# TP2 : Turtle & Langages de programmation

Les fichiers Python de ce TP sont disponibles sur mon site internet : bouillotvincent.github.io .

## I. Résumé de l'épisode précédent (TP1)

10 min 

### 1) Opérateurs mathématiques en Python

Le TP1 nous faisait manipuler les conditionnelles : nous avons besoin de connaitre **quelques opérateurs mathématiques** pour faire des tests.
Les plus classiques en Python sont les suivants :

| +, -, *, / | a // b | a % b | a**b | a == b | a != b | <, <=, >, >=, |
|---|---|---|---|---|---|---|
| Addition, soustraction, mutliplication, 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 divisible par 5, il nous faut savoir si son reste dans une division par 5 est égal de 0.

En Python, on pourra avoir le code ci-contre :

```
a = 12
reste = a % 5
if reste == 0:
    print(a, ' non divisible par 5')
else:
    print(a, ' divisible par 5')
```

### 2) Conditionnelles et boucles

Lorsque nous souhaitons réaliser une spirale constituée de `maxTraits` traits plutôt qu'un polygone de `nombreCote` côtés, nous devons diminuer de plus en plus la valeur de la `longueur` d'un côté. Nous arrivons toutefois à une situation où la `longueur` est négative, ce qui crée un comportement imprévisible.

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

Pour éviter cela, nous choisissons de tracer un trait seulement si la variable `longueur` est positive. Nous incrémentons aussi `nombreTraits` d'une unité. Dans le cas où la variable `longueur` est négative, nous ajoutons tout de même une unité à la variable `nombreTraits` afin d'éviter une boucle non bornée infinie.

**Remarque** : Nous avons ici mélangé la boucle et la conditionnelle ! Il est bien sur possible de tout **imbriquer** : une boucle dans une conditionnelle, une boucle dans une boucle, une conditionnelle dans une boucle dans une boucle etc.

## II. Boucles bornées

**30 min** 

Le programme de la page 1 est toutefois extrêmement lourd car on **sait combien de fois** on doit répéter le bloc B à l'intérieur de la **boucle non bornée Tant que**. On aurait donc du utiliser une **boucle bornée Pour**.

### — Exercice **1** —

Dans le cours, nous avons fait la somme des entiers de 0 à 4-1 grâce au code ci-dessous :

```
p = 0
for i in range(4):
    p = p + i
print(p)
```

- Dans un nouveau programme appelé **bouclePour.py**, recopiez puis testez ce code. Changez `range(4)` en `range(10)` : on doit trouver 45.
- À l'aide de l'instruction `print`, affichez sur la même ligne chaque valeur de `i` et de `p`. Testez à nouveau votre programme et observez les valeurs prises par `i`.
- Modifiez votre programme pour effectuer **la somme des carrés des entiers de 0 à 9**. Vous devez trouver 285.

Remarque : `i` est appelé un **compteur**. On peut éviter de l'utiliser si, dans la boucle, nous n'en avons pas besoin. Dans ce cas, on utilise une variable anonyme avec un **underscore** :

```
for _ in range(4):
```

### — Exercice **2** —

Nous allons à présent reprendre quelques programmes du TP1 en nous allons les réécrire à l'aide d'une boucle bornée Pour.

- Téléchargez le programme **Exercice 2** sur bouillotvincent.github.io .
- En vous aidant des encadrés de la page suivante, remplacez la boucle non bornée `while` par une boucle bornée `for` avec i allant de 0 à 8.
- Modifiez le code précédent afin de prendre en compte n'importe quel nombre de côtés.

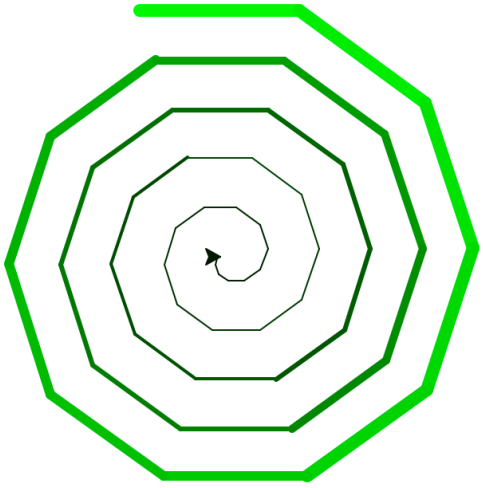
```python
import turtle
nombreCote = 8
longueur = 100
angle = 360/nombreCote

fred = turtle.Turtle()
fred.goto(0,0)
nombreTraits = 0
while nombreTraits < 8:
    fred.forward(longueur)
    fred.right(angle)
    print(nombreTraits)
    nombreTraits = nombreTraits + 1

turtle.exitonclick()
```

❐ — Exercice 3 —

- Téléchargez le programme **Exercice 3** sur bouillotvincent.github.io .
- Que fait la ligne `if nombreTraits % 5 == 0` ?
- De la même manière qu'à l'exercice 2, modifiez ce programme afin de remplacer la boucle `while` par une boucle `for` appropriée.
  Vous devez observer la figure ci-contre :

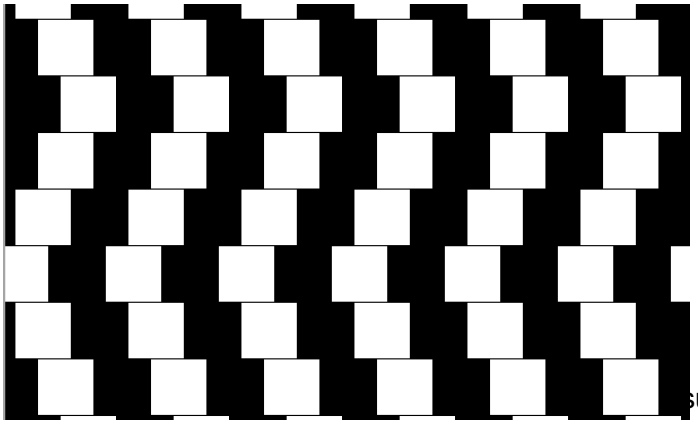


## III.Fonctions

70 min

On souhaite à présent créer l'illusion d'optique ci-contre :

Rien de plus simple ! Vous n'avez qu'utiliser le code téléchargeable **illusion.py** sur bouillotvincent.github.io .



## ❐ — Exercice 4 —

- Téléchargez le programme **illusion.py** sur bouillotvincent.github.io .
- Dans ce programme, combien de boucles **Tant que** sont imbriquées les unes dans les autres ?
- Pensez-vous que ce code soit : ☐ organisé ? ☐ simple à comprendre ? ☐ facile à utiliser ? ☐ facile à modifier ?

Jusqu'à maintenant, nous avons écrit de petits codes. Mais, nos programmes vont rapidement avoir des tailles importantes : des problèmes de compréhension et d'organisation vont apparaitre.
Nous allons donc devoir organiser davantage notre programme en utilisant des **fonctions.**

**Les fonctions sont des "boites noires de code" réutilisables partout au sein d'un programme et cachant des opérations complexes.**

## ❐ — Exercice 5 —

Ces deux codes tracent un angle droit : à gauche, le code du TP1 et à droite le code avec fonction.

- Encadrez et reliez par des flèches les points communs ;
- Surlignez (ou soulignez) les différences.

```
import turtle

fred = turtle.Turtle()

fred.forward(70)
fred.right(90)
fred.forward(70)
fred.right(90)

turtle.exitonclick()
```

```
import turtle                    1
                                 2
def angleDroit():                3
    fred.forward(70)             4
    fred.right(90)               5
    fred.forward(70)             6
    fred.right(90)               7
                                 8
fred = turtle.Turtle()           9
angleDroit()                     10
                                 11
turtle.exitonclick()             12
```

- Expliquez le fonctionnement de la ligne 10. Demandez vous en particulier dans quel ordre le programme est-il lu ?
- À l'aide de la fonction `angleDroit`, comment feriez-vous pour tracer deux angles droits, l'un après l'autre ?

Une fonction est définie par le mot clé `def` suivi du nom de la fonction. Tout ce qui fait partie de la fonction doit être indenté correctement grâce aux tabulations.
Dans notre exemple, `def angleDroit()` nous permet de définir une fonction appelée angleDroit.

❐ — Exercice **6** —

- Dans le programme illusionEnCours.py, complétez le code ci-dessous pour **créer** une fonction appelée **`carre`** et traçant un carré :

```
_____ _________():
    for _ in range(4):
        fred.forward(70)
        fred.left(90)
```

- Appelez cette fonction et observez ce qui est tracé à l'écran.
- Changez la longueur du côté du carré à 100, puis observez le résultat à l'écran.

Les fonctions que nous avons écrites jusqu'à maintenant ne nous permettent pas de contrôler la longueur du côté de notre carré.
Il faut faire une modification manuelle, ce qui est dramatique. Allons-nous faire une fonction carre70 pour les côtés à 70 pixels et une fonction carre100 pour les côtés à 100 pixels ? Ce n'est pas très DRY…

Pour éviter cela, les fonctions peuvent prendre des paramètres (ou arguments) lors de leurs appels.

❐ — Exercice **7** —

Dans la fonction **`carre`** ci-dessous , on ajoute une variable x, qui est un **paramètre (ou argument)** de la fonction. Tout comme une fonction mathématique, x n'a pas de valeur lors de la définition de la fonction.

```
_____ _________(x):
    for _ in range(4):
        fred.forward(x)
        fred.left(90)
```

- Faites la modification dans votre programme.
- Appelez la fonction en donnant une valeur 100 à votre côté. On utilisera l'instruction :

```
carre(100)
```

- En conservant le premier appel à la fonction `carre`, appelez à nouveau la fonction en donnant 50 à votre côté. Observez le résultat.

**Attention ! La variable x n'existe QUE dans la fonction. Tous paramètres ou variables créées au sein d'une fonction disparait une fois la fonction terminée.**

Mais, du coup… comment faire pour récupérer des résultats utiles que la fonction a calculé ? En Python, nous allons utiliser l'instruction **`return`** .

Exemple :

```
def maFonction(x):
    variableInterne = 2*x + 4
    return variableInterne

res = maFonction(8)     # res vaut 2*8 + 4
```

❐ — Exercice **8** —

Pour réaliser l'illusion d'optique, on souhaite colorier un carré sur deux. Nous allons créer une fonction appelée **`choixCouleur`** qui va prendre en paramètre un **`nombre`** et va renvoyer comme résultat la **`couleur`** choisie.

Dans cette fonction, traduire en Python l'algorithme suivant :

si le nombre est divisible par 2, la couleur est 'black' ;

sinon, la couleur est 'white' ;

renvoyez la couleur

Testez votre programme en recopiant en fin de code la ligne ci-dessous, puis en lançant votre programme : `assert(choixCouleur(3)=='white')`

❐ — Exercice **9** —

Nous aurons besoin par la suite d'une fonction appelée **`decalage`** qui prend en paramètre un entier **`indice`** et va renvoyer comme résultat :

- 20 si le reste de la division de **`indice`** par 6 vaut 0, 1 ou 2 ;
- -20 sinon.

Dans le même programme, créez cette fonction.

En Python, toute structure est imbriquable dans une autre structure. Nous pouvons donc utiliser une fonction dans une autre fonction, ce qui est très utile pour organiser vos programmes !

## ❐ — Exercice 10 —

On propose une fonction à trous **`carreColore`** qui colore un carré de taille **`x`** en fonction de la valeur d'un **`nombre`** et ne renvoie rien :

```
def carreColore(x, nombre):
    couleur = choixCouleur(nombre)   # ...................
    fred.begin_fill()   # commence le coloriage
    fred.fillcolor(.........)    # remplit de la couleur choisie
    ..............     # trace un carré de côté x
    fred.end_fill()     # fin du coloriage
```

- Que fait la ligne 2 ?
- Compléter les lignes 4 et 5.
- Testez votre programme en appelant `carreColore(80, 1)`. Un carré blanc doit apparaitre. Vérifiez que `carreColore(80, 2)` fait apparaitre un carré noir.

## ❐ — Exercice 11 —

Nous allons créer une fonction **`ligneDeCarre`** permettant de tracer une ligne de carrés. Cette fonction prendra en paramètre la longueur **`x`** d'un côté ainsi que le nombre de carrés **`n`** par ligne. Cette fonction ne renvoie rien.

a) À l'aide d'une boucle, imaginez un algorithme permettant de tracer plusieurs carrés colorés les uns à la suite des autres. Programmez et testez votre programme.

L'inconvénient majeur de notre fonction est que les carrés n'occupent pas tout l'écran. Nous devons donc contrôler leurs positions en ajoutant deux nouveaux paramètres gérant l'abscisse de départ (**`posx`**) et l'ordonnée (**`posy`**) de la ligne de carrés :

```
def ligneDeCarre(posx, posy, x, n):
    fred.pu()
    fred.goto(posx, posy)
    fred.pd()
    .........   # Code du a)
    .........   # Code du a)
    .........   # Code du a)
```

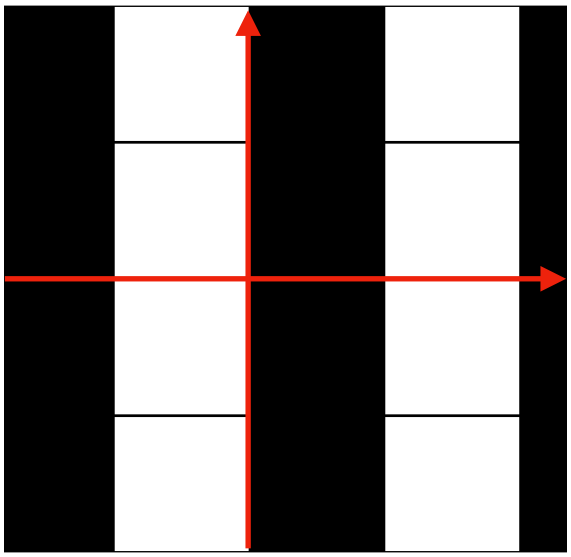
b) Testez votre programme en appelant votre fonction avec les variables globales définies au début du code : `ligneDeCarre(xBax, yHaut, cote, nCarre)`

Avec cette fonction, nous avons la base pour tracer un damier de carrés colorés. En effet, un damier est juste une **succession** de lignes colorées.

## — Exercice **12** —

Nous allons créer une fonction **`damier`** permettant de tracer une succession de ligne de carrés colorés. Cette fonction prendra en paramètre l'abscisse de départ (**`posx`**) et l'ordonnée (**`posy`**) du damier, le nombre de lignes **`n`** et le côté d'un carré **`x`**. Cette fonction ne renvoie rien.

a) Dans un premier temps, nous souhaitons afficher le dessin sur la droite sans créer la fonction **`damier`**. Pour comprendre les coordonnées, l'origine du repère est indiquée par une flèche. Pour un **`cote`** de 100, complétez les lignes ci-dessous permettant d'obtenir ce "damier".



```
ligneDeCarre(-400, 200 ,cote,nCarre)
ligneDeCarre(-..., ... ,cote,nCarre)
ligneDeCarre(-..., ... ,cote,nCarre)
ligneDeCarre(-..., ... ,cote,nCarre)
```

b) En trouvant une relation mathématique entre les ordonnées des lignes, simplifiez votre code en utilisant une boucle. Vérifiez que votre programme fonctionne en le testant et en changeant la valeur de la variable **`cote`**.

c) Modifiez votre programme afin de l'intégrer dans la fonction **`damier`**. Pour les tests, on appellera :

```
damier(xBas, yHaut, nLigne, cote)
```

d) Finalement, en utilisant astucieusement la fonction **`decalage`** de l'exercice 9 et en modifiant la valeur de **`xpos`**, créez les décalages d'abscisses permettant d'obtenir :