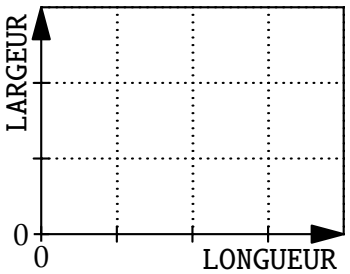
Tracer les fractales

Préparatifs

Afin de réaliser les figures suivantes, vous allez avoir besoin du fichier `module_graphique.py` qui se trouve dans `Echange/Tous/NSI`. Il faut copier ce fichier dans le dossier où vous mettrez vos autres fichiers Python.
Ce module contient les fonctions de bases permettant de tracer des segments, des cercles ou des rectangles.
Vos fichiers doivent commencer par les lignes suivantes :

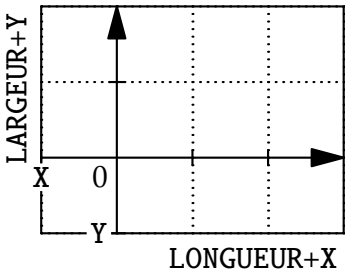
```
from module_graphique import *

figure.modifie_taille(LONGUEUR, LARGEUR)
```

Les valeurs `LONGUEUR` et `LARGEUR` représentent les dimensions de l'images produite.
Par défaut, le centre du repère se trouve en bas à gauche, comme indiqué sur l'image ci-contre.



Il est possible de décaler l'origine du repère en rajoutant un troisième argument correspondant aux coordonnées souhaitée pour le coin inférieur gauche.

```
figure.modifie_taille(LONGUEUR, LARGUEUR, (X, Y))
```



Pour centrer le repère on peut donc mettre `(LONGUEUR/2, LARGEUR/2)`. On peut aussi utiliser la valeur spéciale `'center'`.

```
figure.modifie_taille(LONGUEUR, LARGUEUR, 'center')
```

Chacun de vos fichier doit se terminer par l'instruction suivante :

```
figure.enregistre('nom_fichier.svg')
```

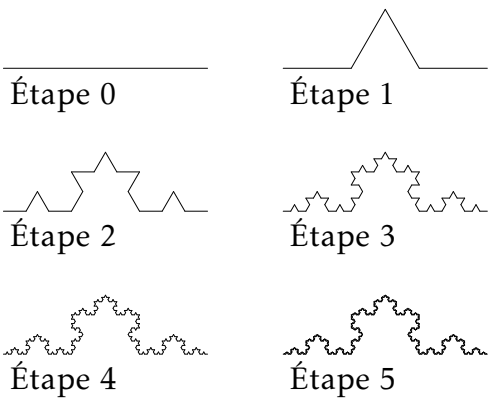
Cela permet d'enregistrer l'image dans un fichier `svg` qu'il faut ouvrir pour voir le résultat.

Premières fractales

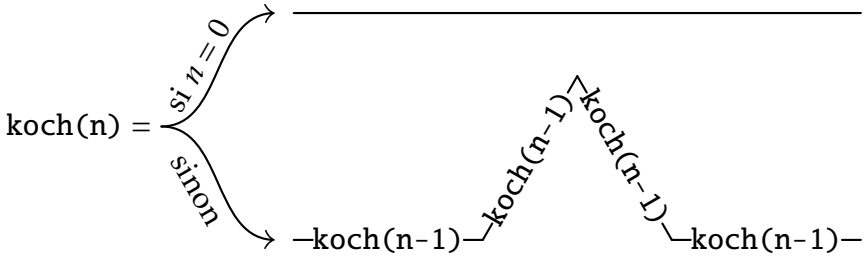
Pour tracer la courbe de Koch à l'étape n, on peut appliquer l'algorithme suivant :

- Si on est à l'étape 0, on trace un segment.
- Sinon, on trace 4 fois la courbe à l'étape $n-1$.

Pour tracer une courbe il faut donc connaître le point de départ, celui à gauche, la longueur du segment reliant le départ et la fin, et l'angle que forme la courbe avec l'axe des abscisses. Les angles utilisés seront donnés en radians. Dans le cas de la courbe de Koch, ce sera des multiples de $\frac{\pi}{3}$.



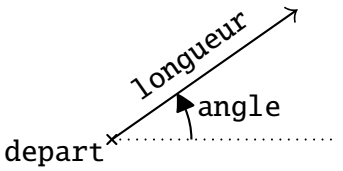
On peut simplifier cet algorithme ainsi :



Cela donne le code suivant :

```
from module_graphique import *

figure.modifie_taille(400, 130)

def koch(etape, depart, longueur, angle=0):
    if etape == 0:
        fin = translation(depart, longueur, angle)
        segment(depart, fin, epaisseur=.2)
    else:
        longueur = longueur / 3
        a1 = pi/3
        for a in [0, a1, -a1, 0]:
            koch(etape-1, depart, longueur, a+angle)
            depart = translation(depart, longueur, a+angle)

etape = 5
depart = (10, 10)
longueur = 380

koch(etape, depart, longueur)
figure.enregistre('koch.svg')
```

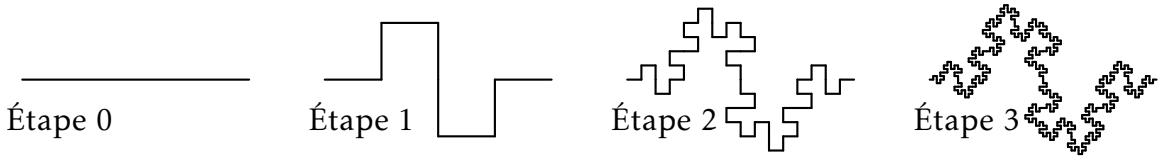
La fonction `translation` renvoie les coordonnées du point obtenu en partant du point `depart`, en se déplaçant selon un vecteur de norme `longueur` formant un angle `angle` par rapport à l'axe des abscisses.



La fonction `segment` trace un segment entre les deux points donnés. On peut rajouter des arguments comme `epaisseur` pour changer l'épaisseur du trait ou `couleur` pour modifier la couleur. On peut donner une couleur en anglais, comme `'red'` ou `'blue'`. On peut aussi donner une valeur en hexadécimal, comme `'#ff30ba'`.

Exercice 1 : Recopier le code précédent dans un nouveau fichier, l'exécuter et ouvrir l'image obtenue.

Exercice 2 : En s'inspirant de l'exemple précédent, faire un nouveau fichier permettant de tracer la saucisse de Minkowski.



Principe de la récursivité

On peut remarquer que la fonction **koch** s'appelle elle-même. On dit que c'est une **fonction récursive**.
Certaines suites sont définies de façon récursives. Pour cela, on donne la valeur du **terme initial** et une formule de récurence permettant de calculer le terme de rang $n+1$ à partir de celui de rang n. Par exemple, on peut définir la suite (u_n) telle que $u_0 = 0$ et $u_{n+1} = u_n^2 + 1$.
Il existe des suites qui sont définies à l'aide de plusieurs termes initiaux et avec une formule de récurence qui utilise plusieurs termes précédents. C'est le cas de la suite de Fibonacci, qui est la suite (F_n) telle que $F_0 = 0$, $F_1 = 1$ et $F_{n+2} = F_{n+1} + F_n$.
Pour calculer un terme particulier d'une suite, par exemple u_3, on doit calculer les termes précédents. On peut le faire en partant des termes initiaux :

$$\begin{aligned} u_0 &= 0 \\ u_1 &= u_0^2 + 1 = 0^2 + 1 = 1 \\ u_2 &= u_1^2 + 1 = 1^2 + 1 = 2 \\ u_3 &= u_2^2 + 1 = 2^2 + 1 = 5 \end{aligned} \qquad \begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_2 &= F_1 + F_0 = 1 + 0 = 1 \\ F_3 &= F_2 + F_1 = 1 + 1 = 2 \end{aligned}$$

On peut aussi faire le conctraire et partir du terme à calculer et développer l'expression jusqu'à arriver aux termes initiaux.

$$\begin{aligned} u_3 &= u_2^2 + 1 \\ &= (u_1^2 + 1)^2 + 1 \\ &= ((u_0^2 + 1)^2 + 1)^2 + 1 \\ &= ((0^2 + 1)^2 + 1)^2 + 1 \\ &= 5 \end{aligned} \qquad \begin{aligned} F_3 &= F_2 + F_1 \\ &= F_1 + F_0 + F_1 \\ &= 1 + 0 + 1 \\ &= 2 \end{aligned}$$

Dans ces deux cas, cela revient sensiblement au même. Ce n'est pas toujours le cas en terme de simplicité ou de temps de calcul. Pour programmer le calcul du terme de rang n pour chacune de ces suites, on peut utiliser les deux approches. La première est dite **itérative** alors que la deuxième est **récursive**.

```python
def u(n):   # version itérative
    val = 0
    for i in range(n):
        val = val**2 + 1
    return val
```

```python
def u(n):   # version récursive
    if n == 0:
        return 0
    else:
        return u(n-1)**2 + 1
```

```python
def fibo(n):   # version itérative
    u, v = 0, 1   # F0 et F1
    for i in range(n):
        u, v = v, u + v   # on avance
    return u
```

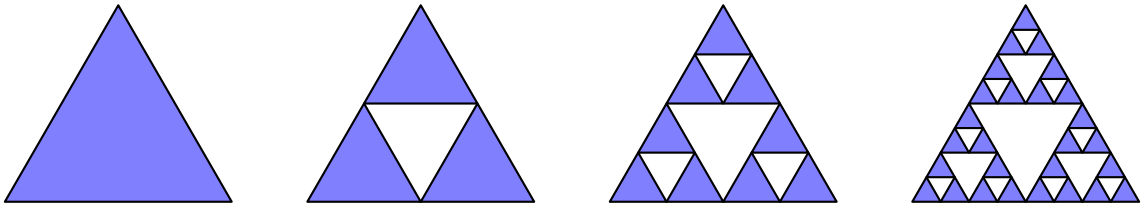
```python
def fibo(n):   # version récursive
    if n == 0 or n == 1:
        return n
    else:
        return fibo(n-1) + fibo(n-2)
```

Dans les versions itératives, on construit explicitement une boucle alors que dans les versions récursives, c'est le fait d'arriver au cas de base qui permet d'arrêter la suite d'appels.
Pour des objets définis de façon récursive, comme c'est le cas pour les fractales, la deuxième approche est généralement plus simple à mettre en œuvre.

D'autres fractales

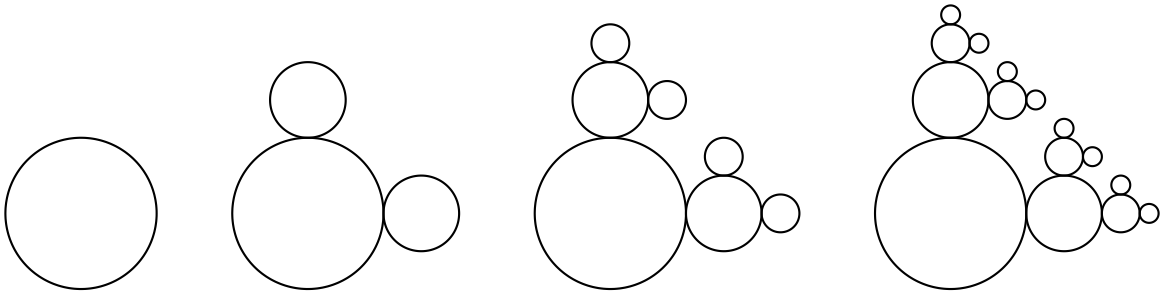
Exercice 3 : Créer un nouveau fichier qui permet de tracer le triangle de Sierpinski. Pour cela, on pourra utiliser la fonction ci-dessous :

```
def triangle(depart, longueur):
    sommet1 = depart
    sommet2 = translation(sommet1, longueur, 0)
    sommet3 = translation(sommet1, longueur, pi/3)
    segments([sommet1, sommet2, sommet3], remplissage='blue', epaisseur=.2)
```



Étape 0 Étape 1 Étape 2 Étape 3

Exercice 4 : Créer un nouveau fichier qui permet de tracer la fractale ci-dessous.



Étape 0 Étape 1 Étape 2 Étape 3

Vous pourrez écrire une fonction `cercles` qui prend comme paramètres :

- `etape` : le numéro de l'étape à afficher.
- `centre` : les coordonnées du centre du plus gros cercle qui reste à tracer.
- `rayon` : le rayon du plus gros cercle qui reste à tracer.
- `k` : le facteur de réduction
- `epaisseur` : l'épaisseur du plus gros cercle qui reste à tracer.

Initialement, vous pouvez prendre `k` = 0,5 et ne pas mettre le paramètre sur l'épaisseur du trait. Vous pouvez utiliser la fonction `cercle` qui prend le centre et le rayon. On peut rajouter un paramètre nommé `epaisseur=...` et éventuellement `remplissage='lightblue'` ou une autre couleur :

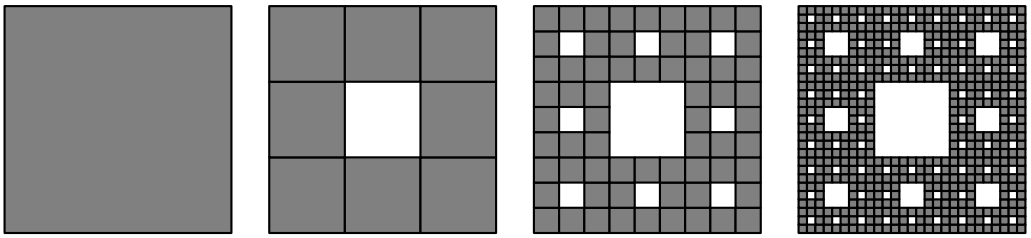
```
  cercle(centre, rayon, epaisseur=epaisseur, remplissage=couleur)
```

Pour les exercices suivants, vous pouvez la fonction `rectangle` de la manière suivante :

```
rectangle(bas_gauche, longueur, hauteur, couleur='none', remplissage='black')
```

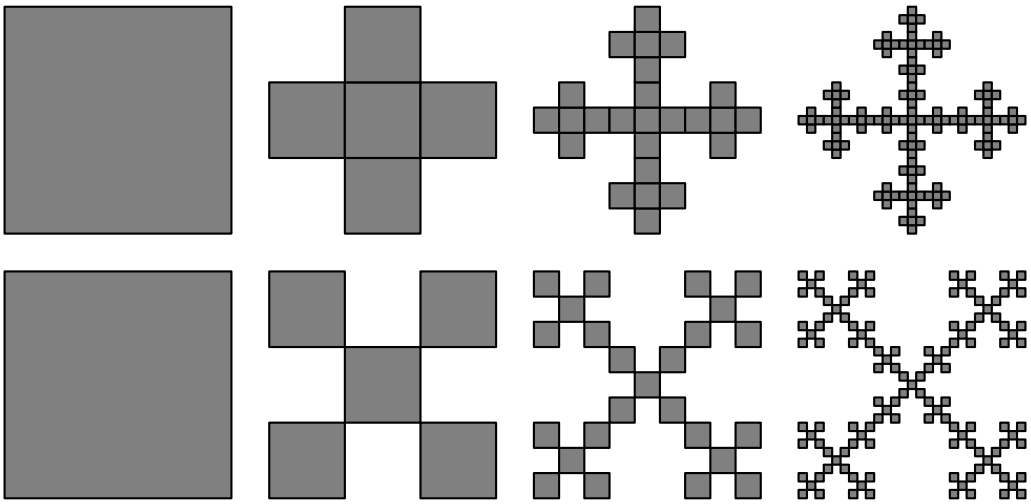
Le point `bas_gauche` correspond aux coordonnées du coin inférieur gauche du rectangle.

Exercice 5 : Créer un nouveau fichier qui permet de tracer le **tapis de Sierpinski**.



Étape 0 Étape 1 Étape 2 Étape 3

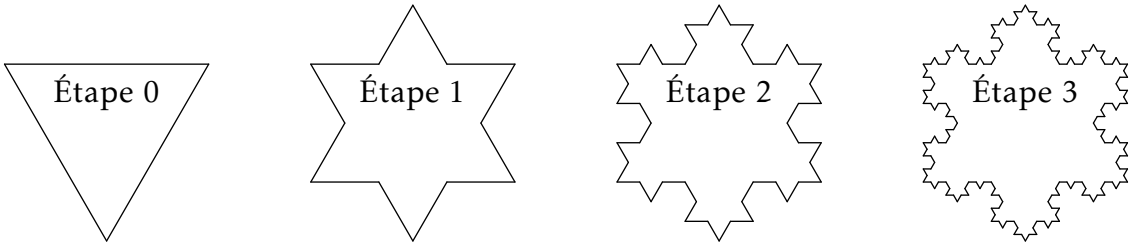
Exercice 6 : Créer un nouveau fichier qui permet de réaliser une des deux fractales de Vicsek ci-dessous :



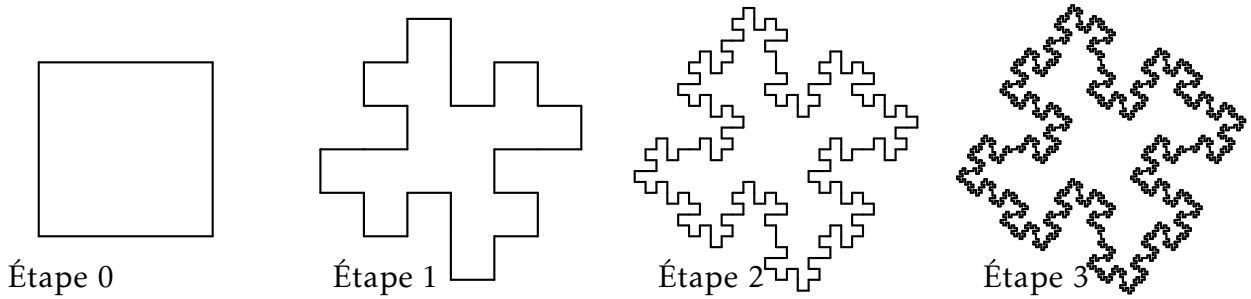
Étape 0 Étape 1 Étape 2 Étape 3

Des fractales plus complexes

Exercice 7 : En reprenant la fonction **koch**, créer un nouveau fichier permettant de réaliser le **flocon de Koch** composé de 3 courbes de Koch.

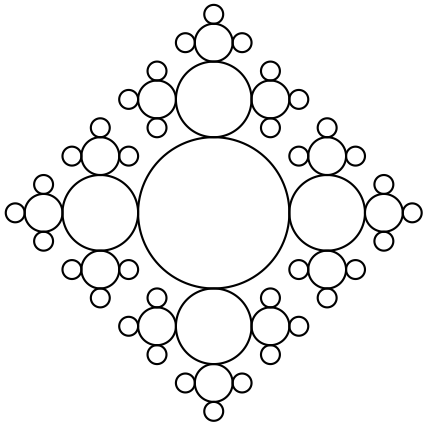


Exercice 8 : De la même manière créer un fichier permettant de réaliser **l'île de Minkowski**, composée de 4 saucisses de Minkowski.



Étape 0 Étape 1 Étape 2 Étape 3

Exercice 9 : Créer un nouveau fichier qui permet de tracer la fractale ci-dessous. Vous pouvez partir de votre fonction `cercles` à laquelle vous rajouterait un autre paramètre `origine` qui est un couple de coordonnées indiquant dans quel sens se trouve le cercle précédent. Les directions peuvent être `(1, 0)`, `(0, 1)`, `(-1, 0)` et `(0, -1)`. Pour faire les appels récursifs, vous parcourerez les différentes directions et vous ne ferez rien dans celle qui correspond à l'origine. Pour le premier appel, vous pourrez donner une origine qui n'existe pas, comme `(0, 0)`.



Zoom infini

Puisque la plupart des fractales que nous avons tracées jusqu'à présent sont auto-similaires, il est possible de zoomer dedans pour retrouver la même image. Le type de fichier `svg` peut justement contenir des animations, comme des zooms.
Nous alons réaliser un zoom infini sur le tapis de Sierpinski.

Exercice 10 : Modifier votre fichier pour que l'origine du repère se trouve en haut à gauche et que le plus grand carré occupe tout l'espace de la figure. Vous pouvez mettre le nombre d'étapes à 4 pour commencer.

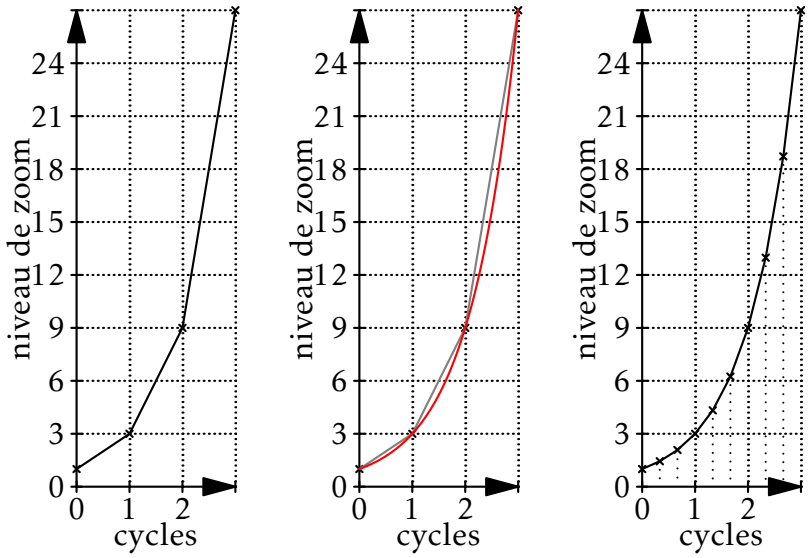
Exercice 11 :

1) Rajouter la ligne suivante après l'appel à `figure.modifie_taille` en remplaçant `Z` par le niveau de zoom qu'il faut pour retrouver une image similaire à l'image de départ :

```
figure.zoom('2s', '1,1;Z,Z', 'indefinite')
```

   Cela va produire un zoom qui dure 2 secondes, qui sera répété en boucle et qui commencera par l'échelle 1 sur les deux axes pour finir à l'échelle `Z`.
2) Générer l'image et l'ouvrir avec Firefox pour observer le résultat.

Vous pouvez remarquer que le zoom semble ralentir progressivement avant d'accélérer brutalement à chaque nouveau cycle. C'est parce que la vitesse du zoom est linéaire (première courbe), alors qu'elle devrait être exponentielle (deuxième courbe). On peut approximer cette courbe en indiquant plus d'étapes (troisième courbe).



Exercice 12 : Remplacer la ligne avec `figure.zoom` par les lignes suivantes et tester le résultat obtenu :

```
def etapes_zoom(nb_etapes):
    res = []
    for i in range(nb_etapes+1):
        k = 3**(i/nb_etapes)
        res.append(f"{k}, {k}")
    return ";".join(res)

zooms = etapes_zoom(4)
```

Exercice 13 : Vous pouvez appliquer cette technique sur d'autres fractales, comme celles de Vicsek.

L'ensemble de Mandelbrot

Pour travailler sur l'ensemble de Mandelbrot, il faut travailler sur les nombres complexes. Python peut le faire nativement :

```
>>> z = complex(1, 2)   # Définition d'un nombre complexe
>>> z
(1+2j)        # j remplace i dans la notation
>>> z * z     # On peut faire des calculs
(-3+4j)
>>> abs(z)   # Calcul de la norme
2.23606797749979
```

Exercice 14 : Écrire une fonction `bornee` qui prend un nombre complexe `c` et un entier `iter_max` et qui renvoie un booléen indiquant si la suite associée à `c` a un terme dont le module dépasse 2 pour un rang inférieur à `iter_max`.

```
>>> bornee(complex(1, 1))
False
>>> bornee(complex(1, 0))
False
>>> bornee(complex(-1, 0))
True
>>> bornee(complex(-.2, .4))
True
>>> bornee(complex(-.4, .5))
True
>>> bornee(complex(-.4, .8))
False
>>> bornee(complex(.3, 0))
False
```

Exercice 15 : Compléter le code de la fonction `affiche` qui prend en entrée les valeurs `xmin`, `xmax`, `ymin`, `ymax`, `pasx` et `pasy` et qui affiche l'ensemble de Mandelbrot pour les points $M(x;y)$ tels que $\texttt{xmin} \leq x \leq \texttt{xmax}$ et $\texttt{ymin} \leq y \leq \texttt{ymax}$. Les valeurs de x partent de `xmin` et vont de `pasx` en `pasx`. Les valeurs de y vont de `ymax` et diminue de `pasy` à chaque étape. Si le point appartient à l'ensemble on affiche $*$. Sinon on affiche un espace `' '`.

```
def afficher(xmin=-1.6, xmax=0.6, ymin=-1, ymax=1, pasx=.025, pasy=0.05):
    y = ...
    while ...:
        x = ...
        ligne = ""
        while ...:
            if bornee(...):
                ligne += ...
            else:
                ligne += ...
            x = ...
        print(ligne)
        y = ...
```

```
afficher(xmin=-1.6, xmax=0.6, ymin=-.9, ymax=.9, pasx=.025, pasy=0.05)
```

Exercice 16 : Pour voir un peu comment la suite (z_n) se comporte en fonction des valeurs de c, vous pouvez utiliser ces pages :

- https://www.geogebra.org/m/BUVhcRSv#material/Npd3kBKn
- https://www.geogebra.org/m/BUVhcRSv#material/XQprvGbW

Exercice 17 : Pour faire de vraies images, vous pouvez utiliser le fichier `mandelbrot-final.py` qui permet de générer des images en couleur. Vous pouvez explorer l'ensemble en choisissant les coordonnées du centre de l'image et le niveau de zoom. Vous pouvez même faire des animations.