PYTHON ET LES CARACTÈRES ACCENTUÉS ET SPÉCIAUX

Python a fortement évolué ces dernières années et, selon les versions, il gère plus ou moins bien les caractères accentués. Dans ce livre, le conseil t'a été donné d'éviter systématiquement ces caractères accentués dans les noms de variables, de fonctions, de classes et même de fichiers. Nous allons donc voir pourquoi.

Les exemples présentés ici sont tirés de Linux (Ubuntu), parce que lorsque tu installes Linux, tu as généralement accès à Python 2, par défaut. Le chapitre 1 explique comment installer Python 3 en plus. Pour bien mettre en évidence quelques différences entre les deux versions, Linux se prête donc bien à ce genre d'expérience.

Différentes versions de Python

Au cours des années, l'auteur de Python, Guido van Rossum, aidé de la communauté des développeurs en Python, a fait évoluer, tant le langage lui-même, que l'environnement (le shell de Python) dans lequel il s'exécute.

Au moment de l'écriture de ces lignes, deux versions de Python coexistent en parallèle : les versions 2.7 et 3.4. Si nous sommes partis sur une des dernières, c'est parce qu'elle améliore bien des choses ; de plus, elle représente tout de même l'avenir. Il faut pourtant savoir que les versions 2 sont encore très utilisées, tout simplement parce que des centaines de modules externes existent, qui permettent de réaliser des choses très complexes, et que ces modules n'ont pas encore été tous convertis pour fonctionner en Python 3. D'ailleurs, sous Linux (Ubuntu), lorsque tu entres dans un terminal la commande `python`, tu obtiens ceci (le signe dollar, $, dans les lignes de code suivantes signifie que tu es dans un terminal, ne l'écris donc pas) :

```
$ python
Python 2.7.6 (default, Mar 22 2014, 22:59:56)
[GCC 4.8.2] on linux2
```

Par défaut, Ubuntu connaît une version 2 (2.7.6 dans ce cas-ci) de Python. Pour accéder à Python 3, il faut entrer `python3` :

```
$ python3
Python 3.4.0 (default, Apr 11 2014, 13:05:11)
[GCC 4.8.2] on linux
```

En conséquence, sous Linux (Ubuntu), où Python 2 est installé d'office, et où tu as installé Python 3 selon nos conseils au chapitre 1, pour exécuter dans un terminal le programme `Bonjour.py` du chapitre 2, page 12, il faut écrire :

```
$ python3 Bonjour.py
Bonjour tout le monde
```

Là où les choses se compliquent, c'est lorsqu'il s'agit de gérer les caractères accentués dans le code et dans l'environnement. Voyons cela en détail.

Caractères accentués et spéciaux

Toutes les langues humaines s'écrivent avec des caractères propres. L'anglais se contente d'une cinquantaine de caractères pour tout faire, le français ajoute une vingtaine de caractères accentués et spéciaux (é, à, è, ù, î, œ, ç, etc.), tandis que le chinois, le russe et le japonais utilisent leurs propres caractères. L'ordinateur, lui, n'en a que faire : il gère des 0 et des 1, peu importe ce que ces 0 et ces 1 représentent. Il a toutefois besoin de repères pour s'y retrouver.

Python 2 et les caractères accentués et spéciaux

Sur le Web, tu trouveras beaucoup de littérature et nombre de forums qui soulèvent les problèmes rencontrés sous Python 2 à propos des caractères accentués et spéciaux, et, si le sujet t'intéresse, consulte par exemple ces deux pages sur l'encodage des caractères : http://info.sio2.be/python/1/9.php et http://sebsauvage.net/python/charsets_et_encoding.html.

Python 2 peut gérer les caractères accentués et spéciaux mais il faut l'aider à le faire.

Ainsi, crée le programme Python suivant et enregistre-le sous le nom `accents.py` :

```
#! /usr/bin/python
print "À chaque problème, sa solution"
```

Exécute ensuite ce programme dans un terminal (pas le shell 3.4 de Python) :

```
$ ./accents.py
```

ou :

```
$ python accents.py
```

Python 2 lève les bras au ciel et crie « on arrête tout » :

```
  File "accents.py", line 2
SyntaxError: Non-ASCII character '\xc3' in file accents.py on line 2, but
no encoding declared; see http://www.python.org/peps/pep-0263.html for
details
```

Maintenant, insère la ligne suivante juste avant la ligne avec le `print` :

```
#! /usr/bin/python
# -*- coding: utf-8 -*-
print "À chaque problème, sa solution"
```

Puis exécute de nouveau le programme :

```
$ python accents.py
À chaque problème, sa solution
```

Ça fonctionne ! La deuxième ligne de commentaire indique en fait à Python d'utiliser le système d'encodage de caractères UTF-8, qui gère parfaitement les caractères accentués et spéciaux. Note qu'ici, c'est dans les chaînes de caractères que le problème apparaît, alors tu imagines le cafouillage quand ce sont les variables, les fonctions ou les classes qui portent des caractères accentués ou spéciaux ! D'ailleurs, essayons, avec le passage dans le code précédent, par une variable `machaîne` (avec un bel accent circonflexe sur le i), avec ou sans la ligne de commentaire `coding` :

```
#! /usr/bin/python
# -*- coding: utf-8 -*-
machaîne = "À chaque problème, sa solution"
print machaîne
```

Le fichier enregistré sous le nom `accents2.py`, son exécution donne :

```
$ python accents2.py
  File "accents2.py", line 3
    machaîne = "À chaque problème, sa solution"
         ^
SyntaxError: invalid syntax
```

Eh non, ça ne va plus du tout ! Et sous Python 3, ça fonctionne ? Essayons, mais d'abord, il y a une petite modification à apporter au programme. Pourquoi ? Parce que Python 3 connaît quelques différences de syntaxe par rapport à Python 2, mises en évidence dans le code :

```
#! /usr/bin/python3
# -*- coding: utf-8 -*-
machaîne = "À chaque problème, sa solution"
print(machaîne)
```

Enregistrons ces modifications dans un nouveau fichier, `accents3.py`, puis essayons :

```
$ python3 accents3.py
À chaque problème, sa solution
```

Et voilà, ça fonctionne ! Moralité, Python est capable de gérer les caractères accentués et spéciaux mais seulement dans les chaînes de caractères en ce qui concerne Python 2, et encore, il faut l'y « aider », alors que tous les exemples du livre démontrent que Python 3 les gère sans problème. Ensuite, Python 2 n'accepte pas les caractères accentués dans les noms de variables, fonctions et classes, tandis que Python 3 les accepte, grâce à l'encodage par défaut en UTF-8 des programmes et, un peu aussi, grâce au fait que le système d'exploitation fonctionne aussi en UTF-8.

Enfin, nous pourrions nous demander ce qu'il en est au niveau des noms de fichiers. Recopions le fichier `accents3.py` sous le nom de fichier `accentués3.py` et voyons ce que cela donne, avec le *é* dans le nom de fichier :

```
$ python3 accentués3.py
À chaque problème, sa solution
```

Ça fonctionne ! Et maintenant, reprenons le fichier `accents.py` et recopions-le sous le nom `accentués.py`, pour l'exécuter en Python 2 :

```
$ python accentués.py
À chaque problème, sa solution
```

Bien, ça fonctionne aussi sous Python 2. Ouf !

Python 3 et les caractères accentués et spéciaux

Pour Python 3, les développeurs ont décidé de faire table rase de toutes ces différences de codages de caractères et de simplifier le tout pour adopter un système de codage beaucoup plus universel, l'UTF-8, accepté aussi par la plupart des systèmes d'exploitation modernes, dont Windows, OS X et Linux dans leurs versions les plus récentes. L'avantage de l'UTF-8 réside dans le fait qu'il s'accommode de tous les caractères connus dans le monde entier et qu'il optimise automatiquement le passage à un autre codage de caractère : l'Unicode.

Donc, en principe, si tu utilises des caractères accentués ou spéciaux pour des noms de variables, de fonctions, de classes, tu ne dois plus rencontrer de problème en Python 3. En revanche, si tu vises ce que l'on appelle la compatibilité avec Python 2, là, tu dois faire attention. De toute façon, il est toujours préférable de les éviter, comme l'explique la section suivante.

Faut-il préférer l'anglais ou le français dans le code ?

Lorsque tu rédiges des programmes, avant tout, le but est que tu comprennes ce que tu fais et écris. Ensuite, si tu fais appel à des copains pour t'aider à construire des programmes plus évolués, il est bien entendu nécessaire qu'eux aussi comprennent ce que tu as écrit, ce qu'ils écrivent et ainsi de suite. Donc le français est parfait pour cela.

En revanche, si tu prolonges l'expérience de la programmation et si tu t'adresses un jour à des étrangers, comme cela se fait de plus en plus dans les forums, les clubs d'entraide et de développeurs, tu verras très vite que ces gens proviennent parfois de Chine, d'Inde, des États-Unis… Bref, il arrive un moment où l'anglais devient incontournable (on dit que, dans la programmation, l'anglais est la langue véhiculaire). Que dirais-tu si tu trouvais un module Python magique qui te permette de réaliser un jeu en trois dimensions en quelques instructions et si ce module était complètement en chinois ? La galère !

Les caractères accentués ou spéciaux dans les noms de fichiers, de variables, de fonctions, de classes représentent la même difficulté pour des gens de langue différente : les subtilités du français ne sont pas toujours à la portée de gens dont ce n'est pas la langue maternelle, et même de gens dont *c'est* la langue maternelle. C'est là une autre raison de les éviter.

Cela ne sert à rien d'ajouter des difficultés inutiles, alors que tu découvres la programmation. Retiens cette règle que nous avons déjà évoquée dans l'avant-propos : simplifier, simplifier et simplifier. Si tu te sens à l'aise en anglais, rappelle-toi que celui qui relit ton code ne se sent peut-être pas du tout à l'aise donc pense à lui aussi. De même, en programmation, le langage texto est strictement interdit.

SOLUTIONS DES PUZZLES DE PROGRAMMATION

Voici les solutions des puzzles de programmation proposés dans les fins des chapitres. Il n'existe pas nécessairement une seule solution à un puzzle donc ta solution n'est peut-être pas exactement celle que tu trouveras ici, mais ces exemples t'offrent une idée des approches possibles.

Chapitre 3

1. Favoris

Voici une solution avec trois loisirs favoris et trois nourritures préférées.

```
>>> jeux = ['Pokemon', 'LEGO Mindstorms', 'VTT']
>>> nourritures = ['crêpes', 'chocolat', 'pommes']
>>> favoris = jeux + nourritures
>>> print(favoris)
['Pokemon', 'LEGO Mindstorms', 'VTT', 'crêpes', 'chocolat', 'pommes']
```

Voir CH03_1_Favoris.py.

2. Compter les combattants

Plusieurs manières existent de compter les combattants. Nous avons 3 bâtiments avec 25 ninjas cachés sur chaque toit, puis 2 tunnels avec 40 samouraïs cachés dans chacun. Nous pouvons compter d'abord le nombre total de ninjas, puis le nombre total de samouraïs, puis ajouter ces deux nombres :

```
>>> 3*25
75
>>> 2*40
80
>>> 75+80
155
```

Il est plus facile et plus judicieux de combiner ces deux calculs à l'aide de parenthèses, même si les parenthèses ne sont pas indispensables, puisque l'ordre de priorité (on dit de « préséance ») des opérateurs mathématiques fait que les multiplications sont effectuées avant l'addition, mais elles apportent plus de clarté dans la lecture du calcul :

```
>>> (3*25)+(2*40)
```

S'il s'agit de faciliter la lecture, un plus joli programme Python utiliserait des variables, un peu dans le genre de la suite, qui clarifie ce qu'il fait :

```
>>> toits = 3
>>> ninjas_par_toit = 25
>>> tunnels = 2
>>> samourais_par_tunnel = 40
>>> print((toits * ninjas_par_toit) + (tunnels * samourais_par_tunnel))
155
```

3. Salutations

Dans cette solution, nous donnons aux variables des noms explicites, c'est-à-dire qui expliquent bien à quoi ils se rapportent, puis nous utilisons des espaces réservés de mise en forme (`%s %s`) dans notre chaîne pour intégrer les valeurs de ces variables :

```
>>> prenom = 'Jules'
>>> nom_famille = 'Martin'
>>> print('Bonjour, %s %s !' % (prenom, nom_famille))
Bonjour, Jules Martin !
```

Chapitre 4

1. Un rectangle

Dessiner un rectangle revient presque exactement au même que de dessiner un carré, sauf que la tortue doit dessiner deux côtés plus longs que les deux autres :

```
>>> import turtle
>>> t = turtle.Pen()
>>> t.forward(100)
>>> t.left(90)
>>> t.forward(50)
>>> t.left(90)
>>> t.forward(100)
>>> t.left(90)
>>> t.forward(50)
```

Voir `CH04_1_Rectangle.py`.

2. Un triangle

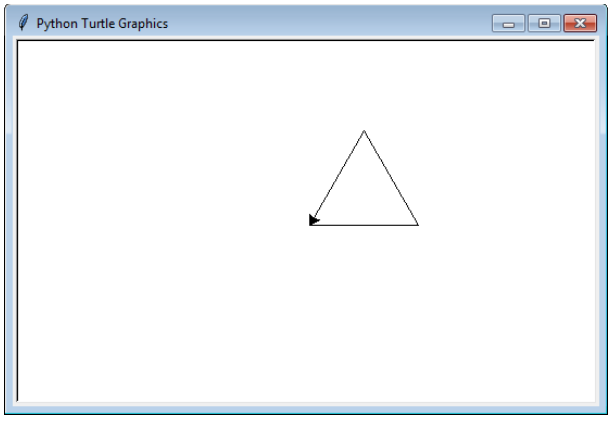
Cet exercice n'indique pas le genre de triangle à dessiner. Il en existe de trois types : équilatéral, isocèle et scalène. Selon ton niveau de connaissance de la géométrie, tu peux dessiner n'importe quel type de triangle, éventuellement en jouant avec les angles pour finir par obtenir quelque chose qui ressemble à un triangle.

Dans l'exemple proposé ici, nous nous concentrons sur les deux premiers types, parce qu'ils sont les plus faciles à tracer. Un triangle équilatéral possède trois côtés de longueurs identiques et trois angles de même mesure :

```
>>> import turtle
>>> t = turtle.Pen()
❶ >>> t.forward(100)
❷ >>> t.left(120)
❸ >>> t.forward(100)
❹ >>> t.left(120)
❺ >>> t.forward(100)
```

Voir CH04_2_Triangle.py.

Pour tracer la base du triangle, nous nous déplaçons vers l'avant de 100 pixels en ❶. Nous tournons vers la gauche de 120 degrés, en ❷, ce qui crée un triangle intérieur de 60 degrés, nous avançons de nouveau de 100 pixels en ❸, En ❹, le virage de 120 degrés suivant nous ramène en direction du point de départ et, en ❺, nous fermons le triangle en avançant encore de 100 pixels. Voici le résultat du code :



Le triangle isocèle possède deux côtés et deux angles égaux :

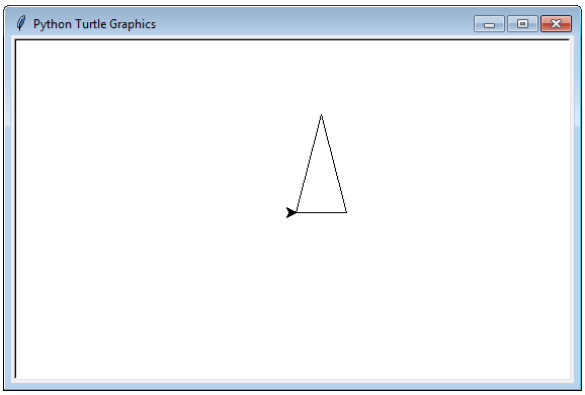
```
>>> import turtle
>>> t = turtle.Pen()
>>> t.forward(50)
>>> t.left(104.47751218592992)
>>> t.forward(100)
>>> t.left(151.04497562814015)
>>> t.forward(100)
```

Dans cette solution, la tortue avance de 50 pixels, pour tourner vers la gauche de 104,47751218592992 degrés. Elle avance ensuite de 100 pixels, pour tourner à gauche de 151,04497562714015 degrés, puis avance de nouveau de 100 pixels. Pour faire tourner la tortue

vers son point de départ, nous entrons une nouvelle fois la ligne suivante :

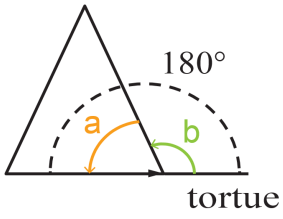
```
>>> t.left(104.47751218592992)
```

Voir CH04_2_TriangleIsocele.py.
Et voici le résultat du code :

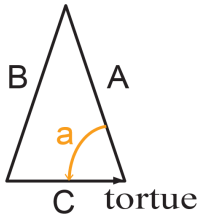


Comment avons-nous obtenu des nombres nébuleux comme 104,47751218592992 degrés et 151,04497562814015 degrés pour ces angles ?

Dès que la décision est prise pour les longueurs des côtés du triangle, il est possible de calculer les angles intérieurs à l'aide de Python et d'un peu de trigonométrie. Le diagramme suivant montre que, lorsqu'on connaît l'angle *a*, nous pouvons déterminer les degrés de l'angle *b* dont la tortue doit tourner à gauche. En effet, la somme des angles *a* et *b* donne toujours 180 degrés.



Il n'est pas difficile de calculer l'angle intérieur quand on connaît la bonne formule. Admettons que nous créions un triangle de 50 pixels pour la base (le côté du bas), que nous appelons *C*, et de 100 pixels pour les deux autres côtés, appelés *A* et *B*.



La formule de calcul de l'angle intérieur a à partir des côtés A, B et C est la suivante :

$$a = \arccos\left(\frac{A^2 + C^2 - B^2}{2AC}\right)$$

Un petit programme permet de calculer la valeur à l'aide du module `math` de Python :

```
>>> import math
>>> A = 100
>>> B = 100
>>> C = 50
❶ >>> a = math.acos((math.pow(A,2) + math.pow(C,2) - \
        math.pow(B,2)) / (2*A*C))
>>> print(a)
1.31811607165
```

La première ligne importe le module `math`. Les trois lignes suivantes définissent les variables qui contiennent les longueurs des trois côtés. En ❶, nous exprimons la formule à l'aide de la fonction `acos` (arc cosinus) pour calculer l'angle, et la fonction `pow` pour porter les longueurs au carré. Le calcul renvoie la valeur en radians de 1,31811607165. Le radian est une autre unité de mesure d'angle, comme les degrés.

NOTE *La barre oblique inverse (\) en fin de ligne ❶ ne fait pas partie de la formule mais sert à indiquer à Python que la suite de cette ligne se situe à la ligne suivante. Au chapitre 16, nous reviendrons sur ce sujet. Ces barres obliques inverses ne sont pas indispensables mais elles permettent d'éclater une longue ligne de code sur plusieurs, pour en améliorer la lecture, quand, comme ici, la largeur de la page ne permet pas de voir toute une ligne.*

La valeur en radians peut être convertie en degrés à l'aide de la fonction `degrees` du module `math`. Ensuite, nous pouvons calculer

l'angle extérieur, dont doit tourner la tortue, en soustrayant l'angle intérieur des 180 degrés :

```
>>> print(180 - math.degrees(a))
104.477512186
```

Pour calculer l'angle de rotation suivant de la tortue, la formule est la même mais adaptée aux autres côtés :

$$b = \arccos\left(\frac{A^2 + B^2 - C^2}{2AB}\right)$$

Le code de cette formule est fort semblable aussi :

```
>>> b = math.acos((math.pow(A,2) + math.pow(B,2) - \
        math.pow(C,2)) / (2*A*B))
>>> print(180 - math.degrees(b))
151.04497562814015
```

Voir CH04_2_CalculAngles_TriangleIsocele.py.

Bien entendu, les formules trigonométriques ne sont pas indispensables et tu peux parfaitement déterminer les bons angles par la méthode des essais-erreurs, jusqu'à ce que tu obtiennes quelque chose qui semble correct.

3. Un carré sans coins

Si tu regardes bien la forme générée, tu devines sans doute que la solution de cet exercice se situe dans un octogone (figure à huit angles). Il s'agit de faire quatre fois à la suite la même chose : avancer, tourner à gauche de 45 degrés, lever le stylo, avancer, abaisser le stylo et encore une fois tourner à gauche de 45 degrés :

```
t.forward(50)
t.left(45)
t.up()
t.forward(50)
t.down()
t.left(45)
```

Donc, l'ensemble final de commandes reprend quatre fois ces commandes, qu'il vaut mieux noter dans une nouvelle fenêtre d'IDLE, puis enregistrer sous le nom de fichier CH04_4_CarreSansCoins.py :

```
import turtle
t = turtle.Pen()
t.forward(50)
t.left(45)
t.up()
t.forward(50)
t.down()
t.left(45)
t.forward(50)
t.left(45)
t.up()
t.forward(50)
t.down()
t.left(45)
t.forward(50)
t.left(45)
t.up()
t.forward(50)
t.down()
t.left(45)
t.forward(50)
t.left(45)
t.up()
t.forward(50)
t.down()
t.left(45)
```

Chapitre 5

1. Es-tu riche ?

Tu obtiens en fait une erreur de retrait (`IndentationError`) lorsque tu atteins la dernière ligne de l'instruction `if` :

```
>>> mes_sous = 2000
>>> if mes_sous > 1000:
❶       print("Je suis riche !!")
else:
        print("Je ne suis pas riche !!")
❷       print("Mais ça viendra...")
SyntaxError: unexpected indent
```

Tu provoques cette erreur parce que le premier bloc, en ❶, débute à quatre espaces, donc Python ne s'attend pas à voir deux espaces supplémentaires à la ligne ❷. En conséquence, il surligne l'endroit

où il détecte le problème avec un rectangle rouge, pour t'indiquer où tu dois apporter une correction.

Voir Es-tu-riche.py.

2. Barres chocolatées

Le code qui permet de vérifier que le nombre de barres chocolatées est plus petit que 100 ou plus grand que 500 est du genre du suivant :

```
>>> barres_choco = 600
>>> if barres_choco < 100 or barres_choco > 500:
>>>     print('Trop peu ou beaucoup trop')
Trop peu ou beaucoup trop
```

Voir Barres_choco.py.

3. Juste le bon nombre

Tu pourrais écrire plusieurs instructions if pour vérifier si la somme d'argent est entre 100 et 500 ou entre 1 000 et 5 000, mais l'utilisation des mots-clés and et or permet d'écrire toutes ces conditions dans une seule instruction if :

```
if (somme >= 100 and somme <= 500) or (somme >= 1000 \
        and somme <= 5000):
    print('La somme est comprise entre 100 et 500 ou entre 1000 et 5000')
```

Vérifie bien la présence des parenthèses autour de chacune des conditions, pour que Python vérifie que la somme est comprise entre 100 et 500 *ou* entre 1000 et 5000.

Pour vérifier le code, donnons à somme plusieurs valeurs, au choix :

```
somme = 800
if (somme >= 100 and somme <= 500) or (somme >= 1000 \
        and somme <= 5000):
    print('La somme est comprise entre 100 et 500 ou entre 1000 et 5000')

somme = 400
if (somme >= 100 and somme <= 500) or (somme >= 1000 \
        and somme <= 5000):
    print('La somme est comprise entre 100 et 500 ou entre 1000 et 5000')
La somme est comprise entre 100 et 500 ou entre 1000 et 5000

somme = 3000
if (somme >= 100 and somme <= 500) or (somme >= 1000 \
        and somme <= 5000):
```

```
    print('La somme est comprise entre 100 et 500 ou entre 1000 et 5000')
La somme est comprise entre 100 et 500 ou entre 1000 et 5000
```

Voir `Somme.py`.

4. Affronter des ninjas

Cet exercice contient un piège ! Es-tu tombé dedans ? Si tu relis l'énoncé de l'exercice et écris l'instruction `if` dans l'ordre indiqué, tu risques de ne pas obtenir les résultats auxquels tu t'attends. Voyons cela :

```
>>> ninjas = 5
>>> if ninjas < 50:
        print("Il y en a trop")
    elif ninjas < 30:
        print("Je vais devoir me battre mais je peux les avoir ")
    elif ninjas < 10:
        print("Je peux affronter ces ninjas !")
Il y en a trop
```

Même si le nombre de ninjas est plus petit que `10`, tu reçois toujours le message « Il y en a trop ». Ceci est dû au fait que la première condition, `< 50`, est évaluée en premier lieu. Autrement dit, Python la vérifie d'abord, avant les autres. Or, comme la valeur de la variable est bel et bien plus petite que `50`, le programme affiche le message auquel tu ne t'attendais pas.

Pour que les vérifications se fassent dans le bon ordre, il faut inverser l'ordre du code précédent, pour vérifier d'abord que la variable est plus petite que `10`, puis que `30`, puis que `50` :

```
>>> ninjas = 5
>>> if ninjas < 10:
        print("Je peux affronter ces ninjas !")
    elif ninjas < 30:
        print("Je vais devoir me battre mais je peux les avoir ")
    elif ninjas < 50:
        print("Il y en a trop")
Je peux affronter ces ninjas !
```

Chapitre 6

1. La boucle Bonjour

L'instruction `print` de cette boucle `for` ne s'exécute qu'une seule fois, tout simplement parce que, lorsque Python atteint l'instruction `if`, `x` est plus petit que 9 donc l'instruction `break` quitte de suite la boucle.

```
>>> for x in range(0, 20):
        print('Bonjour %s' % x)
        if x < 9:
            break
Bonjour 0
```

Voir `Boucle_Bonjour.py`.

2. Nombres pairs

Le paramètre pas de la fonction `range` permet de produire la liste des nombres pairs. Si tu as 14 ans, par exemple, le paramètre de départ sera de `2` et le paramètre de fin sera de `16` car la boucle `for` ne fonctionnera que jusqu'à la valeur juste avant le paramètre de fin.

```
>>> for x in range(2, 16, 2):
        print(x)
2
4
6
8
10
12
14
```

3. Mes cinq ingrédients préférés

Plusieurs approches permettent d'afficher des nombres en face des éléments d'une liste. En voici une :

```
>>> ingredients = ['escargots', 'sangsues', 'tranche de gorille', ↵
          'sourcils de chenilles', 'orteils de mille-pattes']
❶ >>> x = 1
❷ >>> for i in ingredients:
❸         print('%s %s' % (x, i))
❹       x = x + 1
```

```
1 escargots
2 sangsues
3 tranche de gorille
4 sourcils de chenilles
5 orteils de mille-pattes
```

Nous créons en ❶ une variable `x` pour mémoriser le numéro que nous voulons afficher. Ensuite, nous créons en ❷ une boucle `for` pour boucler parmi les éléments de la liste et les affecter à la variable `i`. En ❸, nous affichons les deux variables à l'aide des espaces réservés `%s`. Nous ajoutons `1` à la variable `x` en ❹, pour qu'à chaque tour de boucle, le numéro affiché augmente.

4. Ton poids sur la lune

Pour calculer ton poids en kilogrammes sur la lune sur 15 ans, crée d'abord une variable pour mémoriser ton poids de départ :

```
>>> poids = 30
```

Pour chaque année, tu peux calculer ton nouveau poids en ajoutant un kilo, puis multiplier le tout par 16,5 % (`0.165`), pour connaître le poids équivalent sur la lune :

```
>>> poids = 30
>>> for annee in range(1, 16):
        poids = poids + 1
        poids_sur_lune = poids * 0.165
        print("Poids de l'année %s vaut %s" % (annee, poids_sur_lune))
Poids de l'année 1 vaut 5.115
Poids de l'année 2 vaut 5.28
Poids de l'année 3 vaut 5.445
Poids de l'année 4 vaut 5.61
Poids de l'année 5 vaut 5.775
Poids de l'année 6 vaut 5.94
Poids de l'année 7 vaut 6.105
Poids de l'année 8 vaut 6.2700000000000005
Poids de l'année 9 vaut 6.4350000000000005
Poids de l'année 10 vaut 6.6000000000000005
Poids de l'année 11 vaut 6.765000000000001
Poids de l'année 12 vaut 6.930000000000001
Poids de l'année 13 vaut 7.095000000000001
Poids de l'année 14 vaut 7.260000000000001
Poids de l'année 15 vaut 7.425000000000001
```

Voir `Poids_sur_lune.py`.

Chapitre 7

1. Fonction de base du poids sur la lune

La fonction doit attendre deux paramètres, le `poids` et l'`augmentation`, le poids supplémentaire chaque année. Le reste du code ressemble beaucoup à la solution du puzzle 4 du chapitre précédent.

```
>>> def poids_lune(poids, augmentation):
>>>     for annee in range(1, 16):
            poids = poids + augmentation
            poids_sur_lune = poids * 0.165
            print("Poids de l'année %s vaut %s" % (annee, \
                poids_sur_lune))
>>> poids_lune(40, 0.5)
Poids de l'année 1 vaut 6.6825
Poids de l'année 2 vaut 6.765000000000001
Poids de l'année 3 vaut 6.8475
Poids de l'année 4 vaut 6.930000000000001
Poids de l'année 5 vaut 7.0125
Poids de l'année 6 vaut 7.095000000000001
Poids de l'année 7 vaut 7.1775
Poids de l'année 8 vaut 7.260000000000001
Poids de l'année 9 vaut 7.3425
Poids de l'année 10 vaut 7.425000000000001
Poids de l'année 11 vaut 7.5075
Poids de l'année 12 vaut 7.590000000000001
Poids de l'année 13 vaut 7.6725
Poids de l'année 14 vaut 7.755000000000001
Poids de l'année 15 vaut 7.8375
```

Voir `Fonc_Poids_sur_lune.py`.

2. Fonction poids sur la lune avec les années

Il ne faut qu'une modification mineure à notre fonction pour tenir compte du nombre d'années en paramètre.

```
>>> def poids_lune(poids, augmentation, annees):
>>>     annees = annees + 1
>>>     for annee in range(1, annees):
            poids = poids + augmentation
            poids_sur_lune = poids * 0.165
            print("Poids de l'année %s vaut %s" % (annee, \
                poids_sur_lune))
```

```
>>> poids_lune(40, 0.5)
Poids de l'année 1 vaut 5.8245
Poids de l'année 2 vaut 5.874
Poids de l'année 3 vaut 5.923499999999999
Poids de l'année 4 vaut 5.972999999999998
Poids de l'année 5 vaut 6.022499999999998
```

Voir Fonc_Poids_sur_lune_pour_annees.py.

Remarque bien la deuxième ligne de la fonction, qui ajoute 1 au nombre d'années, pour que la boucle for se termine à l'année correcte et non un an trop tôt.

3. Programme de poids sur la lune

Pour que l'utilisateur puisse entrer les valeurs qu'il souhaite, nous pouvons exploiter l'objet stdin du module sys, avec sa fonction readline. Comme sys.stdin.readline renvoie une chaîne, nous devons convertir ces chaînes en nombres utilisables dans les calculs.

```
import sys
def poids_lune():
    print('Entre ton poids actuel sur la terre :')
❶   poids = float(sys.stdin.readline())
    print('Entre l\'augmentation de poids annuelle :')
❷   augmentation = float(sys.stdin.readline())
    print('Entre le nombre d\'années :')
❸   annees = int(sys.stdin.readline())
    annees = annees + 1
    for annee in range(1, annees):
        poids = poids + augmentation
        poids_lune = poids * 0.165
        print('Poids de l\'année %s vaut %s' % (annee, poids_lune))
```

Voir Fonc_Poids_lune_entrees.py.

La ligne ❶ lit l'entrée effectuée dans sys.stdin.readline puis convertit le résultat en un nombre en virgule flottante, à l'aide de la fonction float. Cette valeur est stockée dans la variable poids. Le même procédé est utilisé en ❷ pour obtenir la variable augmentation. En ❸, nous convertissons en revanche le nombre d'années en un nombre entier (sans partie décimale). Le reste du code est exactement le même à partir de là que dans la solution précédente.

Lorsque nous appelons la fonction, nous obtenons quelque chose du genre de ce qui suit :

```
>>> poids_lune()
Entre ton poids actuel sur la terre :
45
Entre l'augmentation de poids annuelle :
.4
Entre le nombre d'années :
12
Poids de l'année 1 vaut 7.4910000000000005
Poids de l'année 2 vaut 7.5569999999999995
Poids de l'année 3 vaut 7.622999999999999
Poids de l'année 4 vaut 7.688999999999999
Poids de l'année 5 vaut 7.754999999999999
Poids de l'année 6 vaut 7.820999999999999
Poids de l'année 7 vaut 7.886999999999999
Poids de l'année 8 vaut 7.9529999999999985
Poids de l'année 9 vaut 8.018999999999998
Poids de l'année 10 vaut 8.084999999999997
Poids de l'année 11 vaut 8.150999999999998
Poids de l'année 12 vaut 8.216999999999997
```

Chapitre 8

1. Moulinet de girafe

Avant d'ajouter les fonctions pour permettre à Régine d'exécuter ces pas de danse, regardons d'un œil plus attentif les classes `Animaux`, `Mammiferes` et `Girafes`. Voici la classe `Animaux`, sous-classe de la classe `Animes`, que nous avons ignorée ici pour simplifier :

```
class Animaux():
    def respirer(self):
        print("respire")
    def bouger(self):
        print("bouge")
    def manger(self):
        print("mange")
```

La classe `Mammiferes` est une sous-classe de `Animaux` :

```
class Mammiferes(Animaux):
    def nourrir_petits_avec_du_lait(self):
        print("nourrit les petits")
```

Et la classe `Girafes` est une sous-classe de `Mammiferes` :

```
class Girafes(Mammiferes):
    def manger_feuilles_des_arbres(self):
        print("mange des feuilles")
```

Il est ensuite facile d'ajouter les fonctions pour déplacer chaque patte :

```
class Girafes(Mammiferes):
    def manger_feuilles_des_arbres(self):
        print("mange des feuilles")
    def pied_gauche_en_avant(self):
        print("pied gauche en avant")
    def pied_droit_en_avant(self):
        print("pied droit en avant")
    def pied_gauche_en_arriere(self):
        print("pied gauche en arrière")
    def pied_droit_en_arriere(self):
        print("pied droit en arrière")
```

La fonction `danser` n'a plus qu'à appeler chacune de ces fonctions de déplacement des pattes dans l'ordre adéquat :

```
    def danser(self):
        self.pied_gauche_en_avant()
        self.pied_gauche_en_arrière()
        self.pied_droit_en_avant()
        self.pied_droit_en_arrière()
        self.pied_gauche_en_arrière()
        self.pied_droit_en_arrière()
        self.pied_droit_en_avant()
        self.pied_gauche_en_avant()
```

Enfin, pour que Régine danse, nous créons un objet `regine` et nous appelons cette dernière fonction :

```
regine = Girafes()
regine.danser()
pied gauche en avant
pied gauche en arrière
pied droit en avant
pied droit en arrière
pied gauche en arrière
pied droit en arrière
pied droit en avant
pied gauche en avant
```

Voir `CH08_1_Moulinet.py`.

2. Fourche de tortues

Comme `Pen` est une classe du module `turtle`, nous pouvons créer plusieurs objets à partir de cette classe pour obtenir les pointes de la fourche. Nous affectons chaque objet à une variable différente pour les contrôler tous indépendamment. Ainsi, il devient simple de reproduire les lignes fléchées de l'exemple. L'idée que chaque **objet** est indépendant est fondamentale en programmation, en particulier lorsqu'on parle de classes et d'objets.

```
import turtle
t1 = turtle.Pen()
t2 = turtle.Pen()
t3 = turtle.Pen()
t4 = turtle.Pen()
t1.forward(100)
t1.left(90)
t1.forward(50)
t1.right(90)
t1.forward(50)
t2.forward(110)
t2.left(90)
t2.forward(25)
t2.right(90)
t2.forward(25)
t3.forward(110)
t3.right(90)
t3.forward(25)
t3.left(90)
t3.forward(25)
t4.forward(100)
t4.right(90)
t4.forward(50)
t4.left(90)
t4.forward(50)
```

Voir `CH08_2_Fourche.py`.

Il y a bien entendu plusieurs façons de dessiner la même chose donc ton code peut ne pas être absolument identique à celui-ci.

Chapitre 9

1. Code mystère

La fonction `abs` renvoie la valeur absolue d'un nombre, ce qui signifie qu'un nombre négatif devient positif. Donc, dans le code mystère, la première instruction `print` affiche `20`, tandis que la seconde affiche `0`.

```
❶ >>> a = abs(10) + abs(-10)
  >>> print(a)
  20

❷ >>> b = abs(-10) + -10
  >>> print(b)
  0
```

En ❶, le calcul équivaut à 10 + 10. En ❷, le calcul équivaut à 10 – 10.

2. Message caché

L'astuce, ici, consiste à créer une chaîne avec le message, puis à utiliser la fonction `dir` pour voir les fonctions disponibles pour une chaîne :

```
>>> message = "ceci si n'est tu pas peux une lire très ceci bonne ↵
alors manière c'est de que cacher tu un t'es message trompé"
>>> print(dir(message))
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
'__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__
ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__',
'__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith',
'expandtabs', 'find', 'format', 'format_map', 'index', 'isalnum',
'isalpha', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join',
'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate',
'upper', 'zfill']
```

En regardant cette liste et en lisant bien le conseil que nous avons fourni dans l'exercice, la fonction `split` semble utile. Voyons avec `help` ce que fait la fonction `split` :

```
>>> help(message.split)
Help on built-in function split:

split(...) method of builtins.str instance
    S.split(sep=None, maxsplit=-1) -> list of strings

    Return a list of the words in S, using sep as the
    delimiter string.  If maxsplit is given, at most maxsplit
    splits are done. If sep is not specified or is None, any
    whitespace string is a separator and empty strings are
    removed from the result.
```

Selon la description en anglais, nous découvrons que `split` renvoie une liste avec les mots, découpés en fonction du caractère `sep`, quel qu'il soit. Or, si on ne précise pas le caractère de séparation pour `sep`, la fonction utilise l'espace, par défaut. Donc cette fonction est capable de découper notre chaîne en une liste de mots.

Maintenant que nous savons quelle fonction utiliser, il nous reste à boucler parmi les mots de la liste obtenue. Ensuite, plusieurs manières permettent de n'afficher qu'un mot sur deux, en commençant par le premier. Voici une des possibilités :

```
❶ >>> message = "ceci si n'est tu pas peux une lire très ceci bonne ↵
  alors manière c'est de que cacher tu un t'es message trompé"
❷ >>> mots = message.split()
❸ >>> for x in range(0, len(mots), 2):
❹         print(mots[x])
```

La ligne ❶ crée la chaîne. La ligne ❷ utilise la fonction `split` pour découper la chaîne en une liste de mots séparés. La boucle `for` de la ligne ❸ utilise la fonction `range` pour extraire les mots. Le premier paramètre de `range` vaut `0`, soit le début de la liste ; le deuxième appelle la fonction `len` pour connaître la longueur de la liste et donner la fin de la plage ; le troisième paramètre impose un pas de ❷, donc la plage des nombres sera de 0, 2, 4, 6, 8 et ainsi de suite. Nous utilisons la variable `x` dans la boucle pour afficher les valeurs dans la liste en ligne ❹.

Voir `CH09_2_Message.py`.

3. Copier un fichier

Pour copier un fichier, nous l'ouvrons, nous en lisons le contenu que nous affectons à une variable. Nous ouvrons ensuite le fichier de destination en écriture, avec le paramètre `'w'` qui signifie écriture, puis nous y écrivons le contenu de la variable. Le code final est le suivant :

```
f = open('test.txt')
s = f.read()
f.close()
f = open('sortie.txt', 'w')
f.write(s)
f.close()
```

Voir CH09_3_Copier_fichier.py.

Si cet exemple fonctionne très bien, il existe toutefois une manière plus efficace de copier un fichier, à l'aide du module shutil de Python :

```
import shutil
shutil.copy('test.txt', 'sortie.txt')
```

Chapitre 10

1. Copier des voitures

Ce programme contient deux instructions print et nous devons comprendre ce qu'elles affichent.

Voici la première :

```
>>> voiture1 = Voiture()
❶ >>> voiture1.roues = 4
❷ >>> voiture2 = voiture1
>>> voiture2.roues = 3
>>> print(voiture1.roues)
3
```

Pourquoi le résultat de l'instruction print est-il de 3, alors que nous avons clairement défini à 4 la valeur de roues de la voiture1 en ❶ ? Ceci, parce qu'en ❷, les deux variables voiture1 et voiture2 pointent vers le même objet.

Voyons ensuite la deuxième instruction print :

```
>>> voiture3 = copy.copy(voiture1)
>>> voiture3.roues = 6
>>> print(voiture1.roues)
3
```

Dans ce cas-ci, voiture3 est une copie de l'objet. Cette variable n'étiquette pas le même objet que voiture1 et voiture2. Donc, quand

nous définissons le nombre de roues à 6, il n'a pas d'influence sur les roues de voiture1.

2. Objets favoris avec pickle

Nous utilisons le module pickle pour enregistrer le contenu d'une variable (ou de variables) dans un fichier :

```
❶ >>> import pickle
❷ >>> favoris = ['PlayStation', 'Caramels', 'Films', 'Python pour les ↵
        enfants']
❸ >>> f = open('favoris.dat', 'wb')
❹ >>> pickle.dump(favoris, f)
>>> f.close()
```

Voir CH10_2_enregistre_favoris.py.

La ligne ❶ importe le module pickle. La ligne ❷ crée la liste des favoris. La ligne ❸ ouvre avec open un fichier nommé favoris.dat, avec la chaîne 'wb' passée en second paramètre, pour indiquer que le fichier doit être ouvert en écriture (w comme *write*) et en mode binaire (b). La fonction dump de pickle permet en ❹ d'enregistrer le contenu de la variable favoris dans le fichier.

La seconde partie de la solution vise à relire le fichier. En supposant que tu aies fermé et rouvert le shell, il est nécessaire d'importer à nouveau le module pickle.

```
>>> import pickle
>>> f = open('favoris.dat', 'rb')
>>> favoris = pickle.load(f)
>>> print(favoris)
['PlayStation', 'Caramels', 'Films', 'Python pour les enfants']
```

Voir CH10_2_charge_favoris.py.

Ce code ressemble fort au précédent, sauf que nous ouvrons le fichier avec le deuxième paramètre 'rb' (r pour *read*, lecture, et b pour binaire), puis que nous utilisons la fonction load du module pickle.

Chapitre 11

1. Dessiner un octogone

Un octogone possède huit côtés donc nous aurons besoin d'au moins une boucle for pour tracer la forme. Réfléchissons un moment à la

direction de la tortue qui, pour tracer un octogone, doit tourner complètement sur elle-même, comme l'aiguille d'une montre, pendant qu'elle dessine et termine son tracé. Cela signifie qu'elle doit tourner de 360 degrés sur elle-même. Si nous divisons 360 par le nombre d'angles de l'octogone, nous obtenons le nombre de degrés de rotation à chaque angle, soit 45 degrés, comme l'indique le conseil.

```
>>> import turtle
>>> t = turtle.Pen()
>>> def octogone(taille):
        for x in range(1,9):
            t.forward(taille)
            t.right(45)
```

Il suffit ensuite d'appeler la fonction pour la tester avec `100` comme longueur d'un des côtés :

```
>>> octogone(100)
```

Voir `CH11_1_octogone.py`.

2. Dessiner un octogone plein

Si nous modifions la fonction pour simplement dessiner un octogone plein, nous aurons quelques difficultés à dessiner les traits des côtés. Une meilleure approche consiste à passer à la fonction un autre paramètre pour contrôler si l'octogone à tracer doit être plein ou pas.

```
>>> import turtle
>>> t = turtle.Pen()
>>> def octogone(taille, plein):
❶       if plein == True:
❷           t.begin_fill()
        for x in range(1,9):
            t.forward(taille)
            t.right(45)
❸       if plein == True:
❹           t.end_fill()
```

La ligne ❶ vérifie d'abord si le paramètre `plein` est vrai. S'il l'est, alors la ligne ❷ indique à la tortue de commencer à remplir, à l'aide de la fonction `begin_fill`. Nous dessinons ensuite l'octogone aux trois lignes suivantes, comme à l'exercice précédent. La ligne ❸ vérifie si le paramètre `plein` est vrai et, si c'est le cas, la ligne ❹ appelle la fonction d'arrêt du remplissage, `end_fill`, qui provoque le remplissage à ce moment précis.

Pour tester la fonction, nous imposons la couleur jaune et nous appelons d'abord la fonction avec le paramètre réglé à `True`, pour qu'elle effectue le remplissage. Ensuite, nous réglons la couleur à noir et le paramètre à `False`, pour ne plus remplir la figure.

```
>>> t.color(1, 0.85, 0)
>>> octogone(40, True)
>>> t.color(0, 0, 0)
>>> octogone(40, False)
```

Voir `CH11_2_octogone_plein.py`.

3. Autre fonction de dessin d'étoile

L'astuce pour cette fonction de dessin d'étoile consiste à diviser les 360 degrés du tour complet par le nombre de branches, ce qui donne l'angle intérieur de chaque branche de l'étoile (en ligne ❶ du programme suivant). Ensuite, pour trouver l'angle extérieur, dont la tortue doit tourner, il faut soustraire l'angle intérieur de 180 degrés, en ligne ❹.

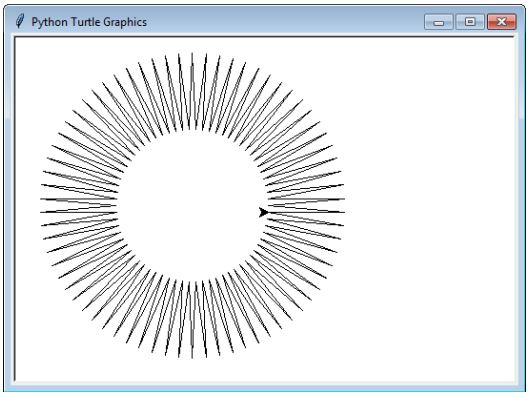
```
import turtle
t = turtle.Pen()
def dessine_etoile(taille, branches):
❶    angle = 360 / branches
❷    for x in range(0, branches):
❸        t.forward(taille)
❹        t.left(180 - angle)
❺        t.forward(taille)
❻        t.right(180-(angle * 2))
```

Nous bouclons de 0 jusqu'au nombre de branches en ❷, puis nous déplaçons la tortue vers l'avant du nombre de pixels indiqué dans le paramètre `taille` ❸. En ❹, nous faisons tourner la tortue du nombre de degrés que nous avons calculé précédemment, puis nous avançons à nouveau la souris en ❺, pour former une des « épines » de l'étoile. Pour pouvoir évoluer selon un modèle circulaire, nous devons juste multiplier l'angle calculé par deux en ❻ et tourner à droite.

Ainsi, appelle cette fonction avec 80 pixels et 70 branches :

```
>>> dessine_etoile(70, 70)
```

Et tu obtiens la figure suivante :



Voir CH11_3_etoile.py.

Chapitre 12

1. Remplir l'écran de triangles

Pour remplir l'écran de triangles, la première étape consiste à définir le canevas. Donnons-lui une largeur et une hauteur de 400 pixels.

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

Un triangle possède donc trois angles, ceci signifie trois ensembles de coordonnées *x* et *y*. Nous pouvons utiliser la fonction randrange du module random (comme pour l'exemple de rectangle aléatoire au chapitre 12), pour générer au hasard les coordonnées des trois points, donc six nombres au total. Nous utilisons ensuite la fonction create_polygon pour dessiner le triangle.

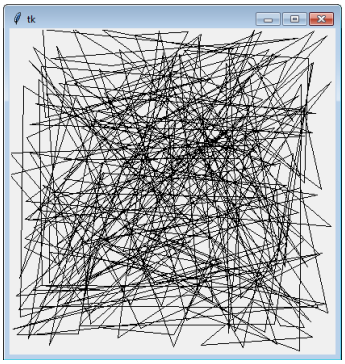
```
>>> def triangle_aleatoire():
        p1 = random.randrange(larg)
        p2 = random.randrange(haut)
        p3 = random.randrange(larg)
        p4 = random.randrange(haut)
        p5 = random.randrange(larg)
        p6 = random.randrange(haut)
```

```
        canvas.create_polygon(p1, p2, p3, p4, p5, p6, \
                fill="", outline="black")
```

Enfin, nous créons une boucle pour dessiner toute une série de triangles au hasard.

```
>>> for x in range(0, 100):
        triangle_aleatoire()
```

Le résultat varie, puisque ce sont des triangles aléatoires, mais il est du genre du suivant :



Voir CH12_1_Triangles_aleatoires.py.

Pour remplir la fenêtre avec des triangles de couleur, crée d'abord une liste de couleurs. Ajoute-la dans le code de début du programme.

```
>>> from tkinter import *
>>> import random
>>> larg = 400
>>> haut = 400
>>> tk = Tk()
>>> canvas = Canvas(tk, width=larg, height=haut)
>>> canvas.pack()
>>> couleurs = ['red','green','blue','yellow','orange','white','purple']
```

La fonction choice du module random permet ensuite d'extraire au hasard un élément de cette liste de couleurs et de l'utiliser dans l'appel à create_polygon de turtle :

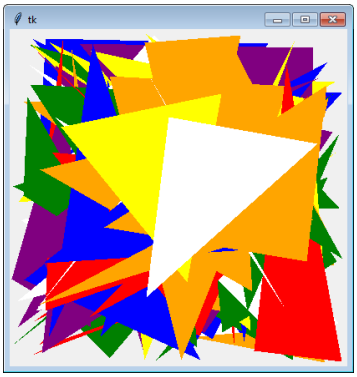
```
>>> def triangle_aleatoire():
        p1 = random.randrange(larg)
        p2 = random.randrange(haut)
```

```
    p3 = random.randrange(larg)
    p4 = random.randrange(haut)
    p5 = random.randrange(larg)
    p6 = random.randrange(haut)
    couleur = random.choice(couleurs)
    canvas.create_polygon(p1, p2, p3, p4, p5, p6, \
            fill=couleur, outline="")
```

Supposons que nous bouclions de nouveau 100 fois :

```
>>> for x in range(0, 100):
        triangle_aleatoire()
```

Voir CH12_1_Triangles_aleatoires_en_couleurs.py.
Le résultat est surprenant et plein de couleurs :



2. Le triangle mobile

Pour ce triangle en mouvement, nous définissons à nouveau, d'abord le canevas, puis nous dessinons le triangle à l'aide de la fonction create_polygon :

```
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)
```

Pour déplacer ensuite le triangle horizontalement à l'écran, la valeur x doit être un nombre positif et la valeur y doit être 0. Pour cela,

nous pouvons mettre en place une boucle `for` et utiliser l'identifiant `1` pour le triangle, `10` comme paramètre `x` et `0` comme paramètre `y`.

```
for x in range(0, 35):
    canvas.move(1, 10, 0)
    tk.update()
    time.sleep(0.05)
```

Le déplacement vers le bas de l'écran est comparable, mais avec une valeur `0` pour le paramètre `x` et une valeur positive pour le paramètre `y` :

```
for x in range(0, 14):
    canvas.move(1, 0, 10)
    tk.update()
    time.sleep(0.05)
```

Pour reculer vers la gauche de l'écran, il faut une valeur négative pour le paramètre `x` et encore `0` pour le paramètre `y`. Pour aller vers le haut, en toute logique, il faut un `x` égal à `0` et un `y` négatif.

```
for x in range(0, 35):
    canvas.move(1, -10, 0)
    tk.update()
    time.sleep(0.05)

for x in range(0, 14):
    canvas.move(1, 0, -10)
    tk.update()
    time.sleep(0.05)
```

Voir `CH12_2_Triangle_mobile.py`.

3. La photo mobile

Le code de la solution de la photo en mouvement dépend forcément de la taille de la photo mais, en admettant que l'image porte le nom de fichier `visage.gif`, et que tu l'as enregistrée sur ta clé USB (`F:`), tu peux l'afficher et la déplacer comme toute autre forme dessinée.

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

```
for x in range(0, 35):
    canvas.move(1, 10, 10)
    tk.update()
    time.sleep(0.05)
```

Ce code déplace l'image en diagonale vers le bas et la droite de l'écran.

Si tu utilises Linux (Ubuntu) ou OS X, le nom de fichier change. Si le fichier `visage.gif` est dans ton dossier personnel sous Linux, le chargement de l'image prendra l'allure suivante :

```
monimage = PhotoImage(file='/home/utilisateur/visage.gif')
```

Sur un Mac, le chargement de l'image prendra plutôt la forme suivante :

```
monimage = PhotoImage(file='/Users/utilisateur/visage.gif')
```

Chapitre 14

1. Retarder le début du jeu

Pour que le jeu ne démarre que lorsque l'utilisateur clique sur le canevas, nous devons apporter quelques petites modifications au programme. Le premier vise l'ajout d'une nouvelle fonction à la classe `Raquette` :

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

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

Cette fonction règle la variable d'objet `jeu_demarre` à `True` quand elle est appelée. Nous devons aussi ajouter la variable d'objet à la fonction `__init__` de la classe `Raquette` et la mettre à `False`, puis ajouter la liaison d'événement du bouton de la souris à la fonction `demarrer_jeu`.

```
    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()
❶       self.jeu_demarre = False
        self.canvas.bind_all('<KeyPress-Left>', self.vers_gauche)
        self.canvas.bind_all('<KeyPress-Right>', self.vers_droite)
❷       self.canvas.bind_all('<Button-1>', self.demarrer_jeu)
```

L'ajout de la variable d'objet apparaît en ❶ et de la liaison d'événement en ❷.

La dernière modification à apporter concerne la boucle principale au bas du programme. Nous devons vérifier que la variable d'objet `jeu_demarre` est à l'état `True` avant de dessiner la balle et la raquette, ce qui vient s'ajouter à l'instruction `if` initiale.

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

Voir `CH14_1_rebondir_clic.py`.

2. Un véritable « Partie terminée »

Nous pouvons utiliser la fonction `create_text` pour afficher le message « Partie terminée ». Nous l'ajoutons juste après le code de création de la balle et de la raquette.

```
raquette = Raquette(canvas, 'blue')
balle = Balle(canvas, raquette, 'red')
texte_fin_partie = canvas.create_text(250, 200, text='PARTIE TERMINEE', \
        state='hidden')
```

La fonction `create_text` possède un argument appelé `state`, qui indique l'état masqué (`hidden`) ou visible (`normal`) du texte à l'écran. Au début, nous réglons `state` à la chaîne `'hidden'` pour masquer le texte, ce qui signifie que Python dessine le texte mais le rend invisible. Ensuite, pour afficher le texte lorsque la partie est terminée, nous ajoutons une instruction `if` à la boucle principale, à la fin du programme :

```
while 1:
    if balle.touche_bas == False and raquette.jeu_demarre == True:
        balle.dessiner()
        raquette.dessiner()
❶   if balle.touche_bas == True:
❷       time.sleep(1)
❸       canvas.itemconfig(texte_fin_partie, state='normal')
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

La ligne ❶ vérifie que la balle a touché le bas de l'écran, autrement dit, que la variable d'objet `touche_bas` est vraie. Si elle l'est, nous mettons le programme en veille pendant une seconde en ❷, pour laisser un petit délai avant l'affichage du texte, puis, en ❸, nous changeons l'état du texte, c'est-à-dire le paramètre `state` en `'normal'` au lieu de `'hidden'`, à l'aide de la fonction `itemconfig` du canevas. Nous passons deux paramètres à cette fonction : l'identifiant du texte dessiné dans le canevas, mémorisé dans `texte_fin_partie`, et le paramètre nommé `state`.

Voir `CH14_2_rebondir_clic_fin_partie.py`.

3. Accélérer la balle

Cette modification est assez simple, à condition que tu comprennes bien de quoi il s'agit et que tu saches où apporter les modifications nécessaires. Le but est d'accélérer la balle lorsqu'elle voyage dans la même direction horizontale que la raquette quand elle la touche, et de ralentir la balle quand elles se touchent dans des directions horizontales opposées. Pour parvenir à cela, la vitesse horizontale, gauche-droite, de la raquette doit venir s'ajouter à la vitesse horizontale de la balle.

L'endroit le plus simple pour apporter cette modification se situe dans la fonction `heurter_raquette` de la classe `Balle` :

```
    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]:
❶           if pos[3] >= pos_raquette[1] and pos[3] <= pos_raquette[3]:
❷               self.x += self.raquette.x
                return True
        return False
```

Voir `CH14_3_rebondir_clic_FP_acceleration.py`.

Dès que nous avons déterminé que la balle a atteint la raquette, en ❶, nous ajoutons en ❷ la valeur de la variable `x` de l'objet raquette à la variable `x` de la balle. Ainsi, si la raquette va vers la droite de l'écran (sa variable `x` est définie à 2, par exemple) et si la balle la touche, alors qu'elle va vers la droite avec une valeur de `x` égale à 3, alors la balle rebondit sur la raquette avec une nouvelle vitesse horizontale de 5. L'addition des deux variables signifie que la balle reçoit une nouvelle vitesse quand elle touche la raquette.

4. Enregistrer le score du joueur

Pour enregistrer les scores à ce jeu, nous créons une nouvelle classe nommée `Score` :

```
class Score:
    def __init__(self, canvas, couleur):
❶       self.score = 0
❷       self.canvas = canvas
❸       self.id = canvas.create_text(450, 10, text=self.score, \
                fill=couleur)
```

La fonction `__init__` de la classe `Score` attend trois paramètres : `self`, `canvas` et `couleur`. La ligne ❶ de cette fonction définit une variable d'objet `score`, avec la valeur `0`. La ligne ❷ mémorise aussi le paramètre `canvas` dans la variable d'objet `canvas` pour un usage ultérieur.

Le paramètre `canvas` nous permet, en ❸, de créer le texte du score de la partie, à l'emplacement `(450, 10)`, et de régler la valeur de remplissage de couleur à celle du paramètre `couleur`. Le texte à afficher est la valeur de la variable `score`, donc `0` à ce stade.

La classe `Score` nécessite une autre fonction pour augmenter ce score et en afficher la nouvelle valeur :

```
class Score:
    def __init__(self, canvas, couleur):
        self.score = 0
        self.canvas = canvas
        self.id = canvas.create_text(450, 10, text=self.score, \
                fill=couleur)

❶   def compter_coup(self):
❷       self.score += 1
❸       self.canvas.itemconfig(self.id, text=self.score)
```

La fonction `compter_coup` ne prend aucun paramètre en ❶, augmente le score de 1 en ❷, puis utilise en ❸ la fonction `itemconfig` du canevas pour changer le texte affiché avec la nouvelle valeur du score.

Nous pouvons ensuite créer un objet de la classe `Score` juste avant de créer les objets `raquette` et `balle`, et nous ajoutons à la création de l'objet `balle` le paramètre `score` (voir ci-après) :

```
score = Score(canvas, 'green')
raquette = Raquette(canvas, 'blue')
balle = Balle(canvas, raquette, score, 'red')
texte_fin_partie = canvas.create_text(250, 200, text='PARTIE TERMINEE', \
        state='hidden')
```

La dernière modification à appliquer se situe au niveau de la classe `Balle`. Nous devons stocker l'objet `score` dans un paramètre lors de la création de l'objet `balle`, pour pouvoir mettre à jour le compteur, puis enclencher la fonction `compter_coup` dans la fonction `heurter_raquette` de la balle.

Le début de la fonction `__init__` de la classe `Balle` reçoit donc le paramètre `score`, que nous utilisons pour créer une variable d'objet, également nommée `score`.

```
class Balle:
    def __init__(self, canvas, raquette, score, couleur):
        self.canvas = canvas
        self.raquette = raquette
        self.score = score
```

La fonction `heurter_raquette` prend l'allure suivante :

```
    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]:
                self.x += self.raquette.x
                self.score.compter_coup()
                return True
        return False
```

Lorsque les quatre puzzles sont réalisés, le code complet de notre jeu devient le suivant :

```
from tkinter import *
import random
import time
```

```
class Balle:
    def __init__(self, canvas, raquette, score, couleur):
        self.canvas = canvas
        self.raquette = raquette
        self.score = score
        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()
        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]:
                self.x += self.raquette.x
                self.score.compter_coup()
                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.jeu_demarre = False
        self.canvas.bind_all('<KeyPress-Left>', self.vers_gauche)
        self.canvas.bind_all('<KeyPress-Right>', self.vers_droite)
        self.canvas.bind_all('<Button-1>', self.demarrer_jeu)
```

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

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

    def demarrer_jeu(self, evt):
        self.jeu_demarre = True

    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

class Score:
    def __init__(self, canvas, couleur):
        self.score = 0
        self.canvas = canvas
        self.id = canvas.create_text(450, 10, text=self.score, \
                fill=couleur)

    def compter_coup(self):
        self.score += 1
        self.canvas.itemconfig(self.id, text=self.score)

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()

score = Score(canvas, 'green')
raquette = Raquette(canvas, 'blue')
balle = Balle(canvas, raquette, score, 'red')
texte_fin_partie = canvas.create_text(250, 200, text='PARTIE TERMINEE', \
        state='hidden')
while 1:
    if balle.touche_bas == False and raquette.jeu_demarre == True:
        balle.dessiner()
        raquette.dessiner()
    if balle.touche_bas == True:
        time.sleep(1)
        canvas.itemconfig(texte_fin_partie, state='normal')
```

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

Voir CH14_4_rebondir_clic_FP_acceleration_score.py.

Chapitre 16

1. Damier

Pour afficher un damier comme image de fond, nous devons apporter une modification aux boucles de la fonction __init__ du jeu, comme suit :

```
        self.ap = PhotoImage(file="arriere-plan.gif")
        larg = self.ap.width()
        haut = self.ap.height()
❶       dessiner_ap = 0
        for x in range(0, 5):
            for y in range(0, 5):
❷               if dessiner_ap == 1:
❸                   self.canvas.create_image(x * larg, y * haut, \
                            image=self.ap, anchor='nw')
❹                   dessiner_ap = 0
❺               else:
❻                   dessiner_ap = 1
```

Voir CH16_1_damier.py.

La ligne ❶ crée une variable dessiner_ap (dessiner l'arrière-plan) et lui donne la valeur 0. La ligne ❷ vérifie que cette variable vaut 1 et, si c'est le cas, dessine en ❸ l'image, pour ramener la variable dessiner_ap à 0 en ❹. En ❺, sinon, nous donnons la valeur 1 à la variable en ❻.

Qu'apportent ces modifications au programme ? La première fois que nous atteignons l'instruction if, nous n'affichons pas l'image de l'arrière-plan mais nous réglons dessiner_ap à 1. La fois suivante, nous affichons l'image et nous réglons la variable à 0. Donc, chaque fois que nous passons par les boucles, nous faisons basculer la valeur de la variable. Une fois, nous affichons l'image, la fois suivante, non. D'où l'effet de damier.

2. Damier à deux images alternées

Dès que tu as bien visualisé le principe du damier, le dessin de deux images alternées est très simple à réaliser. Il faut toutefois charger une nouvelle image en plus de la première. Dans l'exemple suivant, nous chargeons la nouvelle image `arriere-plan2.gif` (que tu dois d'abord dessiner avec GIMP) et nous l'affectons à la variable d'objet `ap2`.

```
        self.ap = PhotoImage(file="arriere-plan.gif")
        self.ap2 = PhotoImage(file="arriere-plan2.gif")
        larg = self.ap.width()
        haut = self.ap.height()
        dessiner_ap = 0
        for x in range(0, 5):
            for y in range(0, 5):
                if dessiner_ap == 1:
                    self.canvas.create_image(x * larg, y * haut, \
                            image=self.ap, anchor='nw')
                    dessiner_ap = 0
                else:
                    self.canvas.create_image(x * larg, y * haut, \
                            image=self.ap2, anchor='nw')
                    dessiner_ap = 1
```

Voir `CH16_2_damier2.py`.

Dans la seconde partie de l'instruction `if` que nous avions créée dans le programme de l'exercice précédent, nous utilisons la fonction `create_image` du canevas pour dessiner, cette fois, la nouvelle image à l'écran.

3. Étagère et lampe

Pour dessiner ces arrière-plans différents, nous partons du programme du damier à images alternées et nous le modifions pour lui ajouter deux images et les déposer par-ci, par-là dans le canevas. Pour débuter, récupère l'image `arriere-plan2.gif` dans GIMP, ajoute-lui le dessin d'une étagère et enregistre-la sous le nom de fichier `etagere.gif`. Ensuite, reprends une nouvelle copie de l'image `arriere-plan2.gif` dans GIMP, ajoute-lui le dessin d'une lampe et enregistre-la sous le nom de fichier `lampe.gif`.

```
        self.ap = PhotoImage(file="arriere-plan.gif")
        self.ap2 = PhotoImage(file="arriere-plan2.gif")
❶       self.ap_etagere = PhotoImage(file="etagere.gif")
❷       self.ap_lampe = PhotoImage(file="lampe.gif")
```

```
        larg = self.ap.width()
        haut = self.ap.height()
❸       compteur = 0
        dessiner_ap = 0
        for x in range(0, 5):
            for y in range(0, 5):
                if dessiner_ap == 1:
                    self.canvas.create_image(x * larg, y * haut, \
                            image=self.ap, anchor='nw')
                    dessiner_ap = 0
                else:
❹                   compteur = compteur + 1
❺                   if compteur == 5:
❻                       self.canvas.create_image(x * larg, y * haut, \
                                image=self.ap_etagere, anchor='nw')
❼                   elif compteur == 9:
❽                       self.canvas.create_image(x * larg, y * haut, \
                                image=self.ap_lampe, anchor='nw')
                    else:
                        self.canvas.create_image(x * larg, y * haut, \
                                image=self.ap2, anchor='nw')
                    dessiner_ap = 1
```

Voir CH16_3_etagere_lampe.py.

Les lignes ❶ et ❷ chargent les nouvelles images, pour les affecter aux variables ap_etagere et ap_lampe, respectivement. La ligne ❸ crée une nouvelle variable compteur. Dans la solution précédente, nous avions une instruction if qui permettait de dessiner une image, puis une autre, en fonction de la valeur de la variable dessiner_ap. Nous partons du même principe ici, sauf qu'au lieu de simplement afficher l'image alternative, nous incrémentons la valeur de la variable compteur (c'est-à-dire que nous lui ajoutons 1 à l'aide de compteur = compteur + 1) en ❹. Selon la valeur du compteur, nous décidons ensuite de l'image à dessiner. En ❺, si sa valeur atteint 5, nous dessinons en ❻ l'image de l'étagère. En ❼, si la valeur atteint 9, nous dessinons l'image de la lampe, en ❽. Dans les autres cas, nous dessinons simplement l'arrière-plan alternatif comme précédemment.

Chapitre 18

1. Tu as gagné !

L'ajout du texte « Tu as gagné ! » se fait dans une variable de la classe Jeu, au niveau de sa fonction __init__ :

```
        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
        self.texte_fin_partie = self.canvas.create_text(250, 250, \
                text='TU AS GAGNE !', state='hidden')
```

Pour afficher le texte quand la partie est terminée, il nous suffit d'ajouter une instruction `else` à la fonction `boucle_principale` :

```
    def boucle_principale(self):
        while 1:
            if self.enfonction == True:
                for lutin in self.lutins:
                    lutin.deplacer()
❶           else:
❷               time.sleep(1)
❸               self.canvas.itemconfig(self.texte_fin_partie, \
                        state='normal')
            self.tk.update_idletasks()
            self.tk.update()
            time.sleep(0.01)
```

La modification apparaît aux lignes ❶ à ❸. La ligne ❶ ajoute une clause `else` à l'instruction `if`, puis Python exécute ce bloc de code si la variable `enfonction` n'est plus vraie. En ❷, nous mettons le programme en sommeil pendant une seconde, pour que le texte ne s'affiche pas trop vite. La ligne ❸ change l'état (`state`) du texte en `'normal'`, ce qui provoque l'affichage du texte sur le canevas.

Voir `CH18_1_Jeufiliforme_gagne.py`.

2. Animer la porte

Pour animer la porte et faire en sorte qu'elle s'ouvre et se referme lorsque le personnage l'atteint, nous devons modifier d'abord la classe `LutinPorte`. Au lieu de lui passer l'image en paramètre, le lutin doit lui-même charger ses deux images, au niveau de sa fonction `__init__` :

```
class LutinPorte(Lutin):
    def __init__(self, jeu, x, y, largeur, hauteur):
        Lutin.__init__(self, jeu)
❶       self.porte_fermee = PhotoImage(file="porte1.gif")
❷       self.porte_ouverte = PhotoImage(file="porte2.gif")
        self.image = jeu.canvas.create_image(x, y, \
                image=self.porte_fermee, anchor='nw')
```

```
        self.coordonnees = Coords(x, y, x + (largeur / 2), y + hauteur)
        self.finjeu = True
```

Ainsi, la définition de la fonction ne contient plus le paramètre `image_photo`, les deux images sont chargées dans deux variables d'objet aux lignes ❶ et ❷. L'image en cours d'affichage est celle de la porte fermée. Nous devons aussi modifier le code de création de l'objet `porte` à la fin du programme, pour enlever le paramètre d'image :

```
porte = LutinPorte(jeu, 45, 30, 40, 35)
```

La classe `LutinPorte` nécessite deux nouvelles fonctions : l'une pour afficher l'image de la porte ouverte et l'autre pour afficher la porte fermée.

```
    def ouvrir_porte(self):
❶       self.jeu.canvas.itemconfig(self.image, \
❷               image=self.porte_ouverte)
        self.jeu.tk.update_idletasks()

    def fermer_porte(self):
❸       self.jeu.canvas.itemconfig(self.image, \
                image=self.porte_fermee)
        self.jeu.tk.update_idletasks()
```

La ligne ❶ utilise la fonction `itemconfig` du canevas pour changer l'image actuelle en celle stockée dans la variable d'objet `porte_ouverte`. La ligne ❷ appelle la fonction `update_idletasks` de l'objet `tk` pour forcer l'affichage de la nouvelle image. Cet appel est indispensable, sinon l'image ne changerait pas immédiatement à l'affichage. La fonction `fermer_porte` est fort semblable et ne diffère que par l'affichage de l'image stockée dans la variable `porte_fermee` en ❸.

Ensuite, une nouvelle fonction vient s'ajouter à la classe `LutinPersonnage` :

```
    def fin(self, lutin):
❶       self.jeu.enfonction = False
❷       lutin.ouvrir_porte()
❸       time.sleep(1)
❹       self.jeu.canvas.itemconfig(self.image, state='hidden')
❺       lutin.fermer_porte()
```

La ligne ❶ force à faux la variable d'objet `enfonction` du jeu et la ligne ❷ appelle la fonction `ouvrir_porte` du paramètre `lutin`. Ce dernier est en fait l'objet de classe `LutinPorte`, comme nous le verrons plus loin. La ligne ❸ met le programme en sommeil pendant une

seconde, puis la ligne ❹ masque le personnage et, enfin, la ligne ❺ appelle la fonction `fermer_porte`, pour donner l'illusion que le personnage est sorti et qu'il a fermé la porte derrière lui.

La dernière modification à apporter se situe au niveau de la fonction `deplacer` de `LutinPersonnage`. Dans la version de base du code, lorsque le personnage entrait en collision avec la porte, nous réglions la variable `enfonction` à faux, mais comme ceci est déplacé vers la fonction `fin`, nous devons appeler cette fonction à la place :

```
            if gauche and self.x < 0 and collision_gauche(co, co_lutin):
                self.x = 0
                gauche = False
❶               if lutin.finjeu:
❷                   self.fin(lutin)
            if droite and self.x > 0 and collision_droite(co, co_lutin):
                self.x = 0
                droite = False
❸               if lutin.finjeu:
❹                   self.fin(lutin)
```

Dans la section de code où nous vérifions si le personnage se déplace à gauche et s'il a touché un lutin à sa gauche, nous vérifions en ❶ si sa variable `finjeu` est vraie. Si c'est le cas, cela signifie que le lutin est de la classe `LutinPorte`, donc, en ❷, nous appelons la fonction `fin` en lui passant la variable `lutin` en paramètre. La même modification intervient dans la section de code où nous vérifions si le personnage de déplace à droite et entre en collision sur sa droite avec un lutin (en ❸ et ❹).

Voir `CH18_2_Jeufiliforme_gagne_porte.py`.

3. Plates-formes mobiles

Une plate-forme mobile se rapproche plus de la classe du personnage car nous devons en recalculer la position au lieu de nous contenter de coordonnées fixes. Nous allons créer une sous-classe de la classe `LutinPlateForme`, dont la fonction `__init__` devient la suivante :

```
    class LutinPlateFormeMobile(LutinPlateForme):
❶       def __init__(self, jeu, image_photo, x, y, largeur, hauteur):
❷           LutinPlateForme.__init__(self, jeu, image_photo, x, y, \
                    largeur, hauteur)
❸           self.x = 2
❹           self.compteur = 0
❺           self.derniere_heure = time.time()
❻           self.largeur = largeur
❼           self.hauteur = hauteur
```

La ligne ❶ capture exactement les mêmes paramètres que la classe `LutinPlateForme`. La ligne ❷ appelle la fonction `__init__` de la classe parente avec les mêmes paramètres. Ceci signifie que tout objet de la classe `LutinPlateFormeMobile` reçoit exactement les mêmes définitions qu'un objet de la classe `LutinPlateForme`. La ligne ❸ crée ensuite une variable `x` de valeur `2` (donc la plate-forme commence par aller à droite) et la ligne ❹ définit un compteur qui permettra d'indiquer quand la plate-forme doit changer de sens. Comme il vaut mieux éviter que la plate-forme bouge trop vite horizontalement, nous mémorisons en ❺ l'heure dans la variable `derniere_heure`. Cette variable servira à ralentir le mouvement de la plate-forme. Enfin, les lignes ❻ et ❼ stockent la largeur et la hauteur.

L'ajout suivant à la nouvelle classe concerne la fonction `coords` :

```
        self.derniere_heure = time.time()
        self.largeur = largeur
        self.hauteur = hauteur

    def coords(self):
        xy = self.jeu.canvas.coords(self.image)
        self.coordonnees.x1 = xy[0]
        self.coordonnees.y1 = xy[1]
❶       self.coordonnees.x2 = xy[0] + self.largeur
❷       self.coordonnees.y2 = xy[1] + self.hauteur
        return self.coordonnees
```

La fonction `coords` est quasi identique à celle de `LutinPersonnage`. La seule différence est qu'au lieu d'utiliser une largeur et une hauteur fixes, nous tenons compte de celles définies dans la fonction `__init__`, aux lignes ❶ et ❷.

Comme il s'agit d'un lutin mobile, nous avons également besoin d'une fonction `deplacer` :

```
        self.coordonnees.x2 = xy[0] + self.largeur
        self.coordonnees.y2 = xy[1] + self.hauteur
        return self.coordonnees

    def deplacer(self):
❶       if time.time() - self.derniere_heure > 0.03:
❷           self.derniere_heure = time.time()
❸           self.jeu.canvas.move(self.image, self.x, 0)
❹           self.compteur = self.compteur + 1
❺           if self.compteur > 20:
❻               self.x = self.x * -1
❼               self.compteur = 0
```

La fonction `deplacer` vérifie si l'heure écoulée dépasse trois centièmes de seconde à la ligne ❶. Si c'est le cas, la variable `derniere_heure` est réglée à l'heure actuelle en ❷. La ligne ❸ déplace l'image de la plate-forme et la ligne ❹ incrémente de `1` (augmente de 1) la variable `compteur`. La ligne ❺ vérifie si le compteur dépasse `20` et, si c'est le cas, la ligne ❻ inverse la direction du mouvement en multipliant la variable `x` par `-1` (de sorte que, si elle est positive, elle devient négative, et si elle négative, elle devient positive), tandis que la ligne ❼ réinitialise le compteur à `0`.

Pour tester les plates-formes mobiles, nous pouvons modifier deux des objets de plates-formes existantes, pour en changer la classe de `LutinPlateForme` en `LutinPlateFormeMobile` :

```
plateforme5 = LutinPlateFormeMobile(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 = LutinPlateFormeMobile(jeu, \
        PhotoImage(file="plate-forme3.gif"), \
        170, 250, 32, 10)
plateforme10 = LutinPlateForme(jeu, PhotoImage(file="plate-forme3.gif"),\
        230, 200, 32, 10)
```

La liste qui suit montre le programme complet, avec toutes ses modifications. Voir `CH18_3_Jeufiliforme_complet.py`.

```
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
        self.texte_fin_partie = self.canvas.create_text(250, 250, \
                text='TU AS GAGNE !', state='hidden')

    def boucle_principale(self):
        while 1:
            if self.enfonction == True:
                for lutin in self.lutins:
                    lutin.deplacer()
            else:
                time.sleep(1)
                self.canvas.itemconfig(self.texte_fin_partie, \
                        state='normal')
            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
```

```
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 LutinPlateFormeMobile(LutinPlateForme):
    def __init__(self, jeu, image_photo, x, y, largeur, hauteur):
        LutinPlateForme.__init__(self, jeu, image_photo, x, y, \
                largeur, hauteur)
        self.x = 2
        self.compteur = 0
        self.derniere_heure = time.time()
```

```
        self.largeur = largeur
        self.hauteur = hauteur

    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] + self.largeur
        self.coordonnees.y2 = xy[1] + self.hauteur
        return self.coordonnees

    def deplacer(self):
        if time.time() - self.derniere_heure > 0.03:
            self.derniere_heure = time.time()
            self.jeu.canvas.move(self.image, self.x, 0)
            self.compteur = self.compteur + 1
            if self.compteur > 20:
                self.x = self.x * -1
                self.compteur = 0

class LutinPorte(Lutin):
    def __init__(self, jeu, x, y, largeur, hauteur):
        Lutin.__init__(self, jeu)
        self.porte_fermee = PhotoImage(file="porte1.gif")
        self.porte_ouverte = PhotoImage(file="porte2.gif")
        self.image = jeu.canvas.create_image(x, y, \
                image=self.porte_fermee, anchor='nw')
        self.coordonnees = Coords(x, y, x + (largeur / 2), y + hauteur)
        self.finjeu = True

    def ouvrir_porte(self):
        self.jeu.canvas.itemconfig(self.image, \
                image=self.porte_ouverte)
        self.jeu.tk.update_idletasks()

    def fermer_porte(self):
        self.jeu.canvas.itemconfig(self.image, \
                image=self.porte_fermee)
        self.jeu.tk.update_idletasks()
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.fin(lutin)
                if droite and self.x > 0 and collision_droite(co, co_lutin):
                    self.x = 0
                    droite = False
                    if lutin.finjeu:
                        self.fin(lutin)
            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)

        def fin(self, lutin):
            self.jeu.enfonction = False
            lutin.ouvrir_porte()
            time.sleep(1)
            self.jeu.canvas.itemconfig(self.image, state='hidden')
            lutin.fermer_porte()
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 = LutinPlateFormeMobile(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 = LutinPlateFormeMobile(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, 45, 30, 40, 35)
jeu.lutins.append(porte)
personnage = LutinPersonnage(jeu)
jeu.lutins.append(personnage)
jeu.boucle_principale()
```