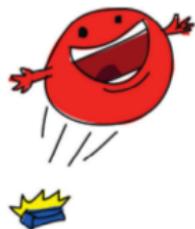
---

Les deux premiers nombres de la liste, `255.0` et `29.0`, correspondent aux coordonnées du « coin » supérieur de l'ovale (`x1` et `y1`), tandis que les deux derniers, `270.0` et `44.0`, sont celles du coin inférieur droit (`x2` et `y2`). Nous exploitons ces valeurs dans les lignes suivantes du programme.

En ❸, nous vérifions que la coordonnée `y1` (qui correspond au sommet de la balle) est inférieure ou égale à `0` (le bord haut de l'écran). Si c'est le cas, nous forçons la variable d'objet `y` à `1`. En pratique, cela signifie que, si la balle touche le haut de l'écran, nous cessons de soustraire `1` à la position verticale, donc nous arrêtons d'aller vers le haut.

De la même manière, nous vérifions que la coordonnée `y2`, c'est-à-dire le bas de la balle, est supérieure ou égale à la variable `hauteur_canevas` ❹. Si elle l'est, alors nous ramenons la variable d'objet `y` à `-1`.

Exécute le programme à nouveau pour constater que la balle rebondit en haut et en bas, jusqu'à ce que tu fermes la fenêtre.



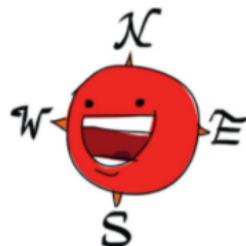
### Changer la direction de départ de la balle

Nous avons obtenu une balle qui rebondit doucement de haut en bas, mais cela ne suffit toujours pas pour faire un jeu. Nous allons donc améliorer un peu les choses et modifier la direction de départ de la balle, c'est-à-dire l'angle de déplacement de celle-ci au départ du jeu. Dans la fonction `__init__` de la classe `Balle`, modifie ces lignes :

---

```
self.x = 0
self.y = -1
```

---



Remplace-les par les suivantes (vérifie que tu as bien laissé les nombres d'espaces corrects, il doit y en avoir huit, au début de chaque ligne) :

---

```
❶     depart = [-3, -2, -1, 1, 2, 3]
❷     random.shuffle(depart)
❸     self.x = depart[0]
❹     self.y = -3
```

---

Nous créons d'abord (❶) la variable `depart` avec une liste de six nombres, que nous mélangeons par l'appel à `random.shuffle` (❷). Nous donnons à `x` la valeur du premier élément de la liste (❸), de sorte que celui-ci peut prendre n'importe quelle valeur au hasard dans cette liste, de `-3` à `3`.

Si nous modifions la valeur de `y` en `-3` pour accélérer la balle (❹), nous devons aussi apporter quelques ajouts pour nous assurer que la balle ne disparaît pas par les côtés de l'écran. Ajoute les lignes suivantes à la fin de la fonction `__init__` pour mémoriser la largeur du canevas dans une nouvelle variable d'objet, `largeur_canevas` :

---

```
self.largeur_canevas = self.canvas.winfo_width()
```

---

Nous utilisons cette nouvelle variable d'objet dans la fonction `dessiner` pour vérifier si la balle touche le côté gauche ou droit du canevas :

---

```
if pos[0] <= 0:
    self.x = 3
if pos[2] >= self.largeur_canevas:
    self.x = -3
```

---

Comme nous réglons `x` à `3` et `-3`, nous faisons de même avec `y`, pour que la balle se déplace à la même vitesse dans toutes les directions. La fonction `dessiner` devient la suivante :

---

```
def dessiner(self):
    self.canvas.move(self.id, self.x, self.y)
    pos = self.canvas.coords(self.id)
    if pos[1] <= 0:
        self.y = 3
    if pos[3] >= self.hauteur_canevas:
        self.y = -3
    if pos[0] <= 0:
        self.x = 3
    if pos[2] >= self.largeur_canevas:
        self.x = -3
```

---

Enregistre et exécute le programme. La balle doit désormais rebondir dans tout l'écran sans disparaître. Le programme complet, à ce stade, est celui-ci :

---

```
from tkinter import *
import random
import time

class Balle:
    def __init__(self, canvas, couleur):
        self.canvas = canvas
        self.id = canvas.create_oval(10, 10, 25, 25, fill=couleur)
        self.canvas.move(self.id, 245, 100)
        departs = [-3, -2, -1, 1, 2, 3]
        random.shuffle(departs)
        self.x = departs[0]
        self.y = -3
        self.hauteur_canevas = self.canvas.winfo_height()
        self.largeur_canevas = self.canvas.winfo_width()

    def dessiner(self):
        self.canvas.move(self.id, self.x, self.y)
        pos = self.canvas.coords(self.id)
        if pos[1] <= 0:
            self.y = 3
        if pos[3] >= self.hauteur_canevas:
            self.y = -3
        if pos[0] <= 0:
            self.x = 3
        if pos[2] >= self.largeur_canevas:
            self.x = -3

    tk = Tk()
    tk.title("Jeu")
    tk.resizable(0, 0)
    tk.wm_attributes("-topmost", 1)
    canvas = Canvas(tk, width=500, height=400, bd=0, highlightthickness=0)
    canvas.pack()
    tk.update()

balle = Balle(canvas, 'red')

while 1:
    balle.dessiner()
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

---

## Ce que tu as appris

Dans ce chapitre, nous avons débuté notre premier jeu à l'aide du module `tkinter`. Nous avons créé une classe pour la balle et l'avons animée pour qu'elle se déplace sur tout l'écran. Grâce aux coordonnées, nous avons vérifié que la balle touche bien les bords du canevas, puis nous avons fait en sorte qu'elle rebondisse. Nous avons également tiré parti de la fonction `shuffle` du module `random` pour que notre balle ne parte pas toujours exactement dans la même direction. Au chapitre suivant, nous ajoutons la raquette pour compléter ce jeu.



14

# ACHEVER TON PREMIER JEU : REBONDIR !

Au chapitre précédent, nous avons débuté la création d'un tout premier jeu, Rebondir ! Nous avons créé le canevas et lui avons ajouté une balle magique, mais cette balle rebondit à l'infini à l'écran, tant que tu ne fermes pas la fenêtre. Ceci n'a pas vraiment d'intérêt. Alors, dans ce chapitre, nous allons ajouter une raquette pour qu'un utilisateur puisse renvoyer la balle. Nous allons aussi ajouter un élément de chance, pour épicer un peu le jeu et apporter plus d'amusement.

## Ajouter la raquette

Comme il n'y a pas vraiment d'amusement à voir rebondir une balle à l'infini, nous allons impliquer l'utilisateur en lui offrant une raquette.

Commence par ajouter le code qui suit pour créer la classe `Raquette`. Entre ces instructions deux lignes après la fonction `dessiner` de la classe `Balle`.



---

```
def dessiner(self):
    self.canvas.move(self.id, self.x, self.y)
    pos = self.canvas.coords(self.id)
    if pos[1] <= 0:
        self.y = 3
    if pos[3] >= self.hauteur_canevas:
        self.y = -3
    if pos[0] <= 0:
        self.x = 3
    if pos[2] >= self.largeur_canevas:
        self.x = -3

class Raquette:
    def __init__(self, canvas, couleur):
        self.canvas = canvas
        self.id = canvas.create_rectangle(0, 0, 100, 10, fill=couleur)
        self.canvas.move(self.id, 200, 300)

    def dessiner(self):
        pass
```

---

Le code ajouté ressemble fort à celui du début de la classe `Balle`, sauf qu'ici, nous appelons `create_rectangle` au lieu de `create_oval` et que nous plaçons le rectangle à 200 pixels du bord intérieur gauche du canevas et à 300 pixels du bord supérieur.

Quasiment à la fin du programme, juste avant la création de l'objet `balle`, crée ensuite un objet de la classe `Raquette` (de couleur bleue), puis ajoute dans la boucle principale un appel à la fonction `dessiner` de la `raquette`, comme suit :

---

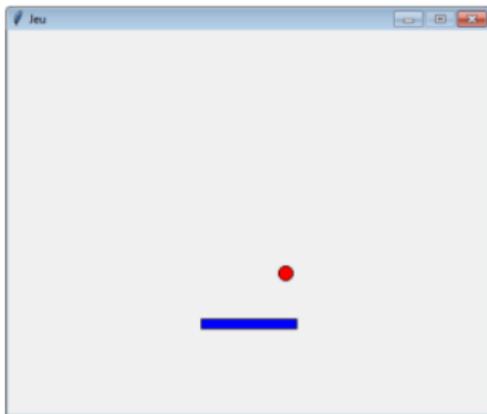
```
raquette = Raquette(canvas, 'blue')
balle = Balle(canvas, 'red')

while 1:
    balle.dessiner()
    raquette.dessiner()
```

---

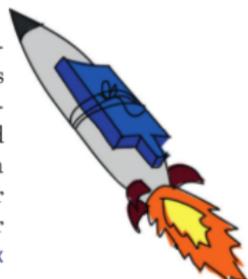
```
tk.update_idletasks()  
tk.update()  
time.sleep(0.01)
```

À partir de là, lorsque tu exécutes le programme, tu vois une balle qui rebondit sur les côtés et un rectangle stationnaire (qui ne bouge pas).



## Déplacer la raquette

Pour que la raquette se déplace, nous utilisons la liaison d'événement pour associer les touches de flèches **Gauche** et **Droite** à de nouvelles fonctions de la classe **Raquette**. Quand l'utilisateur appuie sur la flèche gauche, la variable **x** est réglée à **-2** (donc pour déplacer la raquette vers la gauche). Une pression sur la touche de la flèche droite règle la variable **x** à **2** (vers la droite).



La première étape consiste à ajouter la variable d'objet **x** à la fonction **\_\_init\_\_** de la classe **Raquette**, ainsi qu'une variable pour la largeur du canevas, comme celle que nous avions définie dans la classe **Balle** :

```
def __init__(self, canvas, couleur):  
    self.canvas = canvas  
    self.id = canvas.create_rectangle(0, 0, 100, 10, fill=couleur)  
    self.canvas.move(self.id, 200, 300)
```

```
self.x = 0  
self.largeur_canevas = self.canvas.winfo_width()
```

---

Comme nous devons changer la direction vers la gauche (**vers\_gauche**) et vers la droite (**vers\_droite**) à l'aide de fonctions, nous les ajoutons juste avant la fonction **dessiner** :

---

```
def vers_gauche(self, evt):  
    self.x = -2
```

```
def vers_droite(self, evt):  
    self.x = 2
```

---

Nous pouvons ensuite associer ces fonctions aux touches adéquates dans la fonction **\_init\_** de la classe en deux lignes. Nous avons déjà utilisé la liaison d'un événement à une action (voir la section « Réagir à un événement » du chapitre 12, où Python appelle une fonction lors de la pression d'une touche du clavier). Dans ce cas-ci, nous lions la fonction **vers\_gauche** de la classe **Raquette** à la pression sur la touche flèche **Gauche** à l'aide du nom d'événement '**<KeyPress-Left>**'. De même, nous associons la fonction **vers\_droite** à la touche **Droite** à l'aide de l'événement '**<KeyPress-Right>**'. Dès lors, notre fonction **\_init\_** devient :

---

```
def __init__(self, canvas, couleur):  
    self.canvas = canvas  
    self.id = canvas.create_rectangle(0, 0, 100, 10, fill=couleur)  
    self.canvas.move(self.id, 200, 300)  
    self.x = 0  
    self.largeur_canevas = self.canvas.winfo_width()  
    self.canvas.bind_all('<KeyPress-Left>', self.vers_gauche)  
    self.canvas.bind_all('<KeyPress-Right>', self.vers_droite)
```

---

Quant à la fonction **dessiner** de la classe **Raquette**, elle s'inspire fortement de celle de la classe **Balle** :

---

```
def dessiner(self):  
    ❶    self.canvas.move(self.id, self.x, 0)  
    ❷    pos = self.canvas.coords(self.id)  
    ❸    if pos[0] <= 0:  
        self.x = 0  
    ❹    elif pos[2] >= self.largeur_canevas:  
        self.x = 0
```

---

La fonction **move** ❶ du canevas permet de déplacer la raquette dans la direction donnée par la variable **x**. Le zéro en troisième para-

mètre signifie qu'il n'y a pas de déplacement vertical. Nous lisons ensuite les coordonnées d'emplacement de la raquette ❷ pour vérifier si elle a touché le côté gauche d'abord, droit ensuite, de l'écran, grâce à la valeur contenue dans `pos`.

Au lieu de rebondir comme la balle, la raquette doit simplement s'arrêter de bouger. Donc, quand la coordonnée `x` de gauche (`pos[0]`) est inférieure ou égale à 0 (`<= 0`), nous réglons la variable `x` à 0 ❸. De mêm, quand la coordonnée `x` de droite (`pos[2]`) est supérieure ou égale à la largeur du canevas, c'est-à-dire `>= self.largeur_canevas`, nous réglons aussi la variable `x` à 0 ❹.

#### NOTE

*Si tu exécutes le programme maintenant, tu dois cliquer dans le canevas avant que le jeu ne reconnaisse les actions sur les touches Gauche et Droite. Quand tu cliques dans le canevas, tu donnes le focus au canevas, ce qui signifie qu'il sait alors qu'il doit prendre en charge les pressions de l'utilisateur sur les touches du clavier.*

### DéTECTER quand la balle touche la raquette

À ce point de la programmation, la balle ne heurte pas la raquette, mais elle la traverse, tout simplement. Or, la balle doit savoir quand elle a touché la raquette, au même titre qu'elle sait qu'elle a touché un des murs.



Nous pourrions résoudre ce problème en ajoutant du code à la fonction `dessiner` (là où le code détecte les murs), mais une meilleure idée consiste à déplacer ce genre de code dans de nouvelles fonctions, pour découper les tâches dans de plus petits extraits. Si nous plaçons trop de code au même endroit (dans une seule fonction, par exemple), le programme devient plus difficile à comprendre. Appor-tions les modifications nécessaires.

D'abord, modifions la fonction `__init__` de la balle pour que nous puissions lui passer l'objet `raquette` en paramètre :

---

```
classe Balle:  
❶  def __init__(self, canvas, raquette, couleur):  
      self.canvas = canvas  
❷  self.raquette = raquette  
      self.id = canvas.create_oval(10, 10, 25, 25, fill=couleur)  
      self.canvas.move(self.id, 245, 100)  
      depart = [-3, -2, -1, 1, 2, 3]  
      random.shuffle(departs)
```

```
self.x = depart[0]
self.y = -3
self.hauteur_canevas = self.canvas.winfo_height()
self.largeur_canevas = self.canvas.winfo_width()
```

---

En ❶, nous modifions les paramètres de la fonction `_init_` pour y inclure la raquette. En ❷, nous affectons le paramètre `raquette` à la variable d'objet `raquette`.

L'objet `raquette` mémorisé, nous devons modifier le code où nous avons créé l'objet `balle`, puisqu'un nouveau paramètre y apparaît. La modification a lieu vers la fin du programme, avant la boucle principale :

```
raquette = Raquette(canvas, 'blue')
balle = Balle(canvas, raquette, 'red')

while 1:
    balle.dessiner()
    raquette.dessiner()
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

---

Le code nécessaire pour vérifier si la balle a heurté la raquette est un peu plus compliqué que celui qui détecte les murs. Nous nommons cette fonction `heurter_raquette` et l'ajoutons à la fonction `dessiner` de la classe `Balle`, là où nous vérifions si la balle touche le bas de l'écran :

```
def dessiner(self):
    self.canvas.move(self.id, self.x, self.y)
    pos = self.canvas.coords(self.id)
    if pos[1] <= 0:
        self.y = 3
    if pos[3] >= self.hauteur_canevas:
        self.y = -3
    if self.heurter_raquette(pos) == True:
        self.y = -3
    if pos[0] <= 0:
        self.x = 3
    if pos[2] >= self.largeur_canevas:
        self.x = -3
```

---

Ainsi, si `heurter_raquette` retourne vrai, alors nous changeons la direction verticale, c'est-à-dire la variable d'objet `y` à `-3`. N'essaie pas

d'exécuter le programme à ce stade car la fonction `heurter_raquette` n'existe pas encore. Créons-la maintenant.

Ajoute la fonction `heurter_raquette` juste avant la fonction `dessiner`.

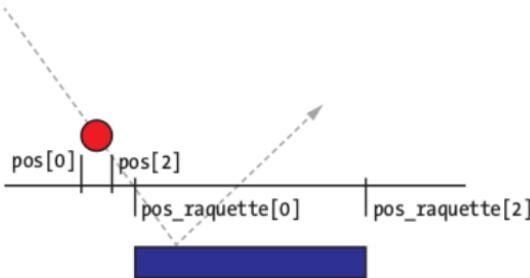
---

```
❶ def heurter_raquette(self, pos):
❷     pos_raquette = self.canvas.coords(self.raquette.id)
❸     if pos[2] >= pos_raquette[0] and pos[0] <= pos_raquette[2]:
❹         if pos[3] >= pos_raquette[1] and pos[3] <= pos_raquette[3]:
❺             return True
❻     return False
```

---

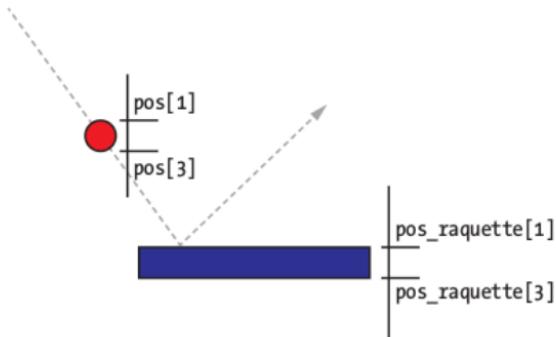
Nous définissons d'abord la fonction avec le paramètre `pos` (position) ❶. Cette ligne contient les coordonnées actuelles de la balle. Puis, nous prenons les coordonnées de la raquette et nous les mémoisons dans la variable `pos_raquette` ❷.

En ❸, nous avons la première partie de notre instruction `if` et nous disons « si le côté droit de la balle est plus grand que le côté gauche de la raquette et si le côté gauche de la balle est plus petit que le côté droit de la raquette, alors... ». Dans cette ligne, `pos[2]` contient la coordonnée  $x$  du côté droit de la balle et `pos[0]` contient celle de son côté gauche. En revanche, `pos_raquette[0]` contient la coordonnée  $x$  du côté gauche de la raquette et `pos_raquette[2]` contient celle de son côté droit. Le diagramme suivant montre mieux comment ces coordonnées se présentent quand la balle est sur le point de toucher la raquette.



La balle tombe sur la raquette et, à cet instant précis, tu vois clairement que le côté droit de la balle (`pos[2]`) n'a pas encore dépassé le côté gauche de la raquette (donc `pos_raquette[0]`).

Nous vérifions ensuite ❹ si le bas de la balle (`pos[3]`) se trouve entre le haut de la raquette (`pos_raquette[1]`) et son bas (`pos_raquette[3]`). Le diagramme suivant montre que le bas de la balle (`pos[3]`) doit encore atteindre le haut de la raquette (`pos_raquette[1]`).



Ainsi, selon cet emplacement actuel de la balle, la fonction `heurter_raquette` renvoie faux.

#### NOTE

*Pourquoi faut-il vérifier si le dessous de la balle est entre le haut et le bas de la raquette ? Pourquoi ne pas simplement regarder si le bas de la balle a heurté le haut de la raquette ? Ceci est dû au fait que, chaque fois que nous déplaçons la balle dans le canevas, nous nous déplaçons par sauts de trois pixels à la fois. Si nous ne vérifions que le fait que la balle a atteint le haut de la raquette (`pos[1]`), il se peut que nous ayons déjà sauté au-delà de cet emplacement. Dans ce cas, la balle poursuivrait son voyage et elle traverserait la raquette sans s'arrêter.*

## Ajouter un facteur chance

Pour que ce programme devienne un véritable jeu et ne se limite pas à une raquette et une balle rebondissante, il doit apporter un facteur chance, c'est-à-dire une possibilité pour l'utilisateur de perdre. Dans le jeu tel qu'il est actuellement, la balle rebondit à l'infini donc il n'y a rien à perdre.

Pourachever notre jeu, nous allons ajouter du code pour que la partie se termine quand la balle touche le bas du canevas, autrement dit quand la balle touche le sol.

En premier lieu, nous ajoutons la variable d'objet `touche_bas` à la fin de la fonction `__init__` de la classe `Balle` :



---

```
self.hauteur_canevas = self.canvas.winfo_height()
self.largeur_canevas = self.canvas.winfo_width()
self.touche_bas = False
```

---

Ensuite, nous devons modifier un peu la boucle principale, au bas du programme, comme ceci :

---

```
while 1:
    if balle.touche_bas == False:
        balle.dessiner()
        raquette.dessiner()
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

---

À partir de maintenant, la boucle vérifie à chaque passage la valeur de **touche\_bas**, pour savoir si la balle a touché le bas de l'écran. Le code continue de déplacer la balle et la raquette seulement si la balle n'a pas touché le bas, comme l'indique l'instruction **if**. La partie se termine quand la balle et la raquette cessent de se déplacer car nous ne les animons plus.

La modification finale concerne la fonction **dessiner** de la classe **Balle** :

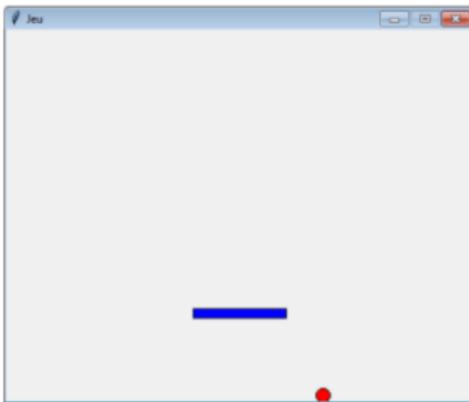
---

```
def dessiner(self):
    self.canvas.move(self.id, self.x, self.y)
    pos = self.canvas.coords(self.id)
    if pos[1] <= 0:
        self.y = 3
    if pos[3] >= self.hauteur_canevas:
        self.touche_bas = True
    if self.heurter_raquette(pos) == True:
        self.y = -3
    if pos[0] <= 0:
        self.x = 3
    if pos[2] >= self.largeur_canevas:
        self.x = -3
```

---

Nous avons corrigé l'instruction **if** qui teste si la balle a touché le bas de l'écran, c'est-à-dire si sa position en *y2* est supérieure ou égale à la **hauteur\_canevas**. Si c'est le cas, à la ligne suivante, nous réglons **touche\_bas** à vrai, au lieu de changer la valeur de la variable **y**, parce qu'il n'est plus nécessaire de la faire rebondir dès qu'elle a touché le bas de l'écran.

À présent, quand tu exécutes le programme de jeu et rate la balle avec la raquette, tous les mouvements de l'écran s'arrêtent et la partie se termine quand la balle touche le bas du canevas :



Le programme complet a désormais l'allure suivante. S'il ne fonctionne pas comme prévu, compare ton code à celui-ci pour trouver les éventuelles erreurs.

---

```
from tkinter import *
import random
import time

class Balle:
    def __init__(self, canvas, raquette, couleur):
        self.canvas = canvas
        self.raquette = raquette
        self.id = canvas.create_oval(10, 10, 25, 25, fill=couleur)
        self.canvas.move(self.id, 245, 100)
        depart = [-3, -2, -1, 1, 2, 3]
        random.shuffle(depart)
        self.x = depart[0]
        self.y = -3
        self.hauteur_canevas = self.canvas.winfo_height()
        self.largeur_canevas = self.canvas.winfo_width()
        self.touche_bas = False

    def heurter_raquette(self, pos):
        pos_raquette = self.canvas.coords(self.raquette.id)
        if pos[2] >= pos_raquette[0] and pos[0] <= pos_raquette[2]:
            if pos[3] >= pos_raquette[1] and pos[3] <= pos_raquette[3]:
                return True
        return False
```

```

def dessiner(self):
    self.canvas.move(self.id, self.x, self.y)
    pos = self.canvas.coords(self.id)
    if pos[1] <= 0:
        self.y = 3
    if pos[3] >= self.hauteur_canevas:
        self.touche_bas = True
    if self.heurter_raquette(pos) == True:
        self.y = -3
    if pos[0] <= 0:
        self.x = 3
    if pos[2] >= self.largeur_canevas:
        self.x = -3

class Raquette:
    def __init__(self, canvas, couleur):
        self.canvas = canvas
        self.id = canvas.create_rectangle(0, 0, 100, 10, fill=couleur)
        self.canvas.move(self.id, 200, 300)
        self.x = 0
        self.largeur_canevas = self.canvas.winfo_width()
        self.canvas.bind_all('<KeyPress-Left>', self.vers_gauche)
        self.canvas.bind_all('<KeyPress-Right>', self.vers_droite)

    def vers_gauche(self, evt):
        self.x = -2

    def vers_droite(self, evt):
        self.x = 2

    def dessiner(self):
        self.canvas.move(self.id, self.x, 0)
        pos = self.canvas.coords(self.id)
        if pos[0] <= 0:
            self.x = 0
        elif pos[2] >= self.largeur_canevas:
            self.x = 0

tk = Tk()
tk.title("Jeu")
tk.resizable(0, 0)
tk.wm_attributes("-topmost", 1)
canvas = Canvas(tk, width=500, height=400, bd=0, highlightthickness=0)
canvas.pack()
tk.update()

raquette = Raquette(canvas, 'blue')
balle = Balle(canvas, raquette, 'red')

```

```
while 1:  
    if balle.touche_bas == False:  
        balle.dessiner()  
        raquette.dessiner()  
        tk.update_idletasks()  
        tk.update()  
        time.sleep(0.01)
```

---

## Ce que tu as appris

Avec ce chapitre, nous avons terminé la réalisation de notre premier jeu à l'aide du module `tkinter`. Nous avons créé la classe de la raquette et exploité les coordonnées pour vérifier si la balle heurte la raquette ou les murs du canevas du jeu. Grâce à la liaison d'événement à des actions, nous avons associé les touches de flèches **Gauche** et **Droite** pour contrôler les mouvements de la raquette. La boucle principale appelle les fonctions `dessiner` des deux objets pour les animer. Enfin, nous avons modifié le code pour ajouter un petit facteur de chance au jeu : quand le joueur rate la balle, la partie se termine au moment où elle touche le bord inférieur du canevas.



## Puzzles de programmation

Pour l'instant, notre jeu demeure un peu simple. Il n'y a pas grand-chose à changer pour créer un programme professionnel de jeu. Essaie d'améliorer le code selon les suggestions suivantes pour le rendre plus intéressant. Compare ensuite tes réalisations à celles du site d'accompagnement du livre.

### 1. Retarder le début du jeu

Le jeu démarre un peu rapidement et il faut cliquer sur le canevas pour qu'il commence à reconnaître les touches du clavier. Peux-tu ajouter un délai au démarrage de la partie, pour donner au joueur suffisamment de temps pour cliquer sur le canevas ? Mieux, peux-tu ajouter une liaison d'événement à un clic de la souris pour ne lancer la partie qu'au moment du clic ?

Conseil 1 : tu as déjà ajouté des liaisons d'événements à la classe `raquette`, donc c'est un bon endroit pour commencer tes recherches.

Conseil 2 : l'événement à détecter pour le bouton gauche de la souris est la chaîne '`<Button-1>`'.

## 2. Un véritable « Partie terminée »

Pour l'instant, le canevas se fige quand la partie se termine, ce qui n'est pas très convivial. Essaie d'ajouter le texte « Partie terminée » quand la balle touche le bord inférieur de l'écran. Tu peux utiliser la fonction `create_text`, mais tu peux trouver aussi une piste du côté du paramètre nommé `state` (état), qui prend des valeurs comme `normal` et `hidden` (caché). Cherche du côté d'`itemconfig` (voir la section « Autres façons d'utiliser l'identifiant » du chapitre 12). En guise de défi supplémentaire, ajoute un délai pour que le texte n'apparaisse pas immédiatement.

## 3. Accélérer la balle

Si tu as déjà joué au tennis, tu sais que quand la balle touche ta raquette, elle part parfois plus vite qu'elle n'est arrivée, selon la force de ton mouvement. La balle de notre jeu évolue actuellement à sa propre vitesse, que la raquette bouge ou pas. Essaie d'améliorer le programme pour que la vitesse de déplacement de la raquette soit communiquée à celle de la balle.

## 4. Enregistrer le score du joueur

Et si on parlait des scores ? Il est facile de compter au fur et à mesure le nombre de fois que la balle touche la raquette. Essaie d'afficher le score dans le coin supérieur droit du canevas. Cherche également du côté d'`itemconfig` (toujours au chapitre 12) pour une piste vers la solution.



PARTIE 3

# M. FILIFORME COURT VERS LA DROITE







15

# CRÉER LES GRAPHISMES DU JEU M. FILIFORME

Lorsqu'il s'agit de créer un jeu, ou n'importe quel autre programme d'ailleurs, le mieux est d'écrire un plan, avec une description de ce qu'il est supposé faire, ainsi que de ses principaux éléments et personnages. Lorsque la programmation débute, ce plan t'aidera à te concentrer sur ce que tu essaies de développer. Note qu'au fur et à mesure de son élaboration, de son encodage, le jeu peut s'écartez un peu de la description initiale, ce qui est tout à fait normal.

Dans ce chapitre, nous entamons le développement d'un jeu amusant, appelé M. Filiforme court vers la sortie.

## Plan du jeu de M. Filiforme

Voici la description de notre nouveau jeu.

- L'agent secret M. Filiforme est piégé dans l'antre du satanique Dr Inoffensif et tu dois l'aider à s'échapper par la sortie située à l'étage du haut.
- Le jeu utilise un personnage en fil de fer qui peut aller à gauche ou à droite et sauter. À chaque étage se situent des plates-formes sur lesquelles il doit sauter.
- Le but du jeu est d'atteindre la porte de sortie, avant qu'il ne soit trop tard et que le jeu ne se termine.



Selon cette description, nous savons qu'il nous faudra plusieurs images différentes : celles de M. Filiforme, de la porte et des plates-formes, entre autres. Nous allons bien entendu devoir programmer un peu pour faire fonctionner tout cela mais, avant toute chose, nous allons créer quelques graphismes destinés au jeu, base qui nous sera utile au chapitre suivant.

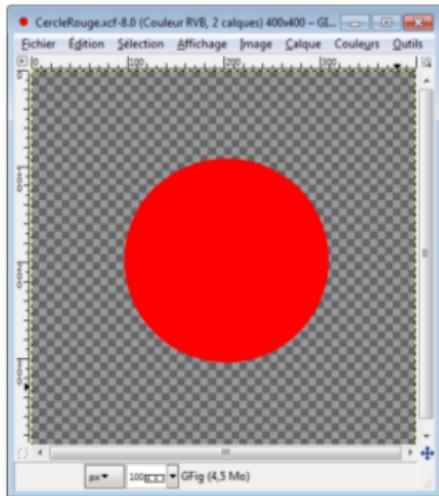
Comment allons-nous dessiner les éléments du jeu ? Nous pourrions utiliser les mêmes graphismes que ceux exploités pour la balle et la raquette des chapitres précédents, mais ils sont beaucoup trop simples pour ce jeu. Nous préférerons donc créer des **lutins**.

Les lutins, ou *sprites*, sont les choses qui interviennent visuellement dans un jeu, généralement les personnages. Ils ont la particularité d'être dessinés à l'avance avant l'exécution même du programme ; ils ne sont donc pas créés par le programme au moyen de polygones, comme dans Rebondir !. Ainsi, M. Filiforme sera un lutin, de même que les plates-formes. Pour fabriquer des images de ce type, il faut un programme de création graphique.

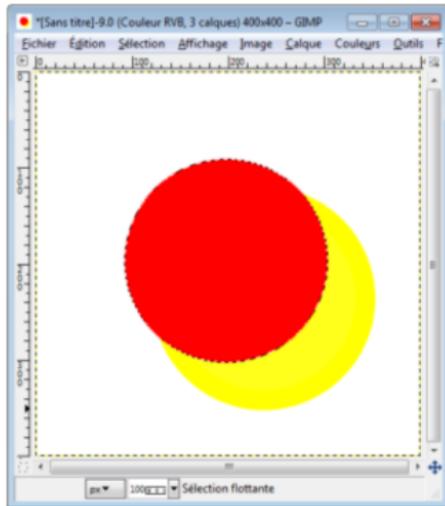
## Obtenir Gimp

Bien des programmes de création graphique existent mais, pour ce jeu, il nous en faut un qui gère la **transparence**, ce que l'on appelle souvent le « canal alpha » : il attribue aux images des portions où aucune couleur n'apparaît à l'écran. Pourquoi avons-nous besoin d'images avec des parties transparentes ? Quand des dessins passent au-dessus ou à côté d'un autre pendant leur mouvement à l'écran, nous ne souhaitons pas qu'ils masquent l'arrière-plan de

l'image ou d'un autre graphisme. Sur la figure suivante, le motif en damier de l'arrière-plan représente une zone transparente :



Par exemple, si nous prenons la totalité de cette image (la boule rouge) et si nous la plaçons sur une autre (le fond blanc et la boule jaune), l'arrière-plan de la première ne cache rien de l'autre :



Gimp (*GNU Image Manipulation Program* ou « programme GNU de manipulation d'images » – <http://www.gimp.org>) est un programme de création graphique libre (c'est-à-dire gratuit) pour Linux, OS X et Windows ; il gère la transparence des images. Pour le télécharger et l'installer, suis les indications ci-après.

- Sous Windows : tu trouveras des installateurs sur la page du projet Gimp à l'adresse <http://download.gimp.org/pub/gimp/stable/windows/>. Au moment de l'écriture de ces lignes, la dernière version de l'installateur était la [gimp-2.8.14-setup-1.exe](#). Note que même si l'installateur est en anglais (*english*), le programme est bel et bien en français à la fin de l'installation.
- Sous Linux, notamment Ubuntu : vérifie que Gimp n'est pas déjà installé par défaut. Sinon, ouvre la Logithèque, entre **gimp** dans la zone de recherche, puis clique sur le bouton **Installer** quand l'éditeur d'images Gimp apparaît dans la liste des résultats.
- Sous OS X : télécharge la suite logicielle complète à partir de l'adresse <http://gimp.lisanet.de/Website/Download.html>.

Pour ton jeu qui contiendra de nombreux fichiers, crée un dossier spécial. Sous Windows, préfère utiliser une clé USB pour les stocker : **Ordinateur>Lecteur amovible (F:\)** par exemple), puis **Nouveau dossier**. Sous Linux (Ubuntu), crée-le avec le gestionnaire de fichiers, puis clique droit dans **Documents** et sélectionne **Nouveau dossier**. Sous OS X, fais de même, avec l'option **Nouveau dossier**. Dans la boîte de dialogue **Nouveau dossier**, saisis **Filiforme** comme nom de dossier.

## Créer les éléments de jeu

Dès que le programme de création graphique est installé, tu es prêt à dessiner. Nous allons réaliser les images suivantes qui constitueront les éléments de notre jeu :

- pour le personnage en fil de fer, qui peut courir à gauche ou à droite et sauter ;
- pour les plates-formes, en trois tailles différentes ;
- pour la porte, soit une ouverte et une fermée ;
- pour l'arrière-plan du jeu, afin d'éviter que le fond soit tout blanc ou tout gris (ce qui risque de fatiguer rapidement l'utilisateur).

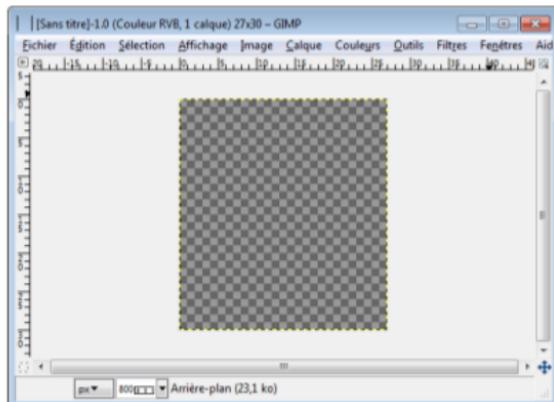
Avant de commencer à dessiner, nous devons préparer les images pour qu'elles aient un fond transparent.

## Préparer une image à fond transparent

Afin de paramétriser une image pour la transparence, donc avec un canal alpha, démarre Gimp, puis suis ces étapes.

1. Clique sur **Fichier>Nouvelle image**.
2. Dans la boîte de dialogue, saisis 27 pixels en largeur d'image et 30 pixels en hauteur.
3. Clique sur le + ou le triangle devant **Options avancées** puis, dans la liste **Remplir avec**, sélectionne **Transparence**.
4. Clique sur **Valider**.

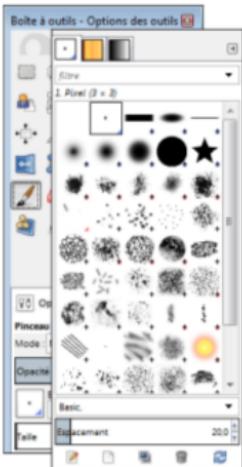
En résultat, tu obtiens une image remplie d'un damier de carrés gris et gris clair, comme ceci (le zoom au bas de la fenêtre permet d'agrandir l'image) :



À partir de là, nous pouvons créer notre agent secret : M. Filiforme.

## Dessiner M. Filiforme

Pour dessiner notre première image de personnage en fil de fer, clique sur l'outil **Pinceau** dans la boîte à outils de Gimp, puis sur la brosse qui ressemble à un petit point (généralement la deuxième brosse dans la liste, voir page suivante).



Nous avons besoin de trois images (*frames*) différentes pour que le personnage en fil de fer donne l'impression de courir et de sauter vers la droite. Nous utiliserons ces images successives pour animer M. Filiforme, comme nous l'avons fait au chapitre 12.

Lorsque tu zoomes sur ces images, elles ressemblent à ceci :



Il n'est pas nécessaire que tes images soient identiques à celles-ci. L'essentiel est qu'elles montrent le personnage en fil de fer dans trois positions de mouvement différentes et qui s'enchainent. Rappelle-toi de fixer pour chacun de tes dessins une largeur de 27 pixels et une hauteur de 30 pixels.

#### M. Filiforme court vers la droite

Nous allons tout d'abord dessiner la séquence d'images de M. Filiforme qui court vers la droite. Crée-la comme suit.

1. Dessine la première image (la plus à gauche de l'illustration précédente).

- Clique sur le menu **Fichier>Exporter vers**. Note que Gimp n'est pas toujours complètement traduit en français : tu peux donc trouver aussi **Export As**.
- Dans la boîte de dialogue **Exporter l'image**, entre **fil-D1.gif** comme nom de fichier, puis clique sur le + ou le triangle devant l'option **Sélectionner le type de fichier**.
- Selectionne **Image GIF** dans la liste qui apparaît.
- Enregistre l'image dans le dossier **Filiforme** que tu as créé précédemment : clique sur l'un des lecteurs de disques dans la colonne **Raccourcis** de gauche, puis va jusqu'au dossier désiré.

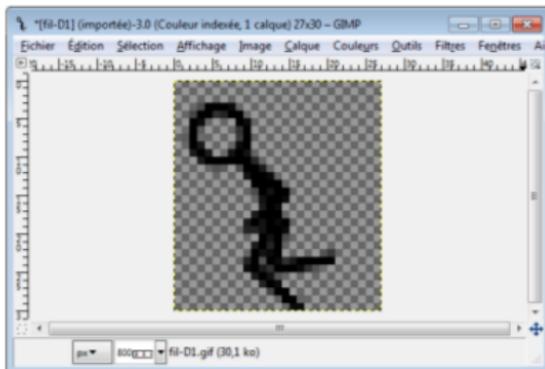
Suis les mêmes étapes pour créer une nouvelle image de 27 pixels de large et 30 pixels de haut, et dessine M. Filiforme en mouvement. Exporte l'image sous le nom de fichier **fil-D2.gif**. Répète enfin la procédure pour le dessin final, **fil-D3.gif**.



#### M. Filiforme court vers la gauche

Au lieu de recréer chacun des dessins pour le personnage en fil de fer qui court vers la gauche, nous allons réutiliser les trois précédents pour leur appliquer un effet miroir.

Dans Gimp, ouvre la première image, puis clique sur le menu **Outils>Outils de transformation>Retourner**. Par défaut, le « retournement » s'effectue horizontalement. Il te suffit ensuite de cliquer sur l'image pour la voir en miroir. Exporte-la sous le nom de **fil-G1.gif**. Procède de la même façon pour les suivantes et enregistre-les sous les noms **fil-G2.gif** et **fil-G3.gif**.



À présent, nous disposons de six dessins de M. Filiforme. Il nous faut encore les images pour les plates-formes et la porte de sortie.

## Dessiner les plates-formes

Nous allons créer trois plates-formes de même hauteur de 10 pixels, mais de largeurs différentes : 100, 60 et 30 pixels. Tu peux leur donner les formes et les couleurs de ton choix, mais veille à ce que leur arrière-plan soit transparent, comme pour les images de M. Filiforme.

Agrandies, voici à quoi peuvent ressembler les plates-formes :



Comme pour les images précédentes, exporte-les dans le dossier **Filiforme**. Nomme l'image la plus courte **plate-forme1.gif**, la moyenne **plate-forme2.gif** et la plus longue **plate-forme3.gif**.

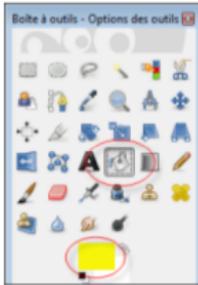
## Dessiner la porte

La taille de la porte doit être proportionnelle à celle de M. Filiforme, soit  $27 \times 30$  pixels. Nous avons besoin de deux images de la porte : fermée et ouverte, par exemple :



Pour les créer, suis les étapes ci-après.

1. Réalise une nouvelle image de la taille indiquée et de fond transparent.
2. Clique sur le rectangle de la **Couleur de premier plan** dans la **Boîte à outils** de Gimp pour afficher le sélecteur de couleur. Choisis celle qui te plaît pour la porte. Sur la figure ci-après, un exemple de jaune a été sélectionné.
3. Clique sur l'**Outil de remplissage** (entouré en rouge dans la **Boîte à outils**) et remplis l'espace de la couleur sélectionnée.
4. Modifie la couleur d'avant-plan en noir.



5. Clique ensuite sur l'outil **Crayon** ou **Pinceau**. Dessine le cadre noir autour de la porte, puis le bouton de porte.
6. Exporte ces deux dessins dans le dossier **Filiforme** et nomme leurs fichiers respectivement **porte1.gif** et **porte2.gif**.

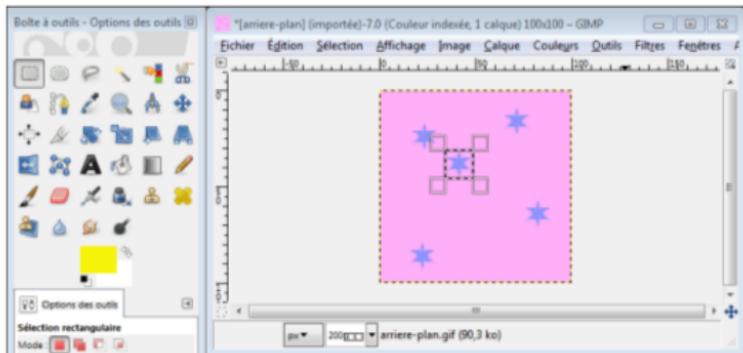
## Dessiner l'arrière-plan

La dernière image dont nous ayons besoin est celle de l'arrière-plan de l'écran du jeu. Pour cela, il faut réaliser une image de  $100 \times 100$  pixels mais, cette fois, il n'est pas nécessaire qu'elle soit transparente : elle sera remplie avec une seule couleur qui servira de « papier peint » derrière tous les autres éléments graphiques du jeu.

Pour créer l'arrière-plan, clique sur le menu **Fichier>Nouvelle image**. Précise une largeur et une hauteur de 100 pixels. Choisis une couleur « d'enfer » pour le fond d'écran de l'antre du méchant Dr Inoffensif, par exemple un rose sombre. Agrémentez ton papier peint avec des fleurs, des traits, des étoiles, etc., bref tout ce qui te semble approprié pour le jeu. Voici comment procéder pour ajouter, par exemple des étoiles.

1. Choisis une autre couleur. Clique sur l'outil **Crayon** et dessine ta première étoile.
2. Clique sur l'outil de **Sélection rectangulaire** pour décrire un rectangle autour de la première étoile.
3. Copie-colle l'image sélectionnée via **Édition>Copier**, puis menu **Édition>Coller**. Rappelle-toi qu'au chapitre 2, nous avons vu des raccourcis clavier pour ces opérations, qui restent valables ici.
4. Tu peux ensuite glisser l'image ainsi collée pour la placer plus loin dans le dessin de fond.

Voici un exemple avec quelques étoiles et l'outil **Sélection rectangulaire** activé dans la Boîte à outils :



Dès que tu es content de ton dessin, exporte l'image sous le nom de fichier `arrière-plan.gif` dans le même dossier `Filiforme`.

### Gérer la transparence

Maintenant que tous nos graphismes sont créés, tu comprends certainement mieux pourquoi ces images (sauf l'arrière-plan) ont besoin de transparence. Si nous placons un M. Filiforme sans cette transparence sur le papier peint, nous obtiendrions le dessin suivant – ce n'est pas très beau :



Dans ce cas, l'arrière-plan blanc de l'image de M. Filiforme masque une portion du papier peint. À l'inverse, quand la transparence est appliquée aux images des lutins, voici le résultat :



Tu constates que l'arrière-plan n'est pas masqué par l'image du premier plan. Seul le personnage se détache du papier peint. Avoue que l'effet est plus professionnel !

## Ce que tu as appris

Dans ce chapitre, tu as appris à définir un plan pour un jeu (M. Filiforme court vers la sortie). Comme il faut un certain nombre d'éléments graphiques avant de se lancer réellement dans la programmation d'un jeu, nous avons vu comment utiliser un programme de création graphique pour réaliser les graphismes de base, essentiels pour la suite du travail. Tu sais maintenant rendre transparents les arrière-plans de ces dessins pour qu'ils ne masquent pas les autres images de l'écran.

Au chapitre suivant, nous allons créer quelques-unes des classes indispensables pour le jeu.







16

# DÉVELOPPER LE JEU DE M. FILIFORME

Nous avons créé les images nécessaires à notre jeu « M. Filiforme court vers la sortie » et nous pouvons à présent entamer l'écriture du programme. La description du jeu du chapitre précédent donne une idée de ce dont nous avons besoin : un personnage en fil de fer qui court et saute, ainsi que des plates-formes sur lesquelles il doit sauter. Nous avons besoin de code pour afficher le personnage et le déplacer à l'écran, ainsi que pour dessiner les plates-formes. Avant de nous lancer dans ce code, nous devons créer le canevas et y afficher notre image de fond.

## Créer la classe Jeu

En premier lieu, nous créons une classe nommée `Jeu`, qui formera le contrôleur principal du programme. C'est dans la fonction `_init_` de cette classe que nous allons initialiser le jeu et une fonction `boucle_principale` se chargera de l'animation.

### Définir le titre de la fenêtre et créer le canevas

Dans la première partie de la fonction `_init_`, nous réglons le titre de la fenêtre et nous créons le canevas. En fait, cette portion du code ressemble fort à celle que nous avons écrite au chapitre 13 pour le jeu Rebondir ! Ouvre l'éditeur et entre le code suivant, puis enregistre-le sous le nom de fichier `jeufigiliforme.py` dans le dossier `Filiforme` que tu as créé au chapitre précédent.

---

```
from tkinter import *
import random
import time

class Jeu:
    def __init__(self):
        self.tk = Tk()
        self.tk.title("M. Filiforme court vers la sortie")
        self.tk.resizable(0, 0)
        self.tk.wm_attributes("-topmost", 1)
        self.canvas = Canvas(self.tk, width=500, height=500, \
                             highlightthickness=0)
        self.canvas.pack()
        self.tk.update()
        self.hauteur_canevas = 500
        self.largeur_canevas = 500
```

---

La première partie de ce programme (depuis `from tkinter` jusqu'à `self.tk.wm_attributes`) crée l'objet `tk`, puis définit le titre de la fenêtre, avec `self.tk.title("M. Filiforme court vers la sortie")`. Nous rendons la fenêtre immuable, pour éviter que le joueur la redimensionne, à l'aide de la fonction `resizable`, puis nous plaçons la fenêtre à l'avant de toutes les autres, grâce à la fonction `wm_attributes`.

Ensuite, nous créons le canevas avec la ligne `self.canvas = Canvas`, puis nous appelons les fonctions `pack` et `update` de l'objet `tk`. Enfin, nous créons deux variables dans la classe `Jeu`, `hauteur_canevas` et `largeur_canevas` pour mémoriser la hauteur et la largeur du canevas.

**NOTE**

La barre oblique inverse (`\`) située à la fin de la première ligne de `self.canvas = Canvas` sert seulement à scinder, découper, la longue ligne de code, pour la répartir sur plusieurs lignes. Elle n'est pas obligatoire, mais elle permet d'améliorer la lecture du code, puisque la totalité de la ligne ne tiendrait pas dans la largeur de la page. Si nous avions utilisé jusqu'ici le caractère `\` pour indiquer que la ligne se poursuit à la ligne suivante, ce caractère ne ferait pas partie du code et tout le code serait supposé se situer sur la même ligne de programme. La barre oblique inverse, en revanche, fait partie du langage Python.

## Terminer la fonction `__init__`

Entre à présent le reste de la fonction `__init__` à la suite du fichier `jeufigiliforme.py`. Ce code charge l'image d'arrière-plan et l'affiche sur le canevas :

---

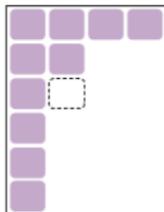
```
    self.tk.update()
    self.hauteur_canevas = 500
    self.largeur_canevas = 500
❶    self.ap = PhotoImage(file="ARRIERE-PLAN.gif")
❷    larg = self.ap.width()
    haut = self.ap.height()
❸    for x in range(0, 5):
❹        for y in range(0, 5):
            self.canvas.create_image(x * larg, y * haut, \
                                      image=self.ap, anchor='nw')
❺    self.lutins = []
❻    self.enfonction = True
```

---

Nous créons une variable `ap` ❶, pour « arrière-plan », qui contient un objet `PhotoImage` avec le fichier image de fond `ARRIERE-PLAN.gif` que nous avons créé au chapitre 15. Ensuite ❷, nous mémorisons la largeur et la hauteur de l'image dans les variables `larg` et `haut`. Les fonctions `width` et `height` de la classe `PhotoImage` retournent la taille de l'image, dès qu'elle est chargée.

Ensuite apparaissent deux boucles dans cette fonction. Pour comprendre ce qu'elles font, imagine que tu aies un petit timbre en caoutchouc, un tampon encreur et une grande feuille de papier. Le but est de remplir la feuille de papier avec les dessins (des rectangles de couleur de l'encre utilisée) du timbre en caoutchouc en les imprimant côté à côté sur la feuille. Comment vas-tu faire pour remplir la feuille ? Une possibilité serait de marquer la feuille au hasard, jusqu'à ce qu'elle soit pleine. Le résultat serait un vrai fouillis, pas

très joli, où les rectangles se superposeraient n'importe comment. Une autre possibilité consiste à marquer la feuille en commençant dans le coin en haut à gauche et à descendre en colonne jusqu'au bas de la feuille, puis de recommencer en haut en une deuxième colonne et ainsi de suite (voir figure).



Dans notre cas, le timbre en caoutchouc est l'image d'arrière-plan que nous avons dessinée au chapitre précédent. Nous savons que le canevas a 500 pixels de large sur 500 pixels de haut, tandis que l'image d'arrière-plan a  $100 \times 100$  pixels. Ceci indique que nous aurons cinq colonnes de cinq rangées de notre image de fond à placer dans le canevas pour le couvrir complètement. La boucle sur `x` calcule le nombre de colonnes ❶, tandis que la boucle sur `y` calcule le nombre de rangées ❷.

Nous multiplions la variable `x` de la première boucle par la largeur de l'image de fond (`x*larg`) pour déterminer à quelle distance du bord gauche du canevas il faut placer le dessin ❸ (le « timbre »), puis nous multiplions la variable `y` de la seconde boucle par la hauteur de l'image (`y*haut`) pour savoir à quelle distance du haut du canevas il faut poser l'image. La fonction `create_image` de l'objet `canvas (self.canvas.create_image)` place l'image à l'écran, aux coordonnées que nous venons de calculer.

Enfin ❹, nous créons la variable `lutins`, une liste qui ne comprend encore aucun lutin, puis la variable `enfonction` qui contient la valeur booléenne `True`. Ces variables nous serviront dans la suite du programme.

## Créer la fonction `boucle_principale`

Nous créons la fonction `boucle_principale` dans la classe `Jeu` pour animer le jeu. Elle ressemble beaucoup à la boucle principale, ou boucle d'animation, que nous avons réalisée dans Rebondir ! au chapitre 13. La voici :

---

```

for x in range(0, 5):
    for y in range(0, 5):
        self.canvas.create_image(x * larg, \
            y * haut, image=self.ap, anchor='nw')
self.lutins = []
self.enfonction = True

def boucle_principale(self):
    ❶    while 1:
    ❷        if self.enfonction == True:
    ❸            for lutin in self.lutins:
                lutin.deplacer()
            self.tk.update_idletasks()
            self.tk.update()
            time.sleep(0.01)

```

---

Nous créons une boucle `while` infinie ❶, qui ne s'arrête que lorsque la fenêtre du jeu est fermée par l'utilisateur. Nous vérifions que la variable `enfonction` est vraie ❷. Si elle l'est, nous bouclons parmi tous les lutins de la liste `self.lutins` ❸ et nous appelons ❹ la fonction `deplacer` pour chacun d'eux. Bien entendu, comme nous n'avons défini aucun lutin, ce programme ne fait encore rien du tout si tu l'exécutes, mais cette fonction `deplacer`, par exemple, nous sera bien utile par la suite.

Les trois dernières lignes ❽ obligent l'objet `tk` à redessiner l'écran et à s'endormir pendant un centième de seconde.

Pour pouvoir exécuter ce code, ajoute enfin les deux lignes suivantes, puis enregistre le fichier du programme. Comme ce sont des lignes du programme principal, aucun retrait n'est à appliquer :

---

```

jeu = Jeu()
jeu.boucle_principale()

```

---



### NOTE

*Ces deux lignes viennent tout à la fin du programme du jeu. Vérifie aussi que toutes les images sont dans le même dossier Filiforme que le fichier du programme en Python.*

La première ligne crée un objet de la classe `Jeu` et le mémorise dans la variable `jeu`, appelant de ce fait la fonction `__init__` de la classe. La dernière ligne appelle la fonction `boucle_principale` du nouvel objet pour dessiner l'écran.

Dès que tu as enregistré le programme, exécute-le dans IDLE (menu **Run>Run Module**). Une fenêtre apparaît, dont le canevas est rempli de l'image *arriere-plan.gif* :

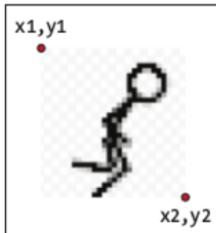


À ce stade, nous avons ajouté un joli papier peint à notre jeu et créé la boucle d'animation qui redessinera les lutins à notre place, du moins quand nous les aurons créés.

## Créer la classe Coords

Maintenant, nous créons une classe qui nous servira plus tard à spécifier l'emplacement d'un objet à l'écran. Cette classe mémorise les coordonnées du coin supérieur gauche ( $x1,y1$ ) et du coin inférieur droit ( $x2,y2$ ) de tout composant du jeu.

Ainsi, tu peux mémoriser l'emplacement de l'image du personnage en fil de fer à partir de ces coordonnées :



Nous nommons cette classe `Coords` et elle ne contient qu'une fonction `__init__`, à laquelle nous passons les quatre paramètres `x1`, `y1`, `x2` et `y2`, pour les mémoriser en interne dans l'objet de la classe. Ajoute ce code dans le fichier `jeufigiliforme.py`, après la classe `Jeu` :

```
class Coords:  
    def __init__(self, x1=0, y1=0, x2=0, y2=0):  
        self.x1 = x1  
        self.y1 = y1  
        self.x2 = x2  
        self.y2 = y2
```

Il s'agit ici de copier chaque paramètre dans une variable d'objet de même nom (`x1`, `y1`, `x2` et `y2`). Les objets de cette classe nous seront bientôt utiles.

**NOTE**

*La classe `Coords`, avec un grand C, donne naissance à des objets `co1`, `co2` et `coordonnees`. Ne perds jamais de vue qu'un nom avec une première majuscule désigne une classe, tandis qu'une variable avec une première minuscule désigne un objet de cette classe. Il s'agit d'une convention. Cette convention vaut pour les fonctions et les fonctions de classe (`coords` aussi), qui s'écrivent avec une première minuscule. En revanche, quand nous parlons de coordonnées, comme cela, avec un accent, il s'agit de coordonnées au sens général.*

## Vérifier les collisions

Lorsque nous savons comment stocker l'emplacement des lutins du jeu, il nous faut un moyen de dire si un lutin entre en collision avec un autre, puisque M. Filiforme se promène un peu partout et saute à l'écran, donc il touchera tôt ou tard un autre objet, notamment une plate-forme. Pour faciliter la résolution de cette question, nous pouvons la découper en plusieurs problèmes élémentaires : vérifier si des lutins entrent en collision verticalement et vérifier s'ils entrent en collision horizontalement. Ensuite, nous pourrons combiner les deux solutions élémentaires pour détecter facilement si des lutins se percutent dans n'importe quelle direction !

## Les lutins entrent en collision horizontalement

Nous allons d'abord créer la fonction `dans_x` pour déterminer si un ensemble de coordonnées en  $x$  (variables `x1` et `x2`, donc) passent au-dessus d'un autre ensemble de coordonnées en  $x$  (variables `x1` et `x2`, également). Il y a plusieurs moyens de résoudre cette question, mais voici une approche simple que tu peux ajouter *après* (mais pas *dans*) la définition de la classe `Coords` :

---

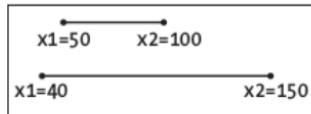
```
class Coords:  
    def __init__(self, x1=0, y1=0, x2=0, y2=0):  
        self.x1 = x1  
        self.y1 = y1  
        self.x2 = x2  
        self.y2 = y2  
  
    def dans_x(co1, co2):  
        ❶ if (co1.x1 > co2.x1 and co1.x1 < co2.x2):  
            return True  
        ❷ elif (co1.x2 > co2.x1 and co1.x2 < co2.x2):  
            return True  
        ❸ elif (co2.x1 > co1.x1 and co2.x1 < co1.x2):  
            return True  
        ❹ elif (co2.x2 > co1.x1 and co2.x2 < co1.x1):  
            return True  
        ❺ else:  
            return False
```

---



La fonction `dans_x` prend les paramètres `co1` et `co2`, tous deux des objets de classe `Coords`. La ligne ❶ vérifie que la position la plus à gauche du premier objet, `co1.x1`, est comprise entre la position la plus à gauche de la deuxième coordonnée, `co2.x1`, et la position la plus à droite de la deuxième coordonnée, `co2.x2`. Si c'est le cas, la ligne ❷ retourne vrai.

Pour bien comprendre comment cela se passe, prenons le cas de deux traits qui se superposent. Chaque trait commence en `x1` et se termine en `x2`.



Le premier trait de ce diagramme (`c01`) débute (`x1`) à l'emplacement 50 en pixels et se termine en 100 (`x2`). Le second trait (`c02`) débute à 40 (`x1`) et se termine en 150 (`x2`). Dans ce cas-ci, comme la position `x1` du premier trait est entre les positions `x1` et `x2` du second trait, la première instruction `if` de la fonction serait vraie pour ces deux ensembles de coordonnées.

Si ce n'est pas le cas, le `elif` ❸ vérifie que la position la plus à droite du premier trait (`c01.x2`) se situe entre la position la plus à gauche (`c02.x1`) et la position la plus à droite (`c02.x2`) du second trait. Si c'est le cas, la ligne ❹ renvoie vrai. Les deux instructions `elif` des lignes ❷ et ❸ font quasiment de même, mais à l'inverse : elles vérifient les positions les plus à gauche et plus à droite de la seconde ligne (`c02`) par rapport à celles de la première (`c01`).

Si aucune des conditions `if` et `elif` n'est vraie, nous atteignons le `else` ❺ et renvoyons faux ❻. Ceci revient à dire que les objets de coordonnées ne se chevauchent pas du tout, les uns, les autres, sur le plan horizontal.

Pour voir cette fonction à l'œuvre, il suffit de l'appeler avec les coordonnées des deux traits du diagramme présenté plus haut et d'afficher le résultat. Les coordonnées `x1` et `x2` du premier objet valent 50 et 100, tandis que celles du second objet valent 40 et 150. Les coordonnées en `y` n'ont pas d'importance dans ce cas-ci. Voici ce qui se passe quand nous appelons directement la fonction `dans_x` :

---

```
>>> c1 = Coords(50, 40, 100, 40)
>>> c2 = Coords(40, 50, 150, 50)
>>> print(dans_x(c1, c2))
True
```

---

La fonction renvoie effectivement `True`. C'est là la première étape qui permet de déterminer si un des lutins entre en collision avec un autre. Ainsi, lorsque nous créerons la classe de M. Filiforme et celle des plates-formes, nous pourrons déterminer si leurs coordonnées en `x` se chevauchent.

Arrêtons-nous là un instant car ce n'est pas une bonne pratique de programmation que de faire se succéder autant d'instructions `if-elif` qui renvoient toutes la même valeur. Pour résoudre le même problème, il suffirait d'entourer de parenthèses chacune des conditions, puis de séparer les conditions par des mots-clés `or`, au lieu des `elif`. Pour aboutir à une fonction plus propre, avec moins de lignes de code, n'hésite pas à modifier la fonction pour qu'elle prenne l'allure suivante :

---

```
def dans_x(co1, co2):
    if (co1.x1 > co2.x1 and co1.x1 < co2.x2) \
        or (co1.x2 > co2.x1 and co1.x2 < co2.x2) \
        or (co2.x1 > co1.x1 and co2.x1 < co1.x2) \
        or (co2.x2 > co1.x1 and co2.x2 < co1.x1):
        return True
    else:
        return False
```

---

Pour une meilleure lisibilité de la condition, n'oublie pas d'étendre l'instruction `if` sur plusieurs lignes, avec la barre oblique inverse (`\`).

## Les lutins entrent en collision verticalement

Nous devons aussi savoir si les lutins entrent en collision verticalement. La fonction `dans_y` ressemble fort à `dans_x`, puisque c'est le même principe qui l'anime. Pour la créer, nous vérifions que la position `y1` de la première coordonnée se situe entre les positions `y1` et `y2` de la seconde coordonnée, puis vice versa. Voici le code de la fonction à ajouter, à la suite de `dans_x`, directement simplifiée pour éviter la succession des `if-elif` :



---

```
def dans_y(co1, co2):
    if (co1.y1 > co2.y1 and co1.y1 < co2.y2) \
        or (co1.y2 > co2.y1 and co1.y2 < co2.y2) \
        or (co2.y1 > co1.y1 and co2.y1 < co1.y2) \
        or (co2.y2 > co1.y1 and co2.y2 < co1.y1):
        return True
    else:
        return False
```

---

## Assembler le tout - Code final de détection de collision

Nous disposons désormais de deux fonctions qui déterminent si les coordonnées horizontales d'une part, verticales d'autre part, de deux objets se chevauchent. Nous pouvons maintenant écrire des fonctions pour savoir si un lutin vient d'en percuter un autre et, si oui, de quel côté. Elles porteront les noms respectifs de `collision_gauche`, `collision_droite`, `collision_haut` et `collision_bas`.

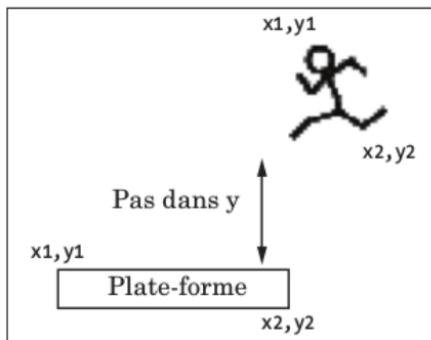
## La fonction collision\_gauche

Voici le code de la fonction `collision_gauche`, à ajouter à la suite de la fonction `dans_y` :

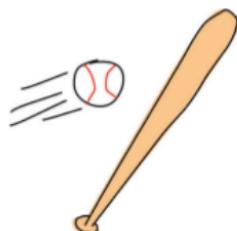
```
❶ def collision_gauche(co1, co2):
❷     if dans_y(co1, co2):
❸         if co1.x1 <= co2.x2 and co1.x1 >= co2.x1:
❹             return True
❺     return False
```

Cette fonction nous indique si le côté gauche (la valeur de `x1`) du premier objet de coordonnées a touché l'autre objet de coordonnées.

La fonction attend deux paramètres ❶ : `co1` (le premier objet) et `co2` (le second objet). Nous vérifions d'abord si les deux objets se sont superposés verticalement, à l'aide de la fonction `dans_y` ❷. Globalement, il n'est pas nécessaire de vérifier si M. Filiforme a touché une plate-forme quand il flotte au-dessus, comme ceci :



Nous vérifions ensuite que la valeur de la position gauche du premier objet (`co1.x1`) a touché la position `x2` du second (`co2.x2`), c'est-à-dire si elle est inférieure ou égale à la position `x2` ❸. Nous contrôlons aussi qu'elle n'a pas dépassé la position `x1`. Si elle a touché le côté ❹, nous renvoyons vrai. Si aucune des conditions `if` n'est vraie, nous retournons faux ❺.



## La fonction collision\_droite

La fonction de vérification de collision à droite ressemble fort à collision\_gauche :

---

```
❶ def collision_droite(co1, co2):
❷     if dans_y(co1, co2):
❸         if co1.x2 >= co2.x1 and co1.x2 <= co2.x2:
❹             return True
❺     return False
```

---

Comme dans collision\_gauche, nous vérifions si les coordonnées en *y* se chevauchent, à l'aide de la fonction dans\_y ❶. Nous vérifions ensuite si la valeur de *x2* est comprise entre les coordonnées *x1* et *x2* du second objet ❷ et nous renvoyons vrai ❸ si c'est le cas. Sinon, nous renvoyons faux ❹.

## La fonction collision\_haut

La fonction ressemble également aux deux précédentes :

---

```
❶ def collision_haut(co1, co2):
❷     if dans_x(co1, co2):
❸         if co1.y1 <= co2.y2 and co1.y1 >= co2.y1:
❹             return True
❺     return False
```

---

La différence se situe dans le fait que nous vérifions cette fois si les objets se chevauchent horizontalement ❶, à l'aide de la fonction dans\_x. Nous vérifions ensuite ❷ si la position supérieure du premier objet (*co1.y1*) a dépassé la position *y2* du second, mais pas sa position *y1*. Si c'est le cas, nous renvoyons vrai (ce qui signifie que le haut du premier objet a touché le second).

## La fonction collision\_bas

Pour la quatrième fonction de la série, il y a une subtile différence. La fonction collision\_bas prend un paramètre de plus :

---

```
❶ def collision_bas(y, co1, co2):
❷     if dans_x(co1, co2):
❸         y_calc = co1.y2 + y
❹         if y_calc >= co2.y1 and y_calc <= co2.y2:
❺             return True
❻     return False
```

---

Cette fonction accepte un paramètre supplémentaire, `y`, une valeur qu'on ajoute à la position `y` du premier objet `co1` de la classe `Coords`. Nous vérifions d'abord si les objets se chevauchent horizontalement ❶, comme dans `collision_haut`. Ensuite, nous ajoutons la valeur du paramètre `y` à la position `y2` du premier objet, `co1`, et nous stockons le résultat dans la variable `y_calc` ❷. Si cette nouvelle valeur est comprise entre les valeurs `y1` et `y2` du second objet ❸, nous renvoyons vrai ❹ parce que le bas de l'objet `co1` a touché le haut de l'objet `co2`. Cependant, si aucune des conditions des instructions `if` n'est vraie, alors nous retournons faux ❺.

Le nouveau paramètre `y` est nécessaire ici, parce que M. Filiforme peut tomber d'une plate-forme. À l'inverse des autres fonctions de détection de collision, nous devons être en mesure de tester s'il *pourrait* heurter le bas d'une plate-forme au lieu de tester s'il l'a *déjà* touché. Si le personnage avançait, quittait une plate-forme et flottait dans l'air, le jeu manquerait de réalisme. Donc, pendant qu'il marche, nous vérifions s'il entre en collision avec quelque chose à gauche et à droite. En revanche, lorsque nous vérifions ce qui se passe en dessous de lui, nous regardons s'il devrait toucher la plate-forme. Sinon, il doit tomber au sol.

## Créer la classe Lutin

La classe parente de tous les lutins du jeu porte le nom `Lutin`. Elle contient deux fonctions : `déplacer` pour bouger le lutin et `coords` pour renvoyer sa position actuelle à l'écran. Voici le code de la classe `Lutin` :

---

```
class Lutin:
❶    def __init__(self, jeu):
❷        self.jeu = jeu
❸        self.finjeu = False
❹        self.coordonnees = None
❺    def déplacer(self):
❻        pass
❼    def coords(self):
➋        return self.coordonnees
```

---

La fonction `__init__` de la classe `Lutin` prend un unique paramètre ❶, qui sera l'objet `jeu` défini globalement au niveau du programme. Nous en avons besoin pour que chaque lutin créé soit capable d'accéder à la liste des autres lutins du jeu. Nous mémorisons le paramètre `jeu` dans une variable d'objet ❷.

Nous déclarons ensuite ❸ la variable d'objet `finjeu`, qui servira à indiquer que la partie est terminée. Pour l'instant, elle est réglée à faux. La dernière variable d'objet, `coordonnees`, est déclarée sans aucune valeur pour l'instant, puisque sa valeur est définie à `None` ❹.

#### NOTE

*La variable `coordonnees` de la classe `Lutin` sert ici à mémoriser les coordonnées du lutin. Ce sera un objet de la classe `Coords`. La fonction de classe, `coords`, permet de renvoyer cet objet qui représente des coordonnées. Pour l'instant, la variable `coordonnees` de la classe `Lutin` est définie à `None`.*

La fonction `deplacer` ❺ ne fait encore rien dans sa classe parente, donc nous utilisons le mot-clé `pass` ❻ dans son corps. Enfin, la fonction `coords` ❼ renvoie simplement la variable d'objet `coordonnees` ❼.

Ainsi donc, nous voici avec une classe `Lutin` dotée d'une fonction `deplacer` qui ne fait rien et d'une fonction `coords` qui ne renvoie aucune coordonnée. En revanche, nous savons que toute classe qui possède `Lutin` en tant que parent disposera toujours des fonctions `deplacer` et `coords`. Au moins, dans la boucle principale du programme, quand nous bouclerons parmi la liste des lutins, nous pourrons appeler leur fonction `deplacer` sans risquer de recevoir d'erreur. Pourquoi ? Parce que tout lutin possède (ou hérite de) cette fonction.



#### NOTE

*Les classes avec des fonctions qui ne font pas grand-chose sont assez courantes en programmation. D'une certaine manière, elles forment une sorte de « contrat » qui garantit que tous les enfants de ces classes offrent les mêmes genres de fonctionnalités, même si, dans certains cas, les fonctions ne font rien dans les classes filles.*

## Ajouter les plates-formes

Il est temps d'ajouter les plates-formes. La classe portera le nom `LutinPlateForme` et sera une classe fille de `Lutin`. Sa fonction `__init__` prendra en paramètre `jeu` (comme la classe parente), plus une image, les positions `x` et `y`, ainsi que les largeur et hauteur d'image. Voici le code de la classe `LutinPlateForme` :

---

```
❶ class LutinPlateForme(Lutin):
❷     def __init__(self, jeu, image_photo, x, y, largeur, hauteur):
❸         Lutin.__init__(self, jeu)
❹         self.image_photo = image_photo
❺         self.image = jeu.canvas.create_image(x, y,\n                image=self.image_photo, anchor='nw')
❻         self.cordonnees = Coords(x, y, x + largeur, y + hauteur)
```

---

Quand nous déclarons la classe `LutinPlateForme` ❶, nous ne lui fournissons qu'un seul paramètre, `Lutin`, le nom de la classe parente. En revanche, la fonction `__init__` ❷ reçoit sept paramètres : `self`, `jeu`, `image_photo`, `x`, `y`, `largeur` et `hauteur`.

La troisième ligne appelle la fonction `__init__` de la classe parente ❸, avec les valeurs de paramètres `self` et `jeu`, parce qu'à part le mot-clé `self`, la fonction `__init__` de la classe `Lutin` prend un seul paramètre : `jeu`.

Si nous devions créer un objet de la classe `LutinPlateForme` à ce stade, il hériterait de toutes les variables d'objet de sa classe parente (`jeu`, `finjeu` et `cordonnees`), simplement parce que nous avons appelé la fonction `__init__` de `Lutin`.

Nous mémorisons ensuite le paramètre `image_photo` ❹ dans une variable d'objet et nous utilisons la variable `canvas` de l'objet `jeu` (passé en paramètre) pour dessiner l'image à l'écran, à l'aide de `create_image` ❺.

Enfin, nous créons un objet `cordonnees` de la classe `Coords` avec les paramètres `x` et `y` en premiers arguments. Nous ajoutons ensuite les paramètres `largeur` et `hauteur` à ces paramètres pour les seconds arguments de coordonnées ❻.

Même si la variable `cordonnees` est mise à `None` dans la classe parente `Lutin`, nous la modifions donc dans sa classe fille `LutinPlateForme`, en fonction de l'objet de classe `Coords` réel, contenant l'emplacement réel de l'image de plate-forme à l'écran.



## Ajouter un objet plate-forme

Ajoutons un objet plate-forme au jeu pour voir à quoi il ressemble. Modifie comme suit les deux dernières lignes du fichier du programme `jeufiliforme.py` :

---

```
❶ jeu = Jeu()
❷ plateforme1 = LutinPlateForme(jeu, PhotoImage(\n        file="plate-forme1.gif"), 0, 480, 100, 10)
```

---

```
③ jeu.lutins.append(plateforme1)
④ jeu.boucle_principale()
```

---

Si les lignes ❸ et ❹ n'ont pas changé, la ligne ❷ crée un objet de la classe `LutinPlateForme` et lui passe la variable `jeu`, un objet `PhotoImage` (avec le premier fichier image de plate-forme, `plate-forme1.gif`). Nous lui passons l'emplacement où nous voulons dessiner cette plate-forme (0 pixel du bord gauche et 480 pixels vers le bas), ainsi que la largeur et la hauteur de la plate-forme. Ce lutin vient s'ajouter à la liste de ceux de l'objet `jeu` ❸.

Si tu exécutes le jeu dès maintenant, tu dois voir une plate-forme apparaître dans le coin inférieur gauche de la fenêtre :



## Ajouter d'autres plates-formes

Ajoutons une série de plates-formes. Chacune d'elles a des emplacements *x* et *y* différents, pour les disséminer dans tout l'écran. Voici le code nécessaire :

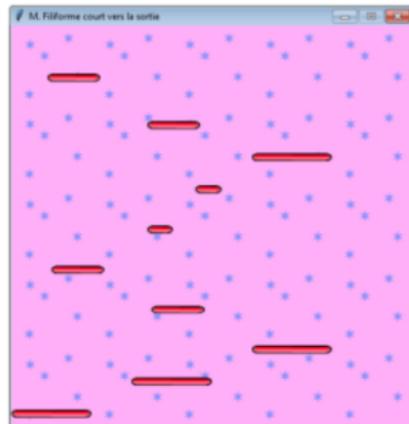
---

```
jeu = Jeu()
plateforme1 = LutinPlateForme(jeu, PhotoImage(file="plate-forme1.gif"), \
    0, 480, 100, 10)
plateforme2 = LutinPlateForme(jeu, PhotoImage(file="plate-forme1.gif"), \
    150, 440, 100, 10)
plateforme3 = LutinPlateForme(jeu, PhotoImage(file="plate-forme1.gif"), \
    300, 400, 100, 10)
plateforme4 = LutinPlateForme(jeu, PhotoImage(file="plate-forme1.gif"), \
    300, 160, 100, 10)
```

```
plateforme5 = LutinPlateForme(jeu, PhotoImage(file="plate-forme2.gif"), \
    175, 350, 66, 10)
plateforme6 = LutinPlateForme(jeu, PhotoImage(file="plate-forme2.gif"), \
    50, 300, 66, 10)
plateforme7 = LutinPlateForme(jeu, PhotoImage(file="plate-forme2.gif"), \
    170, 120, 66, 10)
plateforme8 = LutinPlateForme(jeu, PhotoImage(file="plate-forme2.gif"), \
    45, 60, 66, 10)
plateforme9 = LutinPlateForme(jeu, PhotoImage(file="plate-forme3.gif"), \
    170, 250, 32, 10)
plateforme10 = LutinPlateForme(jeu, PhotoImage(file="plate-forme3.gif"), \
    230, 200, 32, 10)
jeu.lutins.append(plateforme1)
jeu.lutins.append(plateforme2)
jeu.lutins.append(plateforme3)
jeu.lutins.append(plateforme4)
jeu.lutins.append(plateforme5)
jeu.lutins.append(plateforme6)
jeu.lutins.append(plateforme7)
jeu.lutins.append(plateforme8)
jeu.lutins.append(plateforme9)
jeu.lutins.append(plateforme10)
jeu.boucle_principale()
```

---

Nous avons créé ainsi une dizaine d'objets `LutinPlateForme`, pour les mémoriser dans les variables `plateforme1` à `plateforme10`. Nous avons ensuite ajouté ces plates-formes à la liste `lutins`, créée dans la classe `Jeu`. Si tu exécutes le programme à ce stade, tu dois obtenir à peu près ceci :



Les bases du jeu sont à présent établies. Nous allons pouvoir ajouter notre personnage principal, M. Filiforme. Ce sera pour le prochain chapitre.

## Ce que tu as appris

Dans ce chapitre, tu as créé la classe `Jeu` et dessiné l'image d'arrière-plan sur la fenêtre, comme une sorte de papier peint. Tu as appris à déterminer comment une position horizontale ou verticale se situe dans les limites d'autres positions horizontales ou verticales à l'aide des fonctions `dans_x` et `dans_y`. Puis tu as exploité ces fonctions pour en créer d'autres qui déterminent si les coordonnées d'un objet entrent en collision avec celles d'un autre objet. Ces fonctions serviront dans le chapitre suivant, lorsque nous animerons M. Filiforme et que nous devrons déterminer s'il est entré en collision avec une plate-forme en se déplaçant dans le canevas.

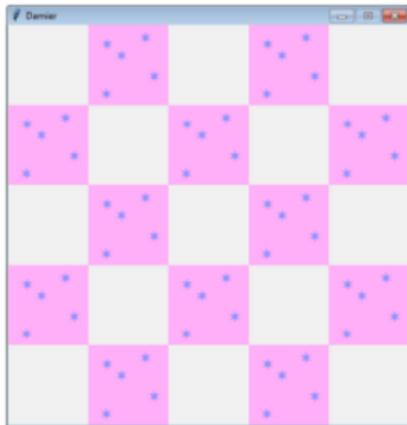
Nous avons également créé la classe parente `Lutin` et sa première classe fille, `LutinPlateForme`, qui nous a servi à dessiner des plates-formes dans le canevas.

## Puzzles de programmation

Les exercices de programmation suivants te proposent de t'entraîner à la création d'images de fond. Les réponses sont disponibles sur le site d'accompagnement du livre.

### 1. Damier

Essaie de modifier la classe `Jeu` pour que l'image de fond apparaisse comme dans un damier :



## 2. Damier à deux images alternées

Dès que tu as compris comment créer l'effet de damier, essaie à nouveau mais avec deux images différentes. Crée une seconde image d'arrière-plan à l'aide du programme de création graphique, puis modifie la classe `Jeu` pour qu'elle affiche un damier avec deux images alternées au lieu d'une sur un fond blanc.

## 3. Étagère et lampe

Pour donner un peu plus de piquant à des jeux comme celui-ci, tu peux créer des images d'arrière-plan différentes. Copie l'image d'arrière-plan sous un nouveau nom de fichier, puis dessine une étagère sur la copie. Tu pourrais aussi dessiner une table avec une lampe ou encore une fenêtre. Répartis ces images autour de l'écran en modifiant la classe `Jeu`, pour qu'elle charge ces images et affiche trois ou quatre images de fond différentes.





17

## CRÉER M. FILIFORME

Ce chapitre nous amène à créer le personnage principal du jeu M. Filiforme court vers la sortie. Il va exiger la programmation la plus complexe que nous ayons dû réaliser jusqu'ici, parce que M. Filiforme doit se déplacer à gauche, à droite, s'arrêter quand il heurte une plate-forme et tomber quand il en dépasse le bord. Nous allons bien entendu exploiter les liaisons d'événements de touches **Gauche** et **Droite** pour déplacer le personnage et nous devrons lui ajouter la possibilité de sauter d'une plate-forme à la suivante quand l'utilisateur presse la barre d'espace.

## Initialiser le personnage en fil de fer

La fonction `__init__` de la classe du personnage en fil de fer (`LutinPersonnage`, qui hérite de la classe parente `Lutin`) reprend l'aspect de celle des autres classes de notre jeu :

---

```
class LutinPersonnage(Lutin):
    def __init__(self, jeu):
        Lutin.__init__(self, jeu)
```

---



Ce code ressemble à celui que nous avions écrit pour la classe `LutinPlateForme` au chapitre 16, sauf qu'ici, aucun paramètre supplémentaire n'est nécessaire à part `self` et `jeu`. En effet, au contraire de la classe `LutinPlateForme`, il n'y aura qu'un seul objet `LutinPersonnage` dans le jeu.

## Charger les images du personnage

Comme nous avions un grand nombre d'objets à l'écran, qui utilisaient chacun des images de tailles différentes, nous avons passé l'image de la plate-forme en tant que paramètre de la fonction `__init__` de `LutinPlateForme`, ce qui revenait à dire au lutin d'une plate-forme de prendre telle ou telle image lors du dessin à l'écran. Pour le personnage en revanche, comme il n'y a qu'une seule instance, cela n'a pas de sens de charger son image en dehors du lutin et de la lui passer en paramètre. La classe `LutinPersonnage` chargera elle-même ses propres images.

Les quelques lignes suivantes de la fonction `__init__` s'occupent exactement de cette tâche : charger chacune des six images qui serviront à l'animation du personnage pendant qu'il court à gauche ou à droite. Nous devons charger ces images à ce stade et non pas (ce qui prendrait du temps et ralentirait beaucoup le jeu) lors de chaque affichage du personnage à l'écran.

---

```
class LutinPersonnage(Lutin):
    def __init__(self, jeu):
        Lutin.__init__(self, jeu)
        ❶ self.images_gauche = [
            PhotoImage(file="fil-G1.gif"),
            PhotoImage(file="fil-G2.gif"),
            PhotoImage(file="fil-G3.gif")
        ]
        ❷ self.images_droite = [
            PhotoImage(file="fil-D1.gif"),
```

```
    PhotoImage(file="fil-D2.gif"),
    PhotoImage(file="fil-D3.gif")
]
❸ self.image = jeu.canvas.create_image(200, 470, \
image=self.images_gauche[0], anchor='nw')
```

---

Nous créons les variables d'objet `images_gauche` ❶ et `images_droite` ❷. Chacune contient une liste d'objets `PhotoImage` à partir des images créées au chapitre 15, avec le personnage en pleine course vers la gauche ou la droite.

Nous dessinons la toute première image ❸, à partir d'`images_gauche[0]`, à l'aide de la fonction `create_image` du canevas et à l'emplacement `(200, 470)`, ce qui place le personnage au milieu de l'écran du jeu, au bas du canevas. Nous mémorisons son identifiant dans la variable d'objet `image`, pour un usage ultérieur (par la suite).

## Définir les variables

La partie suivante de la fonction `__init__` définit d'autres variables qui nous serviront dans le code.

---

```
self.images_droite = [
    PhotoImage(file="fil-D1.gif"),
    PhotoImage(file="fil-D2.gif"),
    PhotoImage(file="fil-D3.gif")
]
self.image = jeu.canvas.create_image(200, 470, \
image=self.images_gauche[0], anchor='nw')
❶ self.x = -2
❷ self.y = 0
❸ self.image_courante = 0
❹ self.ajout_image_courante = 1
❺ self.compte_sauts = 0
❻ self.derniere_heure = time.time()
❼ self.coordonnees = Coords()
```

---

Les variables `x` ❶ et `y` ❷ mémorisent les nombres à ajouter aux coordonnées horizontales (`x1` et `x2`) et verticales (`y1` et `y2`) lorsque le personnage se déplace à l'écran.

Au chapitre 13, nous avons vu que, pour animer quelque chose à l'aide du module `tkinter`, nous devons ajouter des valeurs aux positions `x` et `y` de l'objet pour le déplacer dans le canevas. La définition de `x` à `-2` et de `y` à `0` nous permet, plus loin dans le programme, de soustraire `2` à la position horizontale et rien du tout à la position verticale, pour faire avancer le personnage vers la gauche.

**NOTE**

Pour rappel, un nombre négatif en *x* signifie qu'on se déplace vers la gauche dans le canevas, tandis qu'un nombre positif signifie un déplacement vers la droite. Un nombre négatif en *y* signifie qu'on se rapproche du bord haut du canevas, donc qu'on monte, tandis qu'un nombre positif signifie qu'on descend.

Nous créons ensuite la variable d'objet `image_courante` ❸ pour mémoriser l'indice de position de l'image en cours d'affichage à l'écran. Notre liste d'images, `images_gauche`, contient `f1-G1.gif`, `f1-G2.gif` et `f1-G3.gif`, qui se situent aux indices respectifs 0, 1 et 2.

La variable `ajout_image_courante` ❹ contient le nombre que nous devons ajouter à cet indice de position pour connaître celui de la prochaine image de la liste. Ainsi, si l'image à l'indice 0 est affichée, nous devons ajouter 1 pour obtenir l'image à l'indice 1, puis ajouter encore 1 pour avoir l'image finale de la liste, à l'indice 2. Nous verrons comment exploiter cette variable dans l'animation au chapitre suivant.

La variable `compte_sauts` ❺ est un compteur destiné à la gestion des sauts du personnage. La variable `derniere_heure` mémorise la dernière fois que nous avons changé l'image pendant l'animation du personnage. L'heure actuelle est obtenue ❻ à l'aide de la fonction `time` du module `time`.

Enfin, nous définissons une variable d'objet `coordonnees` ❼ à partir d'un objet de la classe `Coords`, sans aucune initialisation de paramètres (`x1`, `y1`, `x2` et `y2` valent tous 0). Au contraire des plates-formes, les coordonnées du personnage devront changer, donc nous établirons ces valeurs par la suite.

## Lier les touches aux actions

Dans la partie finale de la fonction `__init__`, les liaisons d'événements associent des touches à une portion de code à exécuter lors de la pression sur ces touches.

---

```
self.compte_sauts = 0
self.derniere_heure = time.time()
self.coordonnees = Coords()
jeu.canvas.bind_all('<KeyPress-Left>', self.tourner_a_gauche)
jeu.canvas.bind_all('<KeyPress-Right>', self.tourner_a_droite)
jeu.canvas.bind_all('<space>', self.sauter)
```

---

Nous associons la touche `<KeyPress-Left>` (**Gauche**) à la fonction `tourner_a_gauche`, la touche `<KeyPress-Right>` (**Droite**) à la fonction `tourner_a_`

droite et <Space> (**Espace**) à la fonction **sauter**. Nous devons ensuite créer ces fonctions pour permettre réellement au personnage de se déplacer.

## Tourner le personnage vers la gauche ou la droite

Les fonctions **tourner\_a\_gauche** et **tourner\_a\_droite** vérifient que le personnage ne saute pas, puis définissent la valeur de la variable d'objet **x** pour qu'il puisse se déplacer dans un sens ou dans l'autre. Si le personnage saute, le jeu empêche de changer sa direction pendant qu'il est dans l'air.



---

```
jeu.canvas.bind_all('<KeyPress-Left>',  
self.tourner_a_gauche)  
jeu.canvas.bind_all('<KeyPress-Right>', self.tourner_a_droite)  
jeu.canvas.bind_all('<space>', self.sauter)
```

❶     **def tourner\_a\_gauche(self, evt):**  
❷         **if self.y == 0:**  
❸             **self.x = -2**

❹     **def tourner\_a\_droite(self, evt):**  
❺         **if self.y == 0:**  
❻             **self.x = 2**

---

Python appelle la fonction **tourner\_a\_gauche** (resp. **tourner\_a\_droite**) quand le joueur appuie sur la touche de flèche **Gauche** (resp. droite) et passe en paramètre un objet avec des informations sur ce que le joueur a provoqué. Il s'agit d'un **objet événement**, auquel nous donnons le nom de paramètre **evt** ❶ et ❹.

### NOTE

*L'objet événement n'a pas d'importance particulière dans notre cas mais nous devons l'inclure en paramètre de nos fonctions (❶ et ❹), sinon nous provoquerions une erreur, parce que Python s'attend à ce qu'il soit là. L'objet événement contient des informations comme les emplacements **x** et **y** de la souris (dans le cas d'un événement de souris), un code qui identifie une touche particulière (un événement de clavier), ainsi que d'autres informations.*

Pour savoir si le personnage est en train de sauter, nous examinons la valeur de la variable objet **y** ❷ et ❺. Si elle est différente de **0**, alors le personnage saute (ou tombe) ; nous réglons alors **x** à **-2** pour courir vers la gauche ❸ ou **+2** pour courir vers la droite ❻, notamment

parce que les valeurs **-1** et **+1** ne permettraient pas au personnage de courir suffisamment vite. Dès que l'animation fonctionnera, tu feras varier cette valeur pour voir la différence que cela engendre.

## Faire sauter le personnage

La fonction **sauter** ressemble aux deux précédentes :

---

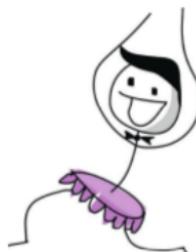
```
def tourner_a_droite(self, evt):
    if self.y == 0:
        self.x = 2

❶ def sauter(self, evt):
❷     if self.y == 0:
❸         self.y = -4
            self.compte_sauts = 0
```

---

Cette fonction prend l'objet événement **evt** en paramètre, que nous pouvons ignorer puisque nous n'avons besoin d'aucune information à propos de l'événement. En effet, nous savons déjà que si cette fonction a été appelée, c'est parce que la barre d'espace a été pressée.

Comme nous voulons que notre personnage saute seulement s'il n'est pas déjà en train de sauter, nous vérifions tout d'abord si **y** est bien égal à **0** ❶. Si c'est le cas (M. Filiforme ne saute pas déjà), nous réglons **y** à **-4** ❷, pour déplacer verticalement le personnage à l'écran, et **compte\_sauts** à **0** ❸. Nous avons besoin de cette dernière variable pour nous assurer que le personnage ne se met pas à sauter à l'infini. Nous devons en effet lui permettre de monter pendant un nombre donné de « sauts », puis le faire redescendre, comme si la gravité l'attirait vers le bas. Nous ajouterons cette portion de code au chapitre suivant.



## Ce que nous avons jusqu'ici

Révisons les définitions des classes et des fonctions dont nous disposons jusqu'ici dans le jeu. Voici comment elles doivent se situer dans le fichier.

Au sommet du programme, tu dois avoir les instructions `import`, suivies des définitions des classes `Jeu` et `Coords`. La classe `Jeu` permet de créer un objet qui sert de contrôleur principal du jeu et les objets de la classe `Coords` servent à conserver les emplacements des objets présents dans le jeu, comme les plates-formes et le personnage de M. Filiforme :

---

```
from tkinter import *
import random
import time

class Jeu:
    ...
class Coords:
    ...
```

---

Ensuite doivent apparaître les fonctions `dans` (qui indiquent si les coordonnées d'un lutin sont « dans » la même zone qu'un autre), la classe `Lutin` (qui constitue la classe parente de tous les lutins du jeu), la classe `LutinPlateForme`, (qui sert à créer les objets des plates-formes sur lesquelles le personnage devra sauter), ainsi que le début de la classe `LutinPersonnage` (pour représenter M. Filiforme) :

---

```
def dans_x(co1, co2):
    ...
def dans_y(co1, co2):
    ...
class Lutin:
    ...
class LutinPlateForme(Lutin):
    ...
class LutinPersonnage(Lutin):
    ...
```

---

Enfin, tout à la fin du programme, vient le code qui crée tous les objets du jeu définis jusqu'ici : l'objet `jeu` lui-même et les plates-formes. La ligne finale appelle la fonction `boucle_principale`.

---

```
jeu = Jeu()
plateforme1 = LutinPlateForme(jeu, PhotoImage(file="plate-forme1.gif"), \
    0, 480, 100, 10)
...
jeu.lutins.append(plateforme1)
...
jeu.boucle_principale()
```

---

Si ton code diffère un peu de ceci, ou si tu éprouves des difficultés à le faire fonctionner, tu peux toujours sauter à la fin du chapitre 18, où figure la liste complète des lignes de code du jeu.

## Ce que tu as appris

Dans ce chapitre, nous avons commencé à travailler sur la classe du personnage. Pour le moment, si nous créions un objet de cette classe, il ne ferait pas grand-chose, à part charger les images nécessaires pour animer M. Filiforme et définir un certain nombre de variables à utiliser plus tard dans le code. Cette classe contient des fonctions pour changer les valeurs des variables d'objet en fonction des événements de clavier (quand l'utilisateur presse les touches **Gauche**, **Droite** ou **Espace**).

Au chapitre suivant, nous terminerons le jeu. Nous écrirons les fonctions nécessaires à la classe **LutinPersonnage** pour afficher, animer M. Filiforme et le déplacer dans tout l'écran. Nous ajouterons aussi la porte de sortie qu'il doit tenter d'attendre.



18

# ACHEVER LE JEU DE M. FILIFORME

Aux trois chapitres précédents, nous avons développé une grande partie de notre jeu M. Filiforme court vers la sortie. Nous en avons créé les graphismes, puis rédigé le code pour ajouter les images du fond, des plates-formes et du personnage en fil de fer. Dans ce chapitre, nous allons compléter l'ensemble avec les pièces manquantes pour animer le personnage et ajouter la porte.

La fin du chapitre reprend la totalité du code du jeu. Si tu te sens un peu perdu dans le programme pendant la rédaction d'une partie de ce code, compare ce que tu as rédigé avec la liste complète pour repérer les différences et la source du problème.

## Animer le personnage

Jusqu'ici, nous avons créé une classe de base représentant le personnage, pour charger les images et associer des touches du clavier à certaines fonctions. Cependant, le code ne fait encore rien de particulièrement intéressant lorsque tu exécutes le programme à ce stade.

Nous allons à présent ajouter les fonctions restantes à la classe `LutinPersonnage` que nous avons créée au chapitre 17 : `animer`, `deplacer` et `coords`. La fonction `animer` devra dessiner les différentes images du personnage, `deplacer` devra déterminer où M. Filiforme doit se déplacer et `coords` renverra sa position actuelle. Au contraire de ce que nous avons écrit pour les lutins des plates-formes, nous devons recalculer l'emplacement du personnage à mesure qu'il se déplace à l'écran.



### Créer la fonction `animer`

Ajoutons d'abord la fonction `animer`, pour vérifier la présence d'un mouvement et changer l'image en fonction de celui-ci.

#### Vérifier la présence d'un mouvement

Nous ne devons pas changer trop vite l'image du personnage dans l'animation, sinon cette dernière paraîtra irréaliste. Rappelle-toi l'animation dessinée sur les coins du carnet de papier : si tu fais tourner les pages trop rapidement, tu ne vois plus correctement l'effet de ce que tu as dessiné.

La première partie de la fonction `animer` vérifie si le personnage court à gauche ou à droite, puis utilise la variable d'objet `derniere_heure` pour décider s'il faut changer l'image en cours. Cette variable nous servira à contrôler la vitesse de l'animation. La fonction `animer` vient à la suite de la fonction `sauter`, que nous avons ajoutée à la classe `LutinPersonnage` au chapitre 17.

---

```
def sauter(self, evt):
    if self.y == 0:
        self.y = -4
        self.compte_sauts = 0

def animer(self):
```

```
❶ if self.x != 0 and self.y == 0:  
❷     if time.time() - self.derniere_heure > 0.1:  
❸         self.derniere_heure = time.time()  
❹         self.image_courante += self.ajout_image_courante  
❺         if self.image_courante >= 2:  
❻             self.ajout_image_courante = -1  
❼         if self.image_courante <= 0:  
➋             self.ajout_image_courante = 1
```

---

La première instruction `if` ❶ vérifie que `x` ne vaut pas `0`, pour déterminer si le personnage se déplace (que ce soit à gauche ou à droite), et vérifie ensuite si `y` est bien égal à `0`, ce qui signifie qu'il ne saute pas. Si ces deux conditions sont vraies, il faut animer le personnage, sinon, il reste tel quel, donc il n'est pas nécessaire d'aller plus loin dans le dessin. Si le personnage ne se déplace pas horizontalement, alors nous laissons de côté le reste de cette suite d'instructions.

La ligne suivante calcule le temps écoulé entre la valeur de la variable `derniere_heure` et l'heure actuelle `time.time()` ❷. Ce calcul permet de savoir s'il faut afficher l'image suivante dans la séquence, ce que nous faisons en pratique si le temps écoulé est supérieur à un dixième de seconde (0.1). Là, nous mémorisons dans `derniere_heure` le temps actuel ❸, ce qui réinitialise le chronomètre pour surveiller le prochain changement d'image.

Nous ajoutons la valeur de la variable d'objet `ajout_image_courante` à la variable `image_courante` ❹, qui stocke l'indice de l'image affichée actuellement. Rappelle-toi que nous avions défini la variable `ajout_image_courante` dans la fonction `__init__` de la classe au chapitre 17 et donc que, quand la fonction `animer` est appelée pour la première fois, la valeur de cette variable est de `1`.

Nous vérifions si la valeur de l'indice de position dans `image_courante` est supérieure ou égale à `2` ❺, auquel cas nous modifions la valeur `d'ajout_image_courante` en `-1` ❻. De même, dès que nous atteignons `0` ❼, nous recommençons à compter vers le haut ➋.

### NOTE

*Si tu peines à te représenter le retrait à appliquer à ce code, voici un conseil : il y a 8 espaces au début de la ligne ❶ et 20 espaces au début de la ligne ➋.*

Pour t'aider à comprendre ce qui se passe dans la fonction jusqu'ici, imagine que tu aies une suite de blocs colorés en ligne sur le plancher. Tu déplaces ton doigt d'un bloc au suivant et chaque bloc que pointe ton doigt (1, 2, 3, 4 et ainsi de suite) porte un numéro,

celui que contient la variable `image_courante`. Le nombre de blocs dont ton index se déplace pour pointer le suivant (il ne vise qu'un bloc à un moment donné) est le nombre mémorisé dans la variable `ajout_image_courante`. Quand ton doigt se déplace d'une rangée vers le haut (les numéros de plus en plus élevés) parmi la ligne de blocs, tu ajoutes 1 à chaque fois et quand il touche la fin de la ligne et redescend (vers les numéros les plus petits), tu soustrais 1, donc tu ajoutes -1.

Le code que nous avons ajouté à la fonction `animer` traite ce processus mais, au lieu de blocs colorés, nous avons les trois images du personnage pour chaque direction, mémorisées dans une liste. Les indices de ces images sont 0, 1 et 2. Comme nous animons le personnage, dès que nous atteignons la dernière image, nous commençons à décompter et, dès que nous atteignons la première image, nous devons recommencer à compter vers le haut. En résultat, nous obtenons l'effet du personnage qui court.

Voici une représentation du phénomène à travers la liste des images, avec les indices de position calculés par la fonction `animer`.

Position 0	Position 1	Position 2	Position 1	Position 0	Position 1
Comptage haut	Comptage haut	Comptage haut	Comptage bas	Comptage bas	Comptage haut
					

### Changer d'image

Dans la seconde partie de la fonction `animer`, nous changeons l'image en cours d'affichage, à l'aide de l'indice de position calculé.

---

```
def animer(self):  
    if self.x != 0 and self.y == 0:  
        if time.time() - self.derniere_heure > 0.1:  
            self.derniere_heure = time.time()  
            self.image_courante += self.ajout_image_courante  
            if self.image_courante >= 2:  
                self.ajout_image_courante = -1  
            if self.image_courante <= 0:  
                self.ajout_image_courante = 1  
①     if self.x < 0:  
②         if self.y != 0:  
③             self.jeu.canvas.itemconfig(self.image, \
```

```
    image=self.images_gauche[2])
❸ else:
❹     self.jeu.canvas.itemconfig(self.image, \
        image=self.images_gauche[self.image_courante])
❺ elif self.x > 0:
❻     if self.y != 0:
❼         self.jeu.canvas.itemconfig(self.image, \
            image=self.images_droite[2])
❽ else:
❾     self.jeu.canvas.itemconfig(self.image, \
        image=self.images_droite[self.image_courante])
```

---

Si `x` est plus petit que `0` ❶, alors le personnage se déplace vers la gauche et Python va dans le bloc de code compris entre les lignes ❷ et ❸. Si `y` est différent de `0` (le personnage monte ou descend, donc saute), la fonction `itemconfig` du canevas change l'image affichée ❹ : comme le personnage saute, nous affichons l'image de lui en pleine extension (`images_gauche[2]`), pour donner plus de réalisme à l'animation :



Si le personnage ne saute pas, c'est-à-dire si `y` est égal à `0`, l'instruction `else` ❽ utilise `itemconfig` pour changer l'image affichée en celle de l'indice donné par la variable `image_courante` ❾.

Ensuite, nous vérifions si le personnage court vers la droite (`x` est plus grand que `0` ❻) et, si c'est le cas, Python passe dans le bloc compris entre les lignes ❼ et ➋. Ce code ressemble fort à celui du bloc précédent, puisqu'il vérifie de nouveau si le personnage saute, pour afficher dans les deux cas l'image correcte mais à partir de la liste `images_droite`, cette fois.

## Connaître l'emplacement du personnage

Comme le personnage est supposé se déplacer à l'écran, nous devons connaître en permanence son emplacement. La fonction `coords` de la classe `LutinPersonnage` utilise la fonction éponyme (`coords`) du canevas pour déterminer où se trouve le personnage, puis exploite ces valeurs pour régler les valeurs `x1`, `y1`, `x2` et `y2` de la variable `coordonnees`, que nous avons déclarée dans la fonction `_init_` au chapitre 17. Voici le code qui vient s'ajouter à la suite de la fonction `animer` :

---

```
    if self.x < 0:
        if self.y != 0:
            self.jeu.canvas.itemconfig(self.image, \
                image=self.images_gauche[2])
        else:
            self.jeu.canvas.itemconfig(self.image, \
                image=self.images_gauche[self.image_courante])
    elif self.x > 0:
        if self.y != 0:
            self.jeu.canvas.itemconfig(self.image, \
                image=self.images_droite[2])
        else:
            self.jeu.canvas.itemconfig(self.image, \
                image=self.images_droite[self.image_courante])

def coords(self):
    ❶    xy = self.jeu.canvas.coords(self.image)
    ❷    self.cordonnees.x1 = xy[0]
    ❸    self.cordonnees.y1 = xy[1]
    ❹    self.cordonnees.x2 = xy[0] + 27
    ❺    self.cordonnees.y2 = xy[1] + 30
    return self.cordonnees
```

---

Lorsque nous avons défini la classe `Jeu` au chapitre 16, nous y avons créé la variable d'objet `canvas`. La fonction `coords` de la variable d'objet `canvas` est appelée ❶, pour obtenir les positions  $x$  et  $y$  de l'image en cours (`self.image`). Cette fonction exploite le numéro mémorisé dans la variable d'objet `image` pour connaître l'identifiant de l'image du personnage affichée à l'écran.

Nous stockons la liste résultante dans la variable `xy`, qui contient donc deux valeurs : la position en  $x$  du coin supérieur gauche, enregistrée dans la variable `x1` de `cordonnees` ❷, et la position en  $y$  de ce même coin supérieur gauche, enregistrée dans la variable `y1` de `cordonnees` ❸.

Comme toutes les images du personnage ont 27 pixels de large sur 30 pixels de haut, nous pouvons déterminer ce qu'il faut ajouter en largeur ❹ et en hauteur ❺, respectivement aux coordonnées  $x$  et  $y$  pour connaître les valeurs des variables `x2` et `y2` de `cordonnees`.

Enfin, la dernière ligne de la fonction renvoie la variable d'objet `cordonnees`.

## Déplacer le personnage

La dernière fonction de la classe `LutinPersonnage`, `deplacer`, prend en charge l'animation réelle du personnage, pendant qu'il se déplace à

l'écran. Elle doit aussi nous indiquer si le personnage heurte quelque chose en se déplaçant.

### Début de la fonction `deplacer`

Voici la première partie du code de la fonction `deplacer`, qui se situe à la suite de `coords` :

---

```
def coords(self):
    # xy = list(self.jeu.canvas.coords(self.image))
    xy = self.jeu.canvas.coords(self.image)
    self.coordonnees.x1 = xy[0]
    self.coordonnees.y1 = xy[1]
    self.coordonnees.x2 = xy[0] + 27
    self.coordonnees.y2 = xy[1] + 30
    return self.coordonnees

def deplacer(self):
    ❶    self.animer()
    ❷    if self.y < 0:
    ❸        self.compte_sauts += 1
    ❹        if self.compte_sauts > 20:
    ❺        self.y = 4
    ❻    if self.y > 0:
    ❼        self.compte_sauts -= 1
```

---

La ligne ❶ appelle la fonction `animer` que nous avons écrite précédemment, pour changer l'image affichée si nécessaire. La ligne ❷ vérifie si la valeur de `y` est plus petite que 0. Si c'est le cas, nous savons que le personnage saute, parce qu'une valeur négative de `y` fait monter le lutin dans le canevas. Rappelle-toi que le 0 se situe au bord supérieur du canevas et que le bas du canevas se trouve en `y` = 500 pixels.

Nous ajoutons 1 à `compte_sauts` ❸. Si la valeur de cette variable atteint 20 ❹, nous devons modifier `y` ❺ pour indiquer que le personnage commence à retomber.

Ensuite, nous vérifions si la valeur de `y` est plus grande que 0 ❻ (ce qui signifie que le personnage retombe) et, si c'est le cas, nous soustrayons 1 de `compte_sauts` ❼. En effet, dès que nous avons compté jusqu'à 20, nous devons recommencer à décompter. Avec ta main, monte progressivement en comptant jusqu'à 20, puis décompte de 20 à 0 en redescendant tout doucement : ainsi, tu as une idée de la manière de calculer le saut du personnage et sa façon de retomber.



Aux lignes suivantes de la fonction `deplacer`, nous appelons la fonction `coords`, qui nous indique où se situe le personnage à l'écran, et nous mémorisons son résultat dans la variable `co`. Nous créons ensuite les variables `gauche`, `droite`, `haut`, `bas` et `tombe`, que nous utiliserons dans la suite de la fonction.

---

```
if self.y > 0:  
    self.compte_sauts -= 1  
co = self.coords()  
gauche = True  
droite = True  
haut = True  
bas = True  
tombe = True
```

---

Note que chacune de ces variables a été définie avec une valeur booléenne, `True`. Elles nous serviront d'indicateurs pour vérifier si le personnage a heurté quelque chose à l'écran ou s'il tombe.

### Le personnage a-t-il touché le bas ou le haut du canevas ?

La section suivante de la fonction `deplacer` vérifie si le personnage touche le haut ou le bas du canevas. Voici le code :

---

```
bas = True  
tombe = True  
❶ if self.y > 0 and co.y2 >= self.jeu.hauteur_canevas:  
❷     self.y = 0  
❸     bas = False  
❹ elif self.y < 0 and co.y1 <= 0:  
❺     self.y = 0  
❻     haut = False
```

---

Si le personnage tombe, `y` est plus grand que `0` et nous devons vérifier qu'il n'a pas touché le bas du canevas (ou sinon, il disparaîtrait en bas de l'écran), c'est-à-dire que sa position `y2` (qui marque le bas du personnage) est supérieure ou égale à la variable `hauteur_canevas` de l'objet `jeu` ❶. Si c'est le cas, alors nous ramenons la valeur de `y` à `0` ❷, pour arrêter le personnage dans sa chute, puis nous donnons à la variable `bas` la valeur `False` ❸ pour indiquer au reste du code de la fonction qu'il n'est plus nécessaire de vérifier si le personnage a touché le bas.

La procédure est similaire pour déterminer si le personnage a touché le haut de l'écran. Nous vérifions d'abord si M. Filiforme saute (`y` plus petit que `0`), puis si le haut du personnage (`y1`) a touché

le bord haut (`<= 0`) ❶. Si ces deux conditions sont vraies, alors nous réglons `y` à `0` ❷, pour arrêter le mouvement. Ensuite, nous réglons à `False` la variable `haut` ❸, pour indiquer au reste du code qu'il n'est plus nécessaire de vérifier si le personnage a touché le haut.

### Le personnage a-t-il touché un côté du canevas ?

Le procédé pour vérifier si le personnage touche un des bords gauche ou droit du canevas, pour corriger le mouvement si nécessaire, s'inspire fort du code précédent :

---

```
    elif self.y < 0 and co.y1 <= 0:  
        self.y = 0  
        haut = False  
❶    if self.x > 0 and co.x2 >= self.jeu.largeur_canevas:  
❷        self.x = 0  
❸        droite = False  
❹    elif self.x < 0 and co.x1 <= 0:  
❺        self.x = 0  
❻        gauche = False
```

---

Nous savons que `x` est plus grand que `0` si le personnage se déplace vers la droite. Nous savons aussi que, s'il touche le côté droit du canevas, alors sa position `x2` (`co.x2`) est plus grande que la largeur du canevas, mémorisée dans `jeu.largeur_canevas`. Si les deux conditions sont vraies ❶, alors nous ramenons le déplacement en `x` à `0` ❷, pour arrêter le personnage dans sa course, puis nous réglons la variable `droite` à `False` ❸. Les lignes ❹ à ❻ font de même pour le côté gauche.

### Collision avec d'autres lutins

Dès que nous avons déterminé si le personnage touche les quatre bords de l'écran, nous pouvons vérifier ensuite s'il heurte autre chose à l'écran. Le code suivant boucle parmi la liste des objets `lutins` stockés dans l'objet `jeu`, pour vérifier si le personnage en touche un.

---

```
    elif self.x < 0 and co.x1 <= 0:  
        self.x = 0  
        gauche = False  
❶    for lutin in self.jeu.lutins:  
❷        if lutin == self:  
❸            continue  
❹        co_lutin = lutin.coords()  
❺        if haut and self.y < 0 and collision_haut(co, co_lutin):  
❻            self.y = -self.y  
❼        haut = False
```

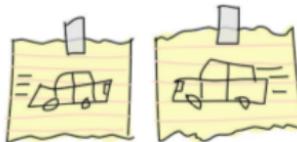
---

La ligne ❶ débute une boucle parmi la liste des `lutins` et affecte chacun d'eux, tour à tour, à la variable `lutin`. Si ce lutin est égal à `self` ❷ (ce qui revient à dire « si ce lutin est égal à moi-même »), alors il n'y a rien à faire puisque le lutin ne peut entrer en collision avec lui-même : le mot-clé `continue` ❸ sert à ignorer la suite du bloc de code de la boucle et à passer directement au lutin suivant de la liste.

Ensuite, nous mémorisons les coordonnées du nouveau lutin par un appel à sa fonction `coords`, dans la variable `co_lutin` ❹.

La ligne suivante ❺ vérifie ceci :

- le personnage n'a pas touché le haut du canevas (la variable `haut` doit encore valoir `True`) ;
- le personnage saute (la valeur de `y` est inférieure à `0`) ;
- le haut du personnage est entré en collision avec le lutin de la liste (à l'aide de la fonction `collision_haut` créée au chapitre 16).



Si toutes ces conditions sont vraies, alors le personnage doit redescendre, donc nous renversons la valeur de `y` à l'aide d'un moins (-) ❻. Nous mettons la variable `haut` à `False` ❼, parce que, dès que le personnage a touché le haut, il n'est plus nécessaire, à ce moment-là, de vérifier la présence d'une collision.

### Collision en bas

La partie suivante de la boucle vérifie si le bas du personnage a touché quelque chose :

---

```
if haut and self.y < 0 and collision_haut(co, co_lutin):
    self.y = -self.y
    haut = False
❶ if bas and self.y > 0 and \
        collision_bas(self.y, co, co_lutin):
❷     self.y = co_lutin.y1 - co.y2
❸     if self.y < 0:
❹         self.y = 0
❺     bas = False
❻     haut = False
```

---

Nous vérifions si la variable `bas` est encore vraie, si le personnage tombe (`y` est plus grand que `0`) et si le bas du personnage touche le lutin ❶. Si ces trois conditions sont vraies, alors nous soustrayons la valeur `y` du bas du personnage (`y2`) de la valeur `y` du haut du

lutin (`y1`) ❸. Ceci peut sembler étrange donc voyons pourquoi il faut le faire.

Supposons que le personnage du jeu tombe d'une plate-forme. Il se déplace vers le bas de 4 pixels à chaque tour de la `boucle_principale`. Or, supposons que le pied du personnage soit à 3 pixels du haut d'une plate-forme. En pratique, imaginons que le bas du personnage (`y2`) soit en position 57 et que le haut de la plate-forme (`y1`) soit en position 60. Dans ce cas-ci, la fonction `collision_bas` renverrait vrai, parce que son code ajouterait la valeur de `y` (qui vaudrait 4) à la variable `y2` du personnage, ce qui donnerait 61.

Cependant, nous ne voulons pas que M. Filiforme s'arrête de tomber dès qu'il semble qu'il va bientôt toucher une plate-forme ou le bas de l'écran, parce que cela reviendrait à le voir tomber brutalement et s'arrêter en l'air, à distance du sol ou de la plate-forme. Ce serait peut-être rigolo mais cela donnerait au jeu un genre bizarre. Au lieu de cela, nous soustrayons la valeur du `y2` du personnage (57) de la valeur du `y1` de la plate-forme (60) pour obtenir 3 ; c'est de cette valeur que le personnage doit tomber pour atterrir proprement sur le sol ou la plate-forme.

Nous vérifions que ce calcul ne donne pas une valeur négative et, si c'est le cas ❹, nous forçons la valeur de `y` à 0 ❺. Si nous laissons cette valeur négative, le personnage rebondirait et s'envolerait, ce que nous ne souhaitons pas vraiment voir dans notre jeu.

Enfin, nous mettons les drapeaux indicateurs `haut` ❻ et `bas` ❼ à `False`, pour ne plus perdre de temps à vérifier si le personnage a touché le haut, le bas de l'écran ou un autre lutin.

Nous devons encore effectuer une vérification sur le bas, pour détecter si le personnage a avancé au-delà du bord d'une plate-forme, avec le code suivant :

---

```
if self.y < 0:  
    self.y = 0  
bas = False  
haut = False  
if bas and tombe and self.y == 0 \  
    and co.y2 < self.jeu.hauteur_canevas \  
    and collision_bas(1, co, co_lutin):  
    tombe = False
```

---

Cinq vérifications sont nécessaires pour mettre la variable `tombe` à `False` :

- le drapeau `bas` doit être à `True` ;
- nous devons vérifier si le personnage peut tomber, donc si le drapeau `tombe` est encore à `True` ;
- il faut aussi que le personnage ne tombe pas déjà (`y` vaut `0`) ;
- le bas du personnage ne doit pas avoir atteint le bas de l'écran (donc il est inférieur à la hauteur du canevas) ;
- il faut enfin que le personnage touche une plate-forme (donc que `collision_bas` renvoie vrai).

Alors seulement, nous réglons la variable `tombe` à `False`.

### Vérifier la gauche et la droite

À ce stade, nous avons vérifié si le personnage a touché un lutin en haut ou en bas. Nous devons ensuite vérifier s'il touche le côté gauche ou droit, avec le code suivant :

---

```

if bas and tombe and self.y == 0 \
    and co.y2 < self.jeu.hauteur_canevas \
    and collision_bas(1, co, co_lutin):
    tombe = False
❶ if gauche and self.x < 0 and collision_gauche(co, co_lutin):
❷     self.x = 0
❸     gauche = False
❹ if droite and self.x > 0 and collision_droite(co, co_lutin):
❺     self.x = 0
❻     droite = False

```

---

Nous vérifions si nous devons toujours chercher la présence de collisions à gauche (`gauche` est encore vrai), si le personnage se déplace vers la gauche (`x` vaut moins de `0`) et s'il entre en collision avec un lutin, à l'aide la fonction `collision_gauche`.

Si ces trois conditions sont vraies ❶,

nous mettons `x` à `0` ❷, pour arrêter le personnage dans sa course, et réglons `gauche` à `False` ❸, pour ne plus devoir vérifier de collision à gauche.

Pour la détection des collisions à droite ❹, le code suit le même principe. Nous réglons également `x` à `0` ❺ et `droite` à `False` ❻, pour cesser de vérifier par la suite des collisions à droite.

Avec les vérifications de collisions dans les quatre directions, la boucle `for` devient à présent :



---

```
        elif self.x < 0 and co.x1 <= 0:
            self.x = 0
            gauche = False
        for lutin in self.jeu.lutins:
            if lutin == self:
                continue
            co_lutin = lutin.coords()
            if haut and self.y < 0 and collision_haut(co, co_lutin):
                self.y = -self.y
                haut = False
            if bas and self.y > 0 and collision_bas(self.y, co, \
                co_lutin):
                self.y = co_lutin.y1 - co.y2
                if self.y < 0:
                    self.y = 0
                bas = False
                haut = False
            if bas and tombe and self.y == 0 \
                and co.y2 < self.jeu.hauteur_canevas \
                and collision_bas(1, co, co_lutin):
                tombe = False
            if gauche and self.x < 0 \
                and collision_gauche(co, co_lutin):
                self.x = 0
                gauche = False
            if droite and self.x > 0 \
                and collision_droite(co, co_lutin):
                self.x = 0
                droite = False
```

---

Il nous reste à ajouter encore les quelques lignes suivantes à la fonction `deplacer` :

---

```
    if droite and self.x > 0 \
        and collision_droite(co, co_lutin):
        self.x = 0
        droite = False
❶    if tombe and bas and self.y == 0 \
        and co.y2 < self.jeu.hauteur_canevas:
❷        self.y = 4
❸    self.jeu.canvas.move(self.image, self.x, self.y)
```

---

Si les deux variables `tombe` et `bas` sont toujours vraies ❶, alors nous avons bouclé parmi tous les lutins des plates-formes de la liste sans avoir détecté de collision en bas.

Le dernier test de la ligne détermine si le bas du personnage est situé à moins de la hauteur du canevas, c'est-à-dire au-dessus du sol (ou du bas du canevas). Si le personnage n'a rien heurté et s'il

est au-dessus du sol, alors il se retrouve suspendu dans l'air (autrement dit, il a dépassé le bord d'une plate-forme pour se retrouver en l'air), donc il doit commencer à tomber. Pour obtenir cet effet, nous réglons `y` à 4 ❷.

Nous pouvons maintenant déplacer l'image à l'écran ❶, selon les valeurs établies dans les variables `x` et `y`. Comme nous avons bouclé parmi tous les lutins pour détecter des collisions, il peut se produire que les deux variables se retrouvent égales à 0, parce que le personnage a touché le bord gauche et qu'il est en contact avec le sol ou une plate-forme. Dans ce cas, l'appel à la fonction `move` ne fait rien, autrement dit, le personnage ne bouge plus.

Dans le cas où M. Filiforme marche au-delà d'une plate-forme, auquel cas `y` est égal à 4, notre agent spécial tombe.

Ouf, c'était là une bien longue fonction !

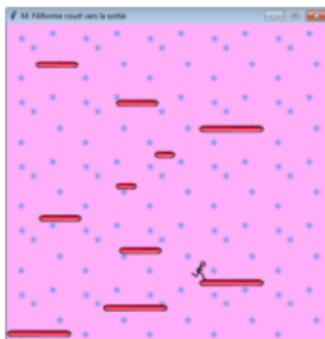
## Tester le lutin du personnage

La classe `LutinPersonnage` achevée, il nous faut l'essayer en situation. Ajoute les deux lignes de code suivantes, juste avant l'appel à la fonction `boucle_principale` :

- 
- ❶ `personnage = LutinPersonnage(jeu)`
  - ❷ `jeu.lutins.append(personnage)`  
`jeu.boucle_principale()`
- 

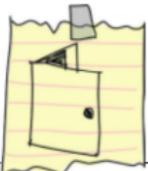
Nous créons un objet `LutinPersonnage` que nous mémorisons dans la variable `personnage` ❶. Comme nous l'avions fait avec les plates-formes, nous ajoutons cette variable à la liste des lutins stockée dans l'objet `jeu` ❷.

Exécute le programme. M. Filiforme peut désormais courir, sauter de plate-forme en plate-forme, et tomber !



## La porte !

Le seul élément qui manque à notre jeu, c'est la porte de sortie. Nous allons donc terminer par l'ajout d'un lutin pour la porte, l'ajout des lignes nécessaires pour détecter cette dernière et la fourniture au programme d'un objet `porte`.



### Créer la classe `LutinPorte`

Tu l'auras deviné, nous devons créer une nouvelle classe, `LutinPorte`, dont voici le code de début :

```
class LutinPorte(Lutin):
    ❶    def __init__(self, jeu, image_photo, x, y, largeur, hauteur):
    ❷        Lutin.__init__(self, jeu)
    ❸        self.image_photo = image_photo
    ❹        self.image = jeu.canvas.create_image(x, y, \
                                             image=self.image_photo, anchor='nw')
    ❺        self.cordonnees = Coords(x, y, x + (largeur / 2), y + hauteur)
    ❻        self.finjeu = True
```

La fonction `__init__` ❶ attend des paramètres pour `self`, un objet `jeu`, un objet `image_photo`, les coordonnées `x` et `y`, ainsi que la `largeur` et la `hauteur` de l'image. La ligne ❷ appelle la fonction `__init__` de la classe parente, `Lutin`, comme dans les autres classes filles de celle-ci.

Le paramètre `image_photo` est mémorisé dans une variable d'objet de même nom ❸, comme nous l'avions fait dans `LutinPlateForme`. Nous créons une image affichée à l'aide de la fonction `create_image` et mémorisons le numéro d'identifiant renvoyé par cette fonction dans la variable d'objet `image` ❹.

Nous définissons les coordonnées du `LutinPorte` avec les paramètres `x` et `y` (qui deviennent les positions `x1` et `y1` de la porte), puis nous calculons les positions `x2` et `y2` ❺. Pour obtenir la coordonnée `x2`, nous ajoutons la moitié de la largeur de l'image au paramètre `x`. Ainsi, si `x` vaut `10` (la coordonnée `x1` aussi vaut `10`) et si la largeur de l'image vaut `40`, la coordonnée `x2` devient `30` (`10` plus la moitié de `40`).

Pourquoi un tel calcul nébuleux ? En fait, alors que nous voulons que M. Filiforme s'arrête de courir dès qu'il entre en collision avec *le bord* d'une plate-forme, il faut au contraire qu'il s'arrête de courir dès qu'il se situe *en face* de la porte. Ce ne serait pas de très bon goût si le personnage s'arrêtait alors qu'il est à côté de cette porte. Tu verras ceci en œuvre lorsque tu joueras une partie et que tu t'approcheras de la porte.

Si la position `x2` est un peu compliquée à calculer, la coordonnée `y2` est très simple à obtenir. Nous ajoutons juste la valeur de la variable `hauteur` au paramètre `y`, c'est tout.

Enfin, nous définissons la variable `finjeu` à `True` ❶. Ceci signifie que, quand le personnage atteint la porte, la partie est terminée.

## Déetecter la porte

Nous devons ensuite modifier la fonction `deplacer` de la classe `LutinPersonnage` qui détecte quand le personnage rencontre un lutin à gauche ou à droite. Voici une première modification :

---

```
if gauche and self.x < 0 and collision_gauche(co, co_lutin):
    self.x = 0
    gauche = False
    if lutin.finjeu:
        self.jeu.enfonction = False
```

---

Nous vérifions ici si le lutin que le personnage touche possède une variable `finjeu` mise à vrai. Si c'est le cas, nous réglons la variable `enfonction` à faux et tout s'arrête : nous avons atteint la fin de la partie.

Nous ajoutons les deux mêmes lignes au code qui vérifient s'il y a collision à droite :

---

```
if droite and self.x > 0 and collision_droite(co, co_lutin):
    self.x = 0
    droite = False
    if lutin.finjeu:
        self.jeu.enfonction = False
```

---

## Ajouter l'objet porte

Notre dernier ajout au programme du jeu est celui d'un objet pour la porte. Nous le plaçons avant l'appel à la boucle principale mais aussi juste avant la création de l'objet `personnage` :

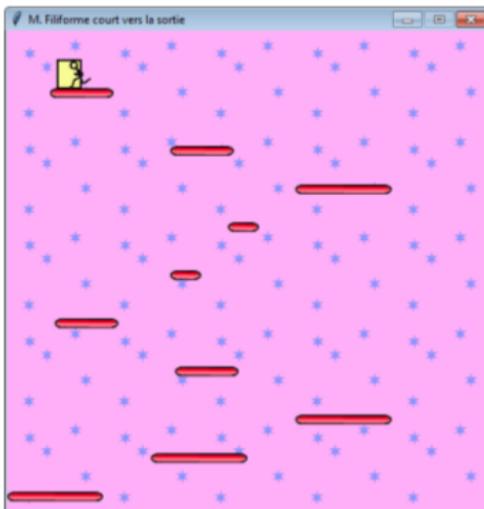
---

```
jeu.lutins.append(plateforme7)
jeu.lutins.append(plateforme8)
jeu.lutins.append(plateforme9)
jeu.lutins.append(plateforme10)
porte = LutinPorte(jeu, PhotoImage(file="porte1.gif"), 45, 30, 40, 35)
jeu.lutins.append(porte)
personnage = LutinPersonnage(jeu)
jeu.lutins.append(personnage)
jeu.boucle_principale()
```

---

Nous créons un objet **porte** à partir de l'objet **jeu**, suivi du **PhotoImage** (l'image de porte créée au chapitre 15). Nous réglons les paramètres **x** et **y** à 45 et 30 pour placer la porte sur une des plates-formes, près du sommet de l'écran, puis nous réglons la **largeur** et la **hauteur** à 40 et 35. Nous ajoutons ensuite l'objet **porte** à la liste, aux côtés des autres lutins du jeu.

L'image suivante montre le résultat quand M. Filiforme atteint la porte. Il cesse de courir lorsqu'il est devant la porte, et non à côté d'elle, comme ceci :



## Le jeu final

Le programme complet de notre jeu occupe un peu moins de 300 lignes de code et est proposé ici en détail. Si tu fais face à des soucis pour faire fonctionner ton jeu, compare chaque fonction et chaque classe à cette liste pour découvrir ce qui ne va pas comme prévu.

---

```
from tkinter import *
import random
import time

class Jeu:
    def __init__(self):
        self.tk = Tk()
```

```

        self.tk.title("M. Filiforme court vers la sortie")
        self.tk.resizable(0, 0)
        self.tk.wm_attributes("-topmost", 1)
        self.canvas = Canvas(self.tk, width=500, height=500, \
                           highlightthickness=0)
        self.canvas.pack()
        self.tk.update()
        self.hauteur_canevas = 500
        self.largeur_canevas = 500
        self.ap = PhotoImage(file="arriere-plan.gif")
        larg = self.ap.width()
        haut = self.ap.height()
        for x in range(0, 5):
            for y in range(0, 5):
                self.canvas.create_image(x * larg, y * haut, \
                                         image=self.ap, anchor='nw')
        self.lutins = []
        self.enfonction = True

    def boucle_principale(self):
        while 1:
            if self.enfonction == True:
                for lutin in self.lutins:
                    lutin.deplacer()
            self.tk.update_idletasks()
            self.tk.update()
            time.sleep(0.01)

    class Coords:
        def __init__(self, x1=0, y1=0, x2=0, y2=0):
            self.x1 = x1
            self.y1 = y1
            self.x2 = x2
            self.y2 = y2

    def dans_x(co1, co2):
        if (co1.x1 > co2.x1 and co1.x1 < co2.x2) \
           or (co1.x2 > co2.x1 and co1.x2 < co2.x2) \
           or (co2.x1 > co1.x1 and co2.x1 < co1.x2) \
           or (co2.x2 > co1.x1 and co2.x2 < co1.x1):
            return True
        else:
            return False

    def dans_y(co1, co2):
        if (co1.y1 > co2.y1 and co1.y1 < co2.y2) \
           or (co1.y2 > co2.y1 and co1.y2 < co2.y2) \
           or (co2.y1 > co1.y1 and co2.y1 < co1.y2) \
           or (co2.y2 > co1.y1 and co2.y2 < co1.y1):
            return True
        else:
            return False

```

```

        return True
    else:
        return False

def collision_gauche(co1, co2):
    if dans_y(co1, co2):
        if co1.x1 <= co2.x2 and co1.x1 >= co2.x1:
            return True
    return False

def collision_droite(co1, co2):
    if dans_y(co1, co2):
        if co1.x2 >= co2.x1 and co1.x2 <= co2.x2:
            return True
    return False

def collision_haut(co1, co2):
    if dans_x(co1, co2):
        if co1.y1 <= co2.y2 and co1.y1 >= co2.y1:
            return True
    return False

def collision_bas(y, co1, co2):
    if dans_x(co1, co2):
        y_calc = co1.y2 + y
        if y_calc >= co2.y1 and y_calc <= co2.y2:
            return True
    return False

class Lutin:
    def __init__(self, jeu):
        self.jeu = jeu
        self.finjeu = False
        self.coordonnees = None
    def deplacer(self):
        pass
    def coords(self):
        return self.coordonnees

class LutinPlateForme(Lutin):
    def __init__(self, jeu, image_photo, x, y, largeur, hauteur):
        Lutin.__init__(self, jeu)
        self.image_photo = image_photo
        self.image = jeu.canvas.create_image(x, y, \
                                             image=self.image_photo, anchor='nw')
        self.coordonnees = Coords(x, y, x + largeur, y + hauteur)

class LutinPersonnage(Lutin):
    def __init__(self, jeu):

```

```

Lutin.__init__(self, jeu)
self.images_gauche = [
    PhotoImage(file="fil-G1.gif"),
    PhotoImage(file="fil-G2.gif"),
    PhotoImage(file="fil-G3.gif")
]
self.images_droite = [
    PhotoImage(file="fil-D1.gif"),
    PhotoImage(file="fil-D2.gif"),
    PhotoImage(file="fil-D3.gif")
]
self.image = jeu.canvas.create_image(200, 470, \
        image=self.images_gauche[0], anchor='nw')
self.x = -2
self.y = 0
self.image_courante = 0
self.ajout_image_courante = 1
self.compte_sauts = 0
self.derniere_heure = time.time()
self.coordonnees = Coords()
jeu.canvas.bind_all('<KeyPress-Left>', self.tourner_a_gauche)
jeu.canvas.bind_all('<KeyPress-Right>', self.tourner_a_droite)
jeu.canvas.bind_all('<space>', self.sauter)

def tourner_a_gauche(self, evt):
    if self.y == 0:
        self.x = -2

def tourner_a_droite(self, evt):
    if self.y == 0:
        self.x = 2

def sauter(self, evt):
    if self.y == 0:
        self.y = -4
    self.compte_sauts = 0

def animer(self):
    if self.x != 0 and self.y == 0:
        if time.time() - self.derniere_heure > 0.1:
            self.derniere_heure = time.time()
            self.image_courante += self.ajout_image_courante
            if self.image_courante >= 2:
                self.ajout_image_courante = -1
            if self.image_courante <= 0:
                self.ajout_image_courante = 1
    if self.x < 0:
        if self.y != 0:
            self.jeu.canvas.itemconfig(self.image, \

```

```

                image=self.images_gauche[2])
        else:
            self.jeu.canvas.itemconfig(self.image, \
                image=self.images_gauche[self.image_courante])
    elif self.x > 0:
        if self.y != 0:
            self.jeu.canvas.itemconfig(self.image, \
                image=self.images_droite[2])
        else:
            self.jeu.canvas.itemconfig(self.image, \
                image=self.images_droite[self.image_courante])

    def coords(self):
        xy = self.jeu.canvas.coords(self.image)
        self.coordonnees.x1 = xy[0]
        self.coordonnees.y1 = xy[1]
        self.coordonnees.x2 = xy[0] + 27
        self.coordonnees.y2 = xy[1] + 30
        return self.coordonnees

    def deplacer(self):
        self.animer()
        if self.y < 0:
            self.compte_sauts += 1
            if self.compte_sauts > 20:
                self.y = 4
        if self.y > 0:
            self.compte_sauts -= 1
        co = self.coords()
        gauche = True
        droite = True
        haut = True
        bas = True
        tombe = True
        if self.y > 0 and co.y2 >= self.jeu.hauteur_canevas:
            self.y = 0
            bas = False
        elif self.y < 0 and co.y1 <= 0:
            self.y = 0
            haut = False
        if self.x > 0 and co.x2 >= self.jeu.largeur_canevas:
            self.x = 0
            droite = False
        elif self.x < 0 and co.x1 <= 0:
            self.x = 0
            gauche = False
        for lutin in self.jeu.lutins:
            if lutin == self:
                continue

```

```

        co_lutin = lutin.coords()
        if haut and self.y < 0 and collision_haut(co, co_lutin):
            self.y = -self.y
            haut = False
        if bas and self.y > 0 \
                and collision_bas(self.y, co, co_lutin):
            self.y = co_lutin.y1 - co.y2
            if self.y < 0:
                self.y = 0
            bas = False
            haut = False
        if bas and tombe and self.y == 0 \
                and co.y2 < self.jeu.hauteur_canevas \
                and collision_bas(1, co, co_lutin):
            tombe = False
        if gauche and self.x < 0 and collision_gauche(co, co_lutin):
            self.x = 0
            gauche = False
            if lutin.finjeu:
                self.jeu.enfonction = False
        if droite and self.x > 0 and collision_droite(co, co_lutin):
            self.x = 0
            droite = False
            if lutin.finjeu:
                self.jeu.enfonction = False
        if tombe and bas and self.y == 0 \
                and co.y2 < self.jeu.hauteur_canevas:
            self.y = 4
        self.jeu.canvas.move(self.image, self.x, self.y)

class LutinPorte(Lutin):
    def __init__(self, jeu, image_photo, x, y, largeur, hauteur):
        Lutin.__init__(self, jeu)
        self.image_photo = image_photo
        self.image = jeu.canvas.create_image(x, y, \
                                             image=self.image_photo, anchor='nw')
        self.coordonnees = Coords(x, y, x + (largeur / 2), y + hauteur)
        self.finjeu = True

    jeu = Jeu()
    plateforme1 = LutinPlateForme(jeu, PhotoImage(file="plate-forme1.gif"), \
        0, 480, 100, 10)
    plateforme2 = LutinPlateForme(jeu, PhotoImage(file="plate-forme1.gif"), \
        150, 440, 100, 10)
    plateforme3 = LutinPlateForme(jeu, PhotoImage(file="plate-forme1.gif"), \
        300, 400, 100, 10)
    plateforme4 = LutinPlateForme(jeu, PhotoImage(file="plate-forme1.gif"), \
        300, 160, 100, 10)
    plateforme5 = LutinPlateForme(jeu, PhotoImage(file="plate-forme2.gif"), \

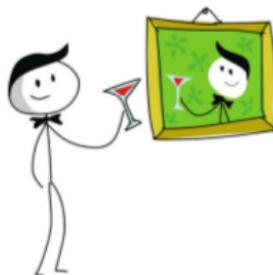
```

```
    175, 350, 66, 10)
plateforme6 = LutinPlateForme(jeu, PhotoImage(file="plate-forme2.gif"), \
    50, 300, 66, 10)
plateforme7 = LutinPlateForme(jeu, PhotoImage(file="plate-forme2.gif"), \
    170, 120, 66, 10)
plateforme8 = LutinPlateForme(jeu, PhotoImage(file="plate-forme2.gif"), \
    45, 60, 66, 10)
plateforme9 = LutinPlateForme(jeu, PhotoImage(file="plate-forme3.gif"), \
    170, 250, 32, 10)
plateforme10 = LutinPlateForme(jeu, PhotoImage(file="plate-forme3.gif"), \
    230, 200, 32, 10)
jeu.lutins.append(plateforme1)
jeu.lutins.append(plateforme2)
jeu.lutins.append(plateforme3)
jeu.lutins.append(plateforme4)
jeu.lutins.append(plateforme5)
jeu.lutins.append(plateforme6)
jeu.lutins.append(plateforme7)
jeu.lutins.append(plateforme8)
jeu.lutins.append(plateforme9)
jeu.lutins.append(plateforme10)
porte = LutinPorte(jeu, PhotoImage(file="porte1.gif"), 45, 30, 40, 35)
jeu.lutins.append(porte)
personnage = LutinPersonnage(jeu)
jeu.lutins.append(personnage)
jeu.boucle_principale()
```

---

## Ce que tu as appris

Dans ce chapitre, nous avons terminé notre jeu, intitulé M. Filiforme court vers la sortie. Nous avons créé une classe pour le personnage animé, rédigé des fonctions pour le déplacer dans tout l'écran et l'animer à mesure qu'il se déplace (le changement successif des images donne une impression de personnage en pleine course). Nous avons exploité la détection de collision de base pour indiquer quand il touche les côtés gauche et droit du canevas, ou quand il touche un autre lutin, comme une plate-forme ou la porte. Nous avons aussi ajouté du code de détection de collision avec le haut ou le bas de l'écran et un autre code pour garantir que, quand le personnage dépasse le bord d'une plate-forme, il tombe correctement. Nous



avons ajouté aussi du code pour dire que, quand M. Filiforme atteint la porte, la partie se termine.

## Puzzles de programmation

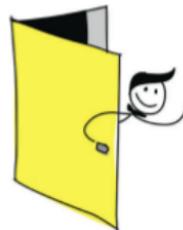
Il est possible d'apporter toutes sortes d'améliorations au jeu. Pour l'instant, il est assez simple. Donc, pour lui donner un aspect plus professionnel et ajouter un peu de plaisir à le voir fonctionner, tu peux ajouter du code. Essaie de développer les fonctionnalités proposées ci-après, puis compare ton code à celui du site d'accompagnement du livre.

### 1. Tu as gagné !

De même que dans le jeu Rebondir ! tu as pu ajouter le texte « Partie terminée » au chapitre 14, ajoute le texte « Tu as gagné ! » quand le personnage atteint la porte, pour que le joueur sache qu'il a gagné la partie.

### 2. Animer la porte

Au chapitre 15, nous avons créé deux images pour la porte : l'une ouverte et l'autre fermée. Quand M. Filiforme atteint la porte, ce serait bien de modifier l'image de celle-ci pour qu'elle apparaisse ouverte, puis faire disparaître le personnage et refermer la porte (revenir à la première image). Ceci donne l'illusion que M. Filiforme sort et referme la porte derrière lui. Pour cela, modifie les classes `LutinPorte` et `LutinPersonnage`.



### 3. Plates-formes mobiles

Encore plus fort : essaie de créer une nouvelle classe nommée `LutinPlateFormeMobile`. Ce genre de plate-forme doit pouvoir bouger horizontalement, d'un côté à l'autre, pour rendre le jeu plus difficile et compliquer la tâche de M. Filiforme de rejoindre la sortie en haut de l'écran. Bien sûr, M. Filiforme doit suivre le mouvement d'une plate-forme quand il est dessus.



## CONCLUSION

# ET À PARTIR DE LÀ ?

Tout au long de ce livre, tu as appris quelques concepts de base de la programmation, grâce à ce voyage au cœur de Python. À partir de cela, tu verras qu'il te sera beaucoup plus aisé de travailler avec d'autres langages de programmation. Si Python est impressionnant de puissance et d'universalité, un seul langage ne suffit pas pour toutes les situations, donc n'hésite pas à découvrir d'autres manières de programmer, ainsi que d'autres langages. Ici, nous allons examiner quelques alternatives existantes pour la programmation de jeux et de graphismes, puis pousser ensuite une petite pointe vers les langages de programmation les plus habituellement utilisés.

## Programmation graphique et de jeux

Si tu veux poursuivre ta découverte de la programmation graphique et des jeux, tu trouveras de nombreuses possibilités, dont voici un petit aperçu.

- BlitzBasic (<http://www.blitzbasic.com/>) utilise une version particulière du langage de programmation BASIC, destinée spécialement aux jeux.
- Adobe Flash est un logiciel spécialisé en animation, conçu pour fonctionner dans un navigateur web. Il possède son propre langage de programmation, appelé ActionScript (<http://www.adobe.com/devnet/actionscript.html>).
- Alice (<http://www.alice.org/>) est un environnement de programmation en trois dimensions, pour Microsoft Windows et Mac OS X.
- Scratch (<http://scratch.mit.edu/>) est un outil libre, conçu par le célèbre MIT, pour développer des jeux à partir de blocs de construction. Il est en français et très facile à utiliser.
- Unity3D (<http://unity3d.com/>) est un autre outil de création de jeux.

Une recherche en ligne te fera découvrir une pléthore de ressources pour t'aider à aborder chacune de ces options.

En outre, si tu souhaites poursuivre ton étude de Python, tu peux aussi utiliser PyGame, le module de Python conçu pour le développement de jeux. Voyons de quoi il s'agit.

### PyGame

La version nommée *Python Reloaded* (`pgreloaded` ou `pygame2`) est conçue pour fonctionner avec Python 3 (les précédentes versions ne fonctionnaient qu'avec Python 2). Des tutoriels sont disponibles en anglais sur <http://code.google.com/p/pgreloaded/> mais un cours OpenClassrooms est également disponible en français sur <http://fr.openclassrooms.com/informatique/cours/interface-graphique-pygame-pour-python>.

#### NOTE

*Au moment de l'écriture de ces lignes, il n'existe pas d'installateur automatique de pgreloaded pour OS X ni Linux, donc il n'y a aucune manière immédiate de l'utiliser sur ces systèmes d'exploitation.*

L'écriture d'un jeu sous PyGame est un peu plus compliquée que sous `tkinter`. Au chapitre 12, par exemple, nous avons affiché une image sous `tkinter` à l'aide de ce code :

---

```
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
monimage = PhotoImage(file='c:\\test.gif')
canvas.create_image(0, 0, anchor=NW, image=monimage)
```

---

Le même programme à base de PyGame (avec chargement d'une image .**bmp** au lieu d'un fichier .**gif**) devient :

---

```
import sys
import time
import pygame2
import pygame2.sdl.constants as constants
import pygame2.sdl.image as image
import pygame2.sdl.video as video
❶ video.init()
❷ img = image.load_bmp("c:\\\\test.bmp")
❸ ecran = video.set_mode(img.width, img.height)
❹ ecran.fill(pygame2.Color(255, 255, 255))
❺ ecran.blit(img, (0, 0))
❻ ecran.flip()
❼ time.sleep(10)
❽ video.quit()
```

---

Après l'importation des modules de **pygame2**, nous appelons la fonction **init** du module **video** de PyGame ❶, ce qui équivaut à peu près à la création du canevas et du **pack** de l'exemple sous **tkinter**. Ensuite, nous chargeons une image BMP ❷ à l'aide de la fonction **load\_bmp**, puis nous créons un objet **ecran** (*screen*) ❸, à l'aide de la fonction **set\_mode**, en lui passant en paramètres la largeur et la hauteur de l'image chargée. La ligne ❹, facultative, vide l'écran et le remplit de blanc. La fonction **blit** de l'objet écran est ensuite appelée ❺ pour y afficher l'image. Les paramètres de cette fonction sont l'objet **img** et un tuple contenant l'emplacement où nous voulons afficher l'image (0 pixel en horizontal et 0 pixel depuis le haut).

PyGame utilise un tampon hors écran (ou *buffer*, que l'on appelle aussi *double buffer*). Le tampon hors écran est une technique utilisée pour dessiner des graphismes dans une zone de la mémoire de l'ordinateur complètement masquée, pour la copier ensuite vers la partie visible de l'écran, le tout en une seule fois. Le tampon hors-écran réduit l'effet de scintillement de l'écran lorsqu'il s'agit de dessiner un grand nombre d'objets à l'affichage. La copie du tampon hors écran vers l'affichage visible est assurée par la fonction **flip** ❻.

Enfin, nous nous mettons en veille pendant 10 secondes ⑦, parce que sans cela, au contraire de ce qui se passe dans le canevas de `tkinter`, l'écran se ferme immédiatement si nous ne l'en empêchons pas. Ensuite, nous nettoyons le tout à l'aide de `video.init` ⑧, pour que PyGame se ferme proprement. Il y a beaucoup à dire à propos de PyGame, mais cet exemple te donne un aperçu de son mode de fonctionnement.

## Langages de programmation

Si tu es intéressé par d'autres langages de programmation, les plus appréciés actuellement sont notamment Java, C/C++, C#, PHP, Objective-C, Perl, Ruby et JavaScript. Nous allons les présenter succinctement et voir à quoi ressemble le traditionnel programme `Bonjour tout le monde` dans chacun de ces langages. Rappelle-toi que nous avons sacrifié à la tradition en réalisant nos premières lignes de code en Python au chapitre 1. Note qu'aucun de ces langages n'est spécialement conçu à l'intention des débutants en programmation et que la plupart sont très différents de Python.

### Java

Java (<http://www.oracle.com/technetwork/java/index.html>) est un langage de complexité modérée, avec une vaste bibliothèque de modules (appelés paquetages ou *packages*). Le Web offre de nombreux cours et documentations à son propos. Java fonctionne sur la plupart des systèmes d'exploitation et c'est aussi le langage utilisé sur les téléphones mobiles Android.

Voici l'exemple du `Bonjour tout le monde` en Java :

---

```
public class Bonjour {  
    public static final void main(String[] args) {  
        System.out.println("Bonjour tout le monde");  
    }  
}
```

---

### C/C++

C (<http://www.cprogramming.com/>) et C++ (<http://www.stroustrup/C++.html>) sont deux langages de programmation très complexes, utilisables sur tous les systèmes d'exploitation. Ils sont même à la base de l'écriture de certains systèmes. Des versions libres (gratuites) et commerciales (payantes) sont disponibles un peu partout. Ces deux

langages (peut-être plus C++ que C) affichent ce que l'on appelle une courbe d'apprentissage ardue. Ainsi, certaines fonctionnalités doivent être écrites en totalité sous ces langages, alors que Python les propose d'emblée, pour un usage immédiat et sans effort. C'est le cas, par exemple, si tu veux indiquer à l'ordinateur que tu veux utiliser une partie de sa mémoire pour stocker un objet. De nombreux jeux commerciaux et consoles de jeu sont programmés sous une forme plus ou moins proche du C ou du C++. Voici un exemple du [Bonjour tout le monde](#) en C :

---

```
#include <stdio.h>
int main ()
{
    printf ("Bonjour tout le monde\n");
}
```

---

L'exemple en C++ ressemblerait plutôt à ceci :

---

```
#include <iostream>
int main()
{
    std::cout << "Bonjour tout le monde\n";
    return 0;
}
```

---

## C#

Prononcé « C sharp », C# (<http://msdn.microsoft.com/fr-fr/vstudio/hh388566/>) est un langage de programmation de complexité moyenne, conçu pour Windows et relativement proche de Java. Il est un peu plus aisés à utiliser que C et C++. Un projet d'environnement de développement pour utiliser ce langage existe aussi sous Windows, OS X et Linux, appelé Mono (<http://monodevelop.com/>).

Voici notre exemple en C# :

---

```
public class Hello
{
    public static void Main()
    {
        System.Console.WriteLine("Bonjour tout le monde");
    }
}
```

---

## PHP

PHP (<http://www.php.net/>) est un langage de programmation utilisé essentiellement pour concevoir des sites web. Il faut donc un serveur web (un logiciel qui permet de fournir des pages web à un navigateur) avec PHP installé, mais tout le reste des logiciels requis est librement accessible pour la plupart des systèmes d'exploitation. Pour pouvoir travailler en PHP, tu dois apprendre le HTML (un langage simple de création de pages web). Sur Internet existent de nombreux tutoriels, notamment <http://www.lephpfacile.com/cours/> et <http://fr.openclassrooms.com/informatique/php/cours>, qui t'enseigneront tous les détails de ces langages.

Une page HTML qui affiche **Bonjour tout le monde** ressemble à ceci :

---

```
<html>
  <body>
    <p>Bonjour tout le monde</p>
  </body>
</html>
```

---

Une page PHP contiendra le code suivant pour faire la même chose :

---

```
<?php
echo "Bonjour tout le monde\n";
?>
```

---

## Objective-C

Objective-C (<http://fr.openclassrooms.com/informatique/cours/programmez-en-objective-c> ou <http://fr.tuto.com/objective-c/>) ressemble fort au C, dont c'est d'ailleurs une extension. Il est surtout utilisé sur les ordinateurs Apple. C'est le langage de programmation de l'iPhone et de l'iPad.

Voici un exemple de **Bonjour tout le monde** en Objective-C :

---

```
#import <Foundation/Foundation.h>
int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    NSLog (@"Bonjour tout le monde");
    [pool drain];
    return 0;
}
```

---

## Perl

Le langage de programmation Perl (<http://www.perl.org/>) est disponible gratuitement sur les principaux systèmes d'exploitation. Il sert généralement à développer des sites web (comme PHP).

Notre exemple, en Perl cette fois :

---

```
print("Bonjour tout le monde\n");
```

---

## Ruby

Ruby (<http://www.ruby-lang.org/>) est un langage de développement disponible gratuitement pour tous les principaux systèmes d'exploitation. Il sert majoritairement à créer des sites web, en particulier dans le cadre de l'environnement de développement Ruby on Rails. Un environnement de développement (ou *framework*) est un ensemble de bibliothèques qui soutiennent le développement de types bien déterminés d'applications.

Voici notre exemple en Ruby :

---

```
puts "Bonjour tout le monde"
```

---

## JavaScript

JavaScript (<https://developer.mozilla.org/fr/docs/Web/JavaScript>) est un langage de programmation généralement utilisé au sein de pages web, mais il sert de plus en plus à développer des jeux. Sa syntaxe ressemble fondamentalement à celle de Java, mais il est probablement un peu plus facile de débuter en JavaScript. Il est possible de créer une simple page HTML qui contient un programme en JavaScript et de l'exécuter dans un navigateur, sans devoir passer par un environnement d'exécution (*shell*), une ligne de commande ou quoi que ce soit d'autre. De nombreux tutoriels sont disponibles en ligne, notamment sur <http://fr.openclassrooms.com/informatique/cours/tout-sur-le-javascript>.

L'exemple **Bonjour tout le monde** en JavaScript diffère selon qu'il fonctionne dans un navigateur ou dans un shell. Dans le shell, l'exemple ressemble à ceci :

---

```
print('Bonjour tout le monde');
```

---

Dans un navigateur, il devient ceci :

```
<html>
  <body>
    <script type="text/javascript">
      alert("Bonjour tout le monde");
    </script>
  </body>
</html>
```

## Et pour finir

Que tu t'accroches à Python ou que tu décides de t'essayer à un autre langage de programmation (et il y en a tellement plus que ceux que nous avons évoqués ici), tu verras que les concepts que tu as découverts dans ce livre sont très utiles. Même si tu ne poursuis pas dans la voie de la programmation, le fait de comprendre déjà certaines de ses idées fondamentales t'aidera dans toutes sortes d'activités, que ce soit à l'école ou plus tard, dans ta vie professionnelle.

Bonne chance et, surtout, amuse-toi bien dans l'univers de la programmation !





## ANNEXE

# MOTS-CLÉS DE PYTHON

Comme tous les langages de programmation, Python utilise des mots-clés, des mots qui possèdent une signification particulière. Ils font partie du langage lui-même et ne peuvent donc être utilisés pour quoi que ce soit d'autre. Ainsi, si tu essaies d'utiliser un mot-clé comme nom de variable ou si tu l'utilises d'une mauvaise manière, tu obtiens des messages d'erreurs, parfois rigolos, parfois très bizarres, de la part de la console de Python.

Cette annexe décrit chacun des mots-clés de Python. Utilise-la comme une référence, lorsque tu rédiges tes programmes.

## AND

Le mot-clé `and` (et) sert à joindre deux expressions dans une instruction (`if`, par exemple), pour indiquer que les deux expressions doivent être vraies ensemble. Voici un exemple :

---

```
if age > 10 and age < 20:  
    print('Attention, voilà les ados !!!!')
```

---

Ce code signifie que la valeur de la variable `age` doit être plus grande que `10` et plus petite que `20` pour que le message s'affiche.

## AS

Le mot-clé `as` (comme) sert à renommer un module importé. Par exemple, si tu as un module avec un très long nom, comme `je_suis_un_module_pas_tres_utile`, ce serait assez embêtant de devoir rappeler ce nom kilométrique, chaque fois que tu en appelles une fonction ou une variable :

---

```
import je_suis_un_module_pas_tres_utile  
je_suis_un_module_pas_tres_utile.faire_ceci()  
J'ai fait ceci, qui n'est pas très utile.  
je_suis_un_module_pas_tres_utile.faire_cela()  
J'ai fait aussi cela, qui n'est pas très utile non plus !!
```

---

Pour alléger ce code, tu peux donner un nom plus concis au module au moment où tu l'importe, puis en utiliser les fonctions avec ce nouveau nom. C'est une sorte de surnom, en fait :

---

```
import je_suis_un_module_pas_tres_utile as pasutile  
pasutile.faire_ceci()  
J'ai fait ceci, qui n'est pas très utile.  
pasutile.faire_cela()  
J'ai fait aussi cela, qui n'est pas très utile non plus !!
```

---

## ASSERT

Le mot-clé `assert` sert à indiquer (littéralement à affirmer) qu'une portion de code doit être vraie. C'est une façon de capturer les erreurs et les problèmes dans un programme, souvent utilisée dans des programmes évolués (ce qui explique que nous ne l'avons pas utilisé dans le livre). Voici un exemple simple d'instruction `assert` :

---

```
>>> monnombre = 10  
>>> assert monnombre < 5
```

---

```
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    assert a < 5
AssertionError
```

---

Cet exemple « affirme » que la valeur de la variable `monnombre` doit être plus petite que `5`. Si elle ne l'est pas, Python affiche une erreur, appelée `AssertionError`.

## BREAK

Le mot-clé `break` arrête l'exécution d'une portion de code. Il intervient souvent dans des boucles `for` et `while`, pour sortir de ces boucles :

```
age = 10
for x in range(1, 100):
    print('Comptage %' % x)
    if x == age:
        print('Fin du comptage')
        break
```

---

Comme la variable `age` est définie avec la valeur `10`, le programme affiche ceci :

```
Comptage 1
Comptage 2
Comptage 3
Comptage 4
Comptage 5
Comptage 6
Comptage 7
Comptage 8
Comptage 9
Comptage 10
Fin du comptage
```

---

Dès que la valeur de la variable `x` atteint `10`, le code affiche le texte `Fin du comptage` et interrompt la boucle.

## CLASS

Le mot-clé `class` sert à définir un *type* d'objet, comme un véhicule, une personne ou un animal. Les classes peuvent posséder une fonction nommée `_init_`, qui sert à réaliser toutes les tâches nécessaires lors de la création d'objets de cette classe. Si nous prenons l'exemple d'un objet de la classe `Voiture`, cette fonction peut par exemple préci-

ser la `couleur`, lors de la création d'un objet `voiture`, membre de cette classe :

---

```
class Voiture:  
    def __init__(self, couleur):  
        self.couleur = couleur  
  
voiture1 = Voiture('rouge')  
voiture2 = Voiture('bleue')  
print(voiture1.couleur)  
rouge  
print(voiture2.couleur)  
bleue
```

---

## CONTINUE

Nous avons vu plus haut le mot-clé `break`, qui interrompt généralement l'exécution d'une boucle. Le mot-clé `continue` sert à « sauter » à l'élément suivant qui fait l'objet d'une boucle, de sorte que le programme s'interrompt au `continue`, ignore le reste du bloc de code et passe directement à l'occurrence (ou l'élément) suivante de la boucle. Au contraire de `break`, qui quitte toute la boucle, `continue` reste *dans* la boucle pour traiter l'élément suivant. Admettons que nous ayons une liste d'éléments et que nous souhaitions ignorer ceux dont le nom commence par `b`. Nous pouvons écrire un code du genre de ceci :

---

```
❶ >>> maliste = ['pomme', 'poire', 'banane', 'blaireau', 'orange', \  
                 'chameau']  
❷ >>> for element in maliste:  
❸         if element.startswith('b'):  
❹             continue  
❺         print(element)  
pomme  
poire  
orange  
chameau
```

---

La ligne ❶ crée une liste d'éléments. La boucle `for` ❷ parcourt un à un tous les éléments de cette liste, pour exécuter le bloc de code qui suit. Si le mot commence (`startswith`) par la lettre `b` ❸, la ligne ❹ s'exécute : le mot-clé `continue` ignore le reste du bloc et passe à l'élément suivant de la boucle, donc en ligne ❺. Dans le cas contraire, la ligne ❺ est exécutée et affiche l'élément.

## DEF

Le mot-clé `def` sert à définir une fonction. Par exemple, pour créer une fonction qui convertit un nombre d'années en le nombre équivalent de minutes, nous pouvons écrire :

---

```
>>> def minutes(annees):
    return annees * 365 * 24 * 60
>>> minutes(10)
5256000
```

---

## DEL

Le mot-clé `del` sert à supprimer quelque chose. Par exemple, si tu as une liste de cadeaux que tu aimerais recevoir pour ton anniversaire, mais si tu changes d'avis pour en supprimer un, tu peux barrer ce cadeau de ta liste et ajouter autre chose :

---

voiture télécommandée  
nouveau vélo  
jeu pour PC  
robot dinosaure

---

En Python, la liste initiale serait :

---

```
ce_que_je_veux = ['voiture télécommandée', 'nouveau vélo', 'jeu PC']
```

---

Pour supprimer le jeu pour ordinateur de la liste, utilise `del` et l'indice de l'élément. Tu peux ensuite ajouter le nouvel élément à l'aide de la fonction `append` des listes :

---

```
del ce_que_je_veux[2]
ce_que_je_veux.append('robot dinosaure')
```

---

Il te reste à afficher la liste :

---

```
print(ce_que_je_veux)
['voiture télécommandée', 'nouveau vélo', 'robot dinosaure']
```

---

## ELIF

Le mot-clé `elif` (sinon, si) ne s'utilise jamais seul mais toujours associé à une instruction `if`, dont il fait partie. Voir le mot-clé `if` pour un exemple.

## ELSE

Le mot-clé `else` (sinon) ne s'utilise jamais seul mais toujours associé à une instruction `if`, dont il fait partie. Vois le mot-clé `if` pour un exemple.

## EXCEPT

Le mot-clé `except` (sauf ou excepté) s'associe au mot-clé `try` décrit plus loin et sert à intercepter, capturer des problèmes dans le code. Il intervient généralement dans des programmes plutôt complexes, donc nous ne l'avons pas utilisé dans ce livre.

## FINALLY

Le mot-clé `finally` (finalement) s'associe aussi au mot-clé `try` et permet, si une erreur se produit, de garantir qu'une certaine portion de code s'exécute, généralement en solution à une situation inextricable et insoluble, qu'une autre portion du code a générée. Ce mot-clé n'intervient pas dans ce livre, parce qu'il concerne une programmation beaucoup plus évoluée et complexe.

## FOR

La boucle que permet de créer le mot-clé `for` (pour chaque) s'exécute un nombre déterminé et prévisible de fois. Voici un exemple :

---

```
for x in range(0, 5):
    print('x vaut %s' % x)
```

---

La boucle `for` de l'exemple exécute le bloc de code (l'instruction `print`) cinq fois et affiche les résultats suivants :

---

```
x vaut 0
x vaut 1
x vaut 2
x vaut 3
x vaut 4
```

---

## FROM

Lors de l'importation d'un module, il est possible de ne charger que la partie indispensable, à l'aide du mot-clé `from` (en provenance de). Ainsi, si le module `turtle` présenté au chapitre 4 possède une classe nommée

`Pen`, qui permet de créer un objet (le canevas sur lequel la tortue évolue), l'importation complète du module `turtle` s'écrit comme ceci :

---

```
import turtle
t = turtle.Pen()
```

---

En revanche, si nous ne voulons importer que la classe `Pen` en elle-même, pour l'utiliser directement (sans aucune référence au module `turtle` ensuite), nous écrirons plutôt :

---

```
from turtle import Pen
t = Pen()
```

---

Cette précision permet, lorsque tu relis le programme après quelque temps, de savoir quels sont les modules, les fonctions et les classes que tu utilises, en examinant simplement le début du code. Ceci s'avère particulièrement intéressant dans de très longs programmes qui font appel à de nombreux modules. Cependant, si tu t'engages dans ce choix, tu ne pourras pas utiliser les portions du module que tu n'auras pas importées. Ainsi, comme le module `time` contient des fonctions nommées `localtime` et `gmtime`, si tu n'importe que `localtime` et essaies d'utiliser `gmtime`, tu obtiens une erreur :

---

```
>>> from time import localtime
>>> print(localtime())
(2007, 1, 30, 20, 53, 42, 1, 30, 0)
>>> print(gmtime())
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'gmtime' is not defined
```

---

Le message d'erreur `name 'gmtime' is not defined` signifie que Python ne connaît rien de la fonction `gmtime`, parce qu'elle n'a pas été importée.

S'il faut faire référence à plusieurs fonctions d'un module déterminé et ne pas les appeler en citant le nom du module (par exemple `time.localtime` ou `time.gmtime`), il est possible d'importer tout ce que contient le module en indiquant un astérisque (\*), comme suit :

---

```
>>> from time import *
>>> print(localtime())
time.struct_time(tm_year=2014, tm_mon=10, tm_mday=9, tm_hour=11, tm_min=10, tm_sec=13, tm_wday=3, tm_yday=282, tm_isdst=1)
>>> print(list(gmtime()))
[2014, 10, 9, 9, 11, 15, 3, 282, 0]
```

---

Cette forme d'importation récupère tout le contenu du module `time` et il est donc possible ensuite d'appeler n'importe laquelle de ses fonctions.

## GLOBAL

Le chapitre 7 présente l'idée de la portée dans les programmes. La portée fait référence à la visibilité d'une variable. Si une variable est définie en dehors d'une fonction, elle est généralement visible dans cette fonction. En revanche, une variable définie dans une fonction n'est visible que dans celle-ci, c'est-à-dire qu'elle est invisible en dehors du corps de la fonction, sauf si elle est déclarée comme globale, avec le mot-clé `global`. La variable devient alors visible de partout. Voici un exemple :

---

```
>>> def test():
    global a
    a = 1
    b = 2
```

---

Dans ce cas, à ton avis, que va-t-il se produire si nous essayons d'afficher la valeur de `a`, puis celle de `b` en dehors de la fonction ? En fait, la première sera visible, mais la seconde provoquera un message d'erreur. Pour t'en convaincre, essaie ceci :

---

```
>>> test()
>>> print(a)
1
>>> print(b)
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    print(b)
NameError: name 'b' is not defined
```

---

Comme la variable `a` a été déclarée comme globale au sein de la fonction, elle est visible en dehors de celle-ci. La variable `b` n'a pas été déclarée comme globale, donc elle n'est visible que dans le corps de la fonction. Note que l'utilisation de `global` sur une variable doit se faire à part et *avant* de donner une valeur à la variable. Si tu écris par exemple `global a = 1`, tu provoques une erreur de syntaxe.

## IF

Le mot-clé `if` (si) sert à prendre une décision en fonction d'une condition ou de plusieurs. Il peut s'utiliser avec les mots-clés `else` (sinon)

et **elif** (sinon, si). L'instruction **if** revient à dire « si une chose est vraie, alors faire telle action ». Voici un exemple :

---

```
❶ if prix_jouet > 1000:  
❷     print("Ce jouet est beaucoup trop cher")  
❸ elif prix_jouet > 100:  
❹     print("Ce jouet est cher")  
❺ else:  
❻     print("Je peux m'offrir ce jouet")
```

---

Cette instruction **if** dit que, *si* le prix d'un jouet vaut plus de **1000** ❶, il faut afficher le message qu'il est beaucoup trop cher ❷ ; *sinon*, *si* son prix est plus grand que **100** ❸, alors il faut afficher le message qu'il est trop cher ❹ ; *sinon* ❺, il faut afficher le message que je peux me l'offrir ❻.

## IMPORT

Le mot-clé **import** indique à Python qu'il doit charger un module pour l'utiliser ensuite. Le code suivant, par exemple, dit à Python de charger le module **sys** :

---

```
import sys
```

---

## IN

Le mot-clé **in** (dans) ne s'utilise jamais seul mais dans des expressions, par exemple dans une instruction **if** pour voir si un élément est présent dans une collection (ou suite) d'éléments, ou dans une boucle **for** pour parcourir un à un tous les éléments d'une liste. Par exemple, le nombre 1 se trouve-t-il dans une liste (ou collection) donnée de nombres ?

---

```
>>> if 1 in [1,2,3,4]:  
>>>     print('Le nombre est dans la liste')  
Le nombre est dans la liste
```

---

Voici comment vérifier si la chaîne '**pantalon**' est présente dans une liste de vêtements :

---

```
>>> liste_vetements = ['short', 'slip', 'boxer', 'bermuda', 'pantacourt']  
>>> if 'pantalon' in liste_vetements:  
        print('pantalon fait partie de la liste')  
    else:  
        print('pantalon ne fait pas partie de la liste')  
pantalon ne fait pas partie de la liste
```

---

## IS

Le mot-clé **is** (est un ou est une) ressemble un peu à l'opérateur de test d'égalité (**==**), qui permet de dire si deux choses sont égales (par exemple **10==10** est vrai mais **10==11** est faux). Une différence fondamentale existe toutefois entre les deux : **==** peut renvoyer vrai et **is** renvoyer faux pour les deux mêmes choses, même si tu penses que ces deux choses sont identiques. Il s'agit d'un concept de programmation avancée. Nous n'avons utilisé que **==** dans ce livre.

## LAMBDA

Le mot-clé **lambda** sert à créer des fonctions anonymes (sans nom) ou « en ligne ». Ce mot-clé est utilisé dans des programmes plus complexes. Nous n'en avons pas parlé dans le livre.

## NOT

Si une expression est vraie, **not** (pas) renvoie faux. Par exemple, si nous créons une variable **vraifaux**, lui donnons la valeur **True**, alors, quand nous affichons **not vraifaux**, nous obtenons **False** :

---

```
>>> x = True  
>>> print(not x)  
False
```

---

Ceci ne semble pas très utile mais, si tu le combines avec une instruction **if**, par exemple pour vérifier si un élément *n'est pas* présent dans une liste, tu commences à en découvrir toute l'utilité :

---

```
>>> liste_vetements = ['short', 'slip', 'boxer', 'bermuda', 'pantacourt']  
>>> if 'pantalon' not in liste_vetements:  
    print('Tu devrais vite t'acheter un pantalon')  
Tu devrais vite t'acheter un pantalon
```

---

## OR

Le mot-clé **or** (ou) sert à joindre deux conditions dans une instruction (**if**, par exemple), pour dire qu'au moins une des conditions doit être vraie. Voici un exemple :

---

```
if dino == 'Tyrannosaure' or dino == 'Allosaure':  
    print('Carnivores')  
elif dino == 'Ankylosaure' or dino == 'Brontosaure':  
    print('Herbivores')
```

---

Ici, si la variable `dino` contient `Tyrannosaure` ou `Allosaure`, alors le programme affiche `Carnivores`. Sinon, si `dino` contient `Ankylosaure` ou `Brontosaure`, alors il affiche `Herbivores`.

## PASS

Il arrive parfois, au cours de l'écriture d'un programme, qu'il faille le tester alors qu'il n'est pas complet. Le problème est qu'il n'est pas possible d'écrire une instruction `if` sans au moins le bloc de code nécessaire lorsque la condition est vraie. Il n'est pas non plus possible d'écrire une boucle `for` sans le bloc de code à répéter. Ainsi, le code suivant fonctionne normalement :

---

```
>>> age = 15
>>> if age > 10:
    print('plus vieux que 10 ans')
plus vieux que 10 ans
```

---

En revanche, si tu n'indiques pas le bloc de code, c'est-à-dire le corps de l'instruction `if`, tu obtiens un message d'erreur :

---

```
>>> age = 15
>>> if age > 10:
File "<stdin>", line 2
    ^
IndentationError: expected an indented block
```

---

C'est le type de message d'erreur que Python affiche quand il s'attend à trouver un bloc de code après une instruction quelconque et qu'il ne la trouve pas. Sous IDLE, il ne te laissera même pas exécuter le programme. Dans des cas comme celui-là, le mot-clé `pass` (passer) écrit à la place du corps de l'instruction indique à Python de ne pas s'inquiéter et de passer à la suite. Supposons par exemple que tu décides de créer une boucle `for` avec une instruction `if` à l'intérieur, mais que tu ne saches pas encore exactement ce que tu veux mettre dans le `if`. Peut-être que ce sera une fonction `print`, un `break` ou autre chose. Le mot-clé `pass` te permet de laisser cela en suspens et de faire en sorte que le programme fonctionne tout de même, même si tu n'es pas certain de ce qu'il faudra y écrire.

Voici la même instruction, avec `pass`, cette fois :

---

```
>>> age = 15
>>> if age > 10:
    pass
```

---

Et voici un autre exemple :

---

```
>>> for x in range(0, 7):
>>>     print('x vaut %s' % x)
>>>     if x == 4:
>>>         pass
x vaut 0
x vaut 1
x vaut 2
x vaut 3
x vaut 4
x vaut 5
x vaut 6
```

---

Python vérifie toujours que la variable `x` contient la valeur `4`, chaque fois qu'il exécute le bloc de code de la boucle, mais il ne fait rien en conséquence, donc il affiche chaque nombre de la plage (`range`) de `0` (inclus) à `7` (exclu).

Par la suite, si tu décides, par exemple, de sortir de la boucle lorsque `x` vaut une certaine valeur, il te suffit de modifier la valeur et de remplacer le `pass` par un `break` :

---

```
>>> for x in range(0, 7):
>>>     print('x vaut %s' % x)
>>>     if x == 5:
>>>         break
x vaut 0
x vaut 1
x vaut 2
x vaut 3
x vaut 4
```

---

Le mot-clé `pass` sert souvent lors de la création d'une fonction dont on ne sait pas encore ce qu'elle va faire, donc dont on n'écrit pas encore le code.

## RAISE

Le mot-clé `raise` (lever) peut servir à provoquer volontairement une erreur. Cela peut sembler étrange mais, en programmation avancée, cela s'avère réellement utile. Nous n'avons pas utilisé ce mot-clé dans le livre.

## RETURN

Le mot-clé `return` (renvoyer) permet de renvoyer une valeur au départ d'une fonction. Par exemple, si tu crées une fonction pour calculer ton âge en secondes à partir de ton âge en années, tu peux la réutiliser plusieurs fois et, dans la fonction, `return` renvoie la valeur calculée :

---

```
def age_en_secondes(age_en_annees):
    return age_en_annees * 365 * 24 * 60 * 60
```

---

Ensuite, quand tu appelles la fonction, la valeur renvoyée peut être affichée ou affectée à une variable :

---

```
>>> secondes = age_en_secondes(9)
>>> print(secondes)
283824000
>>> print(age_en_secondes(12))
378432000
```

---

## TRY

Le mot-clé `try` (essayer) débute un bloc de code qui se termine par les mots-clés `except` et `finally`. Ensemble, ces blocs `try/except/finally` servent à gérer les erreurs, pour être certain que le programme affiche par exemple un message utile à l'utilisateur, au lieu de l'erreur peu conviviale de Python. Ces mots-clés ne sont pas utilisés dans le livre.

## WHILE

Le mot-clé `while` (tant que) offre à peu près les mêmes fonctions de boucle que `for`, sauf que `while` continue de fonctionner tant qu'une condition reste vraie. Sois prudent lorsque tu utilises une boucle `while`, parce que, si l'expression de sa condition est toujours vraie, le `while` boucle à l'infini. D'ailleurs cela s'appelle une « boucle infinie » : la boucle ne s'arrête jamais. En voici un exemple :

---

```
>>> x = 1
>>> while x == 1:
...     print('bonjour')
```

---

Si tu exécutes ce code, il s'exécute à l'infini. Pour forcer l'arrêt du programme, ferme le shell de Python ou appuie sur `Ctrl-C`. En revanche, le code suivant affiche neuf fois `bonjour`, puis s'ar-

rête car chaque passage dans la boucle ajoute 1 à x, jusqu'à ce que x atteigne 10.

---

```
>>> x = 1  
>>> while x < 10:  
    print('bonjour ')  
    x = x +1
```

---

## WITH

Le mot-clé **with** (avec) s'applique à un objet pour créer un bloc de code d'une manière qui ressemble à celle de **try** et **finally**. Ce mot-clé n'est pas utilisé dans le livre.

## YIELD

L'utilisation du mot-clé **yield** ressemble un peu à celle de **return**, sauf que **yield** sert avec une classe bien déterminée d'objet, appelée un « générateur ». Les générateurs créent des valeurs à la volée, c'est-à-dire qu'ils créent ces valeurs à la demande. Sur ce plan, la fonction **range** se comporte comme un générateur. Ce mot-clé n'est pas utilisé dans le livre.



## GLOSSAIRE

Parfois, lorsque tu découvres la programmation, tu trébuches sur un nouveau terme qui ne te parle pas vraiment. Aussitôt, tu t'inquiètes et remets tout en question. Ce défaut de compréhension peut te frustrer. Ne panique pas, la solution n'est pas loin. D'abord, respire un bon coup, puis parcours ces quelques pages.

Ce glossaire est destiné à t'aider à comprendre un mot, un terme, alors que tu es en arrêt devant l'inconnu. Ces pages définissent nombre de termes de la programmation, évoqués dans ce livre donc n'hésite pas à y revenir si tu rencontres des mots que tu ne comprends pas.

**Animation** : processus d'affichage d'une séquence d'images qui va suffisamment vite pour donner l'impression d'un mouvement.

**Appeler** : l'exécution du code d'une fonction. Appeler une fonction signifie l'exécuter.

**Bloc (de code)** : est un groupe d'instructions dans un programme.

**Boîte de dialogue** : une boîte de dialogue est une petite fenêtre qui, dans une application, présente des informations contextuelles, comme un message d'alerte ou d'erreur. Elle peut aussi te demander de répondre à une question. Par exemple, quand tu ouvres un fichier, la boîte de dialogue qui s'affiche s'appelle généralement « Ouvrir... ».

**Booléen** : un type de variable qui n'a que deux valeurs possibles. En Python, cela s'exprime par des valeurs `True` (vrai) ou `False` (faux). Remarque les lettres capitales T et F. Vrai et faux sont des concepts, tandis que `True` et `False` en sont les écritures formelles du langage.

**Boucle** : commande ou groupe de commandes qui sont répétées plusieurs fois.

**Canevas** : il s'agit d'une zone de l'écran où tu peux dessiner. En pratique, ce que nous désignons par `canvas` est un objet d'une classe fournie par le module `turtle` ou `tkinter`.

**Chaîne (de caractères)** : une suite de caractères alphabétiques et numériques (autrement dit, alphanumériques), faite de lettres, de chiffres, de signes de ponctuation et d'espaces (en anglais, `string`).

**Classe** : c'est la description d'un type de chose. En termes de programmation, une classe est un ensemble de fonctions et de variables réservées à cette classe. C'est une sorte de modèle, à partir duquel des objets peuvent être créés.

**Cliquer** : appuyer sur un bouton de la souris pour activer un bouton, une option de menu, marquer un texte à l'écran et ainsi de suite.

**Collision** : dans le contexte des jeux pour ordinateur, ce terme indique qu'un personnage ou un objet du jeu rencontre un autre objet ou personnage de l'écran.

**Condition** : une expression dans un programme qui équivaut à une question. Les conditions sont, en définitive, vraies ou fausses.

**Coordonnées** : l'emplacement d'un point (ou pixel) de l'écran. Pour le décrire, il faut généralement indiquer un nombre *x* de pixels depuis le bord gauche de l'écran et un nombre *y* de pixels, du haut de l'écran vers le bas.

**Degrés** : une unité de mesure d'angles. Fort utilisée dans le module `turtle`.

**Dimensions** : dans le contexte de la programmation graphique, les deux dimensions et les trois dimensions font référence à la manière dont les images sont affichées par l'ordinateur. Les graphismes en deux dimensions (2D) sont des images aplatis à l'écran, avec une hauteur et une largeur, comme les dessins animés à la télé. Les graphismes en trois dimensions (3D) sont des images avec une hauteur, une largeur et une profondeur apparente de champ, et correspondent aux genres d'images que tu connais dans des jeux plus réalistes.

**Données** : généralement au pluriel, ce terme désigne des informations stockées et gérées par un ordinateur.

**Dossier** : l'emplacement d'un groupe de fichiers sur le disque dur ou un disque amovible (mémoire USB) de l'ordinateur.

**Enfant** : quand il s'agit de parler de classes, nous décrivons les relations entre classes comme de parent à enfant(s). Une classe enfant, ou plutôt fille, hérite des caractéristiques de ses classes parentes.

**Erreur** : lorsque quelque chose se passe mal dans un programme, il y a erreur. En programmation Python, toutes sortes de messages peuvent apparaître en réponse à une erreur. Si tu entres incorrectement l'espacement au début d'une ligne, par exemple, tu provoques une erreur de retrait (`IndentationError`).

**Événement** : un événement se produit pendant l'exécution d'un programme, par exemple quand l'utilisateur déplace sa souris, clique sur un de ses boutons ou appuie sur une touche du clavier.

**Exception** : type particulier d'erreur qui se produit pendant l'exécution d'un programme.

**Exécuter** : faire fonctionner du code, par exemple un programme, un petit extrait de code ou une fonction.

**Fonction** : est une commande dans un langage de programmation, constituée généralement de plusieurs instructions pour réaliser une tâche déterminée.

**Hexadécimal** : un système de représentation de nombres, très utilisé en programmation. Les nombres hexadécimaux sont en base 16, ce qui signifie que les chiffres hexadécimaux vont de 0 à 9 et ensuite de A à F. Le chiffre F en base 16 équivaut à 15 en base 10, par exemple.

**Horizontal** : les directions gauche et droite de l'écran, représentées par *x*.

**Identifiant (ou identificateur)** : nombre qui désigne de manière unique quelque chose dans un programme. Par exemple, dans le module `tkinter` de Python, chaque forme dessinée sur le canevas est reliée à un identifiant.

**Image** : une parmi une série d'images (*frames*) qui, enchaînées très vite, produisent une animation. Le terme désigne aussi un dessin ou un graphisme dessiné à l'écran.

**Importer** : en Python, importer signifie mettre un module et son contenu à disposition d'un programme.

**Initialiser** : désigne la définition du tout premier stade, initial, d'un objet, c'est-à-dire le réglage des valeurs des variables de l'objet lors de sa création.

**Intégrer (des valeurs)** : remplacer, insérer des valeurs dans une chaîne. Les valeurs remplaçées sont aussi appelées « espaces réservés ».

**Installation** : c'est le processus de copie des fichiers d'une application logicielle sur un ordinateur, pour que cette application puisse y fonctionner.

**Instance** : une classe est un modèle. Pour pouvoir l'utiliser, il faut créer une instance de cette classe, autrement dit, un objet.

**Logiciel** : ensemble de programmes informatiques. On parle aussi de « suite logicielle », quand plusieurs programmes sont rassemblés pour offrir des services variés, par exemple de bureautique, comme LibreOffice ou Microsoft Office.

**Lutin** : un personnage ou un objet graphique d'un jeu (en anglais, *sprite*).

**Mémoire** : composant de l'ordinateur qui sert à stocker, mémoriser temporairement des informations.

**Module** : ensemble de classes, de fonctions et de variables.

**Mot-clé** : mot particulier utilisé dans un langage de programmation. Les mots-clés (*keywords*) sont aussi appelés « mots réservés » du langage, ce qui signifie qu'ils ne peuvent être utilisés pour quoi que ce soit d'autre. Par exemple, il n'est pas permis d'utiliser un mot-clé pour désigner une variable.

**Nul** : c'est l'absence de valeur. En Python, le mot-clé `None` désigne l'absence de valeur. Attention : 0 est une valeur et non l'absence de valeur.

**Objet** : instance bien spécifique d'une classe. Lors de la création d'un objet d'une classe, Python réserve une portion de mémoire de l'ordinateur pour stocker les informations de ce membre de la classe.

**Opérateur** : élément d'un programme utilisé pour un calcul mathématique ou pour comparer des valeurs.

**Paramètre** : valeur utilisée par une fonction lorsque celle-ci est appelée ou lors de la création d'un objet, par exemple à l'appel de la fonction `_init_` d'une classe par Python. On dit passer une valeur en paramètre ou, aussi, en argument de la fonction.

**Parent** : lorsqu'il s'agit de parler des classes et des objets, le parent d'une classe est une autre classe dont elle hérite. Le mécanisme de l'héritage permet à une classe d'hériter des fonctions (méthodes) et des variables (caractéristiques) de sa classe parente ou de ses classes parentes. Lorsqu'il ne s'agit pas de programmation, un parent est la personne qui te dit de ranger ta chambre ou de te brosser les dents après les repas...

**Pixel** : point lumineux de l'écran de l'ordinateur. C'est le plus petit point qu'un ordinateur soit capable de dessiner.

**Portée** : portion d'un programme dans laquelle une variable est visible, donc utilisable. Une variable définie dans une fonction n'est en principe pas visible (sauf si elle est déclarée comme globale) en dehors de cette fonction. On dit alors que la portée de la variable se limite à la fonction.

**Programme** : ensemble de commandes et d'instructions qui disent à un ordinateur ce qu'il doit faire.

**Shell (de Python)** : de l'anglais *shell*, « coquille », ce terme désigne une interface en ligne de commande, c'est-à-dire un environnement d'exécution qui fonctionne ligne par ligne, où tu peux entrer des commandes. Le terminal de Linux est un shell, l'invite de commande et le Powershell de Windows sont des shells. Le « shell de Python » est utilisé dans ce livre pour désigner l'application IDLE.

**Syntaxe** : disposition obligatoire des mots, dans un ordre bien défini, pour constituer un programme. La syntaxe désigne l'ensemble des règles du langage.

**Transparence** : en programmation graphique, c'est la partie d'une image qui n'est pas affichée, ce qui signifie qu'elle ne masque pas le fond, c'est-à-dire tout ce qui est présent derrière elle.

**Variable** : tout ce qui permet de stocker, de mémoriser des valeurs. Une variable se comporte comme une étiquette qui désigne des informations contenues dans la mémoire de l'ordinateur. Les variables ne sont pas associées en permanence à une valeur déterminée mais peuvent évoluer. C'est d'ailleurs pour cela qu'elles s'appellent « variables », donc qui varient.

**Vertical** : les directions vers le haut et le bas de l'écran, représentées par *y*.

## Index

### Symboles

2D, graphisme en deux dimensions 167  
3D, graphisme en trois dimensions 168  
dessiner  
    avec le module turtle  
        carré plein 15  
    15 160  
{ } (accolades), création de dictionnaire 46  
+ (addition) 23  
\ (barre oblique inverse)  
    caractère d'échappement 35  
    découper une ligne de code sur  
        plusieur lignes 243  
(deux points)  
    dans une condition if 60  
     séparateur d'éléments de  
        dictionnaire 35  
\\" (caractère d'échappement) 35, 129  
"" (chaîne multiligne) 33  
[ ] (crochets)  
    pour création de liste 39  
    pour indice de liste 39  
/ (division) 23  
\_init\_, initialiser un objet 110  
>>> (invite de commande) 17  
\* (multiplication) 22, 23  
. (opérateur point) 103, 111  
( ) (parenthèses)  
    avec les classes et objets 100  
    dans les fonctions 89  
    pour création de tuples 45  
+ (signe plus)  
    pour regrouper des listes 42  
% (signe pourcent)  
    comme espace réservé 36  
= (simple signe égal) 80

# (symbole dièse)  
    commentaire 179  
    nombre hexadécimal 179  
% (symbole pourcent) 36  
    comme espace réservé 179  
intégration de valeur 36  
opérateur modulo 153

**A**  
absence de valeur (None) 67  
abs, fonction 114  
action, lier à un événement 191  
actions de classe 102  
Ada 10  
addition (opérateur +) 23  
Adobe Flash 294  
afficher  
    le contenu d'un dictionnaire 47  
    le contenu d'une liste 39  
    les fonctions disponibles pour une  
        variable ou une valeur 116  
aide  
dir, liste des fonctions et variables  
    116  
help, détails sur une classe, fonction  
    ou variable 118  
kwlist, liste des mots-clés 137  
version, version de Python actuelle  
    142  
ajouter  
    des éléments à une liste 41  
    un objet à une classe 100  
Alice 294  
and, mot-clé 66, 302  
Android, téléphone mobile 296  
animation 168, 188  
ajouter de l'action 204  
avec des lutins 230  
avec tkinter 188  
boucle principale 203  
dans M. Filiforme court vers la sortie  
    234, 270  
définition 316

deux dimensions (2D) 167

image (frame) 168

définition 318

ralentir l'exécution 205

succession d'images 270

sur papier 168

trois dimensions (3D) 168

appeler une fonction 89

définition 316

application. Voir dictionnaire

argument d'une fonction 319

asctime, fonction. Voir time, module

as, mot-clé 302

AssertionError 303

assert, mot-clé 302

association clé-valeur. Voir dictionnaire

## B

barre oblique inverse (\)

dans les chaînes 35, 129

Basic 10

BlitzBasic 294

bloc de code 60, 78

cohérence 62

définition 316

erreur de retrait 62

boîte de dialogue, définition 316

Boolean. Voir booléen

booléen 115

bool 115

définition 316

faux (False) 115

vrai (True) 115

bool, fonction 115

boucle, définition 316

boucle for 73

avec une liste 76

break pour sortir 82

comparée à while 81

comparer le code sans boucle 75

continue 278

dans une boucle 79

erreur de syntaxe 77

fonction range 74

boucle infinie

avec while 83

boucle principale 203

boucle while 81

boucle infinie 83

break pour sortir 82

comparée à for 81

incrémenter 83

break, mot-clé 82, 303

## C

calcul 119

avec le shell Python 22

simple 21

canal alpha 230, 233

canevas 172

canvas, objet

fonction coords 206

fonction winfo\_height 206

fonction winfo\_width 208

créer avec le module tkinter 172

sans bordure 200

créer avec le module turtle 50

définition 316

caractère d'espacement 60

caractéristiques d'une classe 101, 102

chaîne 32

aligner 37

apostrophes () 33

caractère d'échappement (\) 35

classe string 117

convertir en capitales 118

convertir en nombre

float, fonction 69

int, fonction 68

définition 316

différence avec les nombres 67

erreurs de syntaxe 34

%, insertion de valeur 36

intégrer des valeurs 36

hexadécimal (%x) 179

longueur 121

multiligne ("") 33  
multiplier par un nombre 37  
nombre de caractères 121  
problème courant 33  
supprimer espaces et Entrée des entrées de l'utilisateur, rstrip 116  
upper 118  
C, langage de programmation 296  
C#, langage de programmation 297  
C++, langage de programmation 296  
classe 98  
ajouter un objet 100  
caractéristiques 102  
classe enfant 99  
classe parente 99  
class, mot-clé 99  
définir des fonctions 102  
définition 316  
description par le module turtle 105  
diagramme en arbre 98  
fonction appelant d'autres fonctions 108  
fonctions de classe 102  
générateur 314  
hériter des fonctions 107  
initialiser un objet 110  
instance 100  
mot-clé class 99  
opérateur point(.) 111  
parenthèses () 100  
propriétés 110  
relations 99  
self 102  
class, mot-clé 99, 303  
clé mémoire USB 128  
pour fichiers de travail 125  
cliquer, définition 316  
collision, définition 316  
coloration syntaxique 4  
combinaison de couleurs RVB 156  
couleurs primaires 156  
exemples de couleurs 157  
noir et blanc 158  
commande 10  
commentaire 179  
compter des éléments  
list et range 88  
conditions 59  
and, mot-clé 66  
combiner 66  
définition 317  
False (faux) 62  
not, mot-clé 116  
opérateurs 63  
or, mot-clé 66  
True (vrai) 60  
console 19  
en ligne de commande 19  
continue, mot-clé 278, 304  
convertir  
une chaîne en nombre 68  
un nombre en chaîne 68  
coordonnées 172  
définition 317  
coords, fonction 206  
copier-coller 27  
copier des objets  
en profondeur 136  
superficielement 136  
copier un fichier 131  
copy, module 134  
copie en profondeur 136  
copie superficielle 136  
copy, fonction 134  
deepcopy, fonction 136  
corps d'une fonction 89  
couleur  
changer avec la fonction itemconfig 194  
noms prédéfinis dans tkinter 179  
primaire 156  
réglage  
avec le module tkinter 178  
avec le module turtle 155, 162

sélectionner avec tkinter

(colorchooser) 180

créer

une liste de nombres 74, 88

calculer la somme 125

une variable 25

un fichier texte sous Linux 127

un fichier texte sous OS X 126

un fichier texte sous Windows 125

## D

date et heure 142

référence 0 142

selon le fuseau horaire 144

time.asctime() 144

time.localtime() 144

time.sleep() 145

time.time() 142

date sous forme de tuple 144

def, mot-clé 305

dans les fonctions 89

fonction de classe 102

degrés 53

cercle 54

d'arcs 182

définition 317

dessin d'étoiles 151

del, mot-clé 42, 305

dessiner

avec le module tkinter 168, 195

arc 181

carré 174

damier 258

fonction itemconfig du canevas 194

fonction pack 172

identifiant 173, 189, 193

lier un événement 191

ligne 172

objet tk 169

ovale ( cercle ) 183, 201

PhotoImage 187

polygone 183

rectangle 174, 176

texte 185

variable keysym 192

avec le module turtle 49, 149

carré 150

carré plein 150

cercle plein 157

étoile à huit branches 151

étoile en spirale 152

étoile pleine 162

ligne 173

voiture 154

effet miroir 235

pour M. Filiforme court vers la sortie  
232

arrière-plan 237

personnage en fil de fer 233

plates-formes 236

porte 236

retourner une image avec Gimp 235

dessin statique 188

détection de collision 215

M. Filiforme court vers la sortie (jeu)  
276

Rebondir ! (Jeu) 215

deux dimensions, graphisme (2D) 167

diagramme

en arbre 98

en cercle 54

dict. Voir dictionnaire

dictionnaire 45

à partir d'un fichier 147

clé 45

création ( { } ) 46

dict 45

erreur de type 47

nombre d'éléments 122

remplacer une valeur 47

supprimer une valeur 47

trouver une valeur 46

valeur 45

vers un fichier 146

dimensions, définition 317

dir, fonction 116  
division (opérateur /) 23  
données, définition 317  
dossier, définition 317

## E

écrire  
    dans un fichier 129  
    des messages à l'écran 141  
elif, mot-clé 65, 305  
else, mot-clé 64, 306  
emplacement de fichiers 125  
    clé mémoire USB 125  
end-of-line. Voir fin de ligne (EOL)  
enfant  
    de classe 99  
    définition 317  
entier, nombre 69, 120  
entrée au clavier 67  
    éliminer les espaces, rstrip 116  
    input 116, 119, 141  
    sys.stdin.readline() 116, 141  
    touches de curseur 67  
entrée standard (stdin) 93, 141  
environnement  
    de développement intégré 4, 11  
    d'exécution de Python 17  
EOL. Voir fin de ligne (EOL)  
erreur, définition 317  
erreurs  
    AssertionError 303  
    erreur de syntaxe 119  
    NameError, portée de variable 91  
    retrait 62  
    SyntaxError, erreur de syntaxe 34,  
        62, 77  
    SystemExit 140  
    TypeError, erreur de type 43, 44, 47  
    ValueError, erreur de valeur 69, 121  
espace 60  
    réservé 36, 80  
eval, fonction 118

évaluer une expression Python 118, 119  
ordre des opérateurs 24  
parenthèses () 24  
événement 191, 265  
clavier 265  
définition 317  
lier à une action 191  
objet 265  
    event 191  
exception, définition 317  
except, mot-clé 306  
exec, fonction 119  
exécuter un programme 18  
    définition 318  
expression Python 118, 119

## F

False, mot-clé 115  
    condition fausse 62  
fichier  
    créer 125  
        sous Linux 127  
        sous OS X 126  
        sous Windows 125  
    lire 128  
    objet 128, 141  
        copier 131  
        écrire (write) 129, 130  
    fermer (close) 130  
    lire (read) 129  
    ouvrir (open) 128  
    ouvrir 128  
        sous Linux 129  
        sous OS X 129  
        sous Windows 128  
fille, classe. Voir enfant  
finally, mot-clé 306  
float, fonction 69, 120  
fonction 18, 41. Voir aussi fonctions intégrées  
    appeler 89  
    différentes valeurs 92

corps 89  
de classe 101  
définition 318  
def, mot-clé 89  
nom 89  
paramètres 89  
    portée 89  
parties d'une fonction 89  
portée des variables 90  
renvoi de valeur 90  
return, mot-clé 90  
fonctions intégrées 114  
    abs 114  
    bool 115  
    dir 116  
    eval 118  
    exec 119  
    float 69, 120  
    help 118  
    input 116  
    int 68, 121  
    len 121  
    list 88  
        dans des boucles for 74  
    max 122  
    min 123  
    open 128  
    print 18  
    range 88, 123  
        dans des boucles for 74, 122  
    itérateur 124  
    str 68  
    sum 125

forcer une fenêtre à l'avant-plan (avec tkinter) 201

for, mot-clé 306. Voir boucle for

Fortran 10

from, mot-clé 306

**G**

générateur 314  
    mot-clé yield 314

GIF, image 186

exporter avec Gimp 235

Gimp, programme GNU de manipulation d'image 232  
    effet miroir 235  
    retourner une image 235  
    transparence 233

global, mot-clé 308

graphisme  
    animation 232  
    avec Gimp 232  
    deux dimensions (2D) 167  
    isométrique 167  
    lutin 230  
    pseudo-3D 168  
    transparence 233  
    trois dimensions (3D) 168  
    vectoriel 50

**H**

help, fonction 118

héritage 107

hexadécimal, nombre 179  
    définition 318

hiberner 145, 205

horizontal, définition 318

HTML 298

**I**

identifiant 173, 189, 193  
    définition 318

IDLE 17  
    copier-coller 27  
    définition 4  
    démarrer 17  
    exécuter le programme 18  
    installation 11  
    nouvelle fenêtre 18

if, instruction 59, 64  
    elif, mot-clé 65, 305  
    else, mot-clé 64, 306  
    mot-clé 60, 308

image  
    afficher avec le module tkinter 186

GIF. Voir GIF, image  
image (frame)  
    animation 168, 234  
    effet miroir avec Gimp 235  
importer des modules  
    définition 318  
import, mot-clé 309  
incrémenter 83  
initialiser, définition 318  
in, mot-clé 309  
input, fonction 116  
installation, définition 318  
instance 100  
    définition 318  
instruction 10  
int, fonction 68, 121  
integer. Voir entier, nombre  
intégrer des valeurs 36, 179  
    définition 318  
interface utilisateur graphique 2  
invite de commande (>>>) 17  
is, mot-clé 310  
isométrique, graphisme 167  
itérateur 74, 124  
    boucle for 74  
    convertir en liste 124

**J**

Java, langage de programmation 296  
JavaScript, langage de programmation 299  
jeu. Voir Rebondir ! – M. Filiforme court vers la sortie

**L**

lambda, mot-clé 310  
langage de programmation 296  
    Ada 10  
    Basic 10  
    définition 10  
    Fortran 10  
    graphisme et jeu 294  
    Logo 2

Pascal 10  
pour sites web 298, 299  
pour téléphones mobiles 296, 298  
Python 2, 10  
len, fonction 121  
lier une action à un événement  
    avec le module tkinter 191  
Rebondir ! (Jeu) 213  
Linux, installer Python 16  
lire une entrée au clavier  
    éliminer les espaces, rstrip 116  
    input 116, 119, 141  
    sys.stdin.readline() 93, 141  
lire un fichier 129  
    de données. Voir aussi pickle, module  
list, fonction 124  
    et fonction range 88  
liste 39  
    afficher le contenu 39  
    ajouter des éléments (append) 41  
    append, fonction 41  
    avec boucle for 76  
    création ([ ]) 39  
    de listes 41  
    de nombres, création 88  
    différence d'un tuple 45  
    division interdite 43  
    erreur de type 43  
    et fonction range 88  
    indice 39  
        0 pour le premier élément 40  
        de liste ([ ]) 39  
    modifier un élément 40  
    multiplier par un nombre (\*) 43  
    nombre d'éléments 121  
    plage d'éléments ([ \ ]) 40  
    regrouper des listes (+) 42  
    somme des éléments (sum) 125  
    soustraction interdite 43  
    supprimer des éléments (del) 42  
    TypeError, erreur de type 44  
logiciel, définition 9, 318

- Logo (langage) 2  
lutin 230, 253  
    avec Gimp 230  
    définition 319  
    Voir aussi Rebondir ! (Jeu) ; M.  
Filiforme court vers la sortie (jeu)
- M**
- Mac OS X  
    installer Python 13  
paramétrer IDLE 14  
raccourci clavier d'IDLE 20  
map. Voir dictionnaire  
max, fonction 122  
mémoire, définition 319  
message d'erreur. Voir erreurs  
M. Filiforme court vers la sortie (jeu) 229  
    arrière-plan  
        dessiner 237  
    classe Lutin 253  
    classe LutinPersonnage 262  
    classe LutinPlateForme 254  
    classe LutinPorte 283  
    éléments graphiques 232  
    lutin (sprite), création 253  
M. Filiforme 262  
    animation 270  
    charger les images 263  
    classe LutinPersonnage 262  
    déplacer 264  
    dessiner 233  
    liaison des touches du clavier 264  
plan 230  
plates-formes  
    ajouter au jeu 255  
    dessiner 236  
porte, dessiner 236  
min, fonction 123  
module 50, 92  
    contenu 92, 133  
    copy 134. Voir copy, module  
    définition 319  
    importer 50, 93
- pickle. Voir pickle, module  
sys 93. Voir sys, module  
time 93. Voir time, module  
tkinter 2. Voir tkinter, module  
turtle 2. Voir turtle, module  
modulo, opérateur (%) 153  
mots-clés 301  
    and 66, 302  
    as 302  
    assert 302  
    break 82, 303  
    class 99, 303  
    continue 278, 304  
    def 305  
        fonction 89  
        fonction de classe 102  
    définition 319  
    del 42, 305  
    elif 65, 305. Voir aussi if, instruction  
    else 64, 306. Voir aussi if, instruction  
    except 306  
    False 115  
    finally 306  
    for 73, 306. Voir aussi boucle for  
    from 306  
    global 308  
    if 308  
    import 309  
    in 309  
    is 310  
    lambda 310  
    not 116, 310  
    or 66, 310  
    pass 99, 311  
    raise 312  
    return 90, 313  
    True 115  
    try 313  
    while 81, 313  
    with 314  
    yield 314
- multiplication (opérateur \*) 23

multiplier une liste par un nombre 43

## N

NameError, erreur de portée de variable  
91

nombres

à virgule flottante 120

convertir

en chaîne 68

en entier 121

une chaîne en ~ 68

différence avec les chaînes 67

entier 69, 120

erreur de valeur 69, 121

hexadécimal 179

réel 120

valeur absolue 114

nom de fichier

sous Linux (Ubuntu) 129

sous OS X 129

sous Windows 128

nom d'une fonction 89

None

absence de valeur 67

valeur vide 67

not, mot-clé 116, 310

nul, définition 319

n-uplet. Voir tuple

## O

Objective-C, langage de programmation  
298

objet 97, 99, 100

ajouter aux classes 100

définition 319

entrée standard (stdin) 93, 141

événement 265

fichier 128. Voir aussi fichier, objet  
identifiant 193, 202

initialiser 110

instance 100

opérateur point (.) 111

sortie standard (stdout) 141

obtenir la date et l'heure

module time, fonction asctime 93

open, fonction 128

opérateur 23, 24

addition (+) 23

de comparaison 63

de condition 63

définition 319

division (/) 23

modulo (%) 153

multiplication (\*) 22, 23

ordre 24

parenthèses 24

point (.) 103

classes et objets 111

soustraction (-) 23

opération 24

ordre des opérateurs 24

organiser les choses en classes 98

or, mot-clé 66, 310

OS X

intaller Gimp 232

ouvrir un fichier 128

## P

papier peint 235, 237

paramètre 89

définition 319

d'une fonction 89

nommé 171

self 102

parent

de classe 99

définition 319

parenthèses () 24

avec les classes et objets 100

imbriquer 24

Pascal 10

pas, dans la fonction range 124

pass, mot-clé 99, 311

Perl, langage de programmation 299

PHP, langage de programmation 298

pickle, module 146  
dump, fonction 146  
lire un dictionnaire d'un fichier 147  
load, fonction 147  
vidage dans un fichier 146  
pixel 53  
définition 319  
point décimal 22, 120  
portée des variables 308  
dans les fonctions 90  
définition 319  
print (fonction) 18  
profondeur, copie en 136  
programmation informatique  
bases 2  
pourquoi l'apprendre ? 1  
stratégie en cas de problème 3  
programme informatique  
code 2  
commande 10  
découper en classes et objets 107  
définition 9, 320  
enregistrer 20  
établir un plan 230  
exécuter 18, 20  
instruction 10  
répartir le travail entre plusieurs programmeurs 107  
pseudo-3D, graphisme 168  
PyGame2 294  
Python  
environnement d'exécution 17  
fonction 18  
installation 11  
Mac OS X 13  
Ubuntu 16  
Windows 11  
origine 10  
pourquoi ? 2  
shell 17  
turtle, module 49

## R

raccourci clavier 20  
raise, mot-clé 312  
ralentir  
l'exécution d'une animation 205  
un programme 205  
random, module  
créer des rectangles aléatoires 176  
shuffle, fonction 207  
range, fonction 123  
arrêt 123  
dans des boucles for 74  
début 123  
et fonction list 88  
fonction intégrée 74  
itérateur 124  
pas 124  
Rebondir ! (Jeu) 199, 218  
ajouter un facteur chance 218  
balle 201  
changer de direction 207  
déplacer 204  
faire rebondir 206  
boucle principale 203  
canevas 200  
détection de collision 215  
heurter la raquette 215  
raquette 212  
déplacer au clavier 213  
regrouper des listes 42  
réinitialiser une variable 67  
répertoire 116  
reste de la division entière 153  
retourner une image avec Gimp 235  
retrait  
cohérence 62  
dans IDLE 62, 77  
des blocs de code 61  
erreur 77  
de retrait 62  
espaces cohérentes 77  
return, mot-clé 90, 313

RGB. Voir combinaison de couleurs RVB  
rstrip, supprimer espaces et Entrée des entrées de l'utilisateur 116  
Ruby, langage de programmation 299  
run. Voir exécuter un programme  
RVB. Voir combinaison de couleurs RVB

## S

Scratch 294  
self, paramètre 102  
shell de Python 17  
calcul simple 21  
copier-coller 27  
créer une nouvelle fenêtre 18  
définition 320  
fonctions intégrées 113  
réouvrir 19  
sleep, fonction. Voir time, module  
somme des éléments d'une liste 125  
sortie standard (stdout) 141  
soustraction (opérateur -) 23  
sprite. Voir lutin  
stdout, objet. Voir sortie standard  
str, fonction 68  
string, classe 117. Voir aussi chaîne  
rstrip, supprimer les espaces des entrées de l'utilisateur 116  
sum, fonction 125  
superficie, copie 136  
supprimer des éléments d'une liste 42  
syntaxe 33  
définition 320  
erreur 119  
SyntaxError  
dans la fonction eval 119  
erreur de syntaxe 33, 62, 77  
chaînes 34  
sys, module 93  
stdin, objet 93, 141  
readline, fonction 93, 141  
stdout, objet 141  
write, fonction 141

SystemExit 140

## T

tabulation 60  
tant que \ boucle while 81  
time, module 93, 142  
asctime, fonction 93, 144  
localtime, fonction 144  
sleep, fonction 145  
ralentir l'exécution d'une animation 205  
time, fonction 142  
tkinter, module 2, 167  
afficher  
image 186  
texte 185  
animation 188  
askcolor, fonction 180  
canvas (canevas) 172  
créer  
bouton 169  
canevas 172  
dessiner  
arc 181  
carré 174  
ligne 172  
ovale ( cercle ) 183, 201  
polygone 183  
rectangle 174  
et couleurs 178  
fonction pack 172  
identifiant 193  
installer 50  
move, fonction 213  
objet tk 169  
PhotoImage 187  
tk, objet  
définir le titre de la fenêtre 201  
forcer la fenêtre à l'avant-plan 201  
resizable, fonction 201  
title, fonction 201

update, fonction 201, 203  
update\_idletasks, fonction 203  
verrouiller la taille du canevas 201  
wm\_attributes, fonction 201  
variable keysym de l'objet event 192

tortue 49  
    apparence 49  
    canevas 50  
        effacer 55  
        réinitialiser 55  
    turtle, module 49

touches de curseur  
    entrée au clavier 67

transparence dans les images 230, 233, 238  
    définition 320

trois dimensions, graphisme (3D) 168

True  
    condition vraie 60  
    mot-clé 115

try, mot-clé 313

tuple 45  
    avec le module tkinter  
        colorchooser.askcolor 180  
        police, taille 185  
    création (( )) 45  
    différence de la liste 45  
    nombre d'éléments 121  
    non modifiable 45

turtle, module 49, 149  
    backward, fonction 55  
    begin\_fill, fonction 157  
    canevas 50  
    circle, fonction 155  
    clear, fonction 55  
    color, fonction 156  
    degrés. Voir degrés  
    dessiner  
        carré 150  
        carré plein 160  
         cercle plein 157  
        en noir et blanc 158

étoile à huit branches 151  
étoile en spirale 152  
étoile pleine 162  
ligne 173  
voiture 154

down, fonction 55  
end\_fill, fonction 157  
forward, fonction 52  
importer 50  
installer 50  
left, fonction 53  
lenteur 167  
Pen, classe 50, 97  
reset, fonction 55  
right, fonction 54, 55  
setheading, fonction 155  
up, fonction 55  
utiliser des boucles for 150

type de données  
    booléen 115  
    chaîne (string) 32  
    entier 69, 120  
    nombre à virgule flottante 120

TypeError, erreur de type 43, 44, 47

**U**

Ubuntu. Voir Linux  
Unity3D 294  
utilisateur  
    entrée 67  
        input 116  
        supprimer espaces et Entrée 116  
    touches de curseur 67

**V**

valeur absolue d'un nombre 114  
ValueError, erreur de valeur 69, 121  
variable 21, 25  
    caractère accentué déconseillé 26  
    création 25  
    définition 320  
    insérer des valeurs 36, 179  
    nom 26

portée  
    globale 308  
réinitialiser 67  
\_ (soulignement) 26  
vertical, définition 320  
vidage dans un fichier. Voir pickle, module  
virgule  
    décimale 120  
    flottante, nombre 120  
visibilité des variables  
    mot-clé global 308

## W

while, mot-clé 81, 313  
winfo\_height, fonction 206  
winfo\_width, fonction 208  
with, mot-clé 314  
write  
    écrire dans un fichier 130

## Y

yield, mot-clé 314