

# TP : Turtle & Langages de programmation

## I. Démarrage

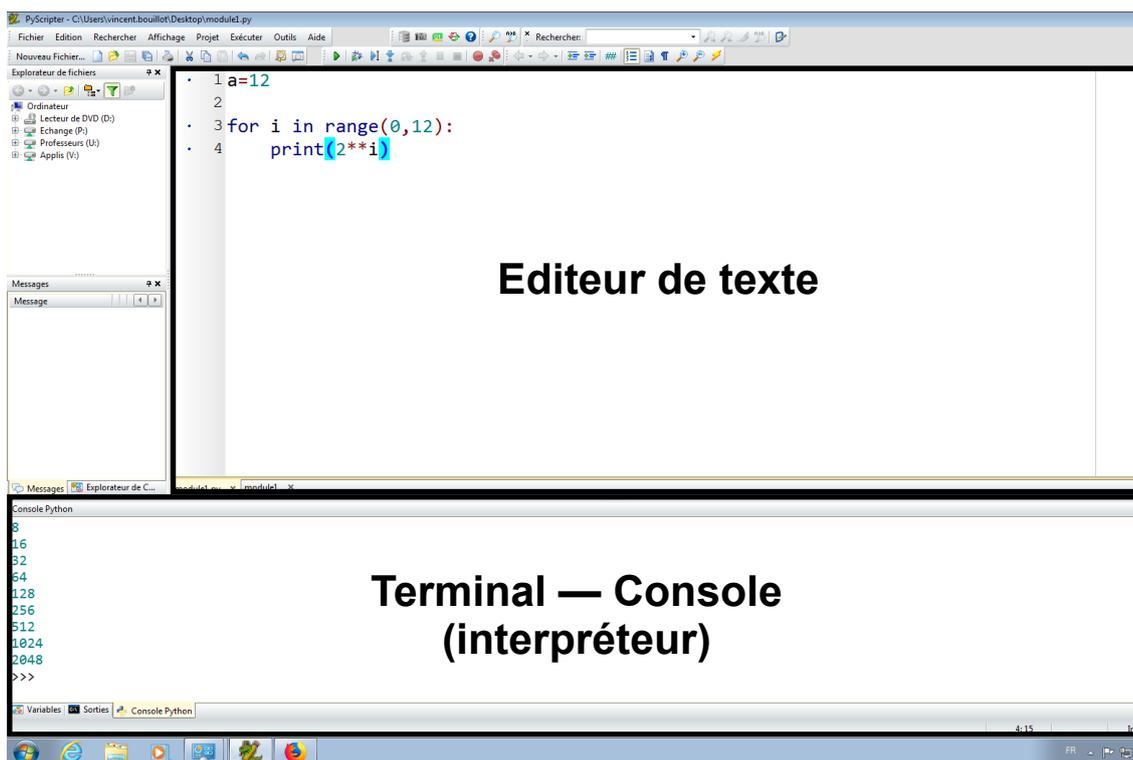
### 1) Environnement de travail

Pour créer des codes informatiques, il nous faut deux outils principaux :

- ❖ un **éditeur de texte** permettant d'écrire notre programme en langage Python ;
- ❖ un **interpréteur ou compilateur** pour exécuter le programme et le rendre compréhensible par l'ordinateur.

Pour commencer, nous utiliserons un environnement complet où l'éditeur de texte et l'interpréteur sont rassemblés dans le même programme. Pour nous, cet environnement sera EduPython (ou Pyzo si nous rencontrons des problèmes avec EduPython).

Une fois Edupython lancé, vous obtenez quelque chose qui ressemble à cela :

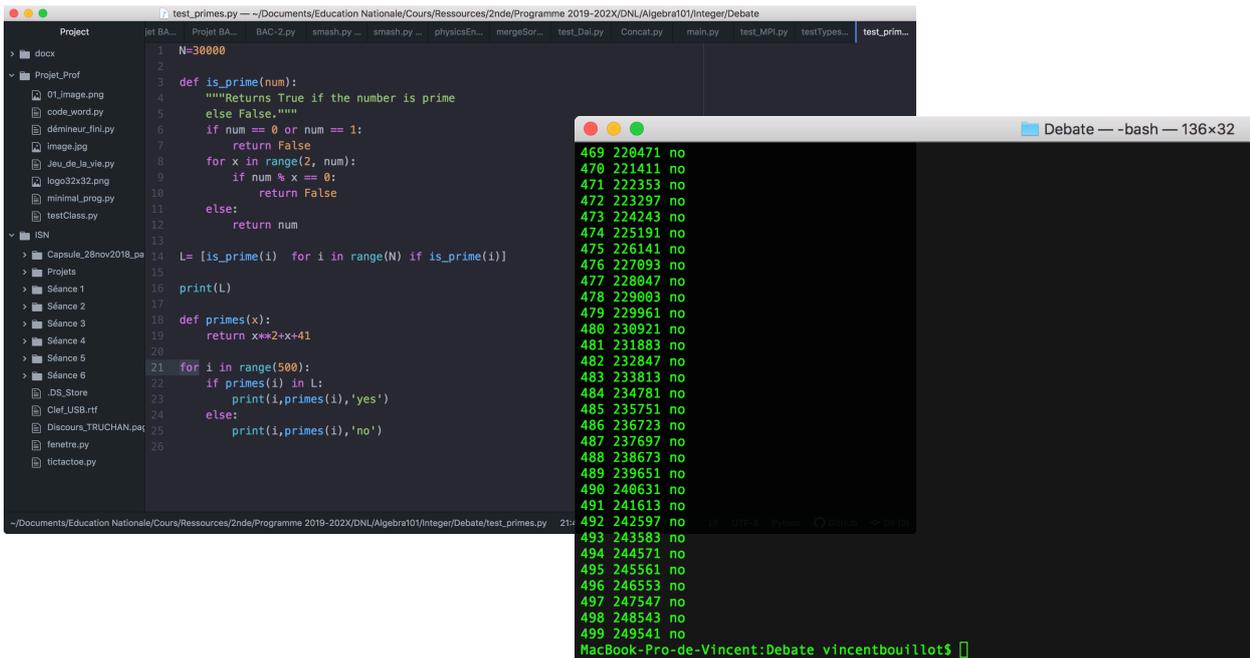


EduPython comporte deux fenêtres principales : la fenêtre "éditeur" et la fenêtre "console". L'éditeur de texte permettra d'écrire les algorithmes, comme nous les avons écrit en cours alors que la console permettra :

- ❖ **d'observer le résultat** de l'exécution de son code ;
- ❖ de tester des instructions Python écrites directement.

Par la suite, en travaillant sur les Raspberry Pi et Linux, notre environnement de travail sera plus flexible.

L'éditeur de texte (**Atom**, **VScode** ou **Geany**) sera différencié du terminal où nous exécuterons le programme.



## 2) Prise de contact avec l'environnement de travail

### — Exercice 1 —

Dans la fenêtre "éditeur", saisissez le programme suivant :  
**print("hello world !")**

Cliquez sur le "triangle vert" dans la barre d'outil afin d'exécuter ce (court) programme qui vient d'être saisi.

EduPython va vous demander d'enregistrer le programme, enregistrez-le dans un dossier qui vous servira de dossier de travail temporaire.

Vous devez voir le message "hello world !" apparaître dans la **console**.

### — Exercice 2 —

Dans la fenêtre "Terminal", écrivez la même ligne de code. Que constatez-vous ?

## 3) Bibliothèque turtle

Tous les langages de programmation proposent des "outils" déjà préparés, proposant des collections d'instructions spécialisées permettant de faire des tâches spécifiques. Cela simplifie la vie de l'utilisateur en lui évitant de réécrire de nombreux codes informatiques : ces outils sont appelés "bibliothèque" ou "module". Nous allons travailler aujourd'hui sur les notions de variables, de boucles et de conditionnelles grâce à une bibliothèque graphique appelée `turtle`. Les instructions de ce module permettent de déplacer une tortue munie

d'un crayon à la surface d'une feuille virtuelle. On peut alors observer l'exécution du programme à travers les mouvements de la tortue et de son tracé.

Les instructions de `turtle` comprennent donc des moyens d'orienter et de déplacer la tortue sur la feuille (en deux dimensions). Pour cela, on utilise un repère standard des mathématiques avec une abscisse sur l'axe horizontal et une ordonnée sur l'axe vertical. Les longueurs sont mesurées en pixels et les angles en degrés.

Voici un résumé des instructions classiques de déplacement en `turtle` :

Instruction	Description
<code>goto(x,y)</code>	aller au point de coordonnées (x,y)
<code>forward(d)</code> ou <code>fd(d)</code>	avancer de la distance d
<code>backward(d)</code> ou <code>bk(d)</code>	reculer de la distance d
<code>left(a)</code> ou <code>lt(a)</code>	pivoter à gauche de l'angle a
<code>right(a)</code> ou <code>rt(a)</code>	pivoter à droite de l'angle a
<code>circle(r,a)</code>	tracer un arc de cercle d'angle a et de rayon r
<code>dot(r)</code>	tracer un disque de rayon r centré sur la tortue

À cela s'ajoute des instructions permettant de modifier les dessins produits par chacun des déplacements :

Instruction	Description
<code>penup()</code> ou <code>pu()</code>	relever le crayon (interruption du dessin)
<code>pendown()</code> ou <code>pd()</code>	redescendre le crayon (reprise du dessin)
<code>pensize(e)</code> ou <code>width(e)</code>	fixer à e la largeur du crayon
<code>color(c)</code>	sélectionner la couleur c pour le tracé
<code>begin_fill()</code>	activer le mode remplissage
<code>end_fill()</code>	désactiver le mode remplissage
<code>fillcolor(c)</code>	sélectionner la couleur c pour le remplissage
<code>speed(s)</code>	définir la vitesse de déplacement (1 à 10, 0=instantané)

Pour plus de précisions sur les fonctions à votre disposition avec la bibliothèque `Turtle`, vous pouvez aller voir sur : <https://docs.python.org/fr/3/library/turtle.html>

## II. Structures de contrôle avec Turtle

### 1) Intérêt et manipulation des variables

#### — Exercice 3 —

Commençons par importer le module turtle. Le plus simple est de simplement dire :

```
import turtle
```

Faites cela dans un nouveau programme que vous enregistrerez sous le nom de TP\_Turtle.py .

Cette méthode pour charger une bibliothèque est le plus précis. Tout **outil** disponible dans le **module** est alors accessible en appelant **module.outil**.

Rem. : Une autre méthode consiste à importer un module en le renommant localement :

```
import module as m.
```

On appelle alors un **outil** à l'aide de **m.outil**.

#### — Exercice 4 —

C'est le moment de tester notre tortue que nous allons appeler fred ! Voici un premier programme :

```
fred = turtle.Turtle()
fred.goto(0, 0)
fred.forward(70)
fred.right(90)
fred.forward(50)
turtle.exitonclick() # fenêtre ouverte jusqu'à un clic
```

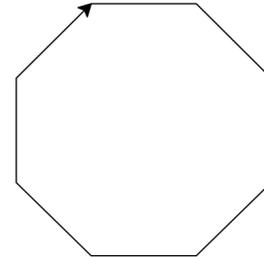
- Sur une feuille de brouillon, dessinez ce que vous pensez obtenir avec ce programme.
- Complétez votre programme avec les lignes ci-dessus puis observez le résultat. Est-ce conforme à vos attentes ?
- Modifiez le code ci-dessus afin de tracer un carré de 100 pixels de côté.
- Modifiez à présent votre programme afin d'afficher un carré de 150 pixels de côté. Que pensez-vous de la méthode que vous avez utilisé ?

Le changement de 100 pixels à 150 pixels est assez pénible à faire avec des copier/coller. Mais bon, il n'y en a que 4 à faire...

### — Exercice 5 —

- a) En adaptant votre code précédent, réalisez un programme permettant d'obtenir un octogone régulier de 80 pixels de côté. Vous aurez la même syntaxe que ci-dessous

```
fred = turtle.Turtle()
.....
.....
.....
turtle.exitonclick()
```



- b) Changez le côté de votre octogone à 50 pixels.  
c) En utilisant une variable longueur que vous initialiserez à 30, changez rapidement le côté de votre octogone à 30 pixels.  
d) Essayez de changer la valeur de longueur et observez si votre octogone change de taille.

## 2) Boucles Tant que

Faire l'octogone était sympa mais au final, il y a énormément de copier/coller ! Un principe fondamental en informatique est le **DRY** : **Don't Repeat Yourself**.

L'introduction d'une variable a déjà amélioré notre code mais nous pouvons faire bien mieux en faisant ! En effet, pour notre octogone, nous avons 8 fois le même bloc d'instructions : avec une boucle, nous allons nous simplifier la vie !

### — Exercice 6 —

- a) Commentez votre code précédent en ajoutant un # devant chaque ligne. On peut également sélectionner toutes les lignes que l'on souhaite commenter, puis cliquer sur "Code source", puis "commenter le bloc".

```
longueur = 50
fred = turtle.Turtle()
nombreTraits = 0 # initialisation
while ..... :
    fred.forward(longueur)
    fred.right(45)
    nombreTraits = .....

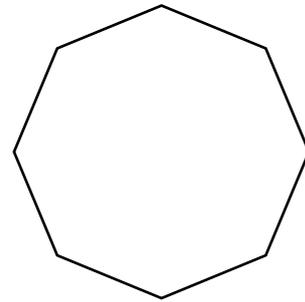
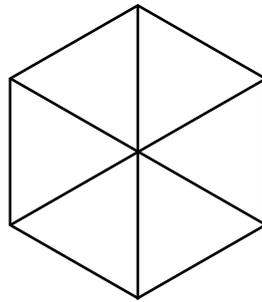
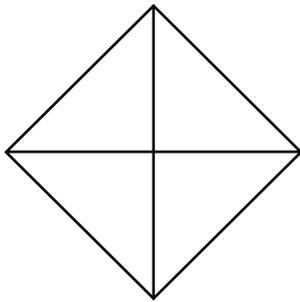
turtle.exitonclick()
```

b) Recopiez le code de la page précédente.

Dans ce code, la boucle "Tant que" va continuer tant que le nombre de Traits que nous avons fait est strictement inférieur à 8. À chaque fois que nous traçons un trait et tournons la tortue de 45 degrés, nous allons ajouter une unité à la variable nombreTraits. Faites les modifications nécessaires dans le code précédent. Si votre code fonctionne, vous devez toujours obtenir un octogone.

Il y a encore beaucoup trop de **nombres** dans notre code. En effet, si nous souhaitons faire un carré, il va falloir changer le nombre de côtés (8) ainsi que l'angle au centre (45). Si nous oublions l'un des deux, c'est le **bug**.

Mais... les maths nous disent que l'angle et le nombre de côtés d'un polygone sont reliés. Utiliser les deux informations est donc contraire au principe DRY !



Démonstration :

Un tour complet fait  $360^\circ$ .

Si je veux 4 côtés, quel est l'angle au centre ? \_\_\_\_\_

Si je veux 6 côtés, quel est l'angle au centre ? \_\_\_\_\_

Si je veux N côtés, quel est l'angle au centre ? \_\_\_\_\_



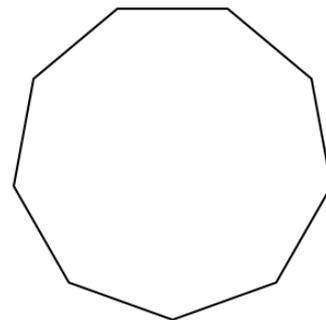
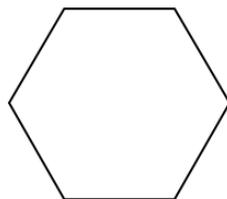
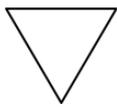
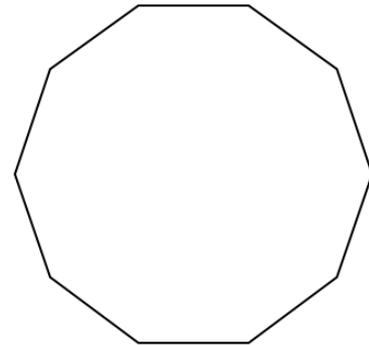
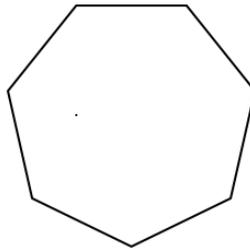
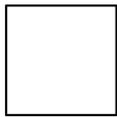
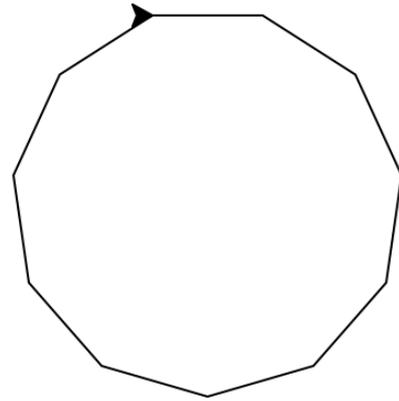
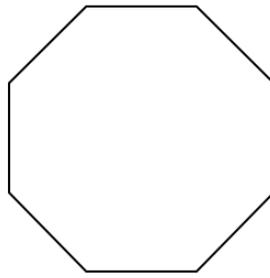
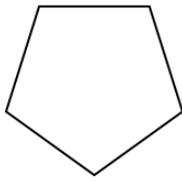
### — Exercice 7 —

- Par rapport à notre petite démonstration, quel est le meilleur choix : l'utilisateur donne l'angle au centre ou l'utilisateur donne le nombre de côté ?
- Introduire une variable nombreCote ainsi qu'une variable angle. Dans cet exercice, la valeur de la variable angle se calculera à partir de la variable nombreCote.
- Modifiez les informations présentes dans votre boucle "tant que" afin de prendre en compte les nouvelles variables. Testez votre code en faisant varier la variable nombreCote (Vous pouvez vérifier vos résultats grâce à la page 7).

Remarque :

Le nom d'une variable ne prend **jamais** d'accents. Le nom d'une variable doit indiquer ce que cette variable représente sans tomber dans l'excès.

On adopte la convention suivante : **maVariableQuiFaitUnTrucTresTresChouette**.



### Exercice 8

- a) Nous allons maintenant essayer d'avoir des comportements plus chaotiques : l'angle et la longueur ne sont plus reliés mathématiquement. Copiez votre programme permettant de réaliser le polygone. Dans le même fichier, collez le sous le programme actuel, puis commentez la première partie du code (voir ci-contre).
- b) Nous allons ajouter une variable **maxTraits** représentant le nombre maximum de traits que nous souhaitons tracer. Initialisez-la à 30. Que devez-vous changer dans votre programme pour que la boucle "Tant que" soit réalisée **30** fois ? **maxTraits** fois ? Effectuez cette modification.
- c) Testez votre programme. Que se passe-t-il ?

```
longueur = 50  
nombreCote = 10  
angle = 360/nombreCote
```

```
# fred = turtle.Turtle()  
# fred.goto(0,0)  
# fred.speed(0)  
# nombreTraits = 0 # initialisation  
# while nombreTrai  
#     fred.forwa  
#     fred.right  
#     nombreTrai
```

Ancien code commenté

```
fred = turtle.Turtle()  
fred.goto(0,0)  
fred.speed(0)  
nombreTraits = 0 # initialisation  
while nombreTrai  
    fred.forward  
    fred.right(a  
    nombreTraits = nombreTraits + 1
```

Nouveau code

### Remarque :

Un code fonctionnel ne doit **JAMAIS** être supprimé en cours de développement. Vous pourriez en avoir besoin au cas où votre nouveau code ne fonctionne pas.

Une bonne pratique est de commenter votre code fonctionnel :

- ❖ avec des hashtag # si nous avons une ligne à commenter ;
- ❖ avec des triple guillemets si nous avons plusieurs lignes à commenter.

""" texte sur plusieurs lignes

à commenter

""".

### ☐ — Exercice 9 —

Choisissez longueur = 100, nombreCote = 10 et maxTraits = 30.

a) Pour réaliser une spirale nous allons diminuer la longueur de chaque trait de 1 pixel à chaque passage dans la boucle.

Pour cela, nous allons ajouter dans notre programme l'affectation :

longueur = longueur - 1

b) Testez votre programme : obtenez-vous bien une spirale ? .....

c) Comment varie la spirale si vous diminuez la longueur de chaque trait :

de 2 pixels ? .....

de 3 pixels ? .....

de 5 pixels ? .....

d) Affichez la valeur de la variable longueur dans votre boucle et expliquez le comportement étrange de votre spirale si on enlève 5 pixels.

### 3) Conditionnelles Si, Sinon

Les conditionnelles permettent de réaliser une partie du code selon une certaine condition. Nous allons nous en servir pour "réparer" notre code sur les spirales.

### ☐ — Exercice 10 —

a) On conserve un décrétement de 5 pixels à la variable longueur. Expliquez le code ci-dessous :

```
nombreTraits = 0 # initialisation
while nombreTraits < maxTraits:
    if longueur >= 0:
        fred.forward(longueur)
        fred.right(angle)
        longueur = longueur - 5
        nombreTraits = nombreTraits + 1
    else:
        nombreTraits = nombreTraits + 1
```

b) Recopiez puis testez ce code. Votre tracé de spirale fonctionne-t-elle pour tous les décréments compris entre 1 et 10 et des valeurs `maxTraits` entre 10 et 100 ?

.....  
 .....  
 .....

c) Ce code ne respecte pas le principe **DRY**.

Expliquez pourquoi : .....

Codez une solution et testez-la pour vérifier votre résultat.

Rem : Une conditionnelle n'a donc pas toujours besoin d'une alternative **else**.

Avec les conditionnelles, nous avons besoin de connaître **quelques opérateurs mathématiques** pour faire des tests. Les plus classiques en Python sont les suivants :

<b>+, -, *, /</b>	<b>a // b</b>	<b>a % b</b>	<b>a**b</b>	<b>a == b</b>	<b>a != b</b>	<b>&lt;, &lt;=, &gt;, &gt;=,</b>
Addition, soustraction, multiplication, division	Quotient de la division euclidienne de a par b	Reste de la division euclidienne de a par b	a puissance b	a est-il égal à b ?	a est-il différent de b ?	supérieur (ou égal)

Exemple : par exemple, pour savoir si un nombre est non divisible par 7, il nous faut savoir si son reste dans une division par 7 est différent de 0.

En Python, on pourra avoir le code suivant :

```
a = 12
reste = a % 7
if reste != 0:
    print('a non divisible par 7')
else:
    print('a divisible par 7')
```



**— Exercice 11 —**

Nous allons rajouter du volume au tracé en faisant varier la **largeur du tracé** en fonction du nombre de traits que nous avons fait.

a) Créez une variable `largeur` que vous initialiserez à 8 puis avec l'instruction `fred.width(...)` indiquez à Fred que sa largeur initiale vaut `largeur` (voir page 3).

b) Dans votre boucle, si le nombre de traits tracé est divisible par 5, on diminuera la valeur de la variable `largeur` de 25% (diminuer de 25% revient à multiplier par .....). Puis, on indiquera la nouvelle largeur du tracé à Fred.

c) Testez votre programme. La largeur de votre tracé change-t-elle ?

```
nombreCote = 10
longueur = 100
angle = 360/nombreCote
maxTraits = 30
largeur = 8
```

Finalement, nous allons ajouter de la couleur dans la vie de Fred.

Définir une couleur en informatique peut se révéler compliqué : nous allons donc partir du principe qu'une couleur est définie par 3 valeurs : la puissance du rouge (0 à 100%), celle du vert (0 à 100%) et celle du bleu (0 à 100%). C'est le codage **RGB** (red, green, blue).

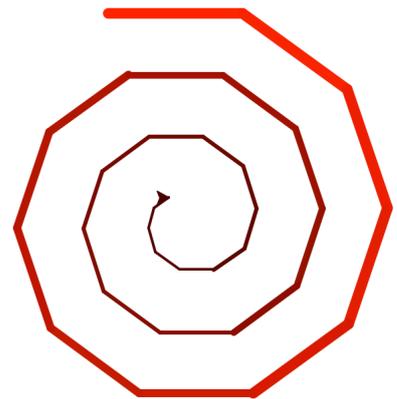
Le rouge a 100% de rouge et 0% de bleu et vert. Donc le rouge se traduit par : (1, 0, 0) .

### — Exercice 12 —

Pour rajouter une couleur rouge à fred, nous allons utiliser l'instruction proposée à la page 3 :

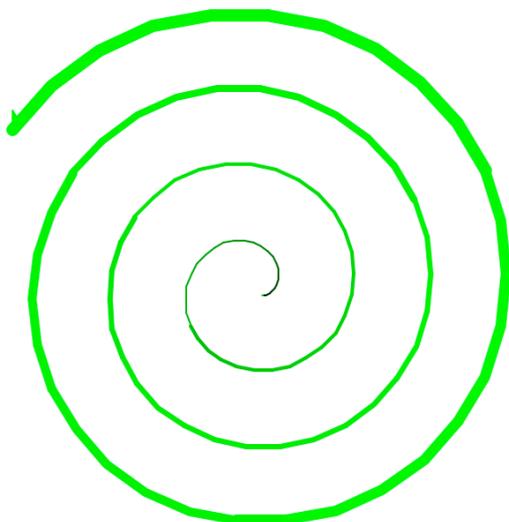
```
fred.color(1, 0, 0)
```

- Faire le tracé de la spirale en rouge, puis en vert, puis en bleu. Vérifier le résultat en testant votre programme.
- Nous souhaitons maintenant que fred change de couleur au fur et à mesure de son avancée. À chaque fois que fred va tracer un trait, ce trait doit devenir plus sombre. Autrement dit, la puissance du rouge de fred doit diminuer de 5% à chaque nouveau trait. Modifiez votre code pour obtenir cela.

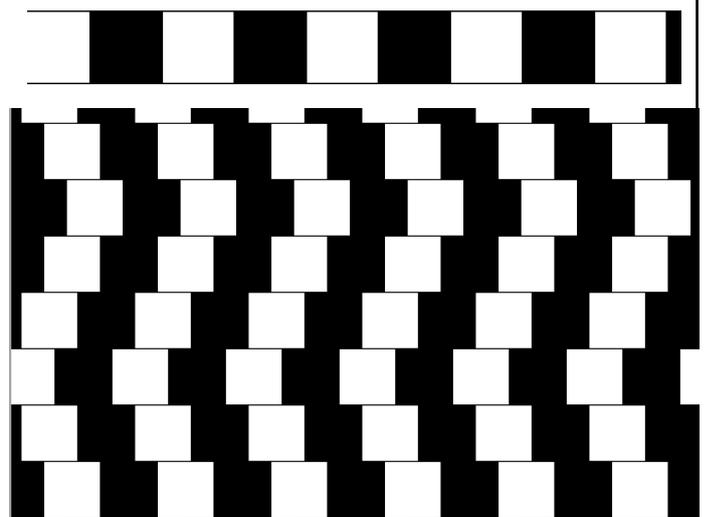


### — Exercice 13 —

Dans cet exercice, vous êtes libre de choisir les structures de contrôle de votre choix dans la limite du principe **DRY**. Nous vous proposons deux images qu'il s'agit de refaire le plus précisément possible.



**Spirale inversée.** Le tracé ne comporte pas d'angles, part du centre et va vers l'extérieur.



**Illusion d'optique.** Faire un carré, puis une ligne de carrés puis le damier. On peut imbriquer une boucle dans une boucle !

### Conclusion :

Une variable a donc deux intérêts majeurs :

- ❖ changer facilement la valeur d'un paramètre (le côté d'un polygone)
- ❖ modifier aisément une quantité en fonction d'une formule

Une conditionnelle permet d'introduire une rupture dans le code et de gérer les cas particuliers. Lorsque vous devez **tester** quelque chose, utilisez une conditionnelle !

Une boucle permet de suivre le principe DRY en évitant les répétitions et les copier/coller. Lorsque vous devez **faire de nombreuses fois** une opération, utilisez une boucle !