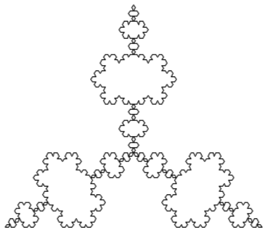
La Tortue

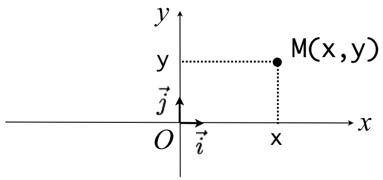


Les deux types de graphisme dans le plan

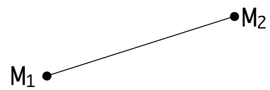
• Il y a deux types de graphisme 2D, mathématiquement parlant :

1. Le graphisme **CARTESIEN**

• Le plan est rapporté à un repère orthonormé direct $(O, \vec{i}, \vec{j})$.

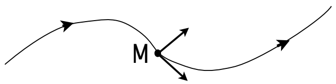


• Une seule opération essentielle :
tracer un segment du point $M_1(x_1,y_1)$ au point $M_2(x_2,y_2)$.



2. Le graphisme **POLAIRE**

• Aucune notion de coordonnées. L'animal traceur porte un repère mobile orthonormé avec une notion de droite et de gauche.



Au point M, la tortue va commencer à tourner sur sa gauche !

• Deux opérations essentielles :
- **tourner** à droite ou à gauche sur place d'un angle a
- **avancer** dans la direction courante d'une distance d

• Opérateurs de ***translation*** et de ***rotation*** plane, qui engendrent le *groupe des déplacements*. La tortue se déplace dans le plan !

• Graphisme moins matheux, plus intuitif. Inutile de calculer les coordonnées des points...

• Une trajectoire qui semble *lisse* sera en fait un polygone !

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.

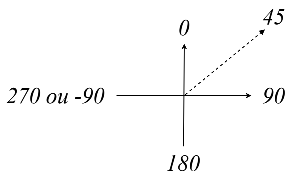
• Ne nommez PAS votre fichier `turtle.py` !

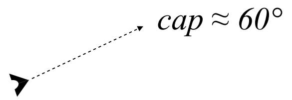
• Placez en dernière ligne de votre fichier l'instruction `mainloop()`.

Le graphisme cartésien

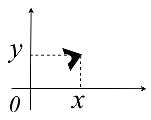
• C'est celui des matheux dans la mesure où il faut calculer les coordonnées des points à relier.

• 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*.

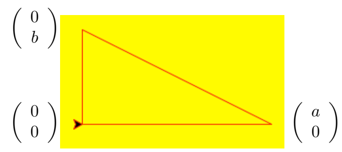
La position	Le cap	Le crayon

`pos()`
`goto(x,y)`
	`heading()`
	`setheading(a)`
		`down()`
		`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)
```



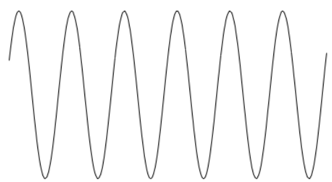

```
bgcolor('yellow')
pencolor('red')
tri_rect(200,100)
```

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

• Exemple : la **courbe** du cosinus comme suite de petits segments !

```
def trace_fonction(f,a,b,dx) :
    x = a ; y = f(x)
    up() ; goto(x,y) ; down()
    while x < b :
        xs = x + dx ; ys = f(xs)
        goto(xs,ys)
        (x,y) = (xs,ys)
```

Code à maîtriser !



```
>>> reset()            # effacement du canvas, réinitialisation
>>> hideturtle()       # pour cacher la tortue
>>> tracer(False)      # tracer(True)  pour une exécution lente !
>>> from math import *
>>> trace_fonction(lambda x : 100 * cos(x/10),-200,200,1.0)
```

• Comme `goto` ou `trace_fonction`, la plupart des fonctions de dessin n'ont **pas de résultat**, seulement des **effets**.

• Le fichier `turtle.cfg`, s'il existe, vous permettra de fixer les dimensions du canvas, la forme de la tortue, et d'autres réglages...

Le **tuple**, une donnée composée

(x,y) = (xs,ys)

• Vous avez noté la présence de **tuples**, ici un couple `(x,y)`. Un triplet se noterait `(x,y,z)`, etc.

• Le résultat de la fonction `pos()` est un couple. Les composantes d'un tuple `p` se notent `p[0]`, `p[1]`, `p[2]`...

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

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

• La très intéressante **affectation entre tuples** :

```
>>> (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...

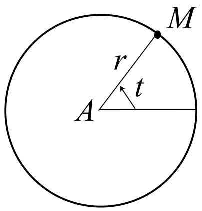
Courbes en **coordonnées paramétriques**

• La **cinématique** (étude du mouvement) s'intéresse à la trajectoire d'un corps dont les coordonnées (x,y) sont fonction d'un ***paramètre*** t. Autrement dit : $x = x(t)$ et $y = y(t)$, pour t dans un certain intervalle I.

• Ces courbes englobent les courbes $y = f(x)$ mais sont plus générales !

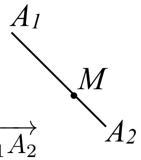
• Exemple : le **cercle** de centre A(a,b) et de rayon r n'est autre que la trajectoire d'un mobile dont les coordonnées sont données par :

$$M \quad \boxed{\begin{array}{l} x = a + r\cos(t) \\ y = b + r\sin(t) \end{array}}$$



• Exemple : le **segment** A_1A_2 joignant le point $A_1(a_1,b_1)$ au point $A_2(a_2,b_2)$ est la trajectoire paramétrée par :

$$M \quad \boxed{\begin{array}{l} x = t\,a_1 + (1 - t)\,a_2 \\ y = t\,b_1 + (1 - t)\,b_2 \end{array}} \qquad \overrightarrow{MA_2} = t\overrightarrow{A_1A_2} \quad t \in [0,1]$$

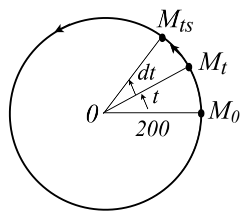


• **Animation** de la tortue parcourant un cercle de centre O et de rayon 200. Le caractère *continu* du mouvement est une illusion d'optique. En fait il est **discrétisé** : le paramètre `t` avance chaque fois de `dt`.

• Le choix de `dt` peut être empirique, guidé par l'esthétique de la simulation. Mais si l'on approche un cercle par un polygone à 40 côtés, on est conduit à prendre `dt = 2*pi/40 ≈ 0.16` radians.

```
def anim_cercle(r) :
    dt = 2 * pi / 40
    t = 0 ; x = r ; y = 0          # départ
    up() ; goto(x,y) ; down()      # en M0
    while t < 6 * pi :             # 3 tours
        ts = t + dt
        xs = r * cos(ts) ; ys = r * sin(ts)
        goto(xs,ys)
        t = ts
```

Comparez avec le code page 7




```
tracer(True)
showturtle()
anim_cercle(200)
```


Le graphisme polaire

• Il s'agit du *vrai* graphisme tortue pour les puristes...

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

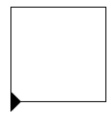
Le cap	La position
`left(a)`	`forward(d)`
`right(a)`	`back(d)`
`towards(p)`	

• 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.

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

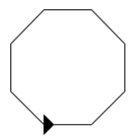
```
def carre(c) :
    i = 0
    while i < 4 :
        forward(c)
        left(90)
        i = i + 1
```



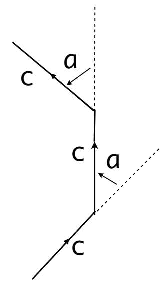
`carre(100)`

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

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



`polygone(8,100)`



Rappel : la boucle for

• Bien pratique lorsque l'on connaît à l'avance le nombre d'itérations.

```
def carre(c) :
    for i in range(4) :      # i = 0..3
        forward(c)
        left(90)
```

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

• `range(n)` pour parcourir `[0,n-1]`

• `range(i,j)` pour parcourir `[i,j-1]`

• `range(i,j,k)` pour parcourir `[i,j-1] de k en k`

• *Grosso modo*, si `i` est une variable inutilisée par ailleurs :

```
for i in range(5,10) :
    <instr>
```

⇔

```
i = 5 ;
while i < 10 :
  <instr>
  i = i + 1
```

• *Attention, la variable i n'est (hélas) pas locale à la boucle :*

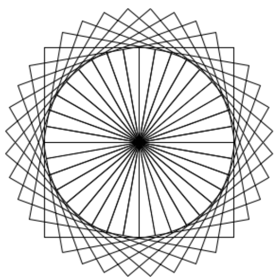
```
>>> i = 1000
>>> for i in range(5) :
        print(i,end=' ')

0 1 2 3 4
>>> i
4
```

• Vous verrez plus tard que la boucle for fonctionne sur n'importe quel **itérateur** en Python, y compris ceux que vous construirez vous-mêmes.

• Exemple de dessin obtenu par des carrés en rotation.

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

```
def fleur() :
    for i 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 plus utilisable par ailleurs !

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

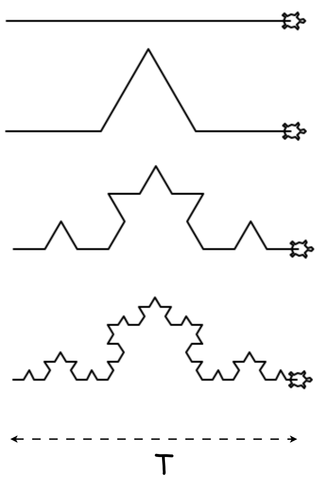
Une fonction locale !

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 :

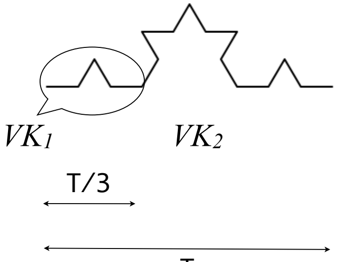
- VK_0 est un segment de longueur T
- VK_1 s'obtient par chirurgie sur VK_0
- VK_2 s'obtient par la même chirurgie sur chaque côté de VK_1
- VK_3 s'obtient par la même chirurgie sur chaque côté de VK_2

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 :

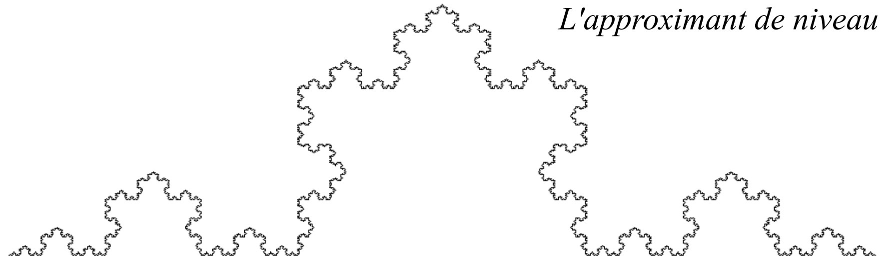
```
def von_koch(n,T) :    # approximant de niveau n et de base T
    if n == 0 :
        forward(T)
    else :
        von_koch(n-1,T/3)
        left(60)
        von_koch(n-1,T/3)
        right(120)
        von_koch(n-1,T/3)
        left(60)
        von_koch(n-1,T/3)
```

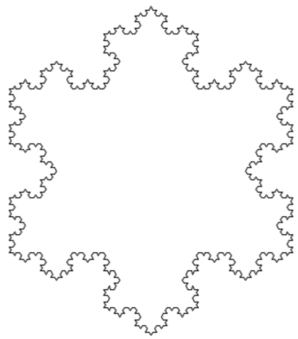


• La courbe de Von Koch VK est la "limite" de la suite : $VK = \lim_{n \to +\infty} VK_n$

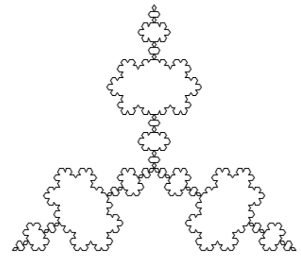
• Découverte en 1906, VK possède d'étranges propriétés. Par exemple, elle est continue mais n'admet de tangente en <u>aucun</u> point !!

L'approximant de niveau 6





Le flocon de Von Koch



L'antiflocon

Les variables <u>locales</u>...

• 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 .

• 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) :
    i = 0
    while i < 4 :
        ...
```

```
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 <u>globales</u>

• 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 !**

• Pourquoi Python a-t-il fait ce choix ? Pour **décourager autant que possible l'utilisation de variables globales** et donc en particulier le style procédural (page suivante). Dont acte !

Le style procédural : des fonctions à effets de bord (?)

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

- Informatiquement, elles produisent une manière bizarre de rédiger un texte de programme. C'est le style procédural. Plutôt que retourner un résultat comme en maths, la fonction va modifier une variable globale définie en-dehors d'elle (effet de bord) ! Brrr…

Le style **normal**
(les fonctions retournent leur résultat)

```
def cube(x) :
    return x * x * x
```

```
>>> cube(2)      # retourne son résultat
8
```

Le style **procédural**
(fonctions sans résultat)

```
c = 0
def cube(x) :
    global c
    c = x * x * x
```

BEURK !

```
>>> cube(2)     # aucun résultat,
>>> c           # mais un effet.
8               # A EVITER !!!
```