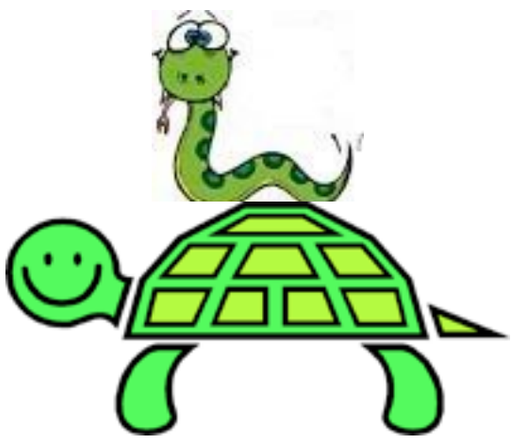
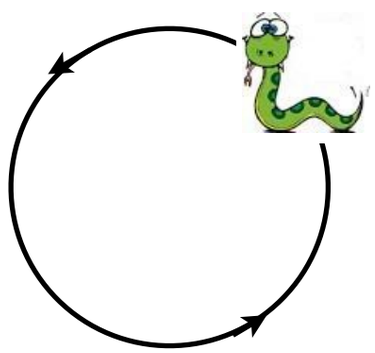
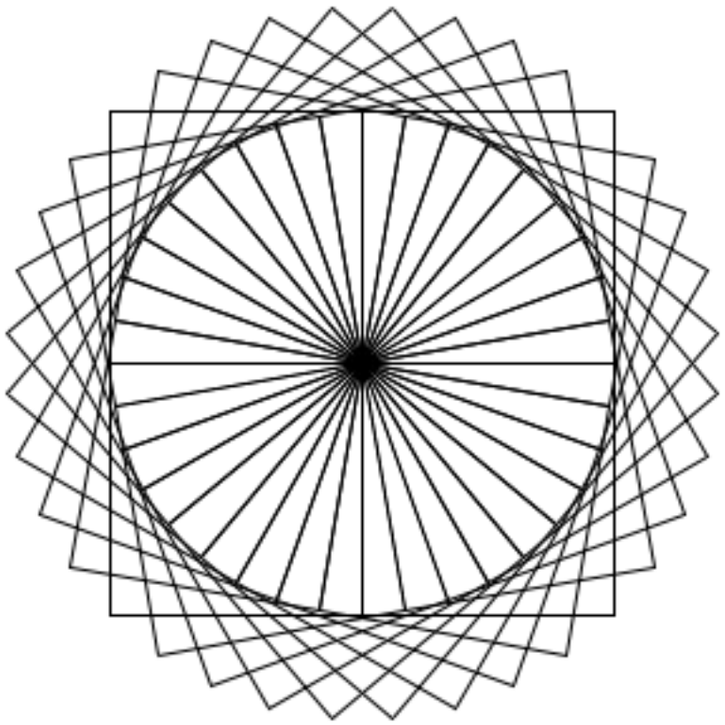
Université Nice Sophia Antipolis
Faculté des Sciences

Etienne Lozes
http://deptinfo.unice.fr/~elozes

La Tortue - Introduction aux boucles







Le module turtle de Python

• Le ***graphisme de la tortue*** a été inventé au Laboratoire d'Intelligence Artificielle du MIT vers 1968 avec le langage LOGO.

• Il est disponible dans quasiment tous les langages de programmation qui offrent des facilités graphiques.

• Et en particulier en Python 3 avec le module **turtle**.

• Ce module est livré avec la distribution Python standard. Mais il faut en importer les noms pour pouvoir les utiliser :

```
from turtle import *
```

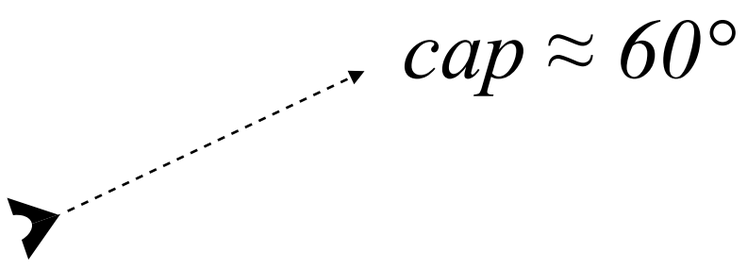
• Un fichier optionnel `turtle.cfg` placé dans le répertoire de travail permet de configurer le monde de la tortue.

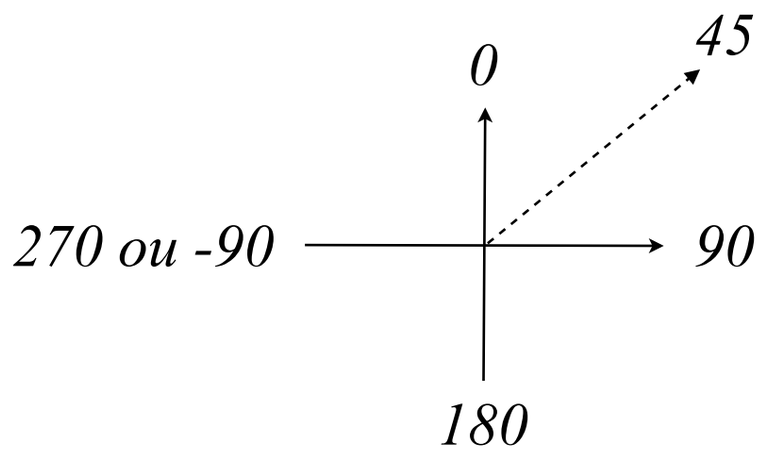
• Ne nommez PAS votre fichier `turtle.py` !

• Placez en dernière ligne de votre fichier l'instruction `mainloop()`.

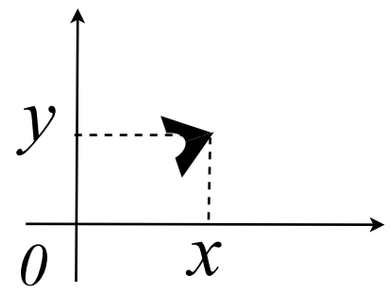
L'état de la tortue : position, cap, crayon

• Une *tortue* est représentée par une flèche qui indique son **cap** en **degrés** :





• Une *tortue* a une **position** : une abscisse et une ordonnée.



• Une *tortue* a un **crayon** (*pen*) qui peut être baissé (*down*) ou levé (*up*). Si le crayon est baissé, la tortue laisse une trace en se déplaçant. On peut choisir la couleur du crayon ainsi que la couleur de fond du canvas.

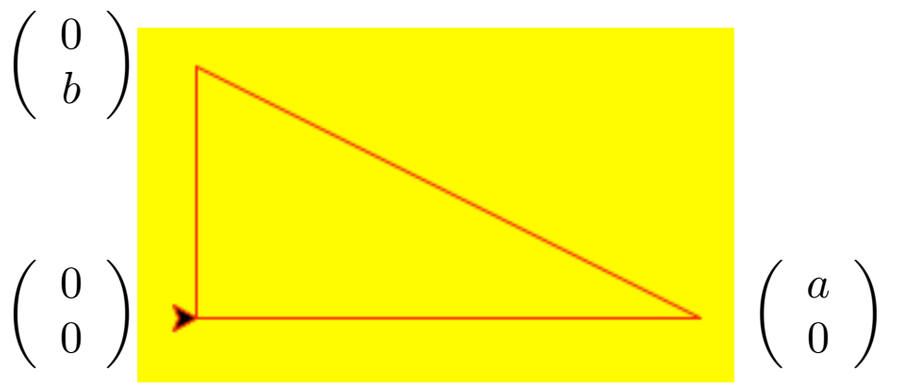
• Une tortue a donc un **ETAT** représenté mathématiquement par trois données : $position$, cap, $crayon$.

La position	Le cap	Le crayon
`pos()`	`heading()`	`down()`
`goto(x,y)`	`setheading(a)`	`up()`
		`color(c)`
		`pencolor(c)`
		`pensize(n)`

• Agir sur le canvas : `reset()` et `bgcolor(c)`.

• Exemple : dessin d'un triangle rectangle de côtés a et b.

```
def tri_rect(a,b) :
    up() ; goto(0,0) ; down()
    goto(a,0)
    goto(0,b)
    goto(0,0)
```




```
bgcolor('yellow')
pencolor('red')
tri_rect(200,100)
```

http://docs.python.org/release/3.2.3/library/turtle.html

Couples, tuples: des données composées

- Le résultat de la fonction pos() est un couple.
- Un couple est un tuple à 2 éléments. Un tuple à 3 éléments serait noté (1, '2', 3.4). Le tuple à un seul élément est (3.14159,)

```
>>> p = pos()
>>> p
(25.0,10.0)
```

Les composantes d'un tuple p se notent p[0], p[1], p[2]…

```
>>> p[0]
25.0
>>> p[1]
10.0
```

L'affectation permet de **déconstruire** un tuple

```
>>> (x,y) = p
>>> x
25.0
>>> y == p[1]
True
```

On peut utiliser cette affectation pour échanger 2 variables

```
>>> (x,y) = (1,2)
>>> (x,y) = (y,x)
>>> (x,y)
(2,1)
```

(x,y) = (α,β) ⇔ x = α, y = β

si α *et* β *sont deux expressions quelconques…*

Manipuler des vecteurs 2D

La somme de deux tuples est leur concaténation

```
>>> v1 = (1,2)
>>> v2 = (3,4)
>>> v1 + v2
(1,2,3,4)
```

```
def somme_vect2d(v1,v2):
    (x1,y1) = v1
    (x2,y2) = v2
    return (x1+x2,y1+y2)
```

Si on veut calculer la somme vectorielle de deux vecteurs, il faudra les déconstruire puis construire le vecteur somme

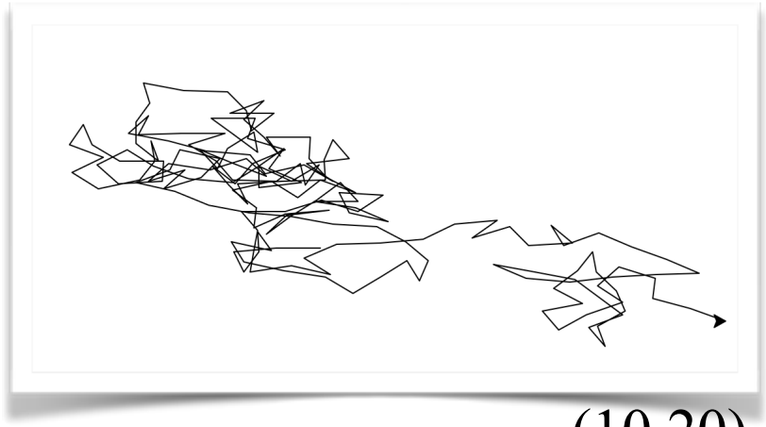
```
def somme_vect2d(v1,v2):
   return (v1[0]+v2[0], v1[1]+v2[1])
```

Si on veut renvoyer un vecteur unitaire au hasard

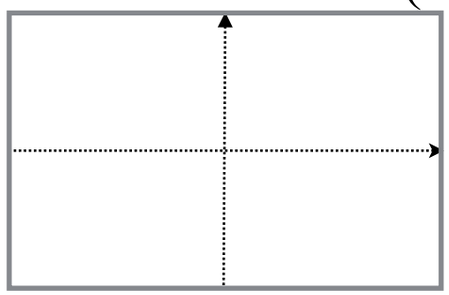
```
def rand_vect2():
   alpha = random.uniform(0,2*pi)
   return (cos(alpha), sin(alpha))
```

Simuler un mouvement Brownien

Une particule se déplace aléatoirement: à chaque étape, elle avance selon un vecteur unitaire v tiré au hasard. Quand elle sort du cadre fixé, on s'arrête.



(10,20)



(-10,-20)

```
# on fixe les coordonnées des angles
setworldcoordinates(-10,-20,10,20)
ht() # « hide turtle » , cache la tortue
p = (0,0)  # la position de la tortue
up(); goto(0,0); down()
while abs(p[0])<10 and abs(p[1])<20 :
    goto(p[0],p[1])
    v = rand_vect2()
    p = somme_vect(p,v)  # future position
                         # de la tortue
```

Boucle while:

tant que p reste dans le cadre, on répète

Les déplacements relatifs de la tortue

• Il s'agit de la *vraie* tortue pour les puristes...

• Nous voulons par principe ignorer la valeur du cap et de la position dans le graphisme polaire pur.

Le cap	La position
`left(a)`	`forward(d)`
`right(a)`	`back(d)`
`towards(p)`	

• Notez que : `right(a)` ⇔ `left(-a)` et `back(d)` ⇔ `forward(-d)`

• Une suite d'appels aux fonctions `left(...)` et `forward(...)` permet donc de décrire une courbe connexe (d'un seul tenant). En levant le crayon, on peut tracer plusieurs courbes non reliées entre elles.

Un mouvement Brownien en tortue pure

On répète les deux opérations:

- tourner d'un angle au hasard
- avancer de 1

```
p = (0,0)  # la position initiale de la tortue
up(); goto(0,0); down()
speed(10) # pour que la tortue avance plus vite

while abs(p[0])<10 and abs(p[1])<20 :
    alpha = random.uniform(0,360) # (les angles de la tortue sont en degrés)
    left(alpha)
    forward(1)
    p = pos()  # la position de la tortue actualisée
```

Les boucles for et while

On a souvent besoin de répéter des opérations
La boucle while peut s'utiliser dans toutes les situations

Cependant, lorsqu'on sait combien de fois on doit répéter une suite d'instructions, on préfère utiliser la boucle for

```
for i in range(5,14,3) :
    print('i =', i)
```



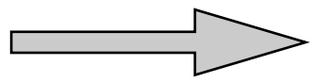
```
i = 5
while i < 14 :
    print('i = ', i)
    i = i + 3
```

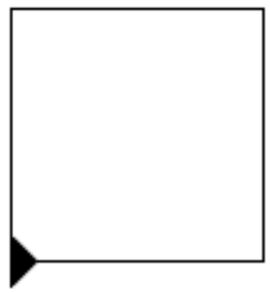
```
i = 5
i = 8
i = 11
```

Dans un boucle for, il faut fixer le début (5), la fin (14), et le pas (3)

- Exemple, dessin d'un **carré** de côté c.

```
def carre(c) :
    i = 0
    while i < 4 :
        forward(c)
        left(90)
        i = i + 1
```

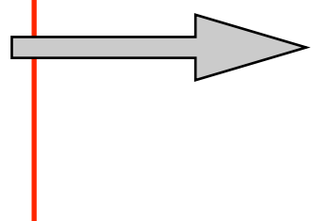


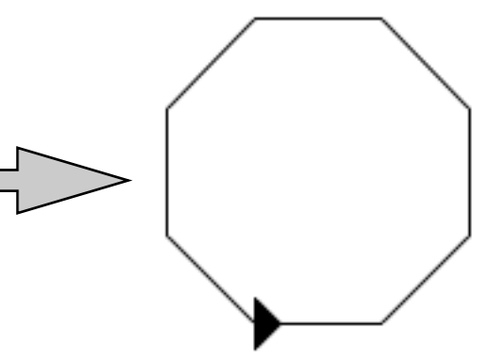


carre(100)

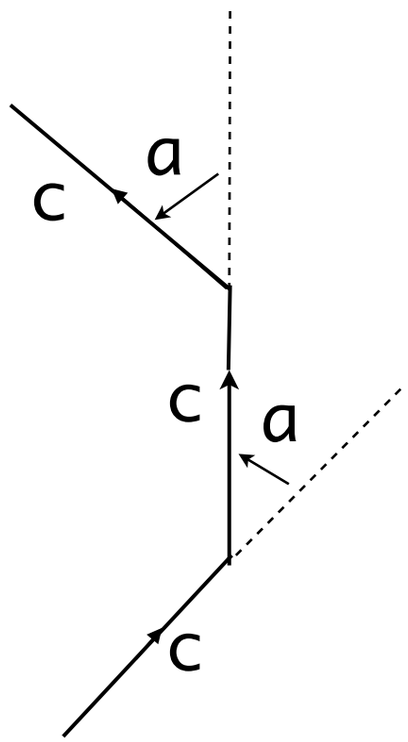
- Généralisation : dessin d'un **polygone régulier** à n côtés.

```
def polygone(n,c) :
    a = 360.0 / n
    for i in range(0,n,1) :
        forward(c)
        left(a)
```





polygone(8,100)



Les différentes utilisations de range dans une boucle for

Dans beaucoup de situations :

- le pas de la boucle est de 1
- la valeur de départ est 0
- on ne se soucie pas de la variable de boucle (la variable i)

Python autorise une écriture simplifiée dans chacun des cas

```
for i in range(0,14,1) : print('hello')
```

⇕

```
for i in range(0,14) : print('hello')
```

pas=1 implicite

⇕

```
for i in range(14) : print('hello')
```

début=0 implicite

⇕

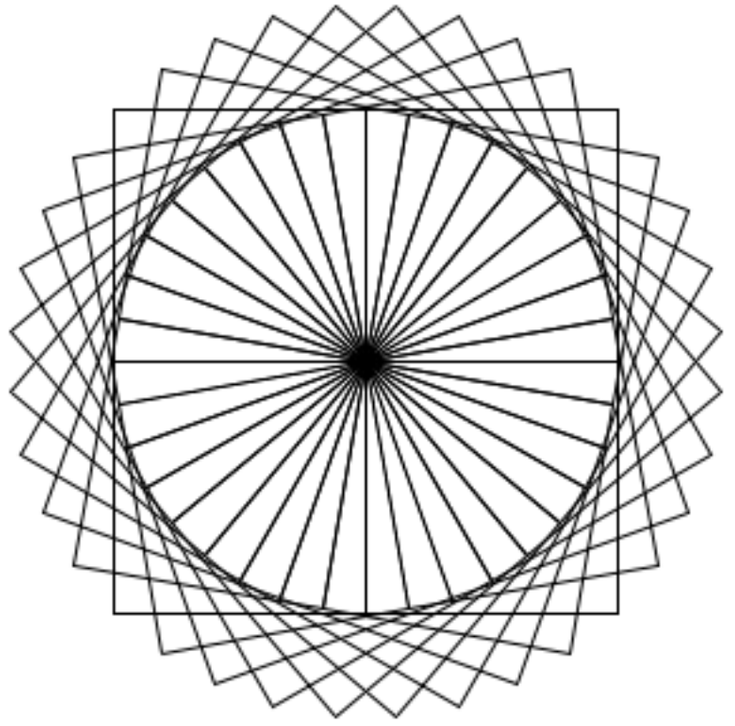
```
for _ in range(14) : print('hello')
```

pas de var de boucle

- Exemple de dessin obtenu par des carrés en rotation.

```
def carre(c) :
    polygone(4,c)
```

```
def fleur() :
    for _ in range(36) :
        carre(100)
        left(10)
```



```
reset() ; hideturtle() ; tracer(False) ; fleur() ; tracer(True)
```

- Il est possible mais non obligatoire de **localiser la fonction auxilliaire** `carre` . Elle ne sera pas utilisable en dehors de fleur() !

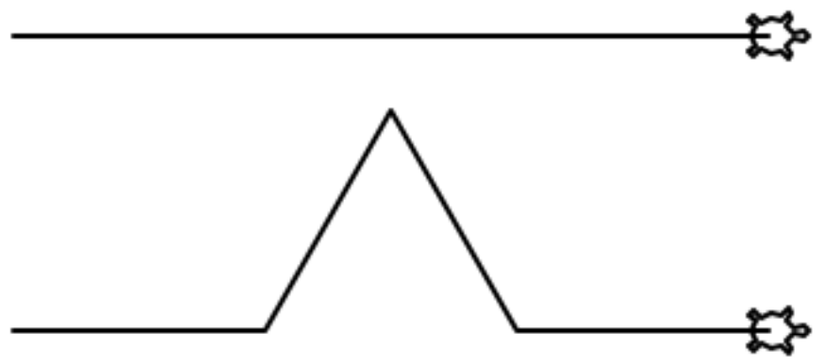
Une fonction locale !

```
def fleur() :
    def carre(c) :
        polygone(4,c)
    for i in range(0,36) :
        carre(100)
        left(10)
```

La courbe fractale de Von Koch

• Petite incursion dans la récurrence graphique. La suite (VK_n) des courbes de Von Koch de base T est construite de proche en proche :

- VK_0 est un segment de longueur T

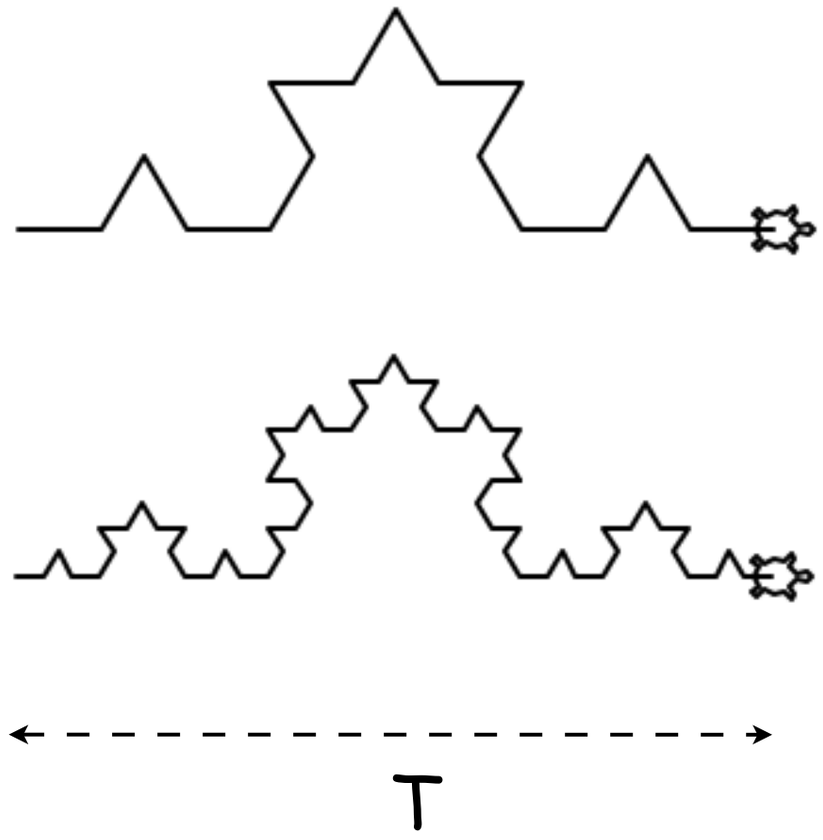


- VK_1 s'obtient par chirurgie sur VK_0

- VK_2 s'obtient par la même chirurgie sur chaque côté de VK_1

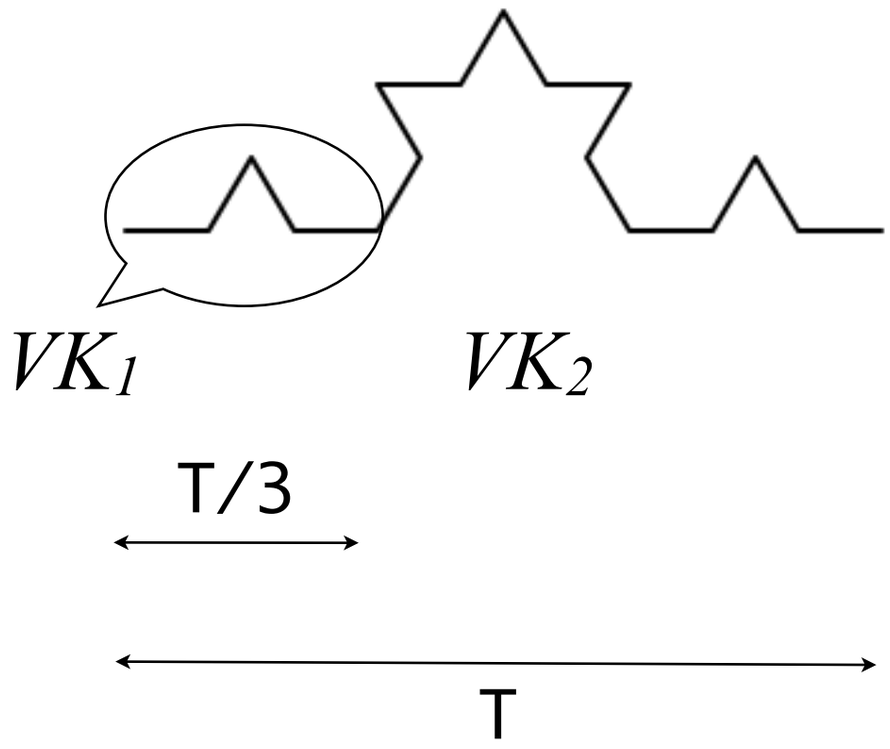
- VK_3 s'obtient par la même chirurgie sur chaque côté de VK_2

etc.



• Mathématiquement, la courbe VK_n s'obtient donc comme assemblage de **quatre** courbes VK_{n-1} . Il s'agit donc d'une RECURRENCE sur n :

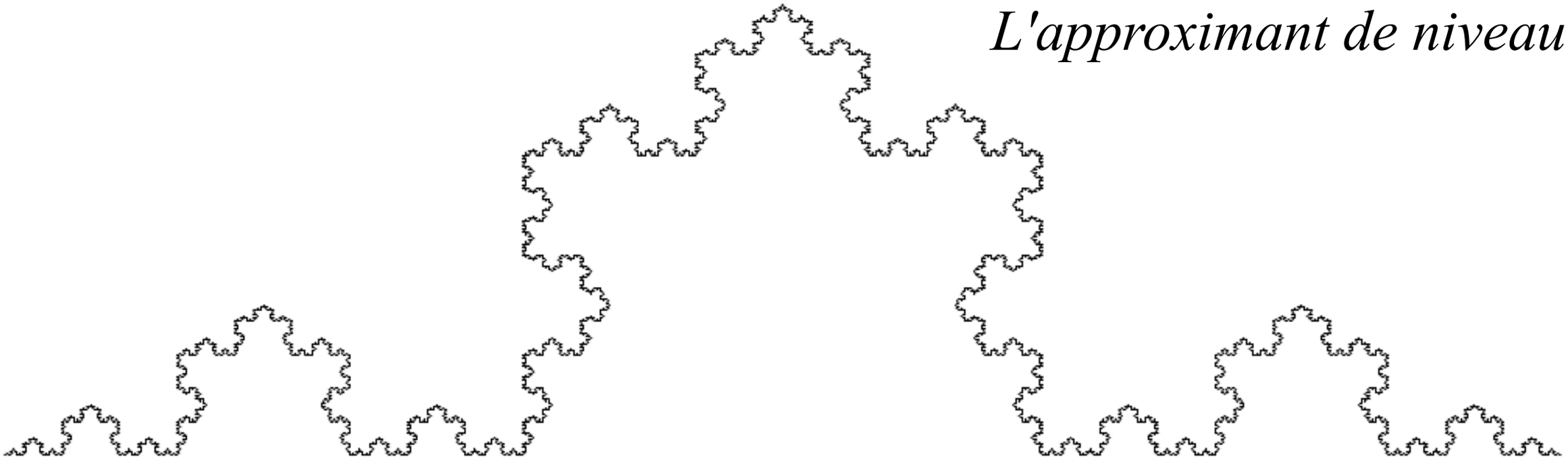
```
def von_koch(n,T) :     # approximant de niveau n et de base T
    if n == 0 :
        forward(T)
    else :
        von_koch(n-1,T/3)
        left(60)
        von_koch(n-1,T/3)
        right(120)
        von_koch(n-1,T/3)
        left(60)
        von_koch(n-1,T/3)
```

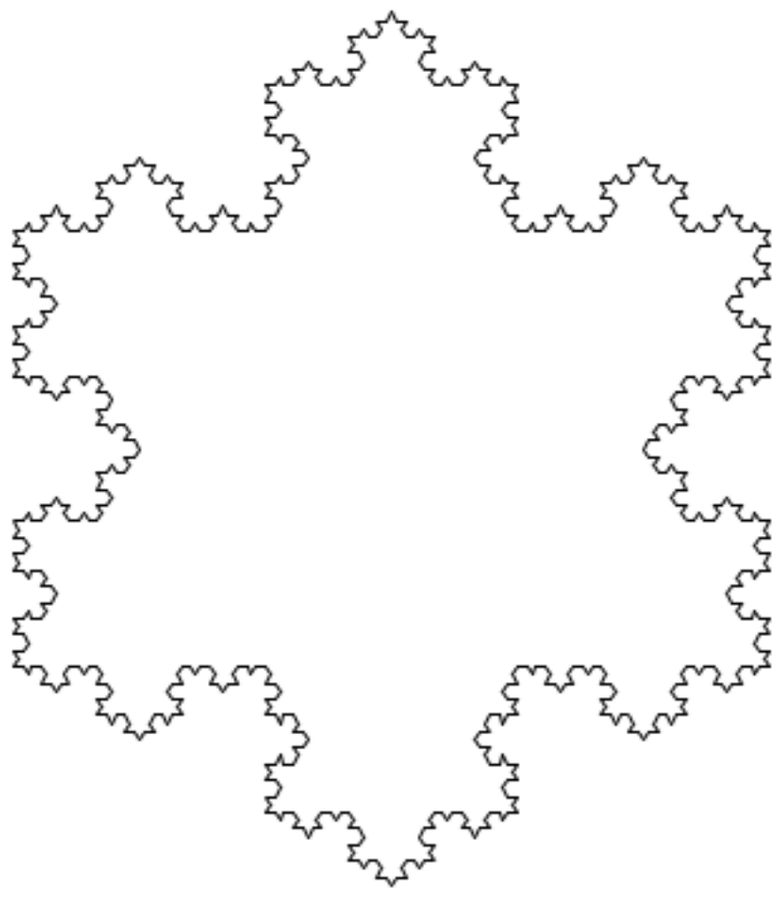


• La courbe de Von Koch VK est la "limite" de la suite : $VK = \lim_{n \to +\infty} VK_n$

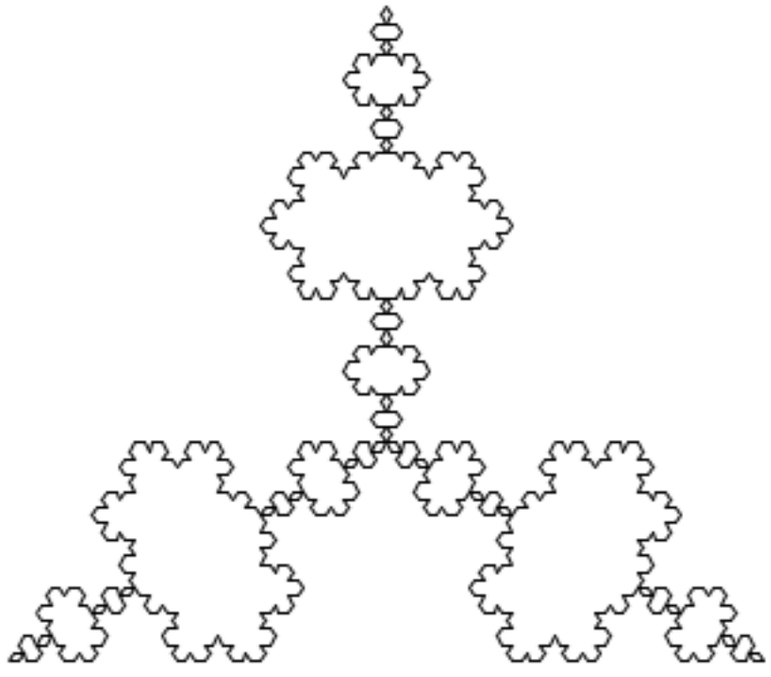
• Découverte en 1906, VK possède d'étranges propriétés. Par exemple, elle est continue mais n'admet de tangente en <u>aucun</u> point !!

L'approximant de niveau 6





Le flocon de Von Koch

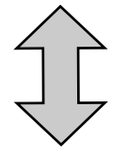


L'antiflocon

Les variables locales...

• Jusqu'à présent, dans plusieurs fonctions, nous avons introduit des variables qui n'étaient pas des paramètres de la fonction. Par exemple, dans la fonction carré ci-contre, la variable `i` .

```
def carre(c) :
    i = 0
    while i < 4 :
        ...
```



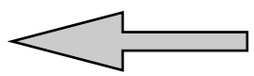
• Une telle variable est dite **locale à la fonction**. Son nom a peu d'importance, elle n'a rien à voir avec une variable de même nom `i` existant en-dehors de cette fonction !

```
def carre(c) :
    j = 0
    while j < 4 :
        ...
```

```
i = 42

def foo() :
    i = 10
    print("i vaut " + i)
```

```
>>> i
42
>>> foo()
i vaut 10
>>> i
42
```



i est locale !

...et les variables globales

• Une variable définie en-dehors de toute fonction est **globale**. Pour y faire référence dans une fonction, il faut le déclarer explicitement !

```
i = 42

def foo() :
    global i
    i = 10
    print("i vaut " + i)
```

```
>>> i
42
>>> foo()
i vaut 10
>>> i
10
```

globale

globale

globale

• On ne peut pas changer impunément `i` en `j` dans la fonction `foo` !

• Conclusion : **par défaut, les variables introduites dans une fonction sont locales !**

Les procédures: les fonctions sans return

- Mathématiquement : aucun intérêt. Soit f : E —> ∅ ???

- Informatiquement, elles permettent de rédiger un programme en **style impératif**.
  - Style déclaratif (~math) : les *expressions* représentent ce qu'on veut calculer
  - Style impératif : les *instructions* sont des ordres pour modifier l'état (effet de bord).

- On appelle état tout ce qui permet de donner un sens aux expressions et aux instructions : les valeurs de variables globales, les entrées-sorties (fichiers ouverts, connections,…) , la position et le cap de la tortue…