Python – Récursivité (suite)

Encore des fonctions récursives

On considère la suite u_n définie de la manière suivante, avec a et b des réels quelconques :

$$u_n = \begin{cases} a & \text{si } n = 0 \\ b & \text{si } n = 1 \\ 3u_{n-1} + 2u_{n-2} + 5 & \text{si } n > 1 \end{cases}$$

Exercice 1 : Compléter les versions récursives et itératives de `suite(n, a, b)` qui renvoie la valeur de u_n avec $a =$ `a` et $b =$ `b`.

```
def suite_i(n, a, b):
    u = ...
    v = ...
    for i in range(...):
        w = ...
        u = ...
        v = ...
    return u

def suite_r(n, a, b):
    if n == 0:
        return ...
    elif ...:
        return ...
    else:
        return 3*suite_r(..., a, b) + 2*suite_r(..., a, b) + 5
```

```
>>> suite(3, -3, 2)
24
>>> suite(5, -3, 2)
314
>>> suite(5, 4, -6)
-252
```

Exercice 2 : Le pgcd de deux nombres est leur plus grand diviseur commun. On le note pgcd$(a;b)$. Il possède les propriétés suivantes :

- pgcd$(a;b) =$ pgcd$(b;a)$
- pgcd$(a;a) = a$
- pgcd$(a;0) = a$
- pgcd$(a;b) =$ pgcd$(a;b-a)$ si $b > a$
- pgcd$(a;b) =$ pgcd$(b;r)$, où r est le reste de la division euclidenne de a par b

Compléter la fonction récursive `pgcd(a, b)` qui renvoie le pgcd des deux entiers donnés.

```
def pgcd(a, b):
    if ...:
        return ...
    else:
        return pgcd(..., ...)
```

```
>>> pgcd(15, 5)
5
>>> pgcd(6, 27)
3
>>> pgcd(11, 13)
1
>>> pgcd(0, 4)
4
```

Exercice 3 : Compléter la fonction `nombre_chiffres(n)` qui renvoie le nombre de chiffres de `n`. Vous ne devez pas convertir le nombre en texte pour obtenir le résultat.

```
def nombre_chiffres(n):
    if ...:
        return 1
    else:
        return 1 + nombre_chiffres(...)
```

```
>>> nombre_chiffres(35315)
5
>>> nombre_chiffres(0)
1
```

Exercice 4 : Compléter la fonction récursive `appartient_r(v, t, i)` qui dit si `v` se trouve dans le tableau `t` entre l'index `i` et la fin du tableau. Elle est utilisée par `appartient(v, t)`.

```
def appartient_r(v, t, i):
    if ...: # on est hors du tableau
        return False
    elif ...: # on a trouvé la valeur
        return True
    else:
        return appartient_r(v, t, i+1)

def appartient(v, t):
    return appartient_r(v, t, 0)
```

```
>>> appartient(4, [3, 7, 1, 9])
False
>>> appartient(4, [3, 7, 4, 9])
True
>>> appartient(4, [4, 7, 1, 9])
True
>>> appartient_r(4, [4, 7, 1, 9], 1)
False
>>> appartient(4, [])
False
```

Exercice 5 : On appelle **coefficient binomiaux** les nombres $C(n;p)$ définis de la manière suivante, pour tous entiers positifs n et $p \leq n$:

$$C(n;p) = \begin{cases} 1 & \text{si } p = 0 \text{ ou } p = n \\ C(n-1;p-1) + C(n-1;p) & \text{sinon} \end{cases}$$

Compléter la fonction `coeff(n, p)` qui renvoie C(n;p).

```
def coeff(n, p):
    if ...:
        return ...
    else:
        return ...
```

```
>>> coeff(5, 4)
5
>>> coeff(5, 3)
10
>>> coeff(7, 3)
35
```

Exercice 6 : Le **triangle de Pascal**, nommé ainsi en l'honneur de Blaise Pascal, est obtenu en affichant à la $(n+1)$-ième ligne les coefficients $C(n;p)$ pour p allant de 0 à n.
Compléter la fonction itérative `triangle_pascal(n)` qui affiche les `n` premières lignes du triangle de Pascal. Le paramètre `end=''` dans la fonction **`print`** permet de ne pas aller à la ligne après un affichage. L'utilisation de **`print()`** fera aller à la ligne au moment voulu.

```
def triangle_pascal(n):
    for i in range(n+1):
        for j in range(...):
            print(f"{coeff(i, j):2} ", end='')
        print()
```

```
>>> triangle(5)
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
```

Exercice 7 (optionnel) : Écrire une version récursive de `triangle_pascal(n)`. Pour cela, vous pouvez utiliser une autre fonction récurisve `afficher_ligne(n, p)` qui affiche tous les coefficient de $C(n;0)$ à $C(n;p)$ et qui va à la ligne après si $n = p$.

Exercice 8 : Compléter la fonction `ack(m, n)` qui calcule le résultat de la fonction de Ackermann de la feuille d'exercices.

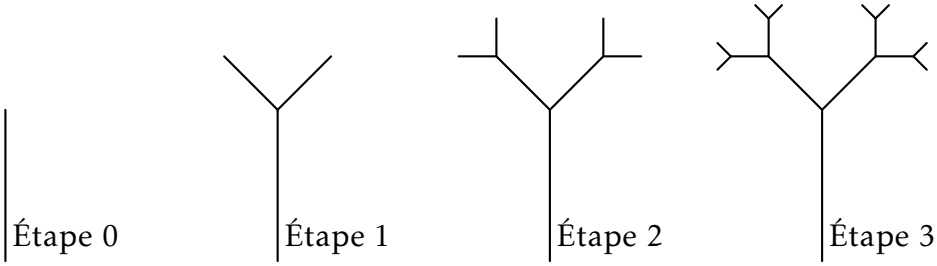
```
def ack(m, n):
    if ...:
        return ...
    elif ...:
        return ...
    else:
        return ...
```

```
>>> ack(2, 0)
3
>>> ack(3, 4)
125
>>> ack(3, 6)
509
```

Retour des fractales

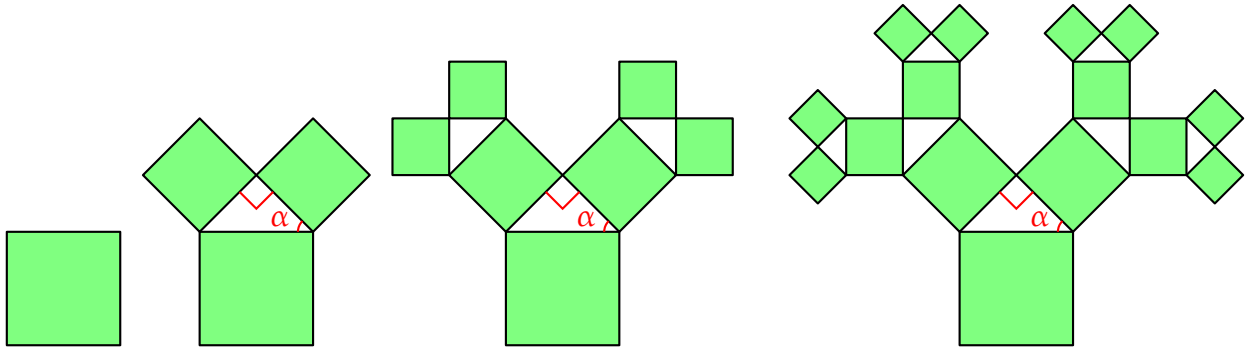
Nous allons continuer le fichier `fractales.py`.

Exercice 9 : Écrire une fonction `arbre(n, longueur)` qui permet de reproduire le fractale suivante :



Chaque branche se partage en deux branches moitié moins longues et formant un angle droit entre elles.
Attention, il faut bien réfléchir à la position de votre tortue à la fin de la figure.

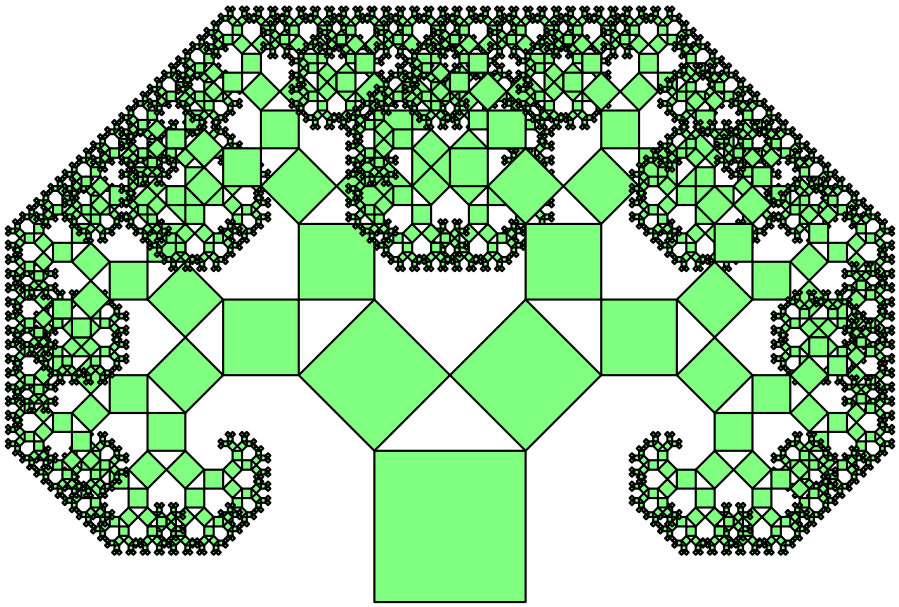
Exercice 10 : Écrire une fonction `arbre_pythagore(n, longueur, alpha)` qui permet de tracer la fractale suivante, appelée **arbre de Pythagore** :



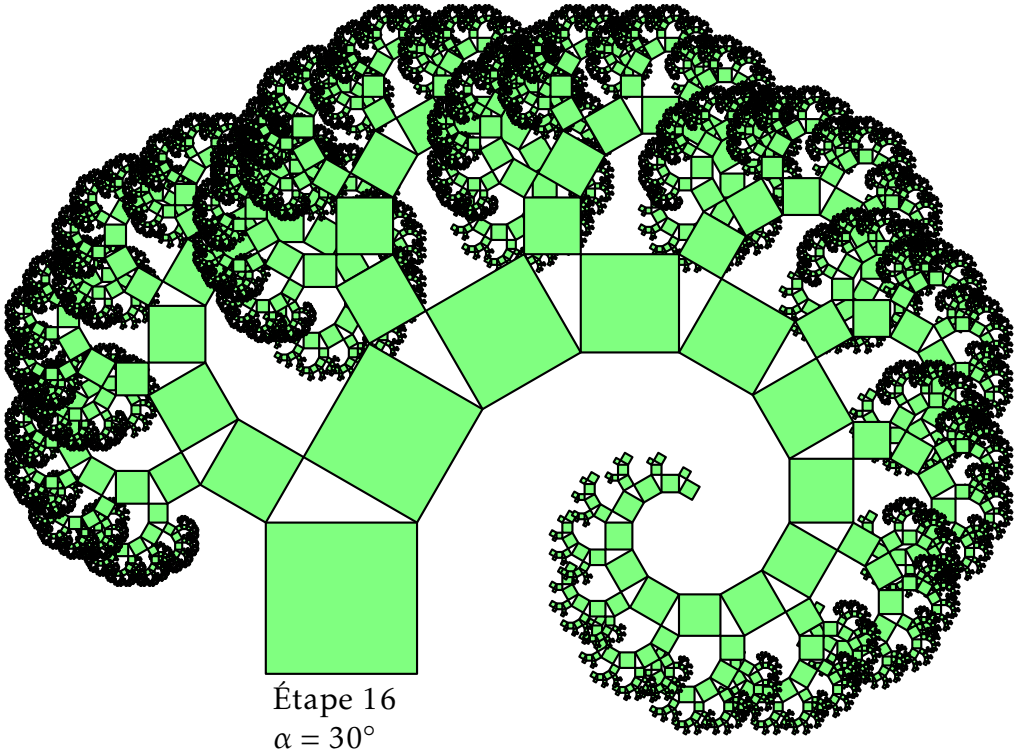
Les branches sont formées par des carrés qui se séparent en deux branches. Les branches sont séparées par un triangle rectangle dont un des angles, noté α, définit l'inclinaison du triangle.
Remarque 1 : Si l'hypoténuse d'un triangle rectangle mesure l, alors le côté adjacent à l'angle α mesure $l \times \cos\alpha$ et le côté opposé $l \times \sin\alpha$. Vous allez avoir besoin des fonctions trigonométriques, qui sont fournies par la bibliothèque `math`. Les angles utilisés sont exprimés en radians. Les calculs se font donc ainsi :

```
>>> from math import cos, sin, radians
>>> cos(radians(30))
0.8660254037844387
>>> sin(radians(30))
0.49999999999999994
```

Remarque 2 : Vous pouvez commencer par le cas $\alpha = 45°$ avant de traiter le cas général.

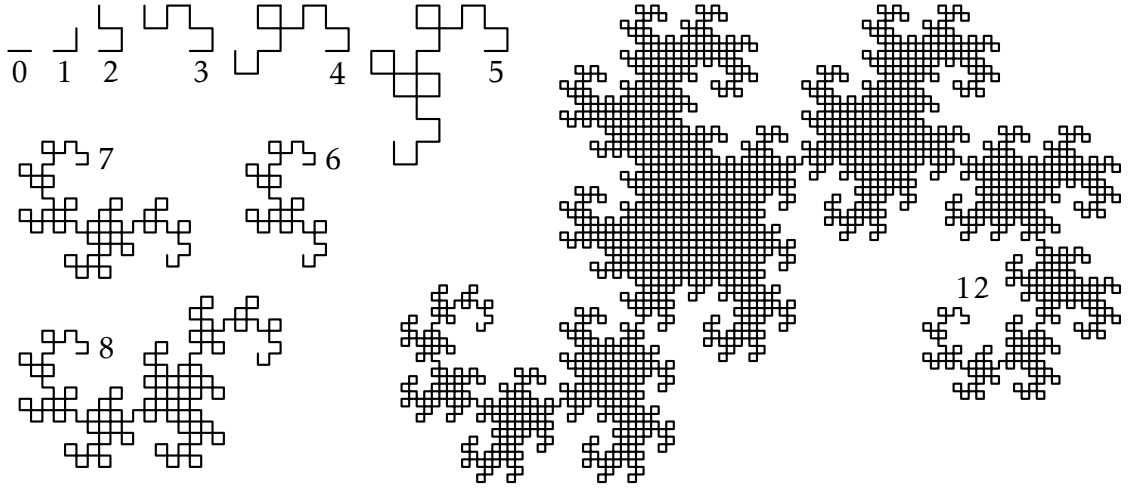


Étape 11
$\alpha = 45°$

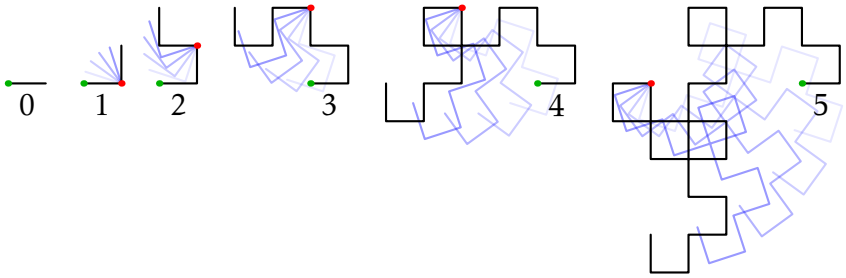


Étape 16
$\alpha = 30°$

La courbe fractale ci-dessous s'appelle le **dragon** :



À l'étape 0, elle est constituée d'un seul segment. Ensuite, pour passer d'une étape à la suivante, on "déplie" une copie de l'étape précédente autour de la fin de la courbe et on fait une rotation de 90° dans le sens des aiguilles d'une montre. Cela revient à dire que cela consiste à tracer la courbe à l'étape $n-1$, tourner à 90° vers la gauche puis tracer à nouveau la courbe à l'étape $n-1$, mais à l'envers.



Exercice 11 : Faire une fonction récursive `dragon(n, longueur, aller=True)` qui trace la courbe à l'étape `n` avec chaque segment qui mesure `longueur`. La variable booléenne `aller` permet de savoir si la courbe est à tracer à l'endroit ou à l'envers.