Programmer des fractales avec Python (1/2)

1 Dressage de la tortue

La bibliothèque de programmation `turtle` du langage Python permet de commander les déplacements d'un objet tortue (une tortue ou un curseur) dans un plan, comme dans le langage Logo. Ce dernier a été créé dans les années 1970 et a été utilisé dans les années 1980 pour l'apprentissage de la programmation.
La tortue est caractérisée par sa **position** (un couple de coordonnées cartésiennes) et **l'angle** entre sa tête (ou la flèche du curseur) et une demi-droite de base qui par défaut est orientée par le vecteur Est de coordonnées $(1,0)$).
Par défaut, la tortue avance en ligne droite d'un certain nombre de pixels sur la demi-droite dont un vecteur directeur a pour origine sa queue et pour extrémité sa tête.

1. **Premier escalier**

   a. Créer un nouveau fichier source et l'enregistrer dans son répertoire sous le nom `escalier.py`, il contiendra les différents programmes de tracé d'escalier.

   Chaque nouveau programme sera introduit par la ligne `## Programme numero`.

   b. Saisir la série d'instructions suivantes, l'enregistrer avec `CTRL + S` puis l'exécuter avec `CTRL + E`. Les commentaires, précédés par la caractère #, ne sont pas exécutés par l'interpréteur. Attention à bien respecter l'indentation.

```
from turtle import *

speed(1)       #parametrage de la vitesse
shape("turtle") #choix de la forme de la tortue
h = 50         #variable donnant la hauteur de la marche
forward(h)     #avancer de d pixels
left(90)       #tourner la tete de la tortue de 90 degres vers la
    gauche
write(heading()) #ecriture de l'angle courant
forward(h)     #avancer de d pixels
right(90)      #tourner la tete de la tortue de 90 degres vers la
    droite
write(pos())   #ecriture de la position courante
mainloop()     #gestionnaire d'evenement, pour l'affichage et l'
    interactivite
```

   c. Compléter le programme pour tracer escalier à deux marches.

   d. Peut-on procéder de la même façon pour tracer un escalier avec 25 marches de 4 pixels de haut ?

2. **Second escalier**

   Pour répéter 25 fois un bloc d'instructions, on peut utiliser une boucle Pour avec l'instruction `for k in range(0, 25):`.

   Il ne faut pas oublier le caractère `:`, il marque le début sur la ligne suivante du bloc d'instructions répété. Toutes les instructions d'un même bloc doivent se trouver au même niveau d'indentation.

a. Saisir puis exécuter les instructions suivantes (dans `Pyzo` on peut exécuter une sélection d'instructions avec `Alt + RETURN`.

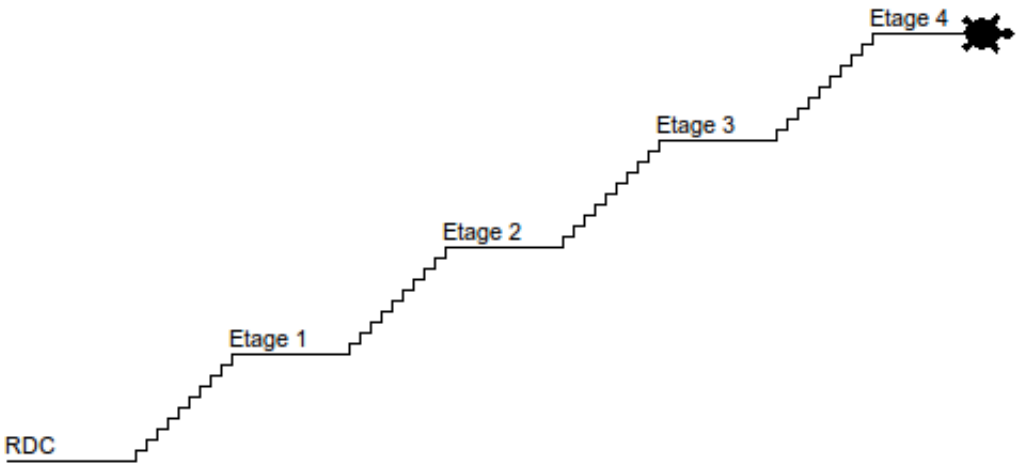
Quelles sont les valeurs prises par la variable `k` ?

```
for k in range(0, 25):
    print("instruction 1 du bloc, iteration ", k)
    print("instruction 2 du bloc", iteration ", k)
```

b. Modifier le programme `escalier.py` pour que la tortue trace un escalier avec 25 marches de 5 pixels de haut.

3. **Troisième escalier**

On veut réaliser l'escalier ci-dessous reliant les différents étages d'un immeuble :



a. Recopier puis compléter le programme ci-dessous pour qu'il trace cet escalier :

```
h = 5          #variable donnant la hauteur de la marche
nbetage = 4      #nombre d'etages de l'immeuble
penup()        #on leve le crayon pour activer le trace
goto(-200, -200) #on positionne la tortue a l'entree de l'immeuble
pendown()      #on baisse le crayon pour reactiver le trace
write("RDC")
forward(55)
for k in range(1, nbetage + 1): #boucle externe
    for j in range(0, 10):    #boucle interne
            #bloc de la boucle interne a completer
    write("Etage %s"%k)
    forward(50)

mainloop()
```

b. On peut améliorer la lisibilité du programme précédent en remplaçant la boucle interne par une seule instruction, indiquant à l'interpréteur de tracer une série de 10 marches de 5 pixels de hauteur.

Comme cette instruction n'existe pas par défaut dans le langage, on la définit avec le mot clef `def` : on lui donne un nom `parcourir_marche` et les deux paramètres dont elle a besoin, le nombre de marches `n` et la hauteur d'une marche `h`.

```
def parcourir_marche(n, h):
    for j in range(0, n):
            #bloc de la boucle interne a completer
```

On dit qu'on a défini la fonction `parcourir_marche` et on peut appeler cette fonction quand on en a besoin, par exemple `parcourir_marche(10,5)` constitue une instruction qui peut remplacer la boucle interne du programme précédent.

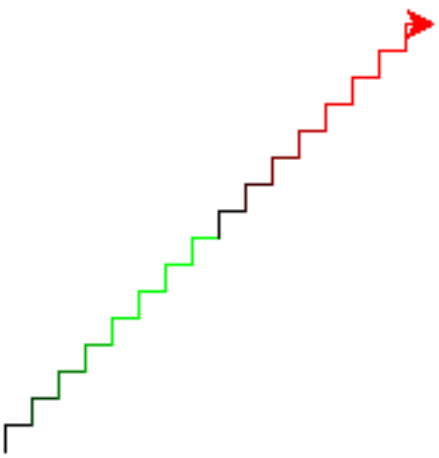
4. Quatrième escalier

On souhaite changer changer la couleur du trait pour chaque marche, en parcourant un nuancier du foncé au clair en rouge pour les huit premières marches, puis en vert.

La fonction `color()` prend comme paramètre le triplet (r, g, b) de représentation de la couleur dans l'espace colorimétrique (Rouge, Vert, Bleu) où chaque composante est mesurée sur une échelle de 0 (intensité nulle) à 255 (intensité maximale).

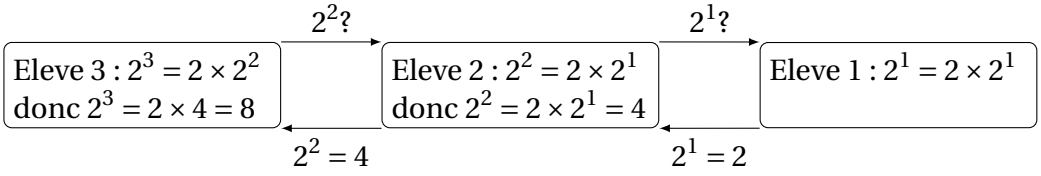
Dans la boucle de la fonction `parcourir_marche`, il suffit de rajouter une structure conditionnelle *Si condition Alors alternative 1 Sinon alternative 2* avec les mots clefs `if` et `else` qui règle la couleur du trait selon le compteur de marche. Compléter et tester la fonction `parcourir_marche2` ci-dessous.

```
1 def parcourir_marche2(n, h):
2     colormode(255)   #choix du mode de couleur (R, G, B) avec echelle de
          0 a 255
3     for j in range(0, n):
4         if j < 8:
5             color((0, min(255, j*60), 0)) #nuance de vert
6         else:
7             color((min(255, (j - 8)*60), 0, 0)) #nuance de rouge
8         # completer comme dans la fonction parcourir_marche
```



2 Fonctions récursives

1. ☞ Imaginons trois élèves possédant les mêmes connaissances mathématiques : ils savent que pour tout entier $n \geqslant 1$, $2^{n+1} = 2 \times 2^n$ et ils savent tous effectuer une multiplication par 2.
Voici une façon de calculer 2^3 :



☞ Une traduction mathématique de ce calcul serait la fonction p définie par :

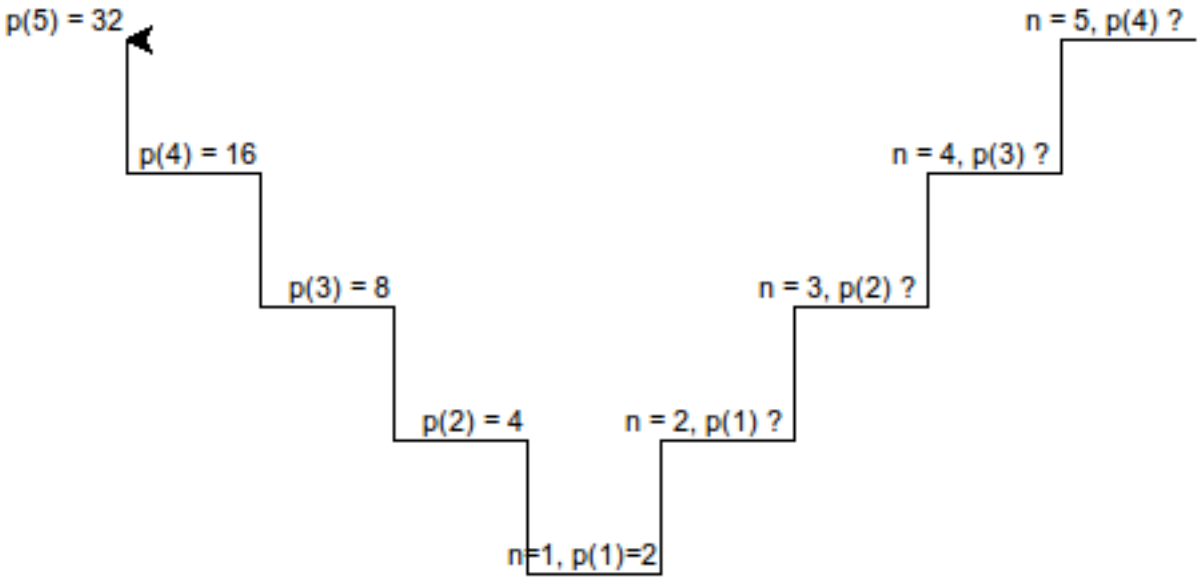
$$p(n) = \begin{cases} \text{si } n = 1 \text{ alors retourne } 1 \\ \text{sinon retourne } 2 \times p(n-1) \end{cases}$$

C'est une fonction qui s'appelle elle même, on parle de **fonction récursive**.

☞ Une traduction en langage Python de cette fonction serait :

```
def p(n):
    if n == 1:
        return 1
    else:
        return 2 * p(n - 1)
```

☞ Voici une illustration de ce qui se passe lors de l'appel de fonction `p(5)`. On peut noter que la fonction cesse de s'appeler et renvoie une valeur pour une condition d'arrêt, sinon on a une **descente infinie**.



☞ Modifier la fonction `p` pour qu'elle retourne la puissance n d'un réel a passé en paramètre en traitant le cas où $n = 0$.

2. Compléter la fonction récursive `parcourir_marche_rec` pour qu'elle trace un escalier de n marches sans utiliser de boucle `for`.

```
def parcourir_marche_rec(n):
    if n == 0:
        return
    else:
            #a completer
```

3 Une courbe fractale, la courbe de von Koch

Les contenus qui suivent sont directement inspirés de l'article « La récursivité de la tortue » écrit par Roger Cuppens et publié dans le Bulletin Vert de l'APMEP n°515.

1. **Définitions**

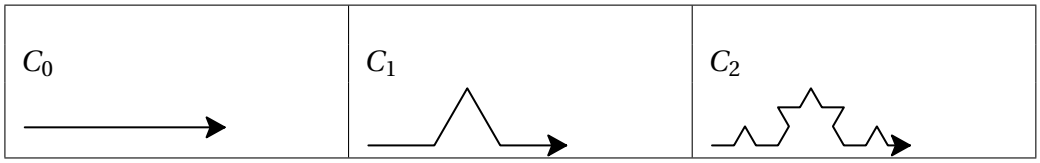
Dans son ouvrage *Les objets fractals – forme, hasard et dimension*, Benoît Mandelbrot a introduit la terminologie suivante :

- Une *courbe fractale* est la limite quand n tend vers l'infini d'une suite de courbes C_n définies récursivement et qui ne font que donner un aperçu de la complexité de la courbe limite.
- La courbe C_0 est *l'initiateur*.
- On obtient C_{n+1} à partir de C_n en remplaçant chaque segment composant C_n par un ensemble de segments appelé *générateur*.
- La courbe C_n est le $n^{\text{ième}}$ *téragone* de la courbe limite.

2. **Courbe de von Koch**

a. Avant la formalisation proposée par Mandelbrot, von Koch a défini en 1904, une courbe fractale dont l'initiateur est un segment de longueur c et dont le générateur s'obtient en découpant le segment en trois parties de même longueur et en remplaçant le segment médian par deux segments formant avec lui un triangle équilatéral.

On représenté ci-dessous avec `turtle` les trois premiers téragones pour la courbe de von Koch.



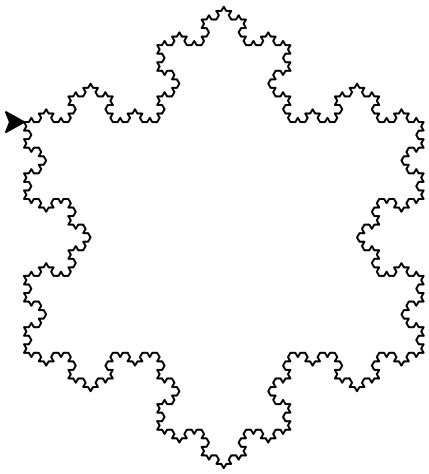
b. Compléter la série d'instructions suivantes pour le tracé de C_1 qui est le *générateur* de cette courbe fractale si on part d'un *initiateur* C_0 de longueur 81 pixels.

```
c = 81
forward(c/3)
left(60)
#a completer
```

c. Compléter la fonction récursive `koch(n, c)` ci-dessous pour qu'elle trace le $n^{\text{ième}}$ *téragone* de la courbe de von Koch en partant d'un segment *initiateur* de longueur c pixels. Dans la série d'instructions décrivant le générateur C_1, il suffit de remplacer les instructions `forward(c/3)` de tracé de segment par des appels récursifs `koch(n - 1, c/3)`.

```
def koch(n, c):
    if n == 0:      #segment initiateur
        forward(c)
    else:           #ensemble de segments generateur
        koch(n - 1, c/3)
        left(60)
        #a completer
```

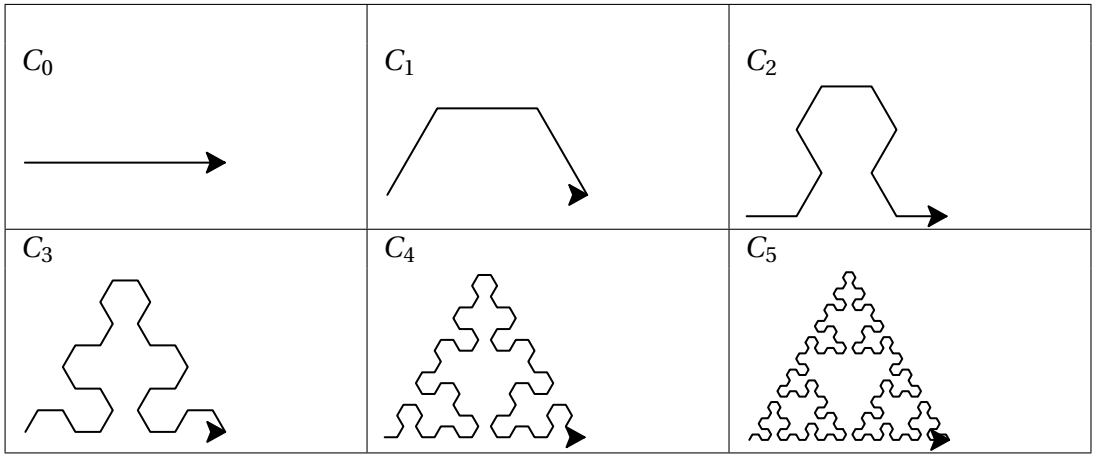
d. À partir de la fonction `koch(n, c)`, écrire une fonction `flocon_koch(n, c)` qui permet d'obtenir le tracé ci-dessous à partir de trois téragones de la courbe de von Koch construits à partir des côtés d'un triangle équilatéral.



4 Courbe de Sierpinski

1. La courbe de Sierpinski est une courbe fractale dont *l'initiateur* est un segment et dont le générateur remplace un segment de longueur c par trois segments de longueurs $c/2$, le premier fait un angle de $\times 60°$ avec le segment remplacé pour le $n^{\text{ième}}$ *téragone*, le second lui est parallèle et le troisième forme un angle de $-60°$ avec le segment remplacé.

A chaque remplacement d'un segment par un générateur, l'orientation de 60° du générateur alterne selon la position 1, 2 ou 3 du segment remplacé : la rotation s'effectue dans le sens opposé pour les segments 1 et 3 et dans le sens opposé pour le segment 2.



2. Compléter la série d'instructions suivantes pour le tracé du *téragone* C_1 de la courbe de Sierpinski si on part d'un *initiateur* C_0 de longueur 100 pixels.

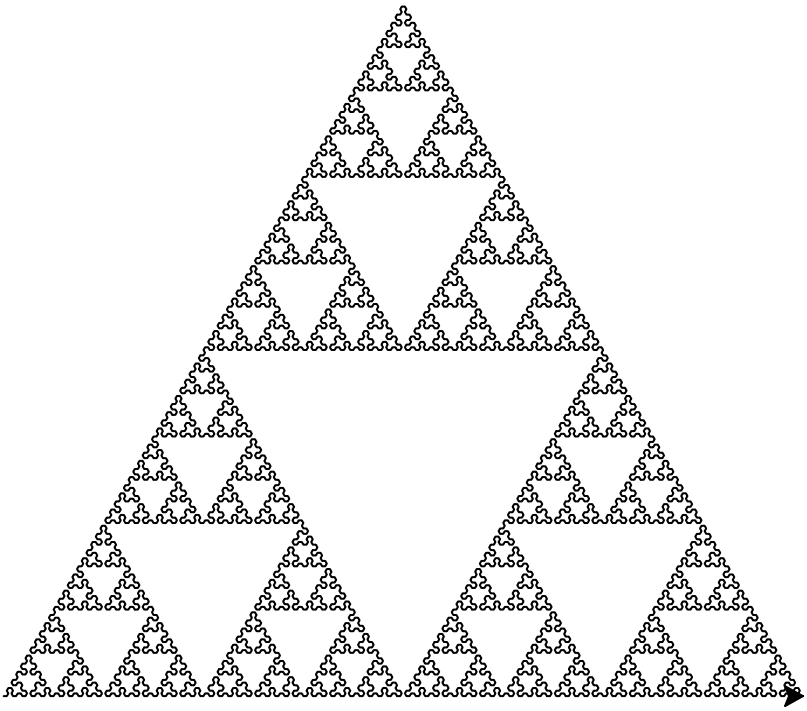
```
k = 1 #coefficient pour choisir l'orientation du generateur.
c = 100
left(k*60)
forward(c/2)
right(k*60)
#a completer
```

3. Compléter la fonction récursive `courbe_sierpinski(n, c, k)` ci-dessous pour qu'elle trace le $n^{\text{ième}}$ *téragone* de la courbe de Sierpinski en partant d'un segment *initiateur* de longueur c pixels.

Dans la description du générateur C_1, on remplace les instructions `forward(c/2)` de tracé de segment par des appels récursifs `courbe_sierpinski(n - 1,c/2,k)` ou `courbe_sierpinski(n - 1,c/2,-k)` selon l'orientation du générateur.

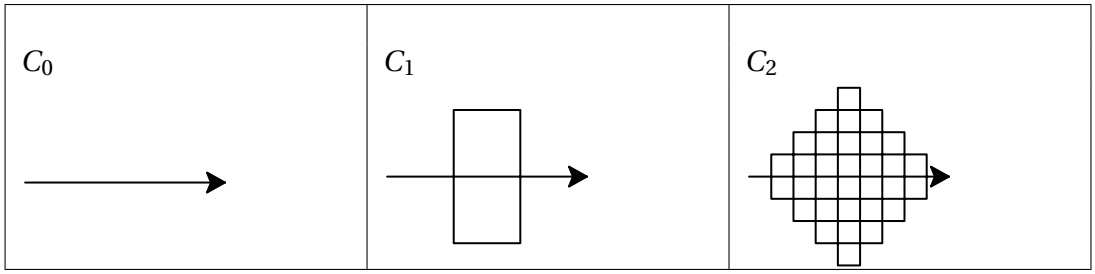
```
def courbe_sierpinski(n, c, k):
    if n == 0:
        forward(c)
    else:
        left(k*60)
        courbe_sierpinski(n - 1, c/2, -k)
        #a completer
```

4. Tracer plusieurs *téragones* C_n avec n de plus en plus grand, observer la forme obtenue et faire une conjecture sur la courbe de Sierpinski.



5 Courbe de Peano

1. La courbe fractale de Peano a pour *initiateur* un segment C_0 de longueur c et pour générateur la courbe C_1 dont tous les segments ont pour longueur $c/3$. On a représenté aussi ci-dessous le *téragone* C_2.



2. Compléter la série d'instructions suivantes pour le tracé du *téragone* C_1 de la courbe de Peano si on part d'un *initiateur* C_0 de longueur 90 pixels.

```
forward(c/3)
left(90)
for k in range(3):
   forward(c/3)
   right(90)
for k in range(4):
    #a completer
right(180)
forward(c/3)
```

3. Compléter la fonction récursive `courbe_peano(n, c)` ci-dessous pour qu'elle trace le $n^{\text{ième}}$ *téragone* de la courbe de Peano en partant d'un segment *initiateur* de longueur c pixels.

```
def courbe_peano(n, c):
    if n == 0:
        forward(c)
    else:
        #a completer
```

4. Tracer plusieurs *téragones* C_n avec n de plus en plus grand, observer la forme obtenue et faire une conjecture sur la courbe de Peano.

Vérifier cette conjecture à l'aide de la page web suivante :

`http://www.mathcurve.com/fractals/peano/peano.shtml`