

Beste de savoir

À la découverte de turtle

20 mars 2019

Table des matières

1. Introduction	3
2. Configurer la fenêtre	4
2.1. Les réglages	4
2.2. TP : Une fenêtre personnalisée	7
Contenu masqué	9
3. Tracer et dessiner	10
3.1. Se repérer et tracer	10
3.2. Dessiner des figures simples	13
3.3. Dessiner des choses plus complexes	18
3.4. TP : De bien jolis dessins	22
Contenu masqué	26
4. Colorier	33
4.1. Couleur de tracé et de remplissage	33
4.2. Notre fonction point	36
4.3. Notre remplissage personnalisé	36
4.4. TP : Atelier coloriage	37
Contenu masqué	39
5. Interagir avec l'utilisateur	44
5.1. Les saisies de l'utilisateur	44
5.2. Les événements	46
5.3. L'écriture à l'écran	50
5.4. L'utilisation de timers	52
5.5. TP : Le jeu des allumettes	53
Contenu masqué	57
6. Aller plus loin dans les configurations	64
6.1. Paramétrer le repère	64
6.2. Paramétrer le crayon	64
6.3. Paramétrer encore et toujours	68
6.4. TP : Configuration avancée	70
Contenu masqué	71
7. S'amuser avec plusieurs tortues	72
7.1. La classe Screen	72
7.2. La classe Turtle	73
7.3. TP : Clique la tortue!	74
Contenu masqué	78

8. Conclusion

85

1. Introduction

Adapté de la fameuse Tortue de [Logo](#) , un langage informatique en partie destiné à apprendre en créant, le module `turtle` de Python offre un vaste espace de jeu avec l'utilisation simple et élargie d'une interface graphique !

À travers ce tutoriel, nous allons découvrir les fonctionnalités de `turtle` tout en pratiquant. Ce tutoriel est ouvert à tous, et peut être un excellent entraînement pour les programmeurs peu expérimentés en particulier. Des bases en Python, que vous pouvez acquérir avec [ce tutoriel](#) par exemple, sont nécessaires pour être à l'aise.

Ce module peut être utilisé de façon procédurale (en manipulant des fonctions) ou orientée objet (en manipulant des objets et des méthodes). Nous l'explorerons avec la première approche, mais nous nous intéresserons aussi aux possibilités qu'offrent la seconde par rapport à la première dans une dernière partie.

Avant de commencer, il convient que vous ayez une version de Python installée ainsi que le module `tkinter` (*Tkinter* pour Python 2). En effet, `turtle` repose en partie sur ce dernier. Si vous avez une version de Python, cela doit être bon puisque *Tcl/Tk* est installé par défaut avec.

i

Il y a quelques **petites différences** concernant le module selon que vous utilisiez Python 2 ou Python 3, c'est pourquoi il faut que vous choisissiez la documentation adaptée à votre version. Pour ma part j'utiliserai la version 3.4 mais généralement le code sera le même que pour une version 2.

Durant la lecture de ce tutoriel, je vous invite à être actif, c'est-à-dire à tester les fonctions décrites, à bidouiller, donc à ne pas vous contenter de lire. Le meilleur moyen de comprendre et d'apprendre est d'essayer par soi-même ! De même, n'hésitez pas à vous reporter à la [documentation officielle](#) correspondant à votre version de Python pour aller chercher vous mêmes des informations sur les fonctions. Un programmeur doit être capable d'aller chercher les renseignements dont il a besoin.

Au cours des exercices proposés, il est possible de bloquer. Dès lors, pour progresser, il faut faire l'effort de se creuser la tête plutôt que de copier-coller la correction, et il ne faut surtout pas hésiter à s'aider du cours ou à demander de l'aide sur les forums.

Enfin, pour les corrections, il est tout à fait possible que votre code ne soit pas semblable au mien. Pas d'inquiétude : le principal est qu'il soit fonctionnel.

Cela étant dit, passons aux choses sérieuses !

2. Configurer la fenêtre

Nous commençons en douceur notre découverte de turtle avec la configuration basique d'une fenêtre. Cette étape est souvent primordiale dans l'utilisation des interfaces graphiques, ne serait-ce que pour choisir les dimensions de la fenêtre.

2.1. Les réglages

Avant d'utiliser le module, nous devons l'importer comme il est coutume de faire avec Python :

```
1 import turtle
```

2.1.0.1. Dimensions et positionnement

Après cela, nous pouvons ouvrir et positionner notre première fenêtre turtle. Pour ce faire, il suffit de faire appel à la fonction `setup` qui peut prendre quatre nombres en paramètre : la largeur (*width*) de notre fenêtre, sa hauteur (*height*), la position en largeur (*startx*) puis en hauteur (*starty*) du coin en haut à gauche de notre fenêtre par rapport au coin en haut à gauche de l'écran. Par défaut, la largeur vaut 50% de l'écran, la hauteur vaut 75% de l'écran et la fenêtre est centrée à l'écran. Voici quelques exemples pour mieux comprendre :

```
1 turtle.setup(640, 480, 100, 100) #Largeur : 640px, Hauteur :  
    480px, pos x : 100px, pos y : 100px  
2 turtle.setup(200, 200) #Largeur : 200px, Hauteur : 200px, position  
    centrée  
3 turtle.setup(startx = 0, starty = 0) #Largeur : 50%, Hauteur :  
    75%, position : coin haut gauche écran  
4 turtle.setup() #Largeur : 50%, Hauteur : 75%, position centrée
```

Si nous omettions de faire appel à cette fonction et que nous commencions par autre chose comme changer le titre par exemple, turtle se chargerait de l'ouvrir.

2. Configurer la fenêtre

2.1.0.2. Titre

Pour changer le titre justement, il suffit de faire appel à `title` qui prend en paramètre le titre sous forme de chaîne de caractères. Rien de bien sorcier, cela donne :

```
1 turtle.title("Ma super fenêtre") #Change le titre
```

2.1.0.3. Couleur de fond

La fonction `bgcolor` permet de modifier la couleur de fond. Elle prend en paramètre la couleur à appliquer soit sous la forme d'une chaîne de caractères (le nom ou le code hexadécimal), soit sous la forme d'un tuple (code RGB : (*Red*, *Green*, *Blue*) avec des valeurs entre 0 et 1 ici). Il existe de nombreux sites sur internet pour vous renseigner sur le code hexadécimal ou le code RGB d'une couleur, en voici [un](#) (pour passer des valeurs d'une échelle de 255 à 1, nous pouvons procéder ainsi : $127 \Rightarrow 127/255 = 0.50$ en arrondissant ; dans le sens inverse, cela donne $0.50 \Rightarrow 0.50*255 = 127$ en tronquant). Si on ne lui passe aucun argument, elle nous retourne la couleur courante. Comme à l'accoutumée, voici quelques exemples :

```
1 turtle.bgcolor("black") #Met fond en noir
2 print(turtle.bgcolor()) #Affiche 'black'
3 turtle.bgcolor("#00FF00") #Met fond en vert
4 turtle.bgcolor((0.5, 0, 1)) #Met fond en violet
```

Au passage, voici un tableau récapitulatif de valeurs de couleur que nous pouvons utiliser. En plus, nous pouvons faire précéder une partie de ces valeurs des termes *"light"* pour clair ou *"dark"* pour foncé. Par exemple, *"lightgrey"* pour gris clair et *"darkgrey"* pour gris foncé. Une exception est retournée avec un message d'erreur lorsque la combinaison n'est pas possible.

Valeur	Couleur
"white"	Blanc
"black"	Noir
"grey"	Gris
"brown"	Marron
"orange"	Orange
"pink"	Rose
"purple"	Violet
"red"	Rouge
"blue"	Bleu

2. Configurer la fenêtre

"yellow"	Jaune
"green"	Vert

2.1.0.4. Image de fond

Ensuite, nous pouvons aussi personnaliser le fond de notre fenêtre avec une image, qui est alors automatiquement centrée et ne prend que la place dont elle a besoin. La fonction `bgpic` permet de faire cela. Elle prend en paramètre une chaîne de caractères représentant le chemin vers l'image. Si nous ne lui fournissons aucune valeur, elle retourne le nom de l'image de fond s'il y a une ou *"nopic"* s'il n'y en a pas. Vous pouvez aussi lui passer *"nopic"* pour enlever l'image de fond. Les formats *bmp* et *jpg* ne sont pas reconnus contrairement aux formats *png* et *gif*. Un exemple :

```
1 turtle.bgpic("image.png") #Ajoute l'image en fond
2 print(turtle.bgpic())    #Affiche 'image.png'
3 turtle.bgpic("nopic")    #Supprime l'image de fond s'il y en a une
4 print(turtle.bgpic())    #Affiche 'nopic'
```

2.1.0.5. Fermeture

Si vous avez testé les fonctions présentées ci-dessus avec la ligne de commande Python ou avec *idle*, vous vous êtes rendu compte que la fenêtre reste ouverte jusqu'à ce qu'on la ferme. Mais si vous exécutez le code autrement, la fenêtre se fermera automatiquement une fois les traitements terminés. Pour le premier cas, nous pouvons remédier à cela avec la fonction `bye`. Placée à la fin de notre code, l'exécution de celle-ci fermera la fenêtre et marquera la fin de l'exécution du programme. Pour le second cas, vous pouvez utiliser `exitonclick` qui permet d'associer le clique gauche à la fermeture de la fenêtre. Ces deux fonctions ne prennent aucun paramètre.

```
1 ###Traitement
2 #...
3 turtle.bye()
4 ###ou bien
5 turtle.exitonclick()
```

2.1.0.6. Afficher ou cacher le curseur

Enfin, de façon plus anecdotique, mentionnons la possibilité de cacher ou d'afficher le curseur (ou le crayon) avec les fonctions respectives `hideturtle` et `showturtle`. De plus, vous pouvez savoir si le crayon est actuellement affiché avec la fonction `isvisible`.

2. Configurer la fenêtre

```
1 turtle.hideturtle() #Cache le crayon
2 turtle.showturtle() #Affiche le crayon
3 print(turtle.isvisible()) #Affiche 'True' : le crayon est visible
```

Il est temps de mettre en œuvre ce que l'on vient de voir !

2.2. TP : Une fenêtre personnalisée

Voilà, nous y sommes. Comme vous allez le voir, ce premier exercice est assez simple. Encore une fois, si vous êtes bloqué au cours de ces travaux pratiques, vous pouvez parcourir ce tutoriel ou demander de l'aide sur les forums pour vous aider.

Le but de l'exercice est de réaliser un programme ouvrant une fenêtre ayant ces caractéristiques :

- Largeur = 640px ; Hauteur = 480px ;
- Position en largeur = Position en hauteur = 50px ;
- Couleur de fond = jaune ;
- Image de fond présente (je vous laisse choisir votre image, faites en sorte qu'elle ne remplisse pas toute la fenêtre) ;
- Fermeture au clique possible une fois fini.

De plus, je vous demande aussi de programmer une fonction récapitulant toutes ces informations (hormis celles relatives à la position de la fenêtre et au clique). Pour connaître la largeur et la hauteur de la fenêtre, il faut respectivement utiliser les fonctions `windows_width` et `windows_height` qui ne prennent aucun paramètre.

Voici le résultat obtenu de mon côté :

2. Configurer la fenêtre

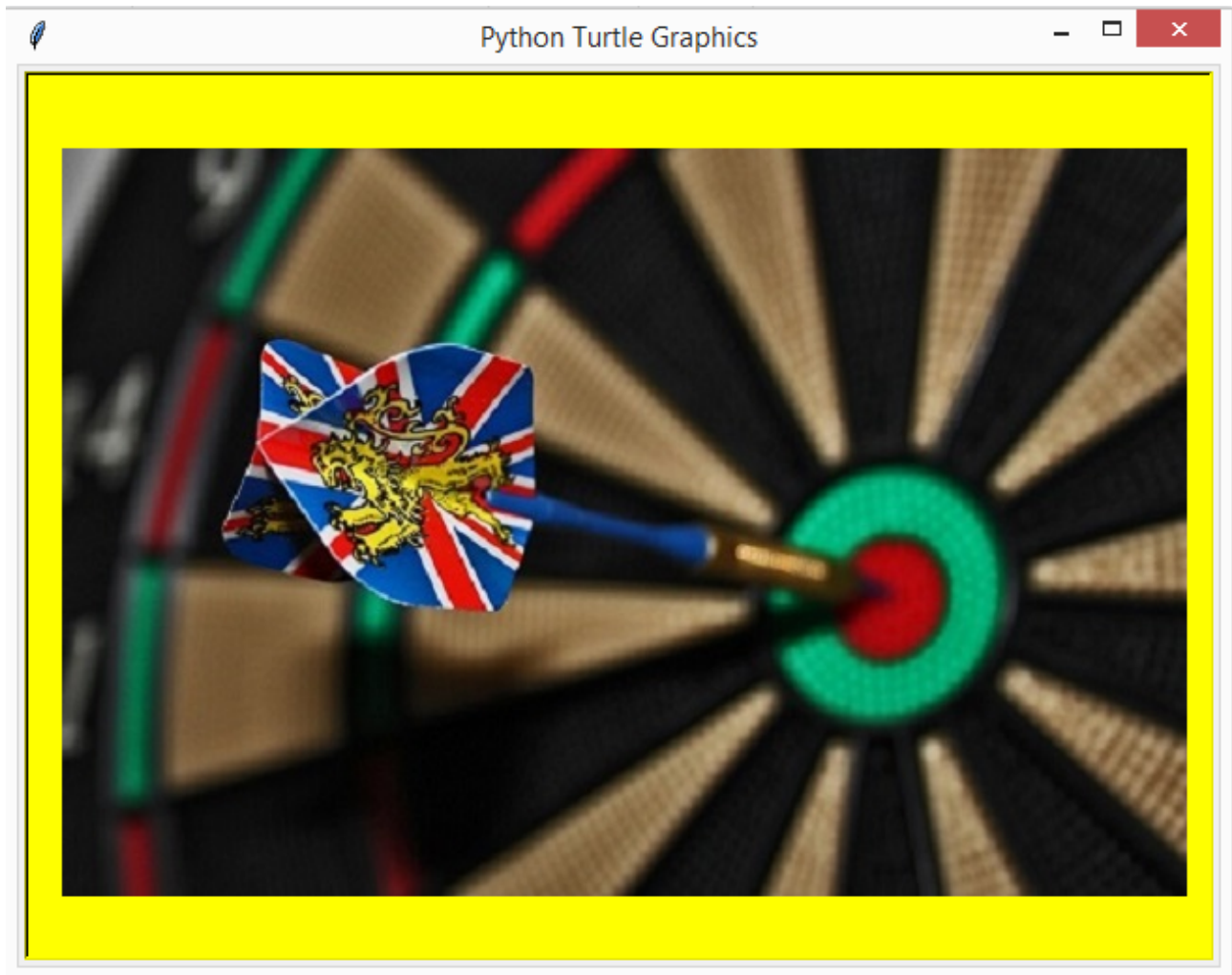


FIGURE 2.1. – Le résultat.

```
1 largeur : 640px
2 hauteur : 480px
3 couleur : yellow
4 image : TP_fenetre_personnalisee.png
```

Nous terminons sur la **correction** :

👁 Contenu masqué n°1

Voilà, vous êtes désormais capable de configurer votre fenêtre avec turtle. Dans la partie suivante, nous allons voir comment tracer et dessiner.

Contenu masqué

Contenu masqué n°1

```
1  #!/usr/bin/env python3
2
3  import turtle
4
5  #Constantes
6  LARGEUR, HAUTEUR = 640, 480
7  POS_X = POS_Y = 50
8  NOM_IMAGE = "TP_fenetre_personnalisee.png"
9
10 def recapitule():
11     """Fonction pour récapituler la largeur, la hauteur,
12     la couleur et l'image de la fenêtre turtle"""
13     print("largeur : {} px".format(turtle.window_width()))
14     print("hauteur : {} px".format(turtle.window_height()))
15     print("couleur : "+turtle.bgcolor())
16     print("image : "+turtle.bgpic())
17
18 if __name__ == "__main__":
19     #On ouvre la fenêtre en choisissant dimensions et positions
20     turtle.setup(LARGEUR, HAUTEUR, POS_X, POS_Y)
21     #On change la couleur de fond
22     turtle.bgcolor("yellow")
23     #On change l'image de fond
24     turtle.bgpic(NOM_IMAGE)
25     #On récapitule la configuration de la fenêtre hormis la
26     position
27     recapitule()
28     #On ferme la fenêtre s'il y a un clique gauche
29     turtle.exitonclick()
```

[Retourner au texte.](#)

3. Tracer et dessiner

Nous allons pouvoir entrer dans le vif du sujet ! À travers cette partie nous verrons comment tracer et dessiner des figures.

3.1. Se repérer et tracer

Pour pouvoir se repérer dans la fenêtre, turtle met en place un repère à deux dimensions. Par défaut, celui-ci est centré dans la fenêtre. Ce repère nous permet de nous déplacer aisément de x sur l'axe des abscisses et de y sur l'axe des ordonnées. Le centre du repère, c'est-à-dire le point $(x = 0, y = 0)$, est l'endroit où le curseur apparaît, sa maison en quelque sorte. Cela n'est pas bien compliqué à comprendre, il faut juste s'habituer à penser dans le plan. L'image suivante permet de mieux visualiser ce que nous venons de dire et comporte quelques exemples de points :

3. Tracer et dessiner

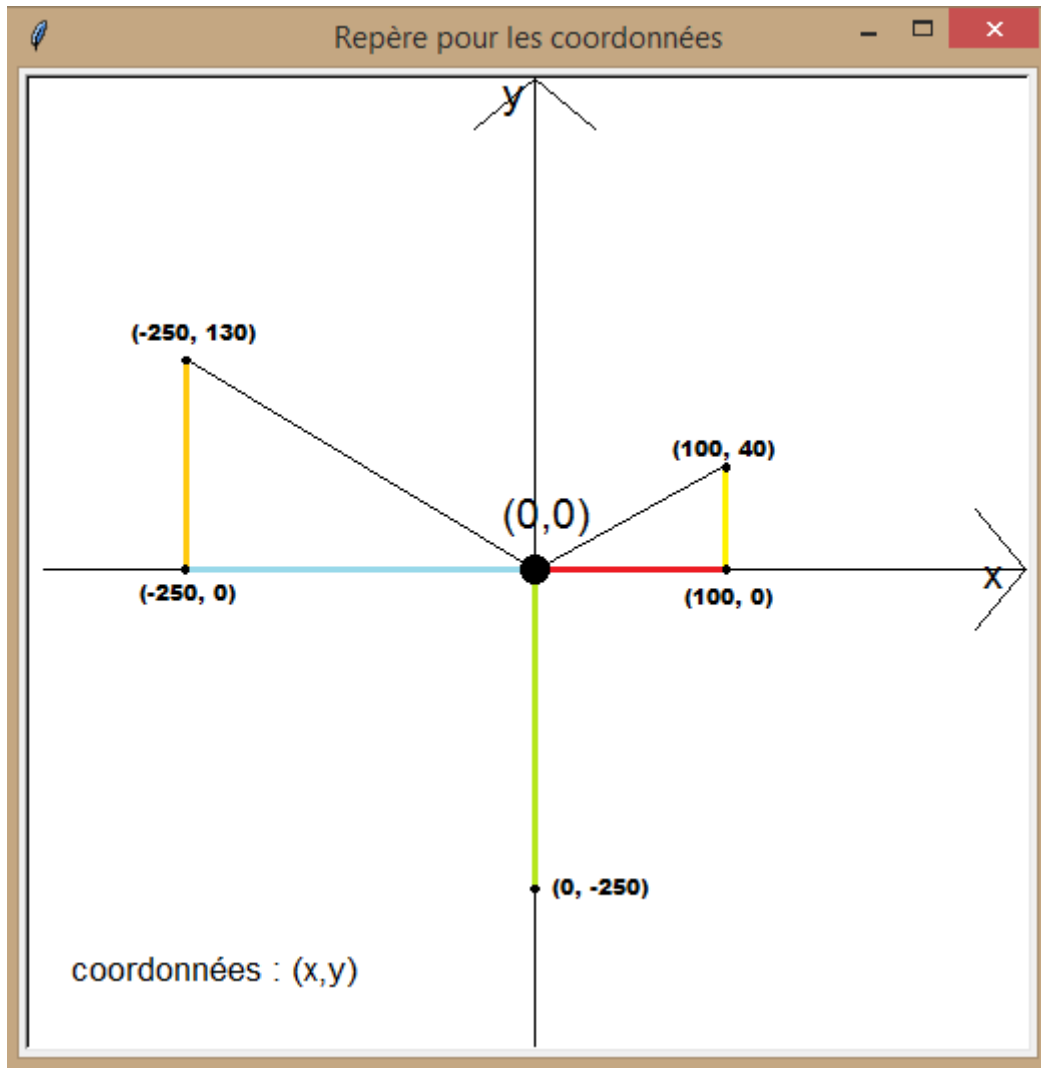


FIGURE 3.1. – Le repère.

3.1.0.1. Nettoyer l'écran

Au cours de notre utilisation de turtle, il est possible que nous ayons besoin de nettoyer l'écran. Pour cela, nous pouvons utiliser `clear` qui permet d'effacer ce que nous avons dessiné. En plus, nous pouvons aussi utiliser `reset` qui fait la même chose et réinitialise les valeurs du curseur à leurs valeurs par défaut (le crayon retourne à l'origine du repère, retrouve son orientation originale, sa largeur de trait par défaut, etc...). Ces deux fonctions ne modifient donc pas les configurations liées à la fenêtre, comme le titre ou la couleur de fond par exemple. Enfin, elles ne prennent aucun paramètre.

```
1 turtle.clear() #Efface les dessins du crayon
2 turtle.reset() #Fait de même et réinitialise le crayon
```

3. Tracer et dessiner

3.1.0.2. Avancer et reculer

Pour tracer, il faut se déplacer. Et pour déplacer le curseur, turtle nous offre plusieurs fonctions, comme `forward` et `backward`, respectivement pour avancer et reculer d'une distance que l'on passe en paramètre. Elles ont aussi chacune leur version abrégée, respectivement `fd` et `bk`.

```
1 turtle.forward(turtle.window_width()/3) #Avance d'un tiers de la
   largeur de la fenêtre
2 turtle.backward(turtle.window_width()/2) #Recul de la moitié de
   la largeur de la fenêtre
3 turtle.bk(50) #Recul de 50px
4 turtle.fd(0) #Avance de 0px, donc n'avance pas
```

3.1.0.3. Se déplacer à des coordonnées données

Avec `goto`, en lui fournissant une coordonnée x et une coordonnée y , nous pouvons nous rendre directement à un point (x, y) donné. De plus, nous pouvons aussi modifier uniquement la position en abscisse du curseur avec `setx` et la position en ordonnée avec `sety`, en leur passant la nouvelle valeur. Enfin, puisque nous avons parlé du centre du repère, notons que la fonction `home` permet d'y retourner.

```
1 turtle.goto(100, 100) #Position (100, 100)
2 turtle.setx(20) #Position(20, 100)
3 turtle.sety(-80) #Position(20, -80)
4 turtle.home() #Position(0, 0) (équivalent à turtle.goto(0, 0))
```

3.1.0.4. Lever ou baisser le crayon

Si nous ne pouvions pas nous déplacer dans la fenêtre sans laisser de trace, ce ne serait pas très amusant. Or, la fonction `up` nous permet de lever le crayon tandis que la fonction `down` nous permet de l'abaisser. Elles ne prennent aucun paramètre. Grâce à elles, nous pouvons choisir de tracer ou non :

```
1 turtle.up() #Lève le crayon
2 turtle.forward(150) #Avance de 150px sans tracer
3 turtle.down() #Abaisse le crayon
4 turtle.backward(50) #Recul de 50px en traçant
```

3. Tracer et dessiner

3.1.0.5. Changer la taille du traçage

Faire des traits, c'est bien, mais pouvoir choisir la taille, c'est encore mieux. En passant la nouvelle largeur de nos traits à `pensize`, nous pouvons le faire. En ne passant rien, la fonction nous renvoie la taille actuelle.

```
1 print(turtle.pensize()) #Affiche '1'
2 turtle.pensize(5.5) #Modifie la largeur du traçage
3 print(turtle.pensize()) #Affiche '5.5'
```

Pour le moment, ce sont des fonctions plutôt basiques. Ce serait plus intéressant de pouvoir faire des figures, en assemblant les traits, et c'est ce que nous allons faire dans la section suivante !

3.2. Dessiner des figures simples

Jusqu'à présent, nous avons vu comment ouvrir une fenêtre et comment nous déplacer dans celle-ci. À présent, nous allons aller encore plus loin en dessinant nos premières figures.

3.2.0.1. Changer l'angle

Pour dessiner aisément, il nous manque tout de même quelque chose, et je pense que vous vous en êtes rendu compte : il faut que l'on puisse choisir l'inclinaison de notre trait, c'est-à-dire l'angle. En effet, jusqu'à présent nous avons été limités dans nos déplacements.

Or, `turtle` nous permet justement de faire varier la direction du curseur. Par défaut, lorsque l'on ouvre une fenêtre avec `turtle`, le crayon est orienté vers l'Est : l'angle est de 0 (ou 360). Pour jouer avec les angles, nous avons les fonctions `right` et `left` qui permettent de tourner respectivement vers la droite ou vers la gauche d'un angle passé. Parfois, il est plus simple d'utiliser `setheading` qui change directement l'angle avec la valeur passée. Vous pouvez aussi connaître la direction actuelle de votre crayon en utilisant la fonction `heading` qui ne prend aucun paramètre. Toutes ces explications sont illustrées avec l'image ci-dessous ainsi que par l'exemple qui la suit :

3. Tracer et dessiner

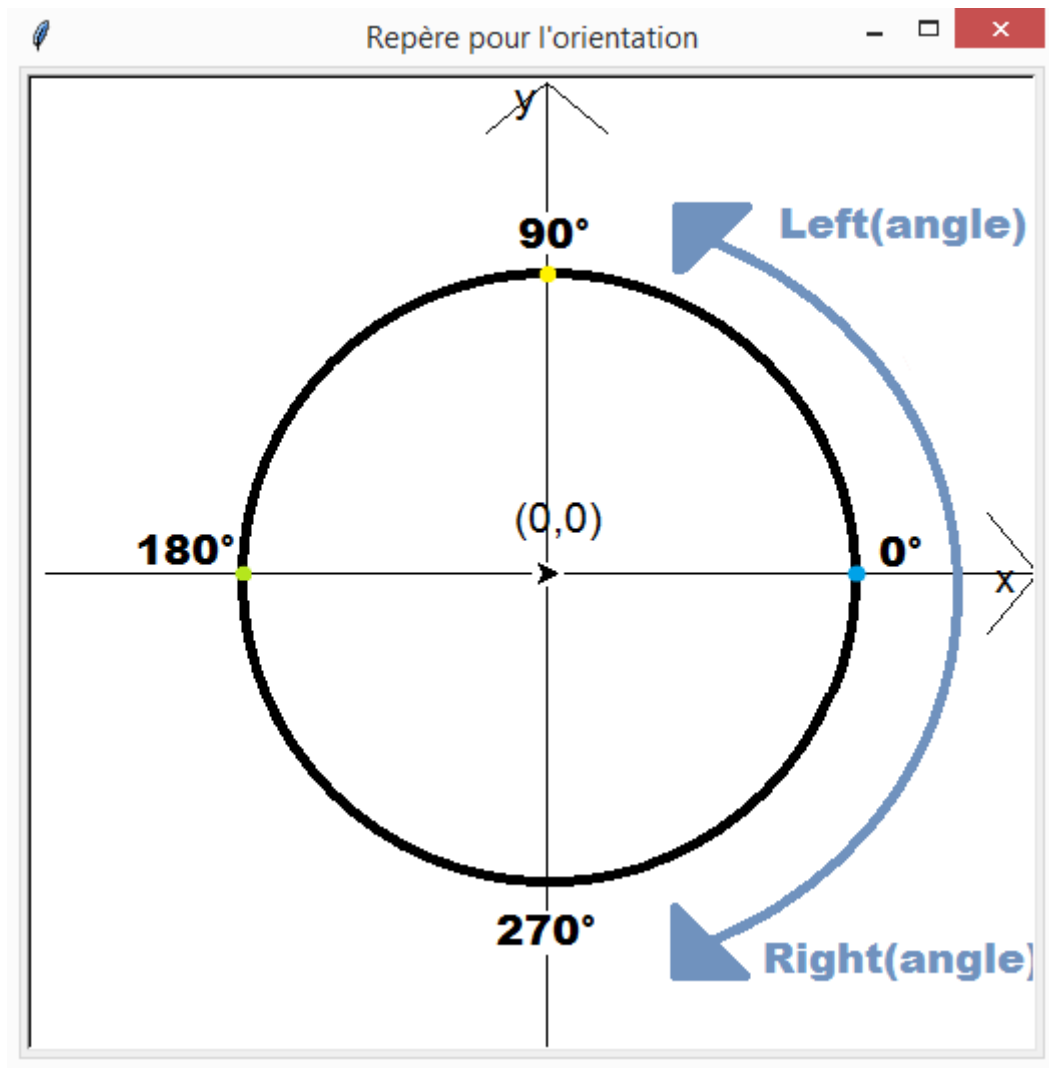


FIGURE 3.2. – L'orientation.

```
1 turtle.setup() #Initialise la fenêtre
2 print(turtle.heading()) #Affiche 0.0 : le crayon pointe vers le
   point bleu : Est
3 turtle.left(90) #Pointe vers le point jaune : Nord
4 turtle.right(270) #Pointe vers le point vert : Ouest
5 turtle.setheading(0) #Pointe de nouveau vers le point bleu
6 turtle.setheading(-90) #Pointe à l'opposé du point jaune : Sud
7 print(turtle.heading()) #Affiche '270.0'
```

Concrètement, vous conviendrez que nous sommes désormais beaucoup plus libres. N'hésitez pas à essayer et à vous approprier ces notions, car elles seront vraiment utiles pour la suite. Voici un exemple d'utilisation de ce que l'on vient d'apprendre :

Code :

👁 Contenu masqué n°2

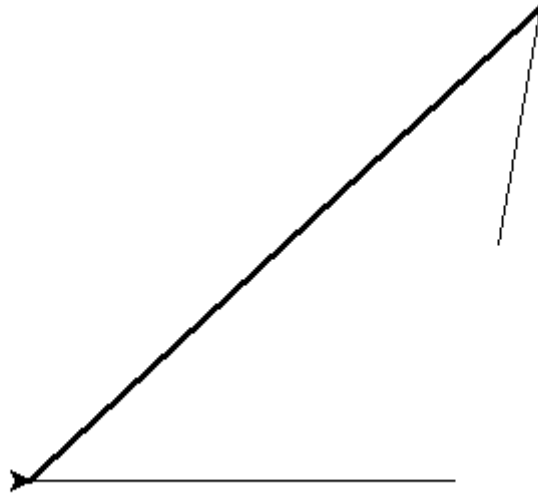


FIGURE 3.3. – Résultat exemple d'utilisation des angles.

Notons au passage que `home` réinitialise aussi l'orientation en plus de la position, c'est pourquoi nous voyons que le curseur a un angle de 0 à la fin puisque nous terminons le traitement par cela.

Pour terminer sur les angles, nous pouvons brièvement parler de la fonction `towards` qui prend en paramètre les coordonnées d'un point et nous retourne l'angle qu'il faudrait pour aller à ce point. Ainsi, ces deux morceaux de code donnent quasiment le même résultat (dans le second cas, l'angle final est celui d'avant l'appel à la fonction)

```
1 angle = turtle.towards(0, 90)
2 print(angle) #Affiche '90.0'
3 turtle.setheading(angle) #Angle : 90.0
4 turtle.forward(90) #Position : (0, 90); Angle : 90.0
```

```
1 turtle.goto(0, 90) #Position : (0, 90); Angle : 0.0
```

3.2.0.2. Dessiner des figures simples

Voilà, nous sommes désormais totalement capable de tracer nos propres figures grâce à ce que nous avons appris. Nous allons donc nous exercer en faisant quelques polygones. Triangle

3. Tracer et dessiner

équilatérale, carré et octogone régulier seront nos invités. Avant que vous lisiez la suite, je vous encourage à essayer de dessiner par vous-mêmes ces figures.

Commençons avec le triangle équilatéral. Pour rappel, un triangle équilatéral est un triangle dont les côtés ont la même longueur ce qui implique que chaque angle a une valeur de 60° (par définition, la somme des angles d'un triangle vaut 180°). Une fois que l'on a cela en tête, nous pouvons implémenter une solution explicite :

```
1 ###Un exemple de triangle équilatéral
2 longueur_cote = 200
3 turtle.forward(longueur_cote) #1er côté
4 turtle.left(360/3) #Angle
5 turtle.forward(longueur_cote) #2ème côté
6 turtle.left(360/3) #Angle
7 turtle.forward(longueur_cote) #3ème côté
```

Pour le carré et l'octogone, nous appliquerons le même principe. Pour le carré, nous avons quatre côtés de même longueur ainsi que quatre angle de 90° . Voici une solution :

```
1 ###Un exemple de carré
2 longueur_cote = 200
3 for i in range(4):
4     turtle.forward(longueur_cote) #Côté
5     turtle.left(90) #Angle
```

L'octogone a quant à lui 8 côtés et des angles de 45° (360° divisé par 8 côtés). Une solution est :

```
1 ###Un exemple d'octogone
2 longueur_cote = 100
3 for i in range(8):
4     turtle.forward(longueur_cote) #Côté
5     turtle.left(360/8) #Angle
```

3. Tracer et dessiner

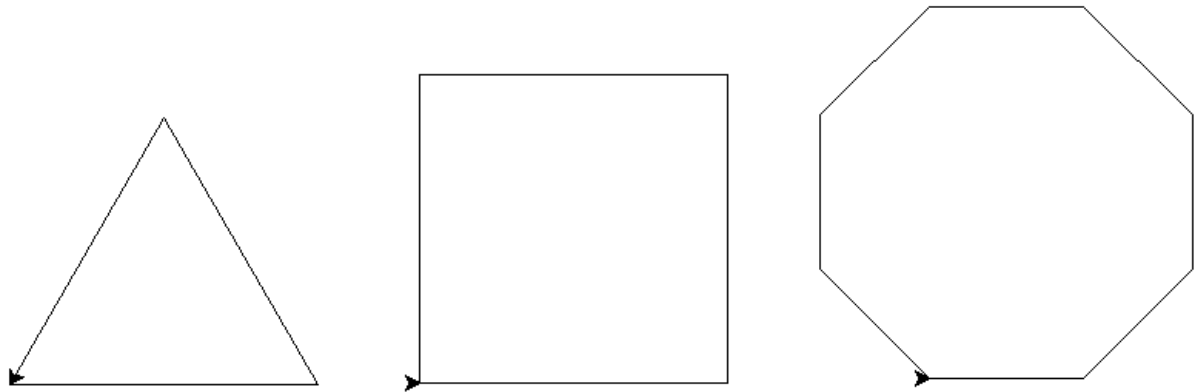


FIGURE 3.4. – Résultats pour ces figures.

Voilà, pour pratiquer, vous pouvez essayer de dessiner des figures plus compliquées voire de généraliser au cas d'un polygone, ou encore de dessiner à d'autres endroits que depuis le centre du repère. Il est temps de parler d'une dernière figure dont nous n'avons pas encore parlé : le cercle.

3.2.0.3. Utiliser les cercles

Pour les cercles, nous pouvons éviter de réinventer la roue puisqu'il existe une fonction déjà toute prête : `circle`. Au minimum, nous devons passer à celle-ci le rayon de notre cercle. De plus, nous pouvons aussi lui passer un angle (*extent*) qui permet de tracer uniquement une partie du cercle, ainsi qu'un nombre (*steps*) qui correspond au nombre d'étapes pour tracer. L'orientation du crayon a des conséquences sur la manière dont le cercle sera tracé. Pour mieux comprendre, voyons ce que ça peut donner :

```
1 turtle.circle(120) #Trace un cercle de rayon 120px
2 turtle.circle(70, 180) #Trace un demi-cercle de rayon 70px
3 turtle.circle(90, steps = 8) #Trace un octogone de longueur 90px
4 turtle.circle(40, 180, 4) #Trace la moitié d'un octogone de
    longueur 40px
```

Voici un exemple de code pour afficher cinq cercles du plus petit au plus grand, avec pour centre l'origine du repère :

Code :

👁 Contenu masqué n°3

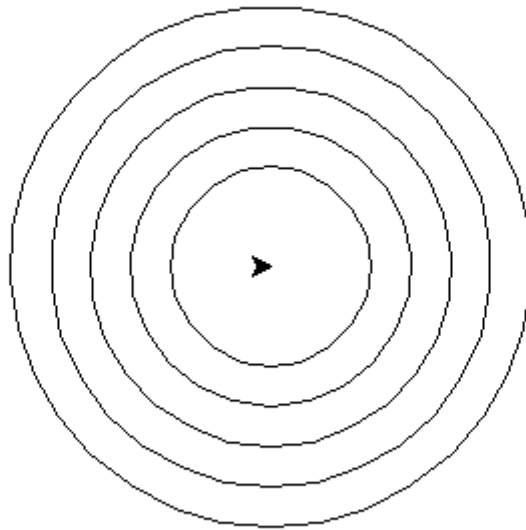


FIGURE 3.5. – Résultat exemple cercles.

Voilà, nous savons désormais dessiner des figures simples. Dans la section suivante, nous allons complexifier nos figures.

3.3. Dessiner des choses plus complexes

3.3.0.1. Un dessin plus complexe (1)

Grâce à ce que nous venons d'apprendre, nous pouvons désormais réaliser des figures plus complexes, comme le montre l'image suivante par exemple. Pour réaliser cela, j'ai utilisé deux fonctions que nous n'avons pas encore vues. Tout d'abord, la fonction `position`, qui ne prend aucun paramètre et retourne, comme son nom l'indique, la position du crayon. Ensuite, j'ai aussi fait appel à la fonction `distance` qui prend en paramètre les coordonnées x et y d'un point et qui retourne la distance entre le curseur et ce point. De cette manière, j'ai pu connaître facilement la distance entre le centre du dessin, le point $(0, 0)$ et le coin des petits carrés par lesquels je souhaitais faire passer le cercle : c'est-à-dire le rayon. Remarquez que le dessin a pour centre le centre du repère.

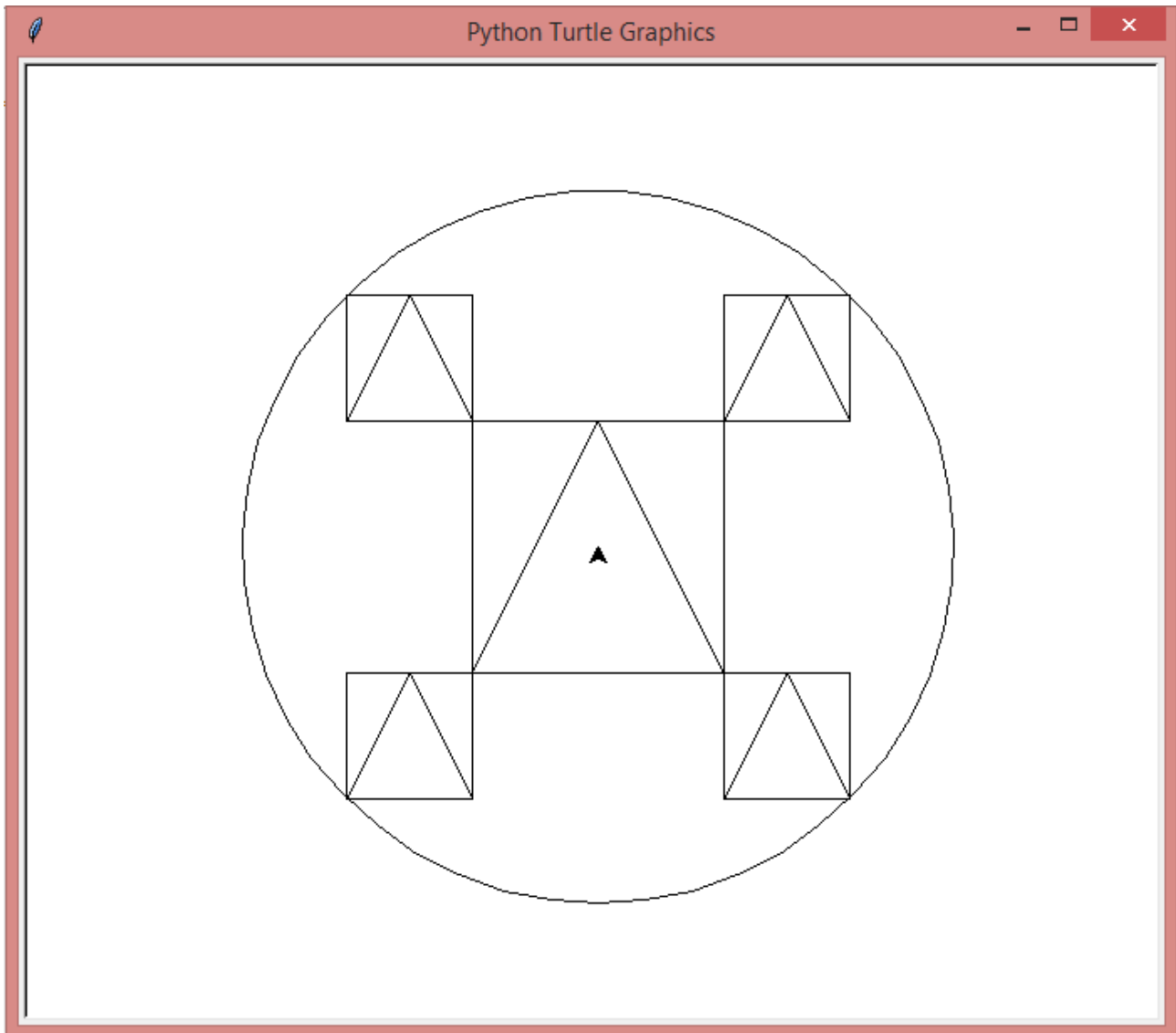


FIGURE 3.6. – Un dessin un peu plus complexe.

Concernant le code, nous commençons par tracer le grand carré, puis les quatre petits, et nous terminons par le cercle. L'ensemble a pour centre le point d'origine du repère. Pour tracer chaque carré, nous nous plaçons à son coin bas gauche. Les commentaires vous aideront à comprendre. Si vous avez du mal, n'hésitez pas à prendre une feuille de papier et un crayon pour vous aider à visualiser.

Code :

© Contenu masqué n°4

3.3.0.2. Les points

La fonction `dot` nous permet d'afficher un point dans le canvas. Pour cela, nous pouvons passer en paramètre le diamètre du point, et nous pouvons aussi, si le cœur nous en dit, passer une

3. Tracer et dessiner

couleur. Si nous ne lui passons rien, le point aura un diamètre par défaut et la couleur de traçage du curseur (noir par défaut). Nous étudierons le coloriage plus en détails dans la partie suivante.

```
1 turtle.dot(100, 'red') #Imprime un point rouge d'un diamètre de
    100px
2 turtle.dot(50, 'yellow') #Imprime un point jaune d'un diamètre de
    50px
3 turtle.dot(25) #Imprime un point noir d'un diamètre de 25px
```

Remarquons que si nous avions imprimé les points dans l'ordre inverse, nous n'aurions vu que le rouge puisque celui aurait masqué le jaune qui lui-même aurait masqué le noir. Nous pouvons aussi noter qu'un point sera tracé même si le crayon est levé.

Puisque comme d'habitude, rien ne nous empêche de jouer avec ce que nous apprenons, voici un programme qui imprime dix points de plus petit en plus grand en allant de gauche à droite et qui ont une couleur différente :

Code :

👁 Contenu masqué n°5

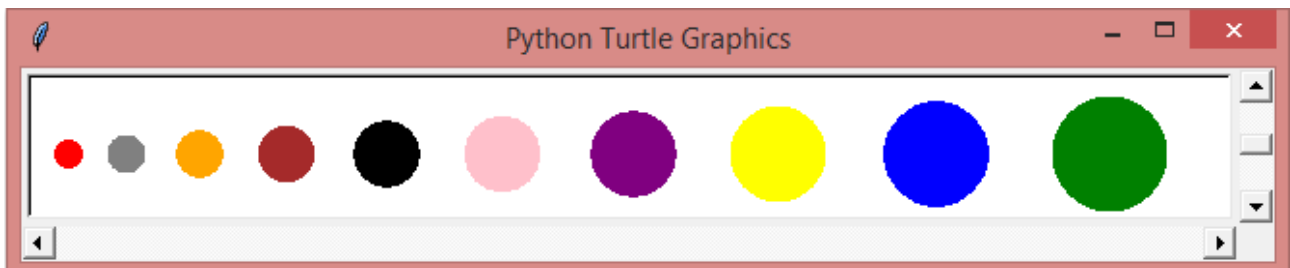


FIGURE 3.7. – Résultat exemple de points.

3.3.0.3. Les tampons

Les points, ce n'est pas tout ! Nous pouvons aussi, de la même manière, imprimer la forme du curseur avec `stamp`. Cette fonction ne prend aucun paramètre et retourne l'identifiant du tampon. Ce dernier nous sert à supprimer le tampon de la fenêtre en le passant à `clearstamp`. Nous pouvons aussi supprimer plusieurs tampons de l'écran en fournissant un nombre à `clearstamps`, voire tous en ne lui passant aucune valeur ou `None`. Voici ci-dessous un exemple d'utilisation illustré :

```
1 id_tampons = []
2 ###L'opération suivante est répétée à maintes reprises tout en se
    déplaçant
```

3. Tracer et dessiner

```
3 id_tampons.append(turtle.stamp()) #Tamponne et on enregistre  
   l'identifiant  
4 turtle.clearstamp(id_tampons[14]) #Supprime le 15ème tampon  
5 turtle.clearstamps(9) #Supprime les 9 premiers tampons  
6 turtle.clearstamps(-10) #Supprime les 10 derniers tampons  
7 turtle.clearstamps() #Supprime tous les tampons restants
```

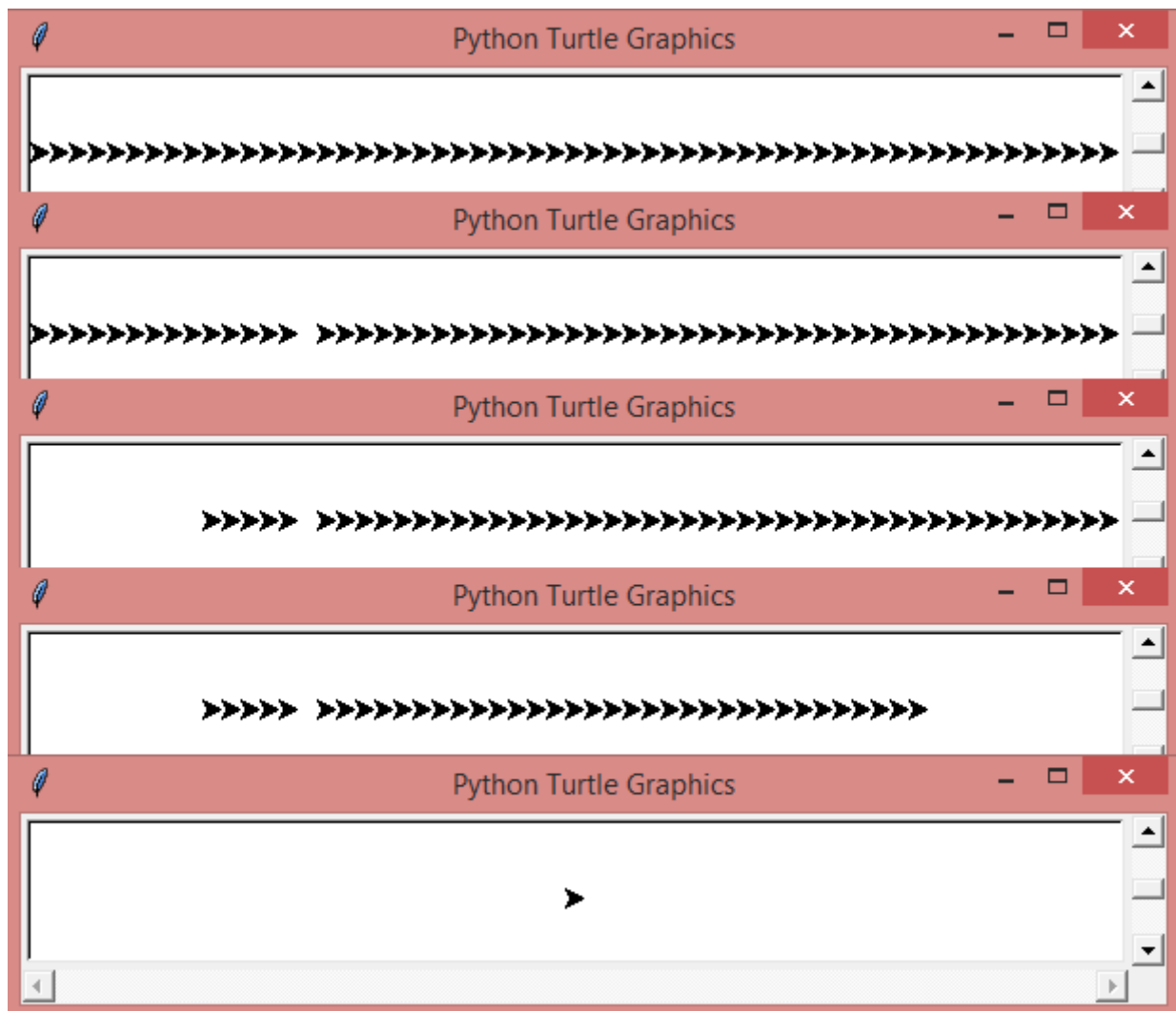


FIGURE 3.8. – Résultat exemple tampon aux différentes étapes.

3.3.0.4. Un dessin plus complexe (2)

Voici un autre exemple un peu plus complexe. Ici, nous dessinons un ciel étoilé. Pour ce faire, nous avons codé une fonction `etoile` qui se charge de tracer une étoile d’une longueur donnée, et nous allons nous en servir dans la boucle principale, tout en veillant à ce que l’étoile ne soit pas dessinée hors de notre fenêtre. Nous pourrions améliorer le code pour éviter qu’une étoile soit tracée par dessus une autre par exemple.

👁 Contenu masqué n°6

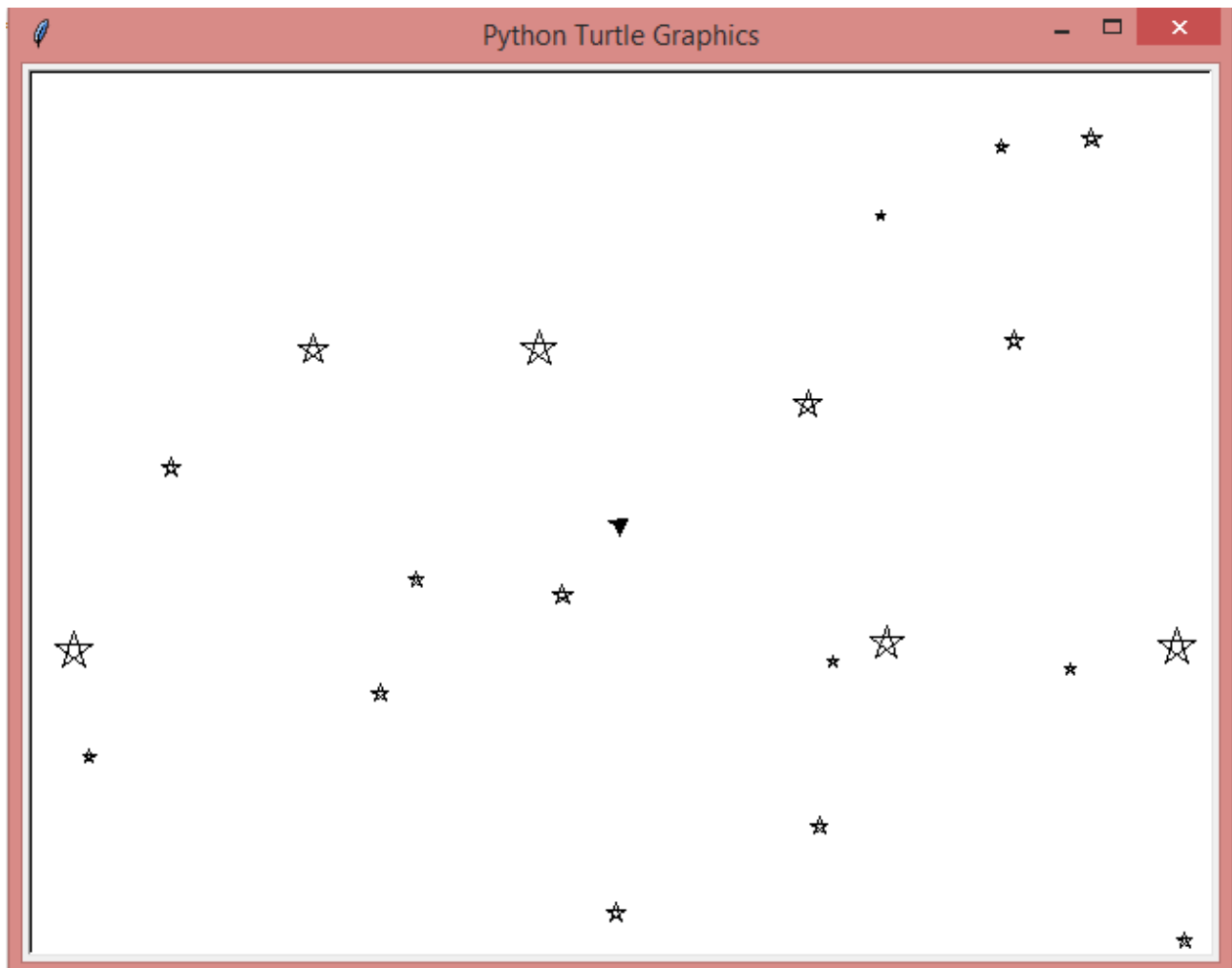


FIGURE 3.9. – Un ciel étoilé.

Voilà, j'espère que vous avez suivi avec attention, car c'est le moment de mettre tout cela en pratique !

3.4. TP : De bien jolis dessins

Nous voilà aux travaux pratiques de cette partie ! C'est le moment de vérifier que vous savez dessiner avec ce que nous avons appris. Pour ce faire, il y aura deux exercices où il faudra coder le programme permettant de reproduire le dessin (ne vous souciez pas des longueurs). Si vous êtes bloqué, n'hésitez pas à retourner voir ce que nous avons précédemment pour vous aider et à procéder par étapes. Enfin, lors du troisième exercice, nous nous intéresserons au Flocon de Von Koch.

3. Tracer et dessiner

3.4.0.1. Exercice 1

Voici la première figure à reproduire :

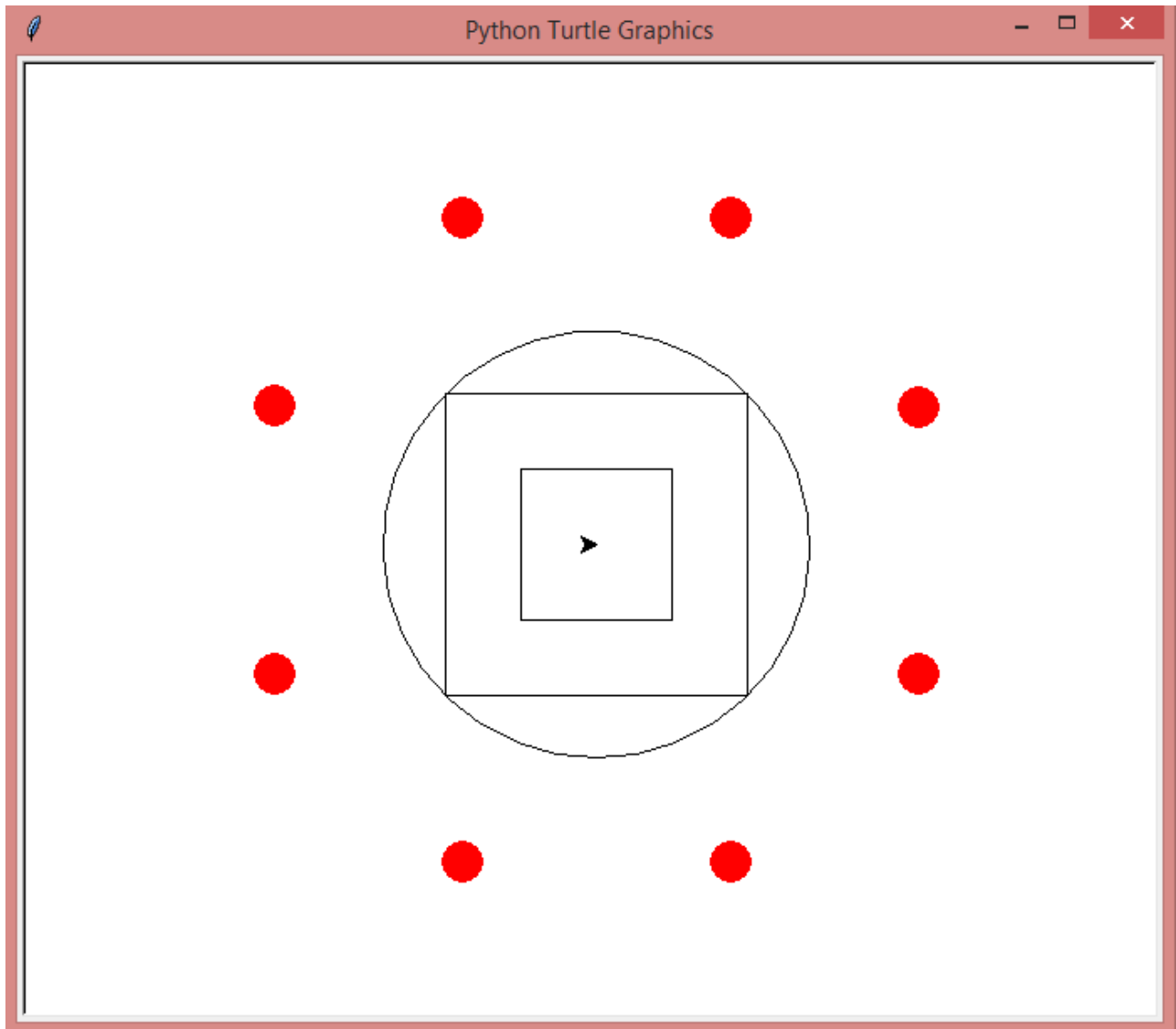


FIGURE 3.10. – Exercice 1.

La **correction** :

© Contenu masqué n°7

3.4.0.2. Exercice 2

Voici la seconde figure à reproduire :

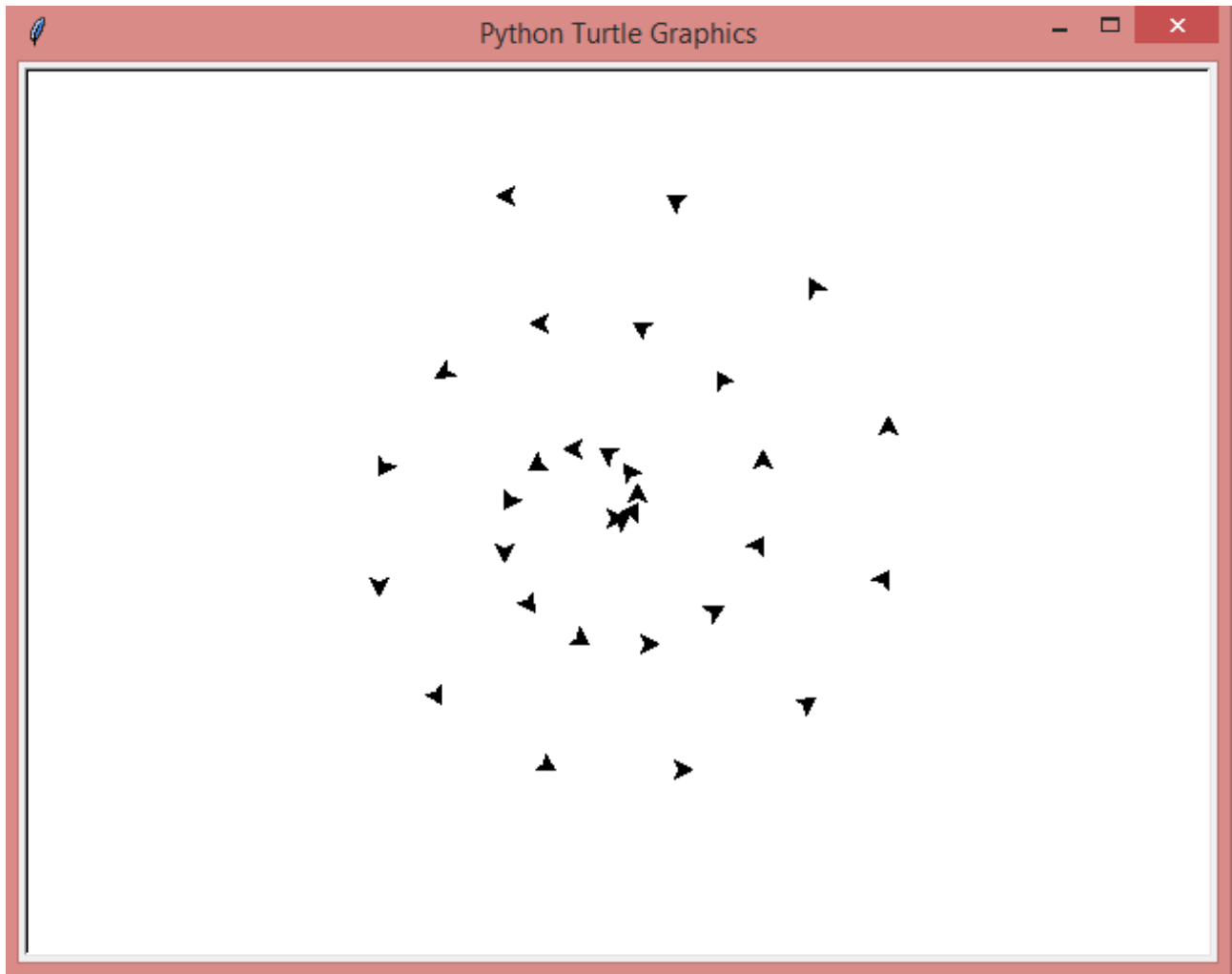


FIGURE 3.11. – Exercice 2.

La **correction** :

👁 Contenu masqué n°8

3.4.0.3. Exercice 3 : Flocon de Von Koch

Un flocon de Von Koch est l'ensemble de trois courbes de Von Koch constituant chaque côté du triangle équilatéral initial. Vous pouvez trouver plus d'explication à propos de cette figure et de sa construction [ici](#) .

À travers cet exercice, il va falloir faire une fonction pour dessiner un flocon de Von Koch en fonction de la longueur des côtés du triangle ainsi que du nombre d'étapes permettant choisir le nombre de pics de notre flocon (par exemple, avec 0 et une longueur non nulle, nous avons juste un triangle équilatéral). Cet exercice étant un peu plus complexe, je vous conseille de découper votre progression ainsi : tout d'abord, essayez de programmer une fonction réalisant une courbe de Von Koch selon les paramètres précédemment mentionnés, puis, dans un second temps, réalisez une fonction pour tracer un flocon selon les paramètres.

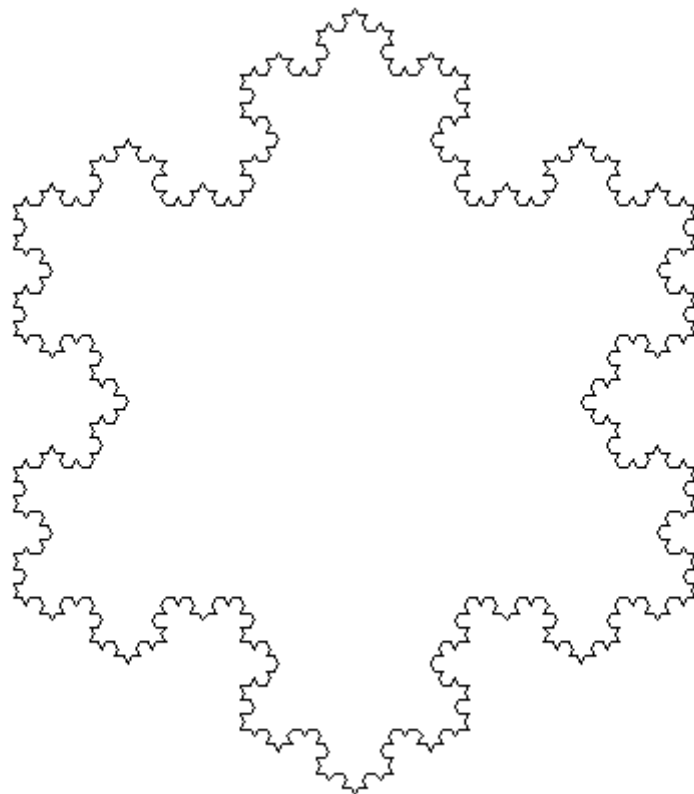


FIGURE 3.12. – Un exemple de flocon de Von Koch avec 3 étapes.

La **correction** :

© Contenu masqué n°9

Voilà, la récursivité s'appliquant bien à ce genre de figure, nous avons une fonction récursive pour tracer une courbe de Von Koch, puis une fonction traçant un flocon de Von Koch à l'aide de cette première.

Désormais, nous savons tracer et dessiner en utilisant turtle. Si vous trouvez que ça manque encore de couleurs, vous allez être ravis, car c'est justement le but de la partie suivante !

Contenu masqué

Contenu masqué n°2

```
1  #!/usr/bin/env python3
2
3  import turtle
4
5  LARGEUR, HAUTEUR = 640, 480
6
7  if __name__ == "__main__":
8      turtle.forward(LARGEUR/3)  #Avance de d'un tiers de la largeur
9      turtle.left(80)  #Tourne de 80° à gauche
10     turtle.up()  #Lève le curseur
11     turtle.forward(HAUTEUR/4)  #Avance d'un quart de la hauteur
12     turtle.down()  #Baisse le curseur
13     turtle.right(180)  #Tourne à 180 à droite
14     turtle.backward(HAUTEUR/4)  #Reculé d'un quart de la hauteur
15     turtle.pensize(3)  #Change l'épaisseur du tracé
16     turtle.home()  #Retourne à la maison
17     turtle.exitonclick()  #Clique gauche pour fermer
```

[Retourner au texte.](#)

Contenu masqué n°3

```
1  #!/usr/bin/env python3
2
3  import turtle
4
5  if __name__ == "__main__":
6      rayon, ecart = 50, 20
7      for i in range(5):
8          turtle.up()
9          turtle.goto(0, -rayon)
10         turtle.down()
11         turtle.circle(rayon)
12         rayon += ecart  #Augmente la valeur de rayon
13     turtle.up()
14     turtle.home()
15     turtle.exitonclick()
```

[Retourner au texte.](#)

Contenu masqué n°4

```

1  #!/usr/bin/env python3
2
3  import turtle
4
5  def deplacer_sans_tracer(x, y = None):
6      """Fonction pour se déplacer à un point sans tracer"""
7      turtle.up()
8      if (isinstance(x, tuple) or isinstance(x, list)) and len(x) ==
9          2:
10         turtle.goto(x)
11     else:
12         turtle.goto(x, y)
13     turtle.down()
14
15 def triangle_dans_carre(long_carre):
16     """Fonction pour tracer un triangle à l'intérieur du carré"""
17     #On prend la position du curseur qui est sur le coin bas gauche
18     pos_coin_bg = turtle.position()
19     #On trace les deux traits restants, la base étant déjà faite
20     turtle.goto(pos_coin_bg[0]+long_carre/2,
21                 pos_coin_bg[1]+long_carre)
22     turtle.goto(pos_coin_bg[0]+long_carre, pos_coin_bg[1])
23
24 def carre_avec_triangle(longueur):
25     """Fonction pour tracer un carré avec un triangle à l'intérieur"""
26     for i in range(4):
27         turtle.forward(longueur)
28         turtle.left(90)
29         triangle_dans_carre(longueur)
30
31 if __name__ == "__main__":
32     #On initialise les longueurs du grand carré et des petits
33     carrés
34     longueur_1, longueur_2 = 150, 75
35     #On se positionne au coin bas gauche de notre futur grand carré
36     deplacer_sans_tracer(-longueur_1/2, -longueur_1/2)
37     #On le dessine
38     carre_avec_triangle(longueur_1)
39     #On prépare les valeurs des coin bas gauche des petits carrés
40     coins = [(longueur_1/2, longueur_1/2),
41              (-longueur_1/2-longueur_2, longueur_1/2),
42              (-longueur_1/2-longueur_2, -longueur_1/2-longueur_2),
43              (longueur_1/2, -longueur_1/2-longueur_2)]
44     #On dessine notre quatre petits carrés
45     for coin in coins:
46         deplacer_sans_tracer(coin)

```

3. Tracer et dessiner

```
44     carre_avec_triangle(longueur_2)
45     #On retourne au centre de notre dessin
46     deplacer_sans_tracer(0, 0)
47     #On prend la distance entre le centre et le coin par lequel le
        cercle passera
48     rayon = turtle.distance(longueur_1/2+longueur_2,
        longueur_1/2+longueur_2)
49     #On se déplace et on trace notre cercle
50     deplacer_sans_tracer(0, -rayon)
51     turtle.circle(rayon)
52     #On retourne à la maison, et on prend un angle de 90°
53     deplacer_sans_tracer(0, 0)
54     turtle.left(90)
55     turtle.exitonclick()
```

[Retourner au texte.](#)

Contenu masqué n°5

```
1  #!/usr/bin/env python3
2
3  import turtle
4  from random import randint
5
6  COULEURS = ['black', 'grey', 'brown', 'orange', 'pink', 'purple',
7             'red', 'blue', 'yellow', 'green']
8
9  if __name__ == "__main__":
10     turtle.setup(650, 100)
11     diametre = 15
12     turtle.up(); turtle.setx(-turtle.window_width()/2+2*diametre);
        turtle.down()
13     #Pour le nombre de couleurs disponibles
14     for i in range(len(COULEURS)):
15         #On choisit un couleur aléatoirement
16         index_choisi = randint(0, len(COULEURS)-1)
17         #On imprime un point de cette couleur
18         turtle.dot(diametre, COULEURS[index_choisi])
19         #On supprime la couleur choisie pour éviter de la rechoisir
20         del COULEURS[index_choisi]
21         #On met à jour le diamètre et on se déplace pour le
            prochain point
22         diametre += 5; turtle.up(); turtle.fd(1.5*diametre);
            turtle.down()
23     turtle.exitonclick()
```

[Retourner au texte.](#)

Contenu masqué n°6

```

1  #!/usr/bin/env python3
2
3  import turtle
4  from random import randint
5
6  LARGEUR, HAUTEUR = 640, 480
7  LONGUEUR_MIN, LONGUEUR_MAX = 5, 20
8
9  def deplacer_sans_tracer(x, y = None):
10     """Fonction pour se déplacer à un point sans tracer"""
11     turtle.up()
12     if (isinstance(x, tuple) or isinstance(x, list)) and len(x) ==
13         2:
14         turtle.goto(x)
15     else:
16         turtle.goto(x, y)
17     turtle.down()
18
19  def etoile(longueur):
20     """Fonction pour dessiner une étoile"""
21     turtle.setheading(180-2*72)
22     for i in range(5):
23         turtle.forward(longueur)
24         turtle.left(180-180/5)
25
26  if __name__ == "__main__":
27     turtle.setup(LARGEUR, HAUTEUR)
28     turtle.speed(0) #Met la vitesse de traçage la plus rapide
29     nb_etoiles, longueur_etoile = 20, 0
30     for i in range(nb_etoiles):
31         longueur_etoile = randint(LONGUEUR_MIN, LONGUEUR_MAX)
32         deplacer_sans_tracer(randint(-LARGEUR//2+LONGUEUR_MAX//2,
33             LARGEUR//2-LONGUEUR_MAX),
34                               randint(-HAUTEUR//2+LONGUEUR_MAX//2,
35             HAUTEUR//2-LONGUEUR_MAX))
36         etoile(longueur_etoile)
37     deplacer_sans_tracer(0, 0)
38     turtle.exitonclick()

```

[Retourner au texte.](#)

Contenu masqué n°7

```
1  #!/usr/bin/env python3
2
3  import turtle
4
5  def deplacer_sans_tracer(x, y = None):
6      """Fonction pour se déplacer à un point sans tracer"""
7      turtle.up()
8      if (isinstance(x, tuple) or isinstance(x, list)) and len(x) ==
9          2:
10         turtle.goto(x)
11     else:
12         turtle.goto(x, y)
13     turtle.down()
14
15 def carre(longueur):
16     """Fonction pour tracer un carré depuis le coin bas gauche"""
17     for nb_cote in range(4):
18         turtle.forward(longueur)
19         turtle.left(90)
20
21 def point_octogone(longueur, diametre = 5, couleur = 'black'):
22     """Fonction pour faire les points des sommets d'un octogone"""
23     for nb_cote in range(8):
24         turtle.dot(diametre, couleur)
25         turtle.up(); turtle.forward(longueur); turtle.down()
26         turtle.left(360/8)
27
28 if __name__ == "__main__":
29     #Deux carrés
30     longueurs_carre = [90, 180]
31     for longueur in longueurs_carre:
32         deplacer_sans_tracer(-longueur/2, -longueur/2)
33         carre(longueur)
34     #Cercle
35     rayon = turtle.distance(0,0) #nous sommes alors sur le coin
36     bas gauche
37     deplacer_sans_tracer(0, -rayon)
38     turtle.circle(rayon)
39     #Octogone de point
40     deplacer_sans_tracer(-1.25*rayon/2, -1.25*rayon-30)
41     point_octogone(1.25*rayon, 25, 'red')
42     #Retour maison
43     deplacer_sans_tracer(0, 0)
44     turtle.exitonclick()
```

[Retourner au texte.](#)

Contenu masqué n°8

```
1 #!/usr/bin/env python3
2
3 import turtle
4
5 LARGEUR, HAUTEUR = 640, 480
6
7 if __name__ == "__main__":
8     turtle.setup(LARGEUR, HAUTEUR)
9     turtle.speed("fast") #Met la vitesse de traçage à rapide
10    ecart = 4
11    for i in range(30):
12        turtle.stamp()
13        turtle.left(30)
14        turtle.up(); turtle.forward(ecart); turtle.down()
15        ecart += 3
16    turtle.exitonclick()
```

[Retourner au texte.](#)

Contenu masqué n°9

```
1 #!/usr/bin/env python3
2
3 import turtle
4
5 def courbe_koch(longueur, etape):
6     """Fonction récursive pour dessiner une courbe de Von Koch
7     (une fonction récursive étant une fonction s'appelant elle-même)"""
8     if etape == 0:
9         turtle.forward(longueur)
10    else:
11        courbe_koch(longueur/3, etape-1)
12        turtle.left(60)
13        courbe_koch(longueur/3, etape-1)
14        turtle.right(120)
15        courbe_koch(longueur/3, etape-1)
16        turtle.left(60)
17        courbe_koch(longueur/3, etape-1)
18
19 def flocon_koch(longueur, etape):
20     """Fonction pour dessiner un flocon de Von Koch
21     depuis le coin haut gauche"""
22     for i in range(3): #Pour chaque côté du triangle initial
```

3. Tracer et dessiner

```
23         courbe_koch(longueur, etape) #Courbe de Von Koch
24         turtle.right(120)
25
26 if __name__ == "__main__":
27     flocon_koch(100, 3)
```

[Retourner au texte.](#)

4. Colorier

À travers cette partie, nous allons voir comment rajouter des couleurs dans nos dessins.

4.1. Couleur de tracé et de remplissage

Nous avons vu dans la première partie portant sur les configurations que nous pouvons changer la couleur de fond en passant soit une chaîne de caractères (nom de la couleur ou bien code hexadécimal) soit un tuple représentant le code (R, G, B) avec des valeurs entre 0 et 1 par défaut. À travers les fonctions utilisées dans cette partie, nous choisirons les couleurs de cette même façon.

Tout d'abord, il faut savoir que notre crayon n'a pas une, mais deux couleurs : une couleur pour tracer ainsi qu'une couleur pour remplir.

4.1.0.1. Couleur du tracé

Pour modifier, la couleur de tracé, nous pouvons utiliser la fonction `pencolor` avec la couleur voulue. Si nous ne lui passons aucun paramètre, elle nous retourne la couleur de tracé actuelle. Voici un exemple d'utilisation :

```
1 turtle.forward(100) #Trace un trait noir de 100px
2 turtle.left(90)    #Tourne de 90°
3 turtle.pencolor("red") #Change la couleur de traçage à rouge
4 turtle.forward(100) #Trace un trait rouge de 100px
5 print(turtle.pencolor()) #Affiche 'red'
```

Nous pouvons nous amuser à faire un dégradé, comme l'illustre l'exemple ci-dessous. Dans celui-ci nous traçons une ligne pour chaque variation de rose allant de 0 à 255 inclus (que nous reportons sur une échelle de 1) en partant du bas et en remontant.

Le code :

👁 Contenu masqué n°10

4. Colorier

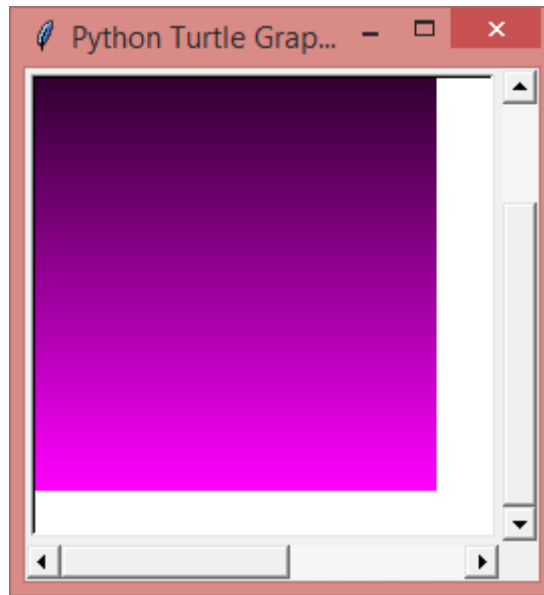


FIGURE 4.1. – Un dégradé de rose.

Si vous le souhaitez, vous pouvez vous entraîner en créant vos propres variations de dégradé (en changeant de sens, de couleur, etc.).

4.1.0.2. Couleur de remplissage

De la même manière que pour le tracé, nous pouvons modifier ou connaître la couleur de remplissage avec la fonction `fillcolor`. Toutefois, nous devons préciser avant de commencer à tracer si nous voulons remplir la figure avec `begin_fill`. De même, une fois celle-ci finie, il nous faut le préciser avec `end_fill` pour que le remplissage devienne actif. Sinon, il ne sera pas visible.

Si nous ne fermons pas notre figure en retournant au point de départ, `end_fill` se charge de la jonction entre notre position actuelle et le point de départ, avec un trait de la même couleur que la couleur de remplissage. Voici deux exemples pour comprendre ce fonctionnement :

```
1  ###Carré noir rempli de rouge
2  print(turtle.fillcolor()) #Affiche 'black'
3  turtle.fillcolor("red")  #Change la couleur de remplissage à rouge
4  turtle.begin_fill()      #Précise le début du remplissage
5  for i in range(4):
6      turtle.forward(120)
7      turtle.left(90)
8  turtle.end_fill()        #Précise la fin du remplissage
```

```
1  ###Moitié d'octogone en vert (sauf le dernier trait) rempli de
   jaune
2  turtle.pencolor("green")  #Change la couleur de traçage à vert
```

4. Colorier

```
3 turtle.fillcolor("yellow") #Change la couleur de remplissage à
   jaune
4 turtle.begin_fill() #Précise le début du remplissage
5 for i in range(4):
6     turtle.forward(75)
7     turtle.left(360/8)
8 turtle.end_fill() #Précise la fin du remplissage
```

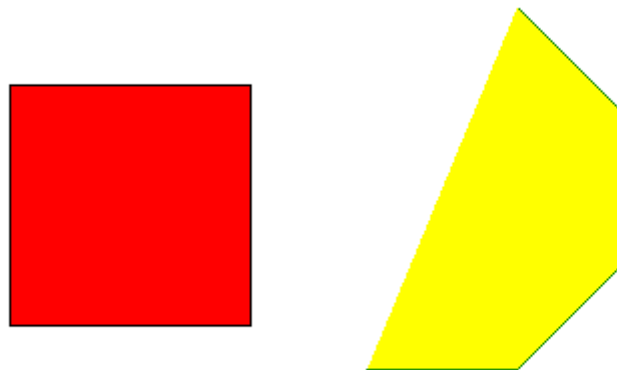


FIGURE 4.2. – Résultats des exemples avec fillcolor.

4.1.0.3. Changer les deux d'un coup

La fonction `color` nous permet de faire d'une pierre deux coups. En effet, nous pouvons lui passer la couleur de tracé ainsi que la couleur de remplissage. Si nous ne lui passons rien, elle nous retourne les valeurs actuelles. Si nous lui passons une seule valeur, alors la couleur de tracé et celle de remplissage prendront toutes deux celle-ci.

```
1 turtle.color("green") #Équivalent de turtle.pencolor("green") et
   turtle.fillcolor("green")
2 turtle.color("red", "yellow") #Équivalent de
   turtle.pencolor("red") et de turtle.fillcolor("yellow")
3 print(turtle.color()) #Affiche '('red', 'yellow')'
```

Vous remarquerez que nous pouvons connaître aisément ces informations en regardant notre crayon si celui-ci est affiché. En effet, la couleur de ses contours correspond à la couleur de tracé tandis que la couleur de son intérieur correspond à la couleur de remplissage.

Comme d'habitude, je vous encourage à tester ces fonctions avant de continuer. Nous allons maintenant créer notre propre fonction pour dessiner des points.

4.2. Notre fonction point

Si vous vous rappelez bien, nous avons vu la fonction `dot`, qui permet d'imprimer un point, au cours de la partie précédente. Or, avec ce que nous venons d'apprendre, nous sommes en mesure de la programmer ! En effet, un point n'est rien de plus qu'un cercle, d'un certain diamètre, tracé et rempli par une certaine couleur. Je vous rappelle que si nous ne passons aucune valeur à `dot`, la fonction affiche un point d'un diamètre par défaut et de la couleur de traçage actuel. Je vous rappelle aussi que le point est dessiné même si le crayon est levé, que le curseur est le centre du point, et que l'orientation n'influe pas sur le sens du traçage (pas comme avec `circle`). Avec ce que nous venons de dire, nous pouvons faire quelque chose comme :

Le **code** :

© Contenu masqué n°11

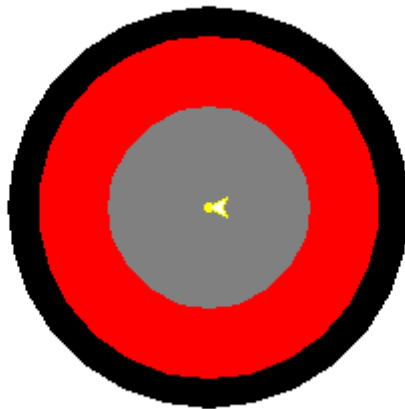


FIGURE 4.3. – Résultat des tests avec la fonction point.

Vous pouvez vous amuser à remplacer les appels à `point` par des appels à `dot` et vous verrez que le résultat est le même ! Toutefois, notre version n'est pas parfaitement identique puisque le diamètre par défaut est sans doute choisi d'une manière différente. De plus, nous ne prenons pas en compte la valeur de la largeur du crayon. Si vous le souhaitez, vous pouvez donc améliorer notre fonction afin qu'elle ait le même comportement que `dot` vis à vis de la taille de traçage (retournée par `pensize`).

4.3. Notre remplissage personnalisé

Nous avons vu que si nous ne fermons pas notre figure alors la fonction de fin de remplissage se charge de la jonction. Toutefois, comme nous l'avons aussi vu, ce dernier trait a la même couleur que celle de remplissage. Or, il serait mieux qu'il ait la couleur de traçage afin que notre

4. Colorier

figure soit harmonieuse. Nous pouvons résoudre ce problème aisément et c'est ce que nous allons faire à travers cette section.

Pour mener à bien notre mission, nous allons utiliser quelques fonctions. Tout d'abord, la fonction `begin_poly` qui nous permet de commencer à enregistrer par quel(s) point(s) passe notre figure (la position actuelle incluse). Ensuite, la fonction `end_poly` qui met fin à cet enregistrement. Enfin, nous utiliserons aussi `get_poly` qui nous retourne un tuple de points et nous permet d'avoir accès à ce que nous venons d'enregistrer. Ces trois fonctions ne prennent aucun paramètre. Voici un petit exemple :

```
1 turtle.begin_poly() #Commence à enregistrer
2 print(turtle.get_poly()) #Affiche '((0.00,0.00),)'
3 for i in range(3):
4     turtle.forward(90)
5     turtle.left(90)
6 turtle.end_poly() #Finit d'enregistrer
7 print(turtle.get_poly()) #Affiche '((0.00,0.00), (90.00,0.00),
    (90.00,90.00), (0.00,90.00))'
```

En définitive, cela va nous être très utile, car nous allons pouvoir savoir quel est le point de départ et tester si le point d'arrivée est identique à ce dernier dans le but savoir s'il faut que nous fassions la jonction ou non. En utilisant ce que l'on vient de dire, voici ce que nous pouvons coder, suivi de quelques explications :

Le **code** :

👁 Contenu masqué n°12

La fonction `notre_begin_fill` nous permet tout simplement de commencer à remplir ainsi qu'à enregistrer les points par lesquels nous passons. Ensuite, à travers `notre_end_fill`, nous terminons l'enregistrement, nous testons si le point actuel correspond au point d'arrivée afin de tracer le dernier trait si besoin et nous pouvons enfin terminer le remplissage.

Pour comparer nos deux points, nous devons être précautionneux. En effet, la comparaison directe de deux nombres flottants est déconseillée, car elle peut aboutir à des erreurs difficiles à cerner. Si besoin, vous pourrez en apprendre plus à travers les réponses de ce [sujet](#) .

C'est le moment de passer à la pratique !

4.4. TP : Atelier coloriage

Comme pour les travaux pratiques de la partie précédente, le but sera de coder le programme permettant d'arriver à l'image ci-dessous (encore une fois, ne vous préoccupez pas des longueurs). Veillez à colorier du plus gros au plus petit afin que le petit ne soit recouvert par le gros à chaque fois.

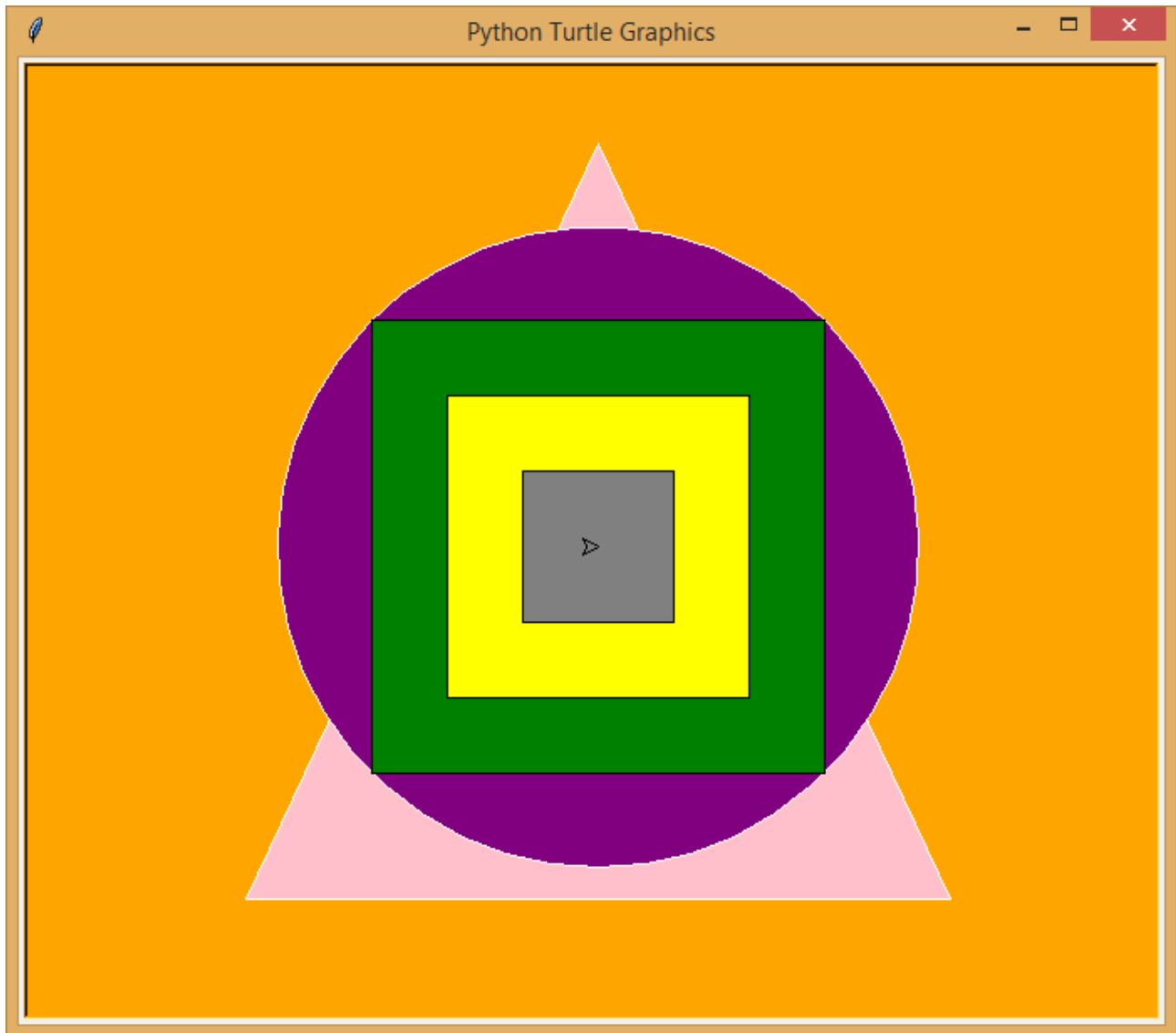


FIGURE 4.4. – Résultat à reproduire.

La **correction** :

👁 Contenu masqué n°13

Si vous souhaitez vous exercer davantage avec le coloriage, vous pouvez, par exemple, ajouter des couleurs au ciel étoilé que nous avons fait dans la partie précédente, ou vous pouvez aussi essayer de dessiner une voiture, une maison ou encore un robot.

Voilà, vous en savez désormais un peu plus sur turtle. Après la configuration et le dessin, nous allons voir comment interagir avec l'utilisateur, en lui demandant de saisir une valeur par exemple !

Contenu masqué

Contenu masqué n°10

```
1 #!/usr/bin/env python3
2
3 import turtle
4
5 LARGEUR, HAUTEUR = 256, 256
6
7 if __name__ == "__main__":
8     pos_y = -HAUTEUR/2 #On initialise la position en hauteur
9     initiale
10    turtle.setup(LARGEUR, HAUTEUR) #On initialise la fenêtre
11    turtle.speed(0) #On met la vitesse de traçage la plus rapide
12    #Pour val allant de 255 à 0 (couleur)
13    for val in range(255, -1, -1):
14        #On se déplace au début de la ligne à tracer
15        turtle.up(); turtle.goto(-LARGEUR, pos_y); turtle.down()
16        turtle.pencolor((val/255, 0, val/255)) #On prépare la
17        couleur
18        turtle.forward(LARGEUR) #On trace la ligne
19        pos_y += 1 #On remonte de 1px à chaque fois
20    turtle.exitonclick()
```

[Retourner au texte.](#)

Contenu masqué n°11

```
1 #!/usr/bin/env python3
2
3 import turtle
4
5 def deplacer_sans_tracer(x, y = None):
6     """Fonction pour se déplacer à un point sans tracer"""
7     turtle.up()
8     if (isinstance(x, tuple) or isinstance(x, list)) and len(x) ==
9         2:
10         turtle.goto(x)
11     else:
12         turtle.goto(x, y)
13         turtle.down()
14
15 def point(diametre = 5, couleur = None):
```

4. Colorier

```
15     """Fonction pour dessiner un point"""
16     #On récupère les valeurs courantes du crayon
17     etat_crayon = turtle.isdown()
18     orientation = turtle.heading()
19     position_crayon = turtle.position()
20     couleurs_crayon = turtle.color()
21
22     #On se prépare pour dessiner le point
23
24     '''Si une couleur est passée alors le point sera de cette couleur,
25     sinon le point aura la même couleur que la couleur de traçage. On vérifie
26     alors que la couleur de remplissage est bien la même que la couleur de
27     traçage'''
28     if couleur != None:
29         turtle.color(couleur, couleur)
30     elif couleurs_crayon[0] != couleurs_crayon[1]:
31         turtle.fillcolor(couleurs_crayon[0])
32
33     deplacer_sans_tracer(position_crayon[0],
34                           position_crayon[1]-diametre//2)
35     turtle.setheading(0)
36
37     if not etat_crayon:
38         turtle.down()
39
40     #On dessine le point
41     turtle.begin_fill()
42     turtle.circle(diametre//2)
43     turtle.end_fill()
44
45     #On remet les valeurs comme au début
46     deplacer_sans_tracer(position_crayon)
47     turtle.setheading(orientation)
48
49     if couleur != None:
50         turtle.color(couleurs_crayon[0], couleurs_crayon[1])
51     elif couleurs_crayon[0] != couleurs_crayon[1]:
52         turtle.fillcolor(couleurs_crayon[1])
53
54     if not etat_crayon:
55         turtle.up()
56         turtle.setheading(orientation)
57
58     #On teste notre fonction
59     if __name__ == "__main__":
60         point(200)
61         turtle.up()
62         point(170, 'red')
63         turtle.left(180)
64         turtle.color('grey', 'white')
```

4. Colorier

```
63     point(100)
64     turtle.pencolor('yellow')
65     point()
66     turtle.exitonclick()
```

[Retourner au texte.](#)

Contenu masqué n°12

```
1  #!/usr/bin/env python3
2
3  import turtle
4
5  EPSILON = 10**-10
6
7  def notre_begin_fill():
8      """Fonction pour commencer à remplir"""
9      turtle.begin_fill()
10     turtle.begin_poly()
11
12     def notre_end_fill():
13         """Fonction pour finir de remplir"""
14         turtle.end_poly()
15         poly = turtle.get_poly()
16         #On teste pour savoir si le point actuel est égal à celui de
            départ
17         if abs(poly[0][0]-poly[-1][0]) > EPSILON or
            abs(poly[0][1]-poly[-1][1]) > EPSILON:
18             turtle.goto(poly[0])
19         turtle.end_fill()
20
21     if __name__ == "__main__":
22         turtle.color("red", "green")
23         notre_begin_fill()
24         for i in range(4):
25             turtle.forward(200)
26             turtle.left(90)
27         notre_end_fill()
```

[Retourner au texte.](#)

Contenu masqué n°13

```

1  #!/usr/bin/env python3
2
3  import turtle
4
5  def deplacer_sans_tracer(x, y = None):
6      """Fonction pour se déplacer à un point sans tracer"""
7      turtle.up()
8      if (isinstance(x, tuple) or isinstance(x, list)) and len(x) ==
9          2:
10         turtle.goto(x)
11     else:
12         turtle.goto(x, y)
13     turtle.down()
14
15 def carre(longueur):
16     """Fonction pour tracer un carré depuis le coin bas gauche"""
17     for nb_cote in range(4):
18         turtle.forward(longueur)
19         turtle.left(90)
20
21 if __name__ == "__main__":
22     turtle.bgcolor("orange")
23     #Le triangle
24     deplacer_sans_tracer(-210, -210)
25     turtle.color("white", "pink") #pencolor("white");
26         fillcolor("pink")
27     turtle.begin_fill()
28     turtle.goto(210, -210)
29     turtle.goto(0, 240)
30     turtle.goto(-210, -210)
31     turtle.end_fill()
32     #Le cercle
33     deplacer_sans_tracer(0, 0)
34     rayon = turtle.distance(135, 135)
35     deplacer_sans_tracer(0, -rayon)
36     turtle.fillcolor("purple")
37     turtle.begin_fill()
38     turtle.circle(rayon)
39     turtle.end_fill()
40     #Les carrés
41     carres = [("green", 270), ("yellow", 180), ("grey", 90)]
42     for var_carre in carres:
43         deplacer_sans_tracer(-var_carre[1]/2, -var_carre[1]/2)
44         turtle.color("black", var_carre[0]) #pencolor("black");
45             fillcolor(var_carre[0])
46         turtle.begin_fill()
47         carre(var_carre[1])

```

4. Colorier

```
45         turtle.end_fill()
46     deplacer_sans_tracer(0, 0)
47     turtle.exitonclick()
```

[Retourner au texte.](#)

5. Interagir avec l'utilisateur

À ce stade, vous vous dites probablement qu'il manque quelque chose pour que nos programmes soient plus dynamiques. Il est donc temps de voir comment turtle nous permet d'interagir avec l'utilisateur !

5.1. Les saisies de l'utilisateur

Avec turtle, il existe deux fonctions pour demander à l'utilisateur de saisir des informations.

5.1.0.1. Demander une chaîne de caractères

Pour commencer, nous pouvons lui demander de saisir une chaîne de caractères avec `textinput`. Pour ce faire, il nous faut passer le titre ainsi que le texte de notre future popup à la fonction. En retour, cette dernière nous renvoie la saisie de l'utilisateur. Grâce à cela, nous pouvons connaître son nom par exemple :

```
1 nom = turtle.textinput("Nom",  
    "Veuillez saisir votre nom s'il vous plaît")  
2 print(nom) #Affiche le nom saisi
```

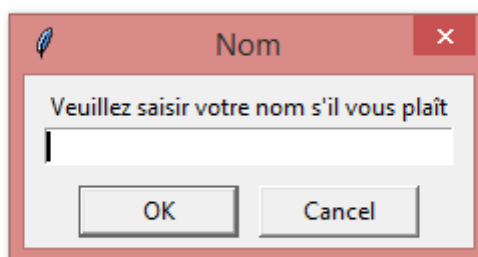


FIGURE 5.1. – La popup résultante.

5. Interagir avec l'utilisateur

5.1.0.2. Demander un nombre

Ensuite, nous pouvons aussi lui demander un nombre avec `numinput`. De la même manière que pour demander une chaîne de caractères, il faut indiquer le titre et le texte de notre future popup. En plus, nous avons la possibilité d'indiquer une valeur par défaut (*default*), une valeur minimum (*minval*) ainsi qu'une valeur maximum (*maxval*). La fonction retourne un nombre flottant. Voici quelques exemples :

```
1 age = int(turtle.numinput("Âge",
    "Veuillez saisir votre âge s'il vous plaît"))
2 age = int(turtle.numinput("Âge",
    "Veuillez saisir votre âge s'il vous plaît", 18))
3 age = int(turtle.numinput("Âge",
    "Veuillez saisir votre âge s'il vous plaît", minval = 5, maxval
    = 125))
```

Vous remarquerez que nous pourrions très bien utiliser `textinput` pour demander un nombre et ensuite convertir la valeur, mais il est plus sage et plus propre d'utiliser `numinput` qui est faite pour ça. En outre, cette dernière vérifie si la valeur saisie est bel et bien un nombre. En vérité, il faut toujours contrôler les données entrées par l'utilisateur pour s'assurer qu'elles sont en adéquation avec ce que nous attendons. Par exemple, lorsque nous lui avons demandé son nom, il aurait très bien répondre n'importe quoi comme "3.14" ou encore " $x = 9$ " et nous aurions affiché cela sans broncher ! Pire encore, il aurait pu cliquer sur `Cancel` et nous n'aurions eu aucune valeur !

Pour vous entraîner, vous pouvez par exemple demander successivement des valeurs à l'utilisateur comme un formulaire (prénom, nom, date de naissance, adresse, etc.) tout en vérifiant la véracité des saisies. Par exemple :

```
1 while True:
2     try :
3         nom = turtle.textinput("Nom",
            "Veuillez saisir votre nom s'il vous plaît")
4         if nom.isalpha() and len(nom) < 21:
5             break
6     except TypeError:
7         pass
8     print(nom)
```

Ici, nous demandons à l'utilisateur d'entrer son nom tant que nous ne jugeons pas sa saisie admissible. Pour ce faire, nous testons si le nom contient au moins une lettre et uniquement des lettres (et non pas des chiffres ou des caractères spéciaux) et nous vérifions que le nom n'excède pas une certaine longueur (20 caractères ici). De plus, nous nous servons de l'exception `TypeError` pour nous prémunir de tout clique sur le bouton `Cancel`. En effet, dans ce cas là, notre variable `nom` n'aura pas de valeur et donc pas de type (`NoneType`). En conséquence, les

méthodes que nous appliquons aux chaînes de caractères ne pourront pas s'appliquer ce qui lèvera une exception.

En plus des saisies, l'utilisateur peut interagir en provoquant des événements.

5.2. Les événements

Grâce à sa simplicité, turtle nous permet d'utiliser facilement les événements.

?

Mais en fait, qu'est-ce qu'un événement ?

Un événement est une information générée lorsque l'utilisateur utilise des périphériques spécifiques tel que le clavier ou encore la souris par exemple. Cela est très utile, car de cette manière l'utilisateur peut faire connaître ses décisions au programme. Par exemple, dans un jeu nous pouvons généralement nous diriger en appuyant sur les flèches ou encore nous pouvons tirer en faisant un clique gauche.

5.2.0.1. Écouter les événements du clavier

Pour commencer, nous devons utiliser la fonction `listen` pour pouvoir écouter et capturer les événements en provenance du clavier. Nous pouvons la placer n'importe où dans le code, mais il est plus propre de la placer avant de gérer les événements afin de mieux s'y retrouver.

```
1 turtle.listen()
2 ###Les événements du clavier
3 #...
```

5.2.0.2. La boucle d'événements

Les fonctions `mainloop` et `done` permettent de démarrer la boucle d'événements. Dans le monde des événements, nous utilisons une boucle infinie. De cette manière, le programme continue d'écouter les interactions jusqu'à ce que l'utilisateur informe qu'il a terminé en fermant la fenêtre par exemple. Dès lors, nous sortons de la boucle et le programme peut s'achever. Ces fonctions mettent en place cette boucle infinie pour nous. D'après la documentation, nous devons utiliser une de ces fonctions tout à la fin de notre programme turtle.

```
1 ###Le code
2 #....
3 turtle.mainloop() #Ou bien turtle.done()
```


5.2.0.3. Le clavier

Commençons avec les événements en provenance du clavier. Il en existe deux types.

5.2.0.3.1. Appuyer sur une touche Tout d'abord, une touche peut être pressée et c'est la fonction `onkeypress` qui nous permet de le savoir. En premier paramètre, nous devons lui passer une fonction qui sera exécutée lors de l'appui. En second paramètre, nous pouvons lui indiquer à quelle touche nous souhaitons associer la fonction sous la forme d'une chaîne de caractères : `"space"` pour la touche `[Espace]` par exemple. Si nous omettons ce second paramètre, alors la fonction sera attribuée à toutes les touches libres.



Pour l'appui, une touche ne peut être associée qu'à une seule fonction.

Avec l'exemple ci-dessous, *"Touche 'a' pressée !"* est affiché lorsque nous pressons la touche `[A]` et *"Touche pressée !"* est affiché lorsque nous appuyons sur une autre touche. Si nous avions inversé les lignes avec les `onkeypress`, nous aurions eu le même comportement puisque la touche `[A]` étant déjà associée à une fonction, elle n'aurait pas été prise en compte par le second appel.

```
1 def appui_sur_a():
2     print("Touche 'a' pressée !")
3
4 def appui_quelconque():
5     print("Touche pressée !")
6
7 if __name__ == "__main__":
8     turtle.listen()
9     turtle.onkeypress(appui_quelconque) #Toutes les touches libres
        sont associées à appui_quelconque
10    turtle.onkeypress(appui_sur_a, "a") #La touche A est désormais
        associée à appui_sur_a
11    turtle.mainloop()
```

5.2.0.3.2. Relâcher une touche Ensuite, une touche peut être relâchée. En passant à `onkeyrelease` une fonction ainsi qu'une touche (ce paramètre **n'est pas facultatif** ici contrairement à `onkeypress`), nous pouvons gérer cela.



Pour le relâchement, une touche ne peut être associée qu'à une fonction.

```
1 def relachement_a():
2     print("Touche 'a' relâchée !")
```

```
3
4 def relachement_haut():
5     print("Touche 'Up' relâchée !")
6
7 if __name__ == "__main__":
8     turtle.listen()
9     turtle.onkeyrelease(relachement_haut, "a") #La touche "A" est
        associée à relachement_haut
10    turtle.onkeyrelease(relachement_a, "a") #La touche "A" est
        maintenant associée à relachement_a
11    turtle.onkeyrelease(relachement_haut, "Up") #La touche "Haut"
        est associée à relachement_haut
12    turtle.mainloop()
```

Ainsi, lorsque nous relâchons la touche **A**, "Touche 'a' relâchée !" est affiché, et lorsque nous relâchons la touche **Haut**, c'est "Touche 'Up' relâchée !" qui est affiché.

5.2.0.3.3. Exemple : une balle qui roule En mettant en application ce que nous venons de voir, nous pouvons réaliser un code basique, permettant de déplacer une balle à l'écran. L'appuie sur les flèches permet de se diriger et la touche **Espace**, lorsqu'elle est relâchée, permet de réinitialiser la position.

Le code :

👁 Contenu masqué n°14

Dans la boucle principale, nous écoutons les événements en provenance du clavier. Ensuite, nous prenons en compte ceux qui nous intéressent en utilisant des **lambda** afin de passer des paramètres aux fonctions. Enfin, nous terminons en démarrant la boucle d'événements. Les scintillements que vous pourrez voir en testant sont explicables par l'utilisation de la fonction **clear** lorsque l'on nettoie l'écran pour afficher la balle.

Vous pouvez améliorer le programme en demandant à l'utilisateur de choisir le diamètre ainsi que la couleur de la balle ou encore en veillant à ce que la balle ne puisse pas sortir de l'écran, par exemple.



FIGURE 5.2. – Une balle qui peut bouger.

5.2.0.4. La souris

La souris peut aussi générer des événements (bouton cliqué, bouton relâché, souris déplacée). Avec turtle, nous pouvons prendre en compte l'appui sur un bouton de la souris. Si vous vous rappelez, nous utilisons déjà cette source d'événements lorsque nous utilisons `exitonclick` pour quitter notre programme avec un clique gauche.

`onscreenclick` est une fonction pour capturer l'appui sur un bouton de la souris. Nous devons lui passer la fonction qui sera associée au clique. Ensuite, nous pouvons lui passer à quel bouton sera associé la fonction (par défaut le clique gauche) et enfin nous avons la possibilité de choisir si la fonction s'ajoutera à (avec `True`) ou remplacera (avec `False` ou rien) la ou les fonctions déjà associée(s). De cette manière, un même clique pourra déclencher plusieurs fonctions.

Notons que les coordonnées du clique seront automatiquement passées à la fonction c'est pourquoi vous devez les prendre en compte dans la définition de votre fonction sans quoi vous aurez un joli message d'erreur. On peut ainsi se déplacer au point cliqué par exemple :

```
1 turtle.up()
2 turtle.onscreenclick(turtle.goto) #Se déplace au point cliqué
3 turtle.mainloop()
```

Le tableau ci-dessous contient les valeurs, correspondant aux boutons, que vous pouvez passer.

Valeur	Bouton
1	Gauche
2	Central
3	Droit

Enfin, voici un petit exemple comme d'habitude pour bien comprendre le comportement de cette fonction :

```
1 def toto(x, y):
2     print("toto : {}, {}".format(x, y))
3
4 def tutu(x, y):
5     print("tutu : {}, {}".format(x, y))
6
7 def tata(x, y):
8     print("tata : {}, {}".format(x, y))
9
10 def titi(x, y):
11     print("titi : {}, {}".format(x, y))
```

```
12
13 def tete(x, y):
14     print("tete : {}, {}".format(x, y))
15
16 if __name__ == "__main__":
17     #Pas besoin de turtle.listen() vu que l'on ne s'occupe pas du
        clavier
18
19     #Clique gauche : juste toto au final
20     turtle.onscreenclick(titi, 1)
21     turtle.onscreenclick(toto, 1) # <=>
        turtle.onscreenclick(toto, 1, False)
22
23     #Clique central : juste tata
24     turtle.onscreenclick(tata, 2)
25
26     #Clique droit : tete et tutu
27     turtle.onscreenclick(tete, 3)
28     turtle.onscreenclick(tutu, 3, True)
29
30     turtle.mainloop()
```

Comme expliqué dans le code, on associe au clique gauche les fonctions `titi` et `toto` (la première étant alors remplacée par la seconde). Ensuite on associe au clique central, la fonction `tata`. Enfin, on associe les fonctions `tete` et `tutu` au clique droit. À chaque fois, on affiche les coordonnées du curseur de la souris lors du clique.

Encore une fois, n'hésitez pas à tester les fonctions que nous avons vues. Nous allons maintenant voir comment écrire dans la fenêtre.

5.3. L'écriture à l'écran

En écrivant un message à l'utilisateur, nous pouvons l'informer ou lui demander une information (c'est ce que nous avons fait avec les saisies par exemple).

Bien que nous pourrions afficher nos messages dans la console grâce à `print`, turtle nous offre `write`, une fonction pour écrire directement dans la fenêtre. Pour ce faire, il suffit de lui passer le texte à écrire. Notons au passage que la couleur d'écriture sera celle de tracé. En plus de cela, nous pouvons choisir si le crayon reprendra sa position préalable ou sera dans le coin bas droit du texte écrit respectivement avec les valeurs `False` et `True` (paramètre *move*). Nous pouvons aussi indiquer comment sera aligné le texte par rapport à la position du crayon (*align*) ainsi que la police utilisée (*font*). Cette dernière se décompose en un nom de police, une taille de police ainsi qu'un formatage de texte. Voici un exemple :

```
1 turtle.write("Bonjour !", True) #Écrit et déplace le crayon à la
    fin du texte
```

5. Interagir avec l'utilisateur

```
2 turtle.write("Salut !", align = "left") #Écrit à droite du crayon
3 turtle.write("Coucou !", font = ("Arial", 15, "bold")) #Écrit
   selon une police
```



Si nous donnons la valeur `True` au paramètre `move`, un trait sera tracé sous notre texte dû au déplacement que notre crayon soit baissé ou levé.

Voici ci-dessous un tableau avec les alignements par rapport au curseur possibles :

Valeur	Alignement
"left"	Droit
"center"	Centré
"right"	Gauche



Oui, vous avez bien lu. Les alignements gauche et droite sont inversés par rapport à ce que nous aurions pu penser. Pour bien comprendre, imaginez-vous que pour écrire, le crayon regarde dans votre direction. Ainsi, quand il se déplacera à la gauche du point d'alignement, pour vous il ira vers la droite, et vice versa.

Voici ci-dessous un autre tableau, contenant les formatages possibles :

Valeur	Formatage
"normal"	Normal
"bold"	Gras
"italic"	Italique
"underline"	Souligné

Dans l'exemple ci-dessous suivi d'un résultat en image, nous écrivons aléatoirement dans la fenêtre chaque fruit de notre dictionnaire. Pour sa couleur, nous prenons la valeur associée tandis que nous choisissons de manière aléatoire les autres caractéristiques du texte (alignement, taille et formatage). En définitive, c'est presque le même procédé qu'avec notre exemple avec les points, si vous vous rappelez.

Le **code** :

👁 Contenu masqué n°15



FIGURE 5.3. – De quoi faire un bon smoothie.

Découvrons à présent une nouvelle notion : les timers.

5.4. L'utilisation de timers

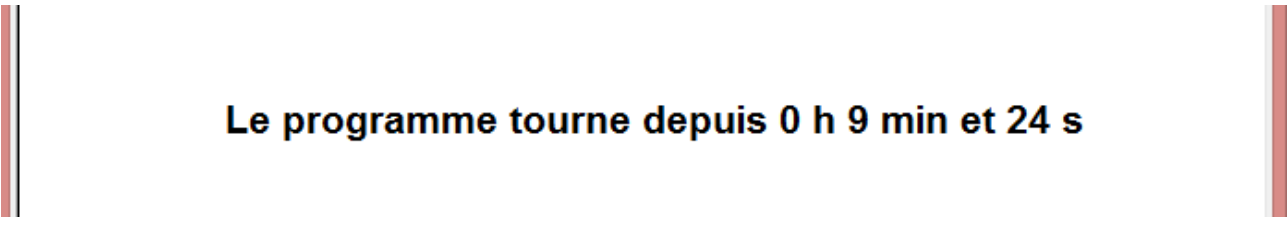
Avec la fonction `ontimer`, nous avons la possibilité de faire appel à une fonction (que l'on passe en premier paramètre) après une durée exprimée en millisecondes (que l'on passe en second paramètre, la valeur par défaut étant 0) :

```
1 turtle.ontimer(lambda : print("tutu"), 2000) #Affiche 'tutu' après
    deux secondes, donc après le 'toto'
2 turtle.ontimer(lambda : print("toto")) #Affiche 'toto'
    immédiatement
```

Notons au passage que l'utilisation de cette fonction n'est pas bloquante puisque 'toto' est bien affiché en premier malgré le fait que l'instruction soit après celle affichant 'tutu'.

Dans le programme ci-dessous, en affichant le temps écoulé depuis le lancement du programme, nous utilisons cette fonction de deux manières différentes. Dans un premier temps, nous nous en servons pour commencer à incrémenter après une seconde écoulée. Puis, dans `incrimente_temps`, nous nous en servons de sorte que la fonction s'appelle elle-même toutes les secondes pour continuer à incrémenter. C'est une sorte de boucle infinie que l'on peut rompre avec la croix rouge ou avec un clic gauche : en fermant la fenêtre.

Le `code` :

A screenshot of a program's output. It features a black background with a red vertical bar on the left and a red vertical bar on the right. In the center, the text "Le programme tourne depuis 0 h 9 min et 24 s" is displayed in a white, monospace-style font.

Le programme tourne depuis 0 h 9 min et 24 s

FIGURE 5.4. – Résultat après un certain temps...

L'utilisation de timers nous ouvre de nouveaux champs de possibilités. Sur ce, nous allons passer à la pratique !

5.5. TP : Le jeu des allumettes

Grâce à ce que nous venons d'apprendre, nous allons maintenant réaliser notre premier jeu avec turtle : le jeu des allumettes !

Il existe plusieurs variantes de ce jeu qui voit deux joueurs s'affronter. Nous partirons du principe qu'une partie commence avec 19 allumettes. Chacun leur tour, les joueurs retirent 1, 2 ou 3 allumettes. Un joueur a perdu lorsque son tour arrive et qu'il ne reste plus qu'une allumette sur le plateau.

Pour notre version, les joueurs choisiront combien d'allumettes ils souhaitent retirer en relâchant les touches `a`, `z` ou `e` pour respectivement 1, 2 ou 3 allumettes. Pour quitter, il suffira de faire un clic droit. Nous afficherons les informations essentielles telles que le nombre d'allumettes restantes, le joueur courant, l'état de la partie et éventuellement le nombre d'allumettes retirées.

Désormais vous avez toutes les clefs en main pour réaliser cet exercice. Si vous ne savez pas par où commencer, essayez de réfléchir à ce que doit faire votre programme et de définir les fonctions que vous avez besoin de programmer. Si besoin, n'hésitez pas à relire cette partie ou à poser vos questions sur le forum.

Enfin, les captures d'écran ci-dessous peuvent vous inspirer si besoin :

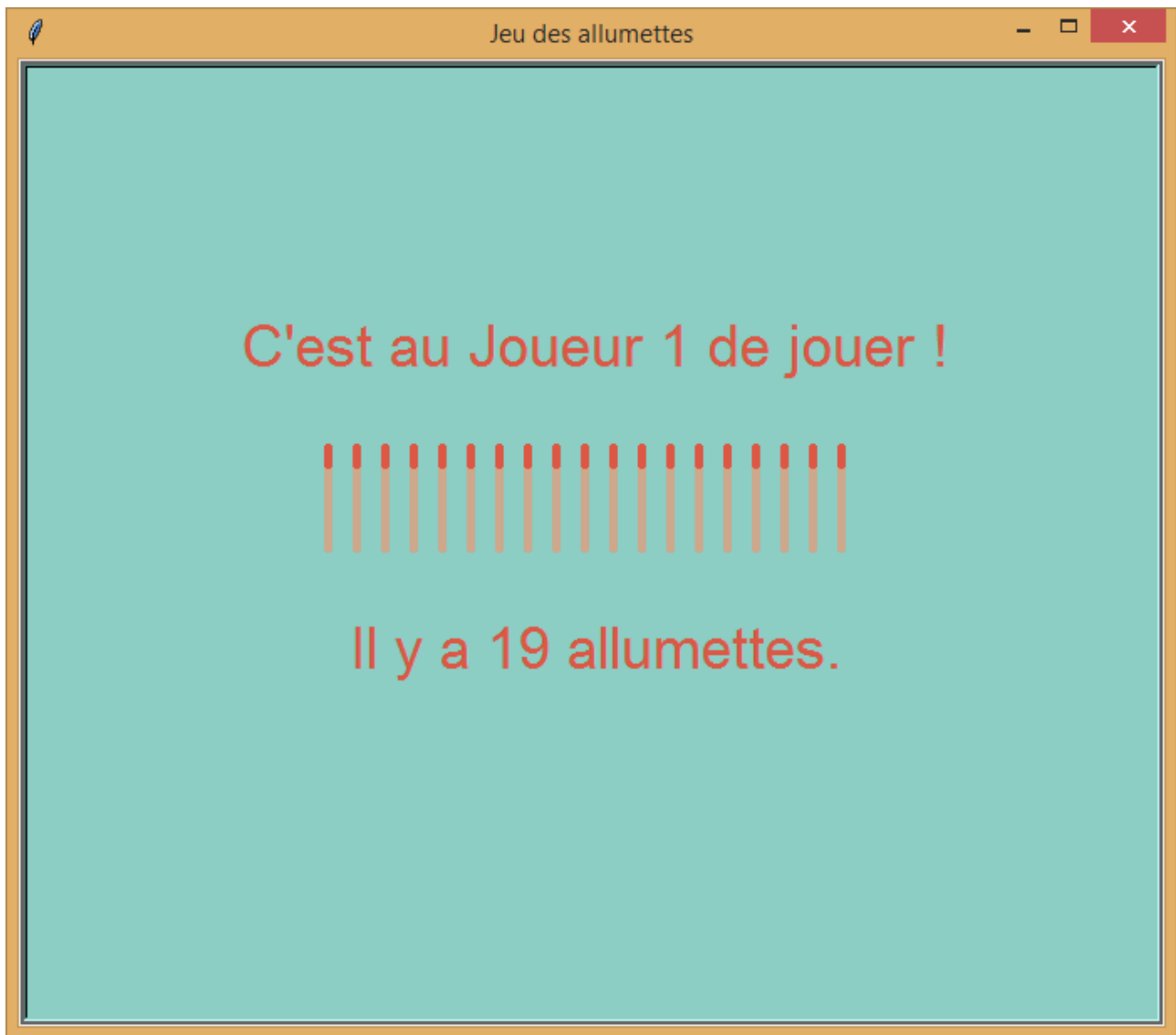


FIGURE 5.5. – Le jeu des allumettes. (1)

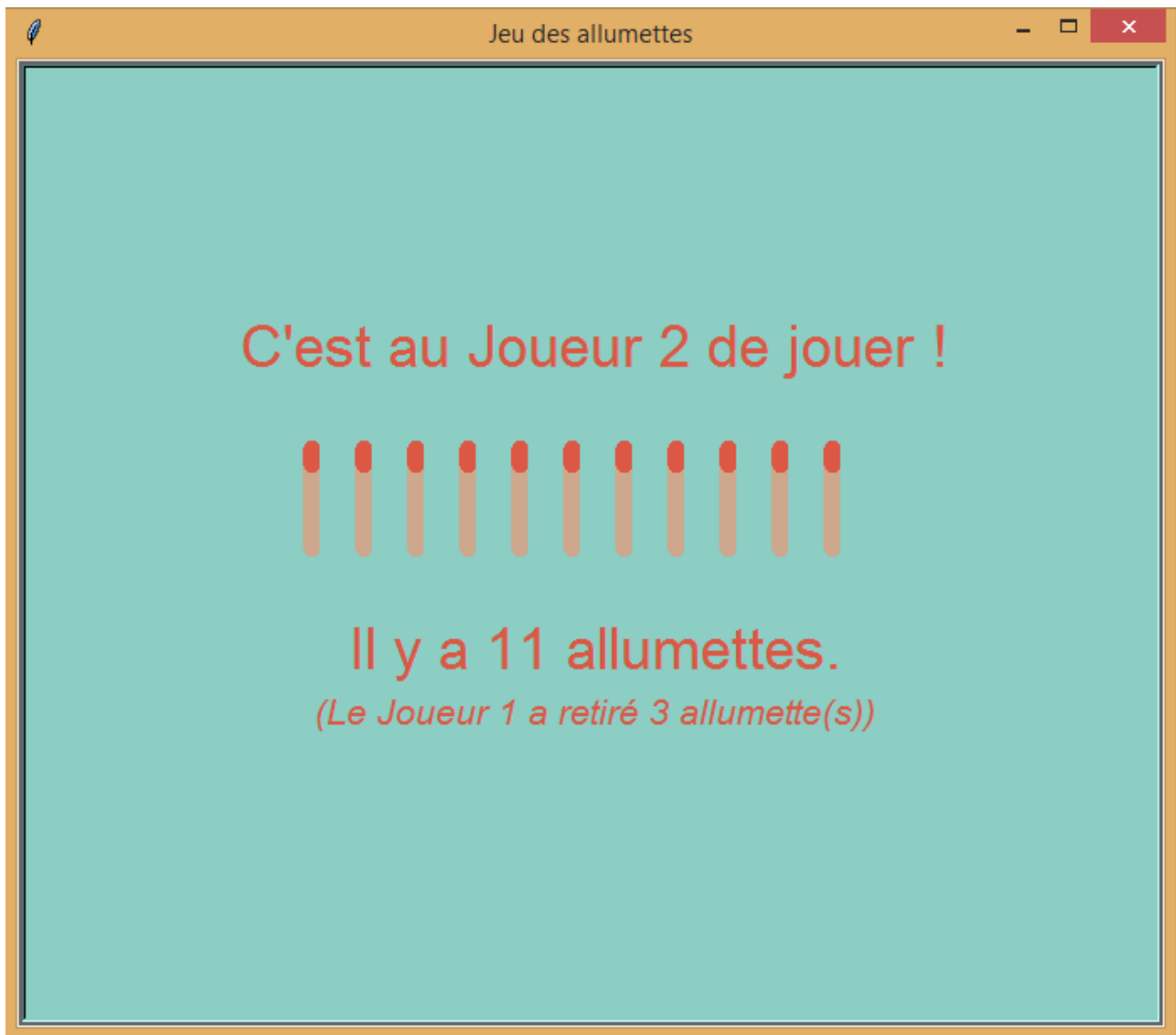


FIGURE 5.6. – Le jeu des allumettes. (2)



FIGURE 5.7. – Le jeu des allumettes. (3)

La **correction** :

👁 Contenu masqué n°17

Comme vous pouvez le voir avec les noms et les commentaires, le but de chaque fonction est assez explicite. Ici, j'ai particulièrement fait attention à désactiver les événements en provenance du clavier lorsque des traitements étaient en cours (que la partie était en train d'être affichée par exemple), sinon il y aurait eu des problèmes vu que la fonction `joue` aurait été appelée plusieurs fois de suite. Comme vous pouvez le tester, on relâche bien les touches selon notre choix et le clic droit permet de quitter en utilisant la fonction de turtle permettant de clore la fenêtre : `bye`.

Vous pouvez améliorer le programme de diverses façons, en rajoutant une `IA` et en demandant au joueur s'il souhaite affronter un joueur humain ou celle-ci par exemple, ou encore en demandant aux joueurs de saisir leur pseudo.

Voilà, avec cette dynamique nouvelle, nos programmes continuent de s'enrichir. C'est le moment de voir comment paramétrer encore plus notre fenêtre.

Contenu masqué

Contenu masqué n°14

```
1  #!/usr/bin/env python3
2
3  import turtle
4
5  def deplacer_sans_tracer(x, y = None):
6      """Fonction pour se déplacer à un point sans tracer"""
7      turtle.up()
8      if (isinstance(x, tuple) or isinstance(x, list)) and len(x) ==
9          2:
10         turtle.goto(x)
11     else:
12         turtle.goto(x, y)
13     turtle.down()
14
15 def affiche_point(diametre, couleur):
16     """Fonction pour afficher seulement la balle"""
17     deplacer_sans_tracer(pos_x, pos_y)
18     turtle.clear()
19     turtle.dot(diametre, couleur)
20
21 def reinit_pos():
22     """Fonction pour réinitialiser la position de la balle"""
23     global pos_x, pos_y
24     pos_x = pos_y = 0
25     affiche_point(diametre_balle, couleur_balle)
26
27 def deplace_x(num = 0):
28     """Fonction pour changer la coordonnée x de la balle"""
29     global pos_x
30     pos_x += num
31     affiche_point(diametre_balle, couleur_balle)
32
33 def deplace_y(num = 0):
34     """Fonction pour changer la coordonnée y de la balle"""
35     global pos_y
36     pos_y += num
37     affiche_point(diametre_balle, couleur_balle)
```

```
38 if __name__ == "__main__":
39     turtle.speed(0)
40     pos_x, pos_y = 0, 0
41     diametre_balle, couleur_balle = 20, 'red'
42
43     turtle.hideturtle()
44     affiche_point(diametre_balle, couleur_balle)
45
46     turtle.listen() #Pour écouter
47     turtle.onkeypress(lambda : deplace_x(-10), "Left") #Touche
         gauche
48     turtle.onkeypress(lambda : deplace_x(10), "Right") #Touche
         droite
49     turtle.onkeypress(lambda : deplace_y(10), "Up") #Touche haut
50     turtle.onkeypress(lambda : deplace_y(-10), "Down") #Touche bas
51     turtle.onkeyrelease(reinit_pos, "space") #Touche espace
52     turtle.mainloop() #Pour démarrer la boucle d'événements
```

[Retourner au texte.](#)

Contenu masqué n°15

```
1 #!/usr/bin/env python3
2
3 import turtle
4 from random import randint
5
6 LARGEUR = HAUTEUR = 400
7
8 ALIGNEMENTS = ["left", "center", "right"]
9 FORMATAGES = ["normal", "bold", "italic", "underline"]
10 FRUITS = {"Clémentine" : "orange", "Ananas" : "brown", "Pomme" :
    "green",
11         "Poire" : "lightgreen", "Banane" : "yellow", "Orange" :
    "orange",
12         "Cerise" : "darkred", "Abricot" : "orange", "Kiwi" :
    "green",
13         "myrtille" : "darkblue", "prune" : "blue"}
14
15 def deplacer_sans_tracer(x, y = None):
16     """Fonction pour se déplacer à un point sans tracer"""
17     turtle.up()
18     if (isinstance(x, tuple) or isinstance(x, list)) and len(x) ==
        2:
19         turtle.goto(x)
20     else:
```

5. Interagir avec l'utilisateur

```
21         turtle.goto(x, y)
22         turtle.down()
23
24     if __name__ == "__main__":
25         turtle.setup(LARGEUR+50, HAUTEUR+50)
26         for fruit, couleur in FRUITS.items():
27             deplacer_sans_tracer(randint(-LARGEUR//2.5, LARGEUR//2.5),
28                                   randint(-HAUTEUR//2.5,
29                                           HAUTEUR//2.5))
29
30         turtle.pencolor(couleur)
31         turtle.write(fruit, align = ALIGNEMENTS[randint(0,
32                                                         len(ALIGNEMENTS)-1)],
33                       font = ("Arial", randint(14, 30),
34                              FORMATAGES[randint(0,
35                                                    len(FORMATAGES)-1)]))
```

[Retourner au texte.](#)

Contenu masqué n°16

```
1  #!/usr/bin/env python3
2
3  import turtle
4
5  def ecrit_temps():
6      """Fonction pour écrire le temps écoulé"""
7      turtle.clear()
8      turtle.write("Le programme tourne depuis "\
9                   + "{} h {} min et {} s".format(heure, minute,
10                                                    seconde),
11                  align = "center",
12                  font = ("Arial", 16, "bold"))
13
14  def incremente_temps():
15      """Fonction pour incrémenter le temps"""
16      global heure, minute, seconde
17      seconde += 1
18      if seconde == 60:
19          minute += 1
20          seconde = 0
21      if minute == 60:
22          heure += 1
23          minute = 0
24      ecrit_temps()
25      turtle.ontimer(incremente_temps, 1000) #Fait appel à elle-même
26      toutes les secondes
```

```
25
26 if __name__ == "__main__":
27     turtle.speed(0)
28     turtle.hideturtle()
29     heure = minute = seconde = 0
30     ecrit_temps()
31     turtle.ontimer(incremente_temps, 1000) #Attend une seconde
        puis commence à incrémenter
32     turtle.exitonclick()
```

[Retourner au texte.](#)

Contenu masqué n°17

```
1  #!/usr/bin/env python3
2
3  import turtle
4  import sys
5
6  #Les constantes
7  NOMBRE_ALLUMETTE = 19
8  HAUTEUR_BOIS_ALLUMETTE = 50
9  HAUTEUR_ROUGE_ALLUMETTE = 10
10 COULEUR_BOIS_ALLUMETTE = "#CDA88C"
11 COULEUR_ROUGE_ALLUMETTE = "#DC5844"
12 COULEUR_FOND = "#8CCDC4"
13 TITRE = "Jeu des allumettes"
14 TAILLE_ECRITURE = 26
15 TAILLE_ECRITURE_2 = 16
16
17 #Les autres variables
18 etat_partie = True
19 nombre_allumettes = NOMBRE_ALLUMETTE
20 joueur_courant = 1
21
22 #Les fonctions
23 def deplacer_sans_tracer(x, y = None):
24     """Fonction pour se déplacer à un point sans tracer"""
25     turtle.up()
26     if (isinstance(x, tuple) or isinstance(x, list)) and len(x) ==
        2:
27         turtle.goto(x)
28     else:
29         turtle.goto(x, y)
30     turtle.down()
31
```

```

32 def initialise_fenetre():
33     """Fonction pour initialiser la fenêtre"""
34     turtle.hideturtle()
35     turtle.setheading(90)
36     turtle.title(TITRE)
37     turtle.bgcolor(COULEUR_FOND)
38     turtle.speed(0)
39
40 def dessiner_allumette():
41     """Fonction pour dessiner une allumette"""
42     turtle.pencolor(COULEUR_BOIS_ALLUMETTE)
43     turtle.forward(HAUTEUR_BOIS_ALLUMETTE)
44     turtle.pencolor(COULEUR_ROUGE_ALLUMETTE)
45     turtle.forward(HAUTEUR_ROUGE_ALLUMETTE)
46
47 def dessiner_allumettes(nombre_allumettes):
48     """Fonction pour dessiner les allumettes"""
49     espace_entre_allumettes = 60 if nombre_allumettes < 8 else
        turtle.window_width()/2//nombre_allumettes
50     taille_crayon = 25 if nombre_allumettes < 8 else
        espace_entre_allumettes//3
51     turtle.pensize(taille_crayon)
52     position_allumettes =
        [-nombre_allumettes/2*espace_entre_allumettes, 0]
53     deplacer_sans_tracer(position_allumettes)
54     for allumette in range(nombre_allumettes):
55         dessiner_allumette()
56         position_allumettes[0] += espace_entre_allumettes
57         deplacer_sans_tracer(tuple(position_allumettes))
58     if nombre_allumettes != 1:
59         afficher_nombre_allumettes(nombre_allumettes)
60
61 def afficher_partie(nombre_allumettes, joueur_courant,
        nombre_retrees = None):
62     """Fonction pour afficher la partie et son état"""
63     turtle.clear()
64     dessiner_allumettes(nombre_allumettes)
65     afficher_qui_joue(joueur_courant)
66     if nombre_retrees != None:
67         joueur = 1 if joueur_courant == 2 else 2
68         affiche_nombre_retre(joueur, nombre_retrees)
69
70 def affiche_nombre_retre(joueur, nombre_retrees, pos = (0,
        -110)):
71     """Fonction pour afficher le nombre d'allumettes retirées"""
72     deplacer_sans_tracer(pos)
73
74     turtle.write("(Le Joueur {} a retiré {} allumette(s))".format(joueur,
        nombre_retrees),
        align = "center",

```

```

75         font = ("Arial", TAILLE_ECRITURE_2, "italic"))
76
77 def afficher_nombre_allumettes(nombre_allumettes, pos = (0, -80)):
78     """Fonction pour afficher le nombre d'allumettes"""
79     deplacer_sans_tracer(pos)
80     turtle.write("Il y a {} allumettes.".format(nombre_allumettes),
81                 align = "center",
82                 font = ("Arial", TAILLE_ECRITURE, "normal"))
83
84 def afficher_qui_joue(joueur_courant, pos = (0, 100)):
85     """Fonction pour afficher qui joue"""
86     deplacer_sans_tracer(pos)
87
88     turtle.write("C'est au Joueur {} de jouer !".format(joueur_courant),
89                 align = "center",
90                 font = ("Arial", TAILLE_ECRITURE, "normal"))
91
92 def bloque_clavier():
93     """Fonction pour désactiver les actions des touches a, z, e"""
94     turtle.onkeyrelease(None, "a")
95     turtle.onkeyrelease(None, "z")
96     turtle.onkeyrelease(None, "e")
97
98 def debloque_clavier():
99     """Fonction pour associer les touches au nombre retiré"""
100     turtle.onkeyrelease(lambda : joue(1), "a")
101     turtle.onkeyrelease(lambda : joue(2), "z")
102     turtle.onkeyrelease(lambda : joue(3), "e")
103
104 def joue(nombre_retire = 1):
105     """Fonction pour prendre en compte le choix du joueur"""
106     bloque_clavier()
107     global nombre_allumettes, etat_partie, joueur_courant
108     if nombre_retire != 0 and nombre_allumettes-nombre_retire > 0:
109         nombre_allumettes -= nombre_retire
110     else:
111         debloque_clavier()
112         return
113     if nombre_allumettes != 1:
114         joueur_courant = 1 if joueur_courant == 2 else 2
115         afficher_partie(nombre_allumettes, joueur_courant,
116                         nombre_retire)
117     else:
118         etat_partie = victoire(joueur_courant)
119         if not etat_partie:
120             quitter()
121             nombre_allumettes = NOMBRE_ALLUMETTE
122             afficher_partie(nombre_allumettes, joueur_courant)
123             turtle.listen()
124             debloque_clavier()

```



```
123
124 def victoire(joueur_courant):
125     """Fonction pour le déroulement de la victoire"""
126     turtle.clear()
127     dessiner_allumettes(1)
128     deplacer_sans_tracer(-35, -100)
129     turtle.down()
130     turtle.write("Le joueur "+str(joueur_courant)+" a gagné !",
131                 align = "center", font = ("Arial", TAILLE_ECRITURE,
132                 "normal"))
131     if (turtle.textinput("Rejouer ?",
132                           "Rejouer ? Veuillez entrer 'oui' si c'est le cas.") ==
133         'oui'):
132         return True
133     return False
134
135 def quitter(x = 0, y = 0):
136     """Fonction pour quitter le jeu et fermer le programme"""
137     turtle.bye()
138     sys.exit(0)
139
140 def main():
141     """Fonction principale"""
142     initialise_fenetre()
143     afficher_partie(nombre_allumettes, joueur_courant)
144     turtle.listen()
145     debloque_clavier()
146     turtle.onscreenclick(quitter, 3)
147
148 if __name__ == "__main__":
149     main()
150     turtle.mainloop()
```

[Retourner au texte.](#)

6. Aller plus loin dans les configurations

En plus de tout cela, turtle nous offre de quoi personnaliser davantage notre fenêtre.

6.1. Paramétrer le repère

Comme vous le savez désormais, le repère est initialement centré dans la fenêtre et ses limites à l'intérieur de celle-ci dépendent de la largeur et de la hauteur. Qu'on se le dise, ce n'est pas toujours très pratique !

Or justement, la fonction `setworldcoordinates`, nous permet de configurer le repère en lui donnant les coordonnées du point bas gauche et celle du coin haut droit du canvas. Notons que la maison reste le point (0, 0) quoiqu'il advienne. Voici des exemples à tester, qui seront sans doute plus parlants :

```
1 setworldcoordinates(0, 0, 100, 100) #L'origine du repère est dans
   le coin bas gauche
2 setworldcoordinates(-50, -50, 50, 50) #L'origine du repère est
   centré dans la fenêtre
3 setworldcoordinates(-100, 0, 0, 100) #L'origine du repère est dans
   le coin bas droit
4 setworldcoordinates(-100, -100, -50, -50) #L'origine du repère est
   en dehors de la fenêtre
```

Au passage, la fonction effectue un `reset`. Ainsi, s'il a des tracés, ceux-ci seront perdus à moins que nous soyons en mode *"world"* (nous étudierons cela un peu plus loin) auquel cas ils seront redessinés d'après les nouvelles coordonnées.

En paramétrant notre repère, il nous sera donc plus simple nous y déplacer. Voyons maintenant comment personnaliser notre crayon.

6.2. Paramétrer le crayon

Il existe différentes façons de personnaliser notre crayon.

6. Aller plus loin dans les configurations

6.2.0.1. Changer la forme

Pour commencer, nous pouvons changer la forme de celui-ci grâce à la fonction `shape`. Pour ce faire, il nous suffit de lui passer la valeur d'une forme disponible en paramètre. Si nous ne lui passons aucun paramètre, celle-ci nous retourne la valeur actuelle de la forme. Les formes disponibles, que nous pouvons obtenir avec la fonction `getshapes`, qui ne prend aucun paramètre, sont :

Valeur	Forme
"arrow"	Flèche
"blank"	Rien
"circle"	Cercle
"classic"	Curseur classique
"square"	Carré
"triangle"	Triangle
"turtle"	Tortue

Voici ci-dessous un programme se servant de ces fonctions et de `ontimer`, fonction étudiée dans la partie précédente, pour afficher les formes, les unes à la suite des autres :

```
1  ###!/usr/bin/env python3
2
3  import turtle
4
5  FORMES_DISPONIBLES = turtle.getshapes() #Récupère les formes
    disponibles
6
7  def forme(index = 0):
8      """Une fonction récursive pour afficher les formes disponibles
9      les unes après les autres"""
10     turtle.shape(FORMES_DISPONIBLES[index]) #Change la forme
11     print("Forme actuelle : "+turtle.shape()) #Affiche la forme
        actuelle
12     if index < len(FORMES_DISPONIBLES)-1:
13         turtle.ontimer(lambda : forme(index+1), 1000)
14     else:
15         return
16
17 if __name__ == "__main__":
18     turtle.ontimer(forme)
```

À l'exécution de celui-ci, vous verrez enfin l'apparition de la fameuse tortue, icône discrète de ce tutoriel :



FIGURE 6.1. – La tortue.

Pour vous entraîner, vous pouvez par exemple faire un programme qui tamponne ces formes de manière aléatoire dans la fenêtre, un peu comme l'exemple du ciel étoilé, et vous pouvez aussi faire en sorte de supprimer les tampons de telle ou telle forme avec l'appui sur telle ou telle touche.

6.2.0.2. Enregistrer une nouvelle forme

En plus des formes initiales, nous avons aussi la possibilité d'ajouter les nôtres (une image au format gif ou encore un polygone) avec `register_shape` en lui passant un nom et/ou un polygone de cette manière :

```
1 turtle.register_shape("triangle", ((-10, 15), (0, 0), (10, 15)))  
   #Enregistre d'après un polygone  
2 turtle.register_shape("clem.gif")  #Enregistre d'après une image
```

i

Si vous enregistrez une forme d'après un polygone, vous verrez que l'orientation de la forme ne sera pas ce que vous pensiez. Il semblerait que les couples (x, y) soient inversés ici. Si vous enregistrez une forme d'après une image, il ne sera pas possible d'agrandir ou d'incliner celle-ci.

Comme convenu, nous avons désormais accès à la forme enregistrée :

```
1 turtle.shape("clem.gif")
```

6. Aller plus loin dans les configurations



FIGURE 6.2. – Clem en forme!

6.2.0.3. Incliner la forme sans impacter l'angle

Pour incliner la forme sans modifier l'angle, nous avons la fonction `tilt` qui permet de tourner d'un angle donné. De plus, nous avons aussi la fonction `tiltangle` pour modifier ou connaître l'orientation de la forme du crayon :

```
1 turtle.tilt(90) #Pointe vers le Nord
2 print(turtle.tiltangle()) #Affiche '90.0'
3 turtle.tiltangle(270) #Pointe vers le Sud
4 print(turtle.heading()) #Affiche '0.0'
```

6.2.0.4. Modifier la taille de la forme

Tout d'abord, il existe aussi un mode pour le redimensionnement et c'est la fonction `resizemode` qui nous permet de modifier ou connaître celui-ci. Les trois modes disponibles sont :

Valeur	Particularité
"noresize"	Pas de redimensionnement avec de mode (par défaut)
"auto"	Le redimensionnement s'effectue en modifiant la taille du tracé (<code>pensize</code>)
"user"	Le redimensionnement s'effectue en faisant appel à la fonction <code>shapessize</code>

Nous connaissons la première fonction puisque nous nous en sommes déjà servis. Pour `shapessize`, celle-ci met le mode de redimensionnement à "user" si elle reçoit des arguments et redimensionne la forme avec possiblement deux dimensions, une perpendiculaire à l'orientation (`stretch_wid`) ainsi qu'une dans le sens de l'orientation (`stretch_len`), plus une largeur de contour (`outline`)

```
1 print(turtle.resizemode()) #Affiche 'noresize'
2 turtle.shapessize(5, 1)
3 print(turtle.shapessize()) #Affiche '(5, 1, 1)'
4 turtle.shapessize(1, 5)
5 turtle.shapessize(4, 2, 2)
```

6. Aller plus loin dans les configurations

Et ce n'est pas tout !

6.3. Paramétrer encore et toujours

En effet, turtle nous permet aussi diverses configurations, sur lesquelles nous n'allons pas trop nous attarder.

6.3.0.1. Le mode

Pour commencer, nous pouvons changer le mode en passant une valeur à `mode`. Si l'on ne passe aucun paramètre, le mode actuel nous est retourné. Par défaut, nous avons travaillé avec le mode `"standard"`, mais il y a au total trois valeurs possibles :

Valeur	Particularité
"standard"	Pointe vers l'Est par défaut (angle : 0°). Rotation dans le sens inverse des aiguilles d'une montre (se reporter à l'image de la section "Dessiner des figures simples" de la partie 2 si besoin).
"logo"	Pointe vers le Nord par défaut (angle : 0°). Rotation dans le sens des aiguilles d'une montre.
"world"	Adapté au paramétrage du repère. Attention, si le ratio x/y est différent de 1 alors les angles apparaîtront distordus.

Par la même occasion, cette fonction effectue un `reset`. Voici un exemple :

```
1 turtle.showturtle()
2 print(turtle.mode()) #Affiche 'standard', pointe vers l'Est
3 turtle.setheading(90) #Pointe vers le Nord
4 turtle.mode("logo") #Modifie le mode, reset, pointe vers le Nord
5 turtle.setheading(90) #Pointe vers l'Est
```

6.3.0.2. L'angle

En plus du changement de mode qui impacte l'orientation, nous avons aussi le choix de travailler en degrés ou en radians, respectivement avec les fonctions `degrees` et `radians`.

```
1 turtle.setheading(90)
2 turtle.radians() #Change l'unité des angles à radians
3 print(turtle.heading()) #Affiche '1.5707963...'
```

6. Aller plus loin dans les configurations

6.3.0.3. Le mode couleur

Si vous vous rappelez, lorsque nous voulions fournir un tuple (R, G, B) pour les couleurs, nous étions obligés de travailler avec des valeurs entre 0 et 1. En fait, il y a deux modes possibles, le mode 1.0 et le mode 255. Nous pouvons modifier ou connaître ce dernier avec `colormode`.

```
1 print(turtle.colormode()) #Affiche '1.0'
2 turtle.colormode(255) #Met le mode de couleur à 255
3 turtle.colormode(1.0) #Remet le mode de couleur à 1.0
```

6.3.0.4. La vitesse

Au détour des programmes, vous avez dû apercevoir la fonction `speed`. Celle-ci nous permet de modifier et de connaître la vitesse de tracé comme ceci :

```
1 turtle.speed("fastest") #Met la vitesse la plus rapide
2 print(turtle.speed()) #Affiche '0'
```

Nous pouvons lui passer une valeur entre 0 et 10. Au delà de 10 et en deçà de 0.5, la valeur attribuée sera 0. De 1 à 10, la vitesse augmente. Les valeurs clefs sont :

Valeur	Particularité
"fastest"	0
"slowest"	1
"slow"	3
"normal"	6
"fast"	10

Nous pouvons aussi travailler sur l'animation et la vitesse de celle-ci à l'aide des fonctions `delay` et `tracer`. Si cela vous intéresse, je vous renvoie à la documentation pour vous renseigner sur celles-ci.

6.3.0.5. Le buffer d'annulation

Nous ne l'avons jamais vu, mais il est possible d'annuler une instruction à l'aide de la fonction `undo`. En fait, celles-ci sont stockées dans un buffer (un tampon mémoire) et nous pouvons faire varier la taille de celui-ci avec la fonction `setundobuffer`. Bon, à moins de vouloir en réduire la proportion, nous sommes larges puisque sa taille initiale est de 1000 ! Voici un exemple d'utilisation :

6. Aller plus loin dans les configurations

```
1 turtle.setheading(90)
2 turtle.undo() #Annule l'instruction
3 print(turtle.heading()) #Affiche '0.0'
4 turtle.setundobuffer(0) #Le buffer a désormais une taille de 0
5 turtle.setheading(90)
6 turtle.undo() #Aucun effet, vu que le buffer ne peut rien contenir
7 print(turtle.heading()) #Affiche '90.0'
```

Voilà, nous avons fait le tour de ces configurations supplémentaires, c'est donc le moment de pratiquer !

6.4. TP : Configuration avancée

C'est l'avant-dernier travaux pratiques et sans doute le plus simple. Comme pour le premier, il suffira de configurer notre fenêtre selon certains critères et de vérifier ces valeurs en les affichant.

Voici les critères en question, suivis du résultat obtenu :

- Mode = *"world"*;
- Origine du repère dans coin haut gauche du canvas;
- Mode de couleur : 255;
- Forme du crayon = tortue;
- Orientation = Sud.



FIGURE 6.3. – Le résultat.

```
1 mode : world
2 position : (0.00,0.00)
3 mode de couleur : 255
4 forme : turtle
5 orientation : 270.0
```

La correction :

👁 Contenu masqué n°18

Pas très compliqué, n'est-ce pas ?

Au cours de cette partie, nous avons vu comment personnaliser encore plus notre fenêtre.

Contenu masqué

Contenu masqué n°18

```
1  #!/usr/bin/env python3
2
3  import turtle
4
5  def recapitule():
6      """Fonction pour récapituler les informations"""
7      print("mode : {}".format(turtle.mode()))
8      print("position : {}".format(turtle.position()))
9      print("mode de couleur : {}".format(turtle.colormode()))
10     print("forme : {}".format(turtle.shape()))
11     print("orientation : {}".format(turtle.heading()))
12
13 if __name__ == "__main__":
14     #On met le mode world
15     turtle.mode("world")
16     #On change l'origine du repère
17     turtle.setworldcoordinates(0, -100, 100, 0)
18     #On met le mode de couleur à 255
19     turtle.colormode(255)
20     #On change la forme du crayon à tortue
21     turtle.shape("turtle")
22     #On oriente le crayon vers le Sud
23     turtle.setheading(270)
24     #On récapitule
25     recapitule()
26     #Clique gauche pour quitter
27     turtle.exitonclick()
```

[Retourner au texte.](#)

7. S'amuser avec plusieurs tortues

À ce stade du tutoriel, nous avons fait le tour d'une bonne partie de turtle. Découvrons maintenant l'utilisation orientée objet de celui-ci ainsi que quelques autres fonctionnalités.

7.1. La classe Screen

Pour commencer, la classe *Screen* est une classe permettant de manipuler une fenêtre avec un canvas. Elle hérite de *TurtleScreen*. Il faut utiliser la première lorsque notre programme turtle est indépendant tandis qu'il faut utiliser la seconde, dont l'instanciation nécessite un canvas, lorsqu'il fait partie d'une application (si nous utilisons turtle à l'intérieur d'un programme *tkinter* par exemple).

La classe *Screen* est instanciable une seule fois, c'est-à-dire que nous pourrions faire autant de fois appel au constructeur de cette classe, celui-ci nous retournera le même objet. En programmation, on appelle cela un [singleton](#) [↗](#). C'est un patron de conception restreignant l'instanciation d'une classe à un unique objet :

```
1 print(turtle.Screen() == turtle.Screen()) #Affiche 'True' : les
    deux objets retournés sont identiques
```

Grâce à notre objet, nous pouvons, comme nous avons appris à le faire au cours de ce tutoriel, paramétrer la fenêtre ou encore interagir avec l'utilisateur :

```
1 ecran = turtle.Screen() #Instanciation
2 ecran.bgcolor('green')
3 ecran.numinput("Nombre", "Un nombre, il me faut un nombre : ", 0,
    0, 100)
4 ecran.exitonclick()
```

Nous savons que notre fenêtre a besoin d'un crayon pour dessiner dedans. Pour savoir s'il y en a, nous pouvons faire appel à la méthode `turtles` qui retourne une liste :

```
1 print(turtle.Screen().turtles()) #Affiche les crayons actuels de
    notre fenêtre
```

7. S'amuser avec plusieurs tortues

Pour le reste, vous pouvez retrouver les méthodes utilisables dans la partie **24.1.2.2. Methods of TurtleScreen/Screen** de la documentation.

7.2. La classe Turtle

La classe *Turtle* nous permet d'instancier et de contrôler un crayon. Si une fenêtre n'a pas encore été créée, le constructeur s'occupe d'instancier un objet de type *Screen*. La classe *Turtle* hérite de la classe *RawInput* (alias *RawPen*) dont le constructeur nécessite un canvas.

Remarquons que comme les objets *Turtle* sont liés à un objet *Screen* unique (dû au singleton), ce n'est pas la solution à choisir si nous souhaitons manipuler plusieurs canvas (ou plusieurs fenêtres).

Avec notre objet, nous pouvons tracer dans la fenêtre de façon personnalisée :

```
1 crayon = turtle.Turtle() #Instanciation
2 crayon.resizemode('user')
3 crayon.pensize(20)
4 crayon.color('green', 'white')
5 crayon.forward(100)
```

De plus, nous pouvons lier des événements à ces crayons, chose que nous n'avons pas encore vue, avec les méthodes `onclick`, `onrelease` et `ondrag`. Elles se comportent comme la fonction `onscreenclick` précédemment étudiée c'est-à-dire qu'elles prennent en paramètre la fonction associée à l'événement ainsi que, optionnellement, le bouton de la souris générant l'événement et si la fonction s'ajoutera à (avec `True`) ou remplacera (avec `False` ou rien) la ou les fonctions déjà associée(s). Voici un exemple, suivi d'une illustration :

Le **code** :

👁 Contenu masqué n°19



FIGURE 7.1. – Cliquer ou relâcher, telle est la question.

Avec cet exemple, le clique gauche sur le crayon rose générera un "cliqué !" tandis qu'un relâchement gauche depuis le crayon gris engendrera un "relâché !".

Pour le reste, vous pouvez retrouver les méthodes utilisables dans la partie **24.1.2.1. Turtle methods** de la documentation.

7.3. TP : Clique la tortue !

Nous voilà quasiment à la fin de ce tutoriel et quoi de mieux qu'un TP pour finir en beauté ? Dans celui-ci, nous allons réaliser un autre jeu : clique la tortue !

7.3.0.1. Cahier des charges

Le principe du jeu de cliques est très simple : il suffit de cliquer sur quelque chose de spécifique pour gagner des points avec lesquels il est possible de monter de niveau voire de débloquent des bonus. Bien sûr, au fur et à mesure de la progression, il est important de gagner de plus en plus de points vu qu'il en faut de plus en plus, ne serait-ce que pour monter de niveau.

Pour notre jeu, nous cliquerons sur des tortues et nous n'utiliserons qu'un seul bonus lié au niveau et qui rapportera **niveau - 1 point(s)** par tortue cliquée. Ainsi, ce bonus rapportera 0 point au niveau 1 (le **niveau initial**) ou encore 17 points au niveau 18, par exemple. **Le coût de chaque niveau** vaut **niveau * 10**.

Les tortues apparaîtront régulièrement dans la fenêtre et auront deux couleurs : une couleur de tracé et une couleur de remplissage, toutes deux choisies **aléatoirement**, de même que leur position et leur orientation. Un clique sur la tortue la fera disparaître et rapportera des points en fonction de ces couleurs et du bonus. Voici la grille des points associés aux couleurs :

Point	Couleur
1	'black' et 'grey'
2	'brown' et 'orange'
4	'pink' et 'yellow'
8	'purple' et 'green'
16	'blue'
32	'red'

En définitive **un clique sur une tortue rapportera** la somme du :

- Nombre de points correspondant à la couleur de remplissage ;
- Nombre de point correspondant à la couleur de tracé si et seulement si celle-ci est la même que la couleur de remplissage ;
- Nombre de point donné par le bonus de niveau

Une partie se terminera lorsque le joueur aura atteint le niveau 20.

Vous êtes libre de choisir comment quitter le jeu, comment monter de niveau ou encore à quel clique associer chaque tortue. Avant de vous lancer, je vous encourage à réfléchir à l'organisation de ce projet, à ce dont vous avez besoin qu'il fasse, etc.

Pour ma part, si vous avez besoin d'aide, j'ai évidemment choisi la programmation orientée objet vu que c'est le thème de cette partie. Ensuite, mon jeu est constitué de la sorte :

7. S'amuser avec plusieurs tortues

👁 Contenu masqué n°20

Enfin, les captures d'écran ci-dessous peuvent vous inspirer si besoin :

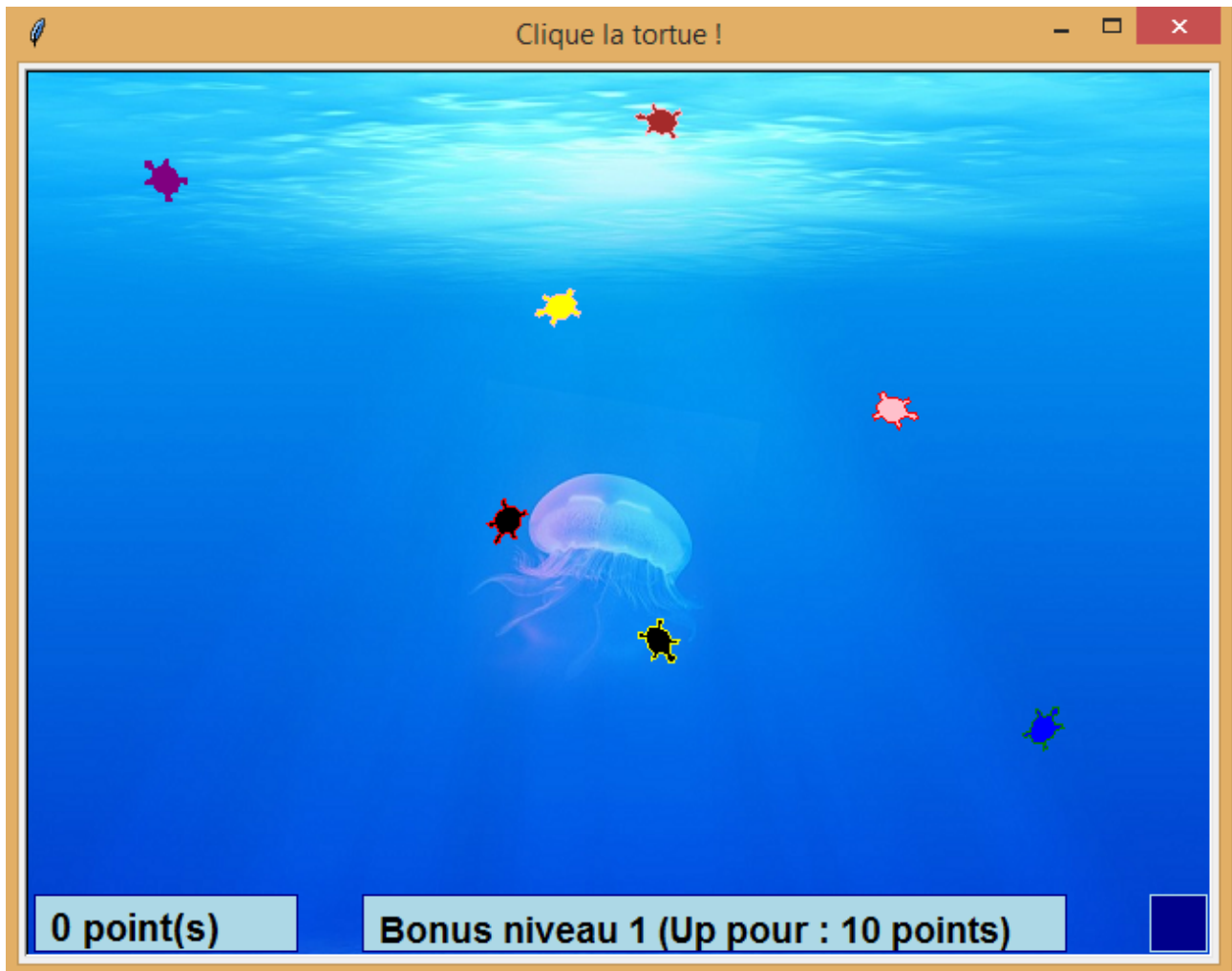


FIGURE 7.2. – Clique la tortue! (1)

7. *S'amuser avec plusieurs tortues*

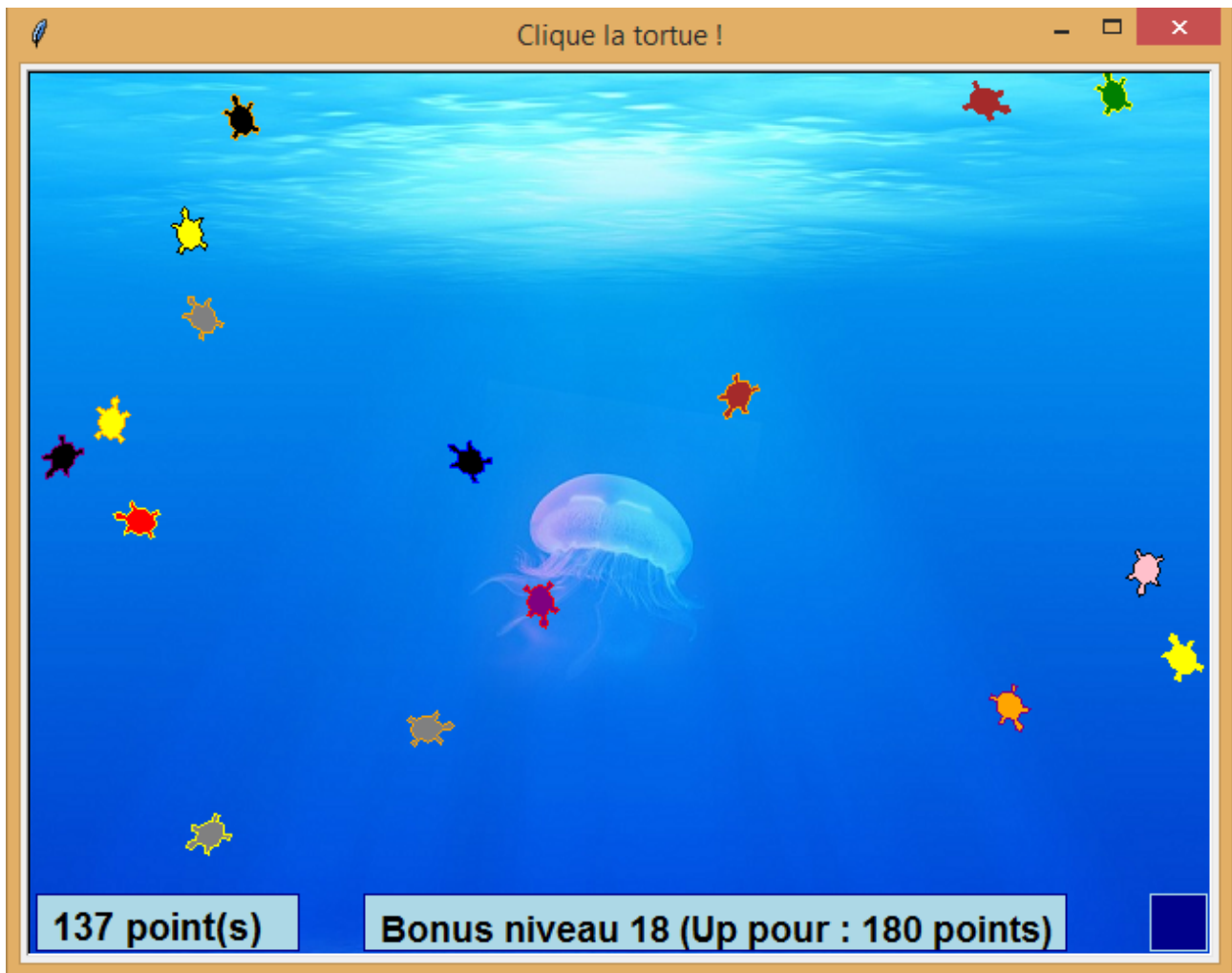


FIGURE 7.3. – Cliquez la tortue ! (2)



FIGURE 7.4. – Clique la tortue! (3)

7.3.0.2. Correction

Puisque le programme est plus consistant que d'habitude, je vais vous présenter les gros morceaux utilisés indépendamment.

Tout d'abord, nous importons évidemment *turtle*, puis la fonction *randint* pour les choix aléatoires et enfin le module *sys* pour quitter le programme. Les constantes ont des noms explicites correspondant à leurs contenus. Enfin, nousinstancions un objet de type *Jeu* qui initialise la fenêtre et le jeu, puis nous lançons l'exécution de celui-ci :

© Contenu masqué n°21

Ensuite, la classe *Bonus* contient le niveau courant et la valeur de points nécessaires pour passer au niveau suivant. En outre, celle-ci permet d'afficher un bouton (qui est en fait un crayon de forme carré) et de gérer le passage au niveau supérieur avec le clique sur celui-ci :

7. S'amuser avec plusieurs tortues

👁 Contenu masqué n°22

La classe *Tortue* permet quant à elle d'afficher une tortue de manière aléatoire (couleurs, orientation, position) à l'écran. En outre, elle gère le clique sur celle-ci :

👁 Contenu masqué n°23

La classe *Jeu* permet d'afficher les informations et de gérer le jeu. Elle contient des méthodes pour afficher notamment le nombre de points ou encore le niveau actuel, à l'aide d'un crayon.

👁 Contenu masqué n°24

i

Si vous testez l'ensemble, vous vous rendrez compte que nous apercevons parfois un curseur noir au centre avant qu'il ne disparaisse. Cela est en fait la nouvelle tortue juste après son début d'initialisation avec `turtle.Turtle.__init__(self)` et juste avant que nous la cachions avec `self.hideturtle`, le temps de la mettre dans le bain .

Encore une fois, il y a de nombreuses améliorations possibles. Par exemple, vous pouvez mieux gérer la courbe de progression, créer de nouveaux bonus, faire varier la vitesse d'apparition des tortues, ajouter des tailles de tortue différentes qui rapporteraient des points comme avec les couleurs, ou encore rajouter de nouvelles couleurs.

Voilà, j'espère que vous vous êtes bien amusé à programmer ce jeu !

Au terme de cette partie, vous savez désormais utiliser turtle de façon orientée objet.

Contenu masqué

Contenu masqué n°19

```
1  #!/usr/bin/env python3
2
3  import turtle
4
5  crayon_clique, crayon_relache = turtle.Turtle(), turtle.Turtle()
6  crayons = { 'pink' : crayon_clique, 'grey' : crayon_relache }
7  pos = [-75, 0]
8  for clef, valeur in crayons.items():
```


7. S'amuser avec plusieurs tortues

```
9     valeur.color(clef, clef)
10    valeur.shapesize(7, 7)
11    valeur.up(); valeur.goto(pos); valeur.down(); pos[0] += 150
12    crayon_clique.onclick(lambda x, y: print("cliqué !"))
13    crayon_relache.onrelease(lambda x, y: print("relâché !"))
```

[Retourner au texte.](#)

Contenu masqué n°20

Premièrement, il y a une classe *Jeu*, qui représente le jeu. Elle est notamment composée d'un écran, d'un nombre de points et d'un bonus. Cette classe permet d'initialiser le jeu, d'afficher la partie et de gérer son fonctionnement. De plus, elle donne son instance à la classe *Tortue* et à la classe *Bonus* de sorte que celles-ci puissent connaître les informations dont elles ont besoin voire faire appel à des méthodes.

Deuxièmement, il y a classe *Bonus*, héritant de *turtle.Turtle*, qui permet gérer le niveau ainsi que le bonus associé, et de progresser en cliquant sur le curseur carré bleu (affiché en bas à droite de la fenêtre).

Troisièmement, il y a la classe *Tortue*, elle aussi héritant de *turtle.Turtle* qui permet de créer, d'afficher et de gérer une Tortue à l'écran.

Au final, chaque tortue correspond à un crayon, le bonus correspond à un crayon et le jeu a un crayon pour dessiner dans la fenêtre.

[Retourner au texte.](#)

Contenu masqué n°21

```
1  #!/usr/bin/env python3
2
3  import turtle
4  from random import randint
5  import sys
6
7  TITRE = "Clique la tortue !"
8  IMAGE_FOND = "fond_clique_tortue.png"
9  LARGEUR, HAUTEUR = 640, 480
10 COULEURS = { 'black' : 1, 'grey' : 1,
11              'brown' : 2, 'orange' : 2,
12              'pink' : 4, 'yellow' : 4,
13              'purple' : 8, 'green' : 8,
14              'blue' : 16,
15              'red' : 32 }
16 CLES_COULEURS = list(COULEURS.keys())
17 TEMPS_APPARITION = 1000
18
```

7. S'amuser avec plusieurs tortues

```
19 #Classes
20
21 if __name__ == "__main__":
22     jeu = Jeu() #Initialisation jeu
23     jeu.ecran.ontimer(jeu.tour_de_jeu, TEMPS_APPARITION)
        #Exécution jeu
```

[Retourner au texte.](#)

Contenu masqué n°22

```
1 class Bonus(turtle.Turtle):
2
3     jeu = None #Attribut de classe
4
5     def __init__(self):
6         """Constructeur"""
7         turtle.Turtle.__init__(self) #Constructeur classe mère
8         self.niveau = 1 #Niveau initial
9         self.cout_niveau = self.niveau * 10 #Cout niveau supérieur
            initial
10        self.bouton_bonus() #Affichage du bouton bonus
11        self.onclick(lambda x, y : self.niveau_suivant()) #Gestion
            du clique sur le bouton
12
13    def bouton_bonus(self):
14        """Affichage du bouton lié au bonus"""
15        self.speed(0)
16        self.shape("square")
17        self.shapesize(1.5)
18        self.color("lightblue", "darkblue")
19        self.up()
20        self.goto(LARGEUR // 2 - 25, -HAUTEUR // 2 + 25)
21
22    def niveau_suivant(self):
23        """Gestion passage au niveau suivant"""
24        if Bonus.jeu != None:
25            if jeu.points >= self.cout_niveau:
26                self.onclick(None) #Désactivation gestion clique
                    le temps des modifications
27                Bonus.jeu.points -= self.cout_niveau
28                self.niveau += 1
29                self.cout_niveau = self.niveau * 10
30                Bonus.jeu.affiche_bonus()
31                Bonus.jeu.affiche_nombre_points()
```

```
32         self.onclick(lambda x, y : self.niveau_suivant())
           #Réactivation gestion clique
```

[Retourner au texte.](#)

Contenu masqué n°23

```
1  class Tortue(turtle.Turtle):
2
3      jeu = None #Attribut de classe
4
5      def __init__(self):
6          """Constructeur"""
7          turtle.Turtle.__init__(self) #Constructeur classe mère
8          self.hideturtle()
9          self.couleur_trace = self.couleur_replissage = None
           #Couleurs
10         self.initialise_tortue() #Initialisation du crayon
11         self.showturtle()
12
13     def initialise_tortue(self):
14         """Initialise la tortue"""
15         self.couleur_trace = CLES_COULEURS[randint(0,
16             len(CLES_COULEURS)-1)]
17         self.couleur_replissage = CLES_COULEURS[randint(0,
18             len(CLES_COULEURS)-1)]
19         self.color(self.couleur_trace, self.couleur_replissage)
20         self.speed(0); self.shape("turtle");
21         self.setheading(randint(0, 359))
22         self.up()
23         self.goto(randint(-LARGEUR//2+15, LARGEUR//2-15),
24             randint(-HAUTEUR//2+40, HAUTEUR//2-15))
25         self.onclick(lambda x, y : self.clique_tortue())
26
27     def clique_tortue(self):
28         """Gestion du clique sur la tortue"""
29         if Tortue.jeu != None:
30             self.onclick(None) #Désactivation du clique sur la
31             tortue
32             points = COULEURS[self.couleur_replissage]
33             if self.couleur_trace == self.couleur_replissage:
34                 points *= 2
35             points += jeu.bonus.niveau - 1
36             Tortue.jeu.points += points
37             Tortue.jeu.affiche_nombre_points()
38             self.hideturtle()
```

35

`del self`[Retourner au texte.](#)

Contenu masqué n°24

```

1  class Jeu():
2
3      def __init__(self):
4          """Constructeur"""
5          self.ecran = turtle.Screen() #Instanciation d'un écran
6          self.points = 0
7          self.bonus = Bonus() #Instanciation d'un bonus
8          self.crayon = turtle.Turtle() #Instanciation d'un crayon
9          self.initialise_crayon() #Initialisation du crayon
10         self.initialise_fenetre() #Initialisation de la fenêtre
11         self.lie_classe()
12
13     def initialise_crayon(self):
14         """Initialise crayon de la fenêtre"""
15         self.crayon.speed(0)
16         self.crayon.hideturtle()
17         self.crayon.up()
18
19     def initialise_fenetre(self):
20         """Initialise la fenêtre"""
21         self.ecran.setup(LARGEUR, HAUTEUR)
22         self.ecran.title(TITRE)
23         self.ecran.bgpic(IMAGE_FOND)
24         self.affiche_nombre_points()
25         self.affiche_bonus()
26         self.ecran.onscreenclick(lambda x, y : self.quitter(), 3)
27
28     def lie_classe(self):
29         """Lie les classe Bonus et Tortue au jeu"""
30         Tortue.jeu = self
31         Bonus.jeu = self
32
33     def affiche_bonus(self):
34         """Affiche le niveau actuel"""
35         self.crayon.up()
36         self.crayon.goto(-140, - HAUTEUR // 2 + 10)
37         self.dessine_rectangle((30, 375), ('darkblue',
38             'lightblue'))
39         self.crayon.pencolor("black")
40         self.crayon.forward(10)

```

```

40         self.crayon.write("Bonus niveau {} (Up pour : {} points)".format
41                             (self.bonus.niveau,
42                             self.bonus.cout_niveau),
43                             align = "left",
44                             font = ("Aria", 14, "bold"))
45     def dessine_rectangle(self, dimensions, couleurs):
46         """Dessine un rectangle personnalisé"""
47         self.crayon.setheading(0)
48         self.crayon.color(couleurs[0], couleurs[1])
49         self.crayon.down()
50         self.crayon.begin_fill()
51         for i in range(4):
52             if i%2 == 0:
53                 self.crayon.forward(dimensions[1])
54             else:
55                 self.crayon.forward(dimensions[0])
56                 self.crayon.left(90)
57         self.crayon.end_fill()
58         self.crayon.up()
59
60     def tour_de_jeu(self):
61         """Gestion du tour de jeu"""
62         if self.bonus.niveau >= 20:
63             self.crayon.up()
64             self.crayon.home()
65             self.crayon.pencolor("darkblue")
66             self.crayon.write("Victoire !",
67                               align = "center",
68                               font = ("Arial", 27, "bold"))
69             self.ecran.ontimer(self.quitter, TEMPS_QUITTER)
70         else:
71             Tortue()
72             self.ecran.ontimer(self.tour_de_jeu, TEMPS_APPARITION)
73
74     def affiche_nombre_points(self):
75         """Affiche le nombre de points"""
76         self.crayon.up()
77         self.crayon.goto(-LARGEUR // 2 + 5, - HAUTEUR // 2 + 10)
78         self.dessine_rectangle((30, 140), ('darkblue',
79                                             'lightblue'))
80         self.crayon.pencolor("black")
81         self.crayon.forward(10)
82         self.crayon.write("{} point(s)".format(self.points),
83                             align = "left",
84                             font = ("Arial", 15, "bold"))
85     def quitter(self):
86         """Fin du game"""
87         self.ecran.bye()

```

7. S'amuser avec plusieurs tortues

87

```
sys.exit(0)
```

[Retourner au texte.](#)

8. Conclusion

Cette présentation pratique et assez complète de turtle est finie. Comme vous avez pu le voir, c'est un module plutôt facile à prendre en main et plein de possibilités.

Maintenant, vous savez réaliser des dessins, jeux, etc... avec turtle! Je vous invite à partager vos créations sur ce [sujet](#) . N'hésitez pas non plus à y proposer vos propres exercices!

Si vous souhaitez renforcer vos connaissances ou en apprendre plus, car bien que nous ayons vu beaucoup de choses, nous n'avons par exemple pas étudié les classes *Shape* et *Vec2D* ou encore les fonctions `delay` et `tracer`, vous pouvez vous référer aux liens suivants (liste non exhaustive) :

- la [documentation](#)
- un tutoriel : [partie une](#) / [partie deux](#)
- un [autre tutoriel](#)
- le [code source](#)

De plus, il existe le module [turtledemo](#) qui regroupe de nombreux exemples d'utilisation (les sources de ceux-ci sont dans **RepertoireDeVotrePython/Lib/turtledemo/**).

Enfin, vous pouvez aller encore plus loin avec les interfaces graphiques en Python en utilisant des bibliothèques spécifiques. Par exemple, il existe [Pygame](#) pour faire des jeux et le module [tkinter](#) qui est plus orienté vers le logiciel.

À bientôt!

Merci à yoch et à Arius pour leurs retours. Merci à Grimur pour sa suggestion d'exercice sur le flocon de Von Koch. Merci à nohar pour la validation.

Liste des abréviations

IA Intelligence Artificielle. 56