Travaux Pratiques de Programmation avec Python

Guillaume Le Blanc Jean-Pierre Vallon

4 octobre 2020

Résumé

Ce cahier est destiné aux élèves de Seconde.

1. Nous avons choisi de faire comprendre les principales constructions d'un langage de programmation (séquence, variable, affectation, test, boucle,fonction) par le biais de l'utilisation de la Tortue car l'élève peut se corriger suivant le principe action-réaction en voyant les effets de son programme.
2. Une séance de 55 minutes est découpée en 2. Durant la première partie l'élève découvre une notion (20 minutes) puis dans un second temps il s'entraîne sur des exercices.

   Il y a toujours deux exercices, le premier est faisable par tous les élèves en classe, le deuxième parfois plus difficile, dans ce cas repéré par une étoile, est à travailler jusqu'à la séance suivante.

   Il est important que dans cette deuxième partie chaque élève puisse expérimenter seul faire des erreurs et se corriger.
3. Il existe plusieurs environnements de Python. Nous avons choisi l'un des plus simples à installer et à utiliser quel que soit le système d'exploitation, IDLE. D'ailleurs il est conseillé dès la première séance de donner un travail à faire à la maison : installer IDLE à la maison, faire tracer un triangle équilatéral par la Tortue puis faire une capture d'écran et l'envoyer par mail au professeur (via monlycee.net)
4. Il est conseillé de commencer les premiers TP dès le début d'année à raison d'un TP tous les quinze jours.
5. Puisque les professeurs de mathématiques n'ont pas la même progression au cours de l'année, néanmoins via l'index du cahier d'exercices ils peuvent choisir pour chaque chapitre du cours de mathématiques au moins un exercice de programmation et d'algorithmique.
6. Nous avons voulu aussi fournir aux élèves des "bonnes" **pratiques**, par exemple éviter les nombres magiques, factoriser le code etc...
7. **La première séance est importante**, l'élève doit se familiariser avec l'environnement de travail constitué du répertoire de sa classe dans lequel se trouve le cahier de TP(lecture) l'environnement IDLE (dans lequel il code) son répertoire personnel (dans lequel il enregistre ses programmes soigneusement)
8. L'élève doit garder une trace (calculs, dessins, programmes) du TP dans un cahier ou sous forme numérique.
9. Il est important aussi de préciser au cours de cette première séance **les règles de comportement en TP** :
   (a) On garde l'environnement propre : pas de déchets .
   (b) On ferme sa session après la séance, et on range sa chaise.
   (c) **En cas de panne** on signale le problème au professeur et on n'essaie pas de réparer seul.

Table des matières

Chapitre 1

Environnement de travail

1.1 Utilisation à la maison

L'apprentissage de la programmation nécessite une pratique régulière donc on recommande dès la première séance, l'installation à la maison de Python au moins avec l'environnement IDLE. Un des exercices de cette première séance est d'installer Python sur un ordinateur à la maison. Télécharger la dernière version ici : `https://www.python.org`

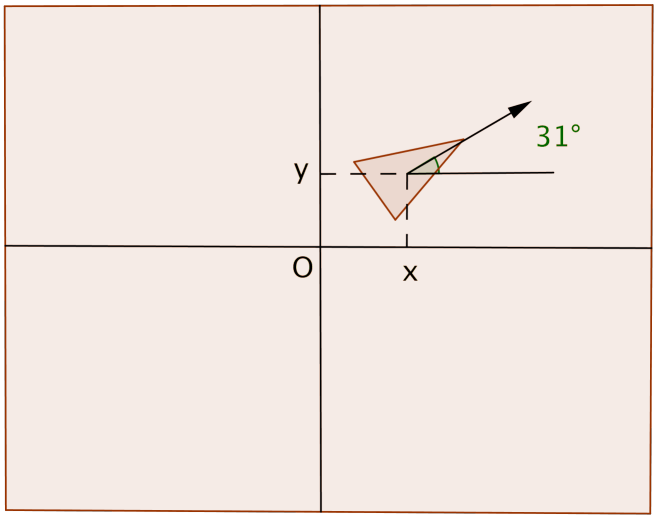
1.2 Utilisation au lycée

Sur le bureau il y a un raccourci IDLE-7 ou IDLE-XP suivant la version de windows

Chapitre 2

Séquence d'instructions

2.1 Découverte de la Tortue



La tortue se déplace en laissant une trace derrière elle comme un escargot. On peut lui demander d'avancer, ou de tourner, de changer la couleur de la trace etc...

Elle apparaît à l'écran sous la forme d'un triangle (voir dessin ci-dessus on a exagéré les dimensions de la tortue). Elle est repérée par des coordonnées $(x; y)$ mais aussi par un angle en degrés entre la direction vers laquelle elle "regarde" et l'axe horizontal

Les commandes usuelles de la tortue ont la même apparence que les **fonctions** en maths. On les écrit toujours avec des parenthèses sous la forme suivante :

nom de la fonction (Liste de paramètres)

La liste des paramètres peut être vide
Quelques exemples :

1. **goto**(x,y) : la tortue va au point de coordonnées (x,y) ;
2. **forward**(d) : la tortue avance dans sa direction de d pixels ;
3. **left**(angle) : la tortue tourne sur la gauche d'un angle donné en degrés ;
   Si l'angle est négatif la tortue tourne à droite

4. **penup**() :relever le crayon pour pouvoir avancer sans dessiner ;
5. **pendown**() :abaisser le crayon pour dessiner ;

2.2 Premières figures géométriques

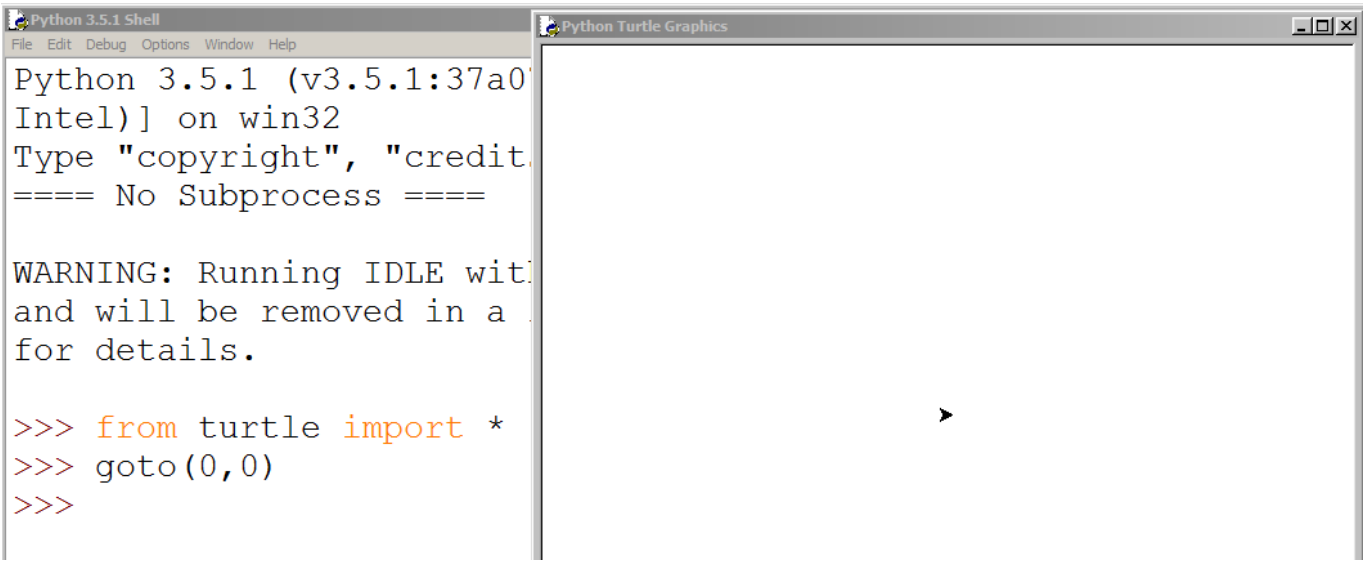
Pour utiliser la tortue il faut charger la bibliothèque python qui gère la tortue, ainsi tout programme python utilisant la tortue doit contenir au préalable la ligne

`from turtle import *`

Dans l'interpréteur python après le signe >>> entrer la commande `from turtle import *`

puis `goto(0,0)` pour faire apparaître la tortue en (0,0), vous allez observer ceci :

Voici sur la fenêtre de gauche **l'interpréteur python**, et à droite **la fenêtre graphique** où on voit la tortue dans son état initial, en (0,0) regardant vers l'ouest

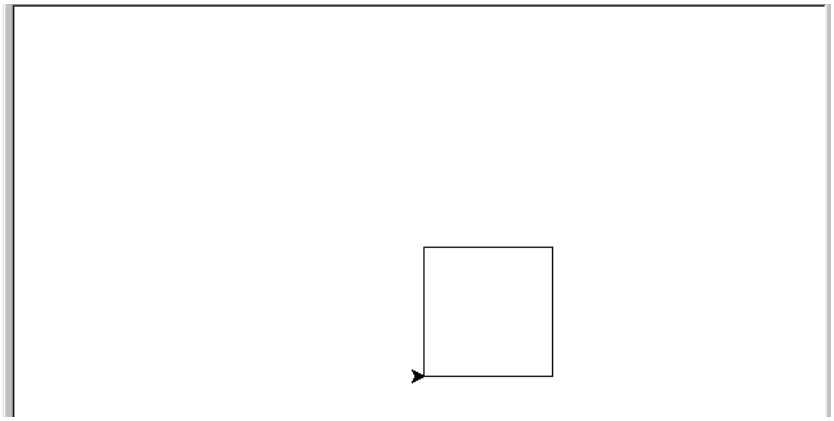


Puis entrer la commande : `forward(100)` pour faire avancer la tortue de 100 pixels dans la direction où elle regarde :

Comment construire un carré ?

On peut faire tourner la tortue sur la gauche d'un angle de 90 degrés par la commande `left(90)` puis recommencer dans l'ordre la séquence `forward(100)`, `left(90)`

```
>>> from turtle import *
>>> forward(100)
>>> left(90)
>>> forward(100)
>>> left(90)
>>> forward(100)
>>> left(90)
>>> forward(100)
>>> left(90)
>>>
```



Cependant **l'interpréteur ne permet pas de sauvegarder le travail**. Si vous voulez montrer le dessin à quelqu'un le lendemain il faudra retaper les commandes dans l'interpréteur.

Heureusement on peut **enregistrer** la séquence d'instructions dans un **fichier** en utilisant **l'éditeur de texte** :

Pour pouvoir éditer du texte, cliquer ,dans la barre des menus, sur **File** puis sur **New File**, une fenêtre de l'éditeur de texte apparaît.

Dans l'éditeur de texte copier-coller (CTRL-C puis CTRL-V) la séquence `forward(100)` et `left(90)` pour écrire 4 fois `forward(100)` et `left(90)` .

Pour régler la vitesse de la tortue au minimum on a rajouté l'instruction `speed(1)`

Dans l'éditeur de texte vous devez observer ceci :

carre.py - K:/Docs/TP Python/carre.py (3.5.1)

File Edit Format Run Options Window Help

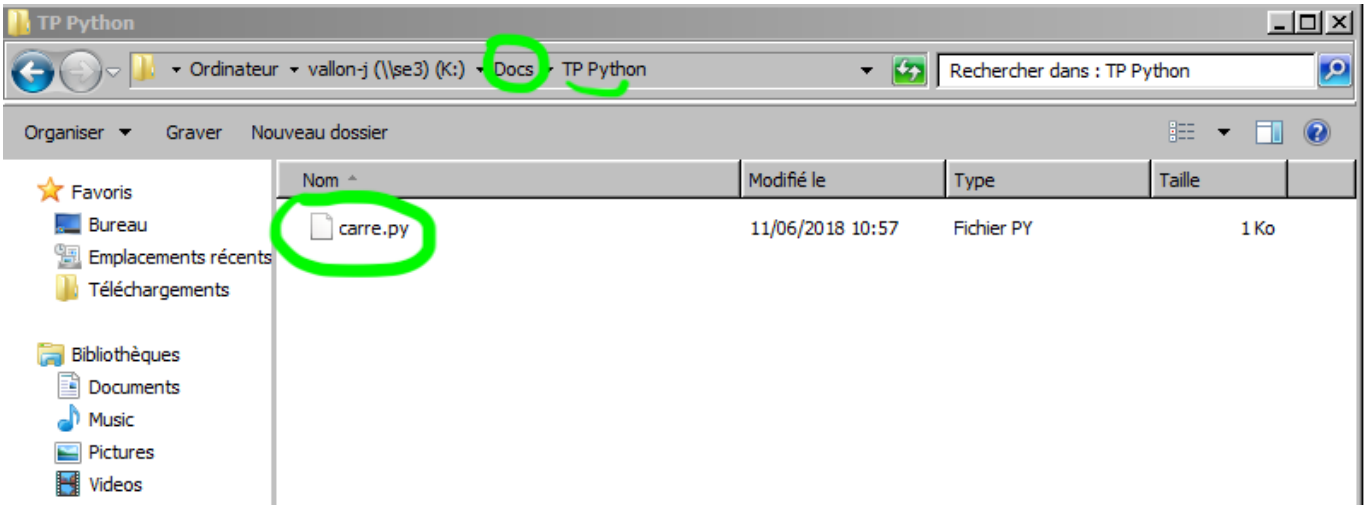
```
from  turtle import *
speed(1)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
forward(100)
left(90)
```

Pour pouvoir **exécuter** ce programme il faut d'abord le **sauvegarder**.

Avant d'enregistrer le programme on va créer un nouveau dossier nommé TP Python dans le répertoire Docs se trouvant dans votre répertoire personnel.

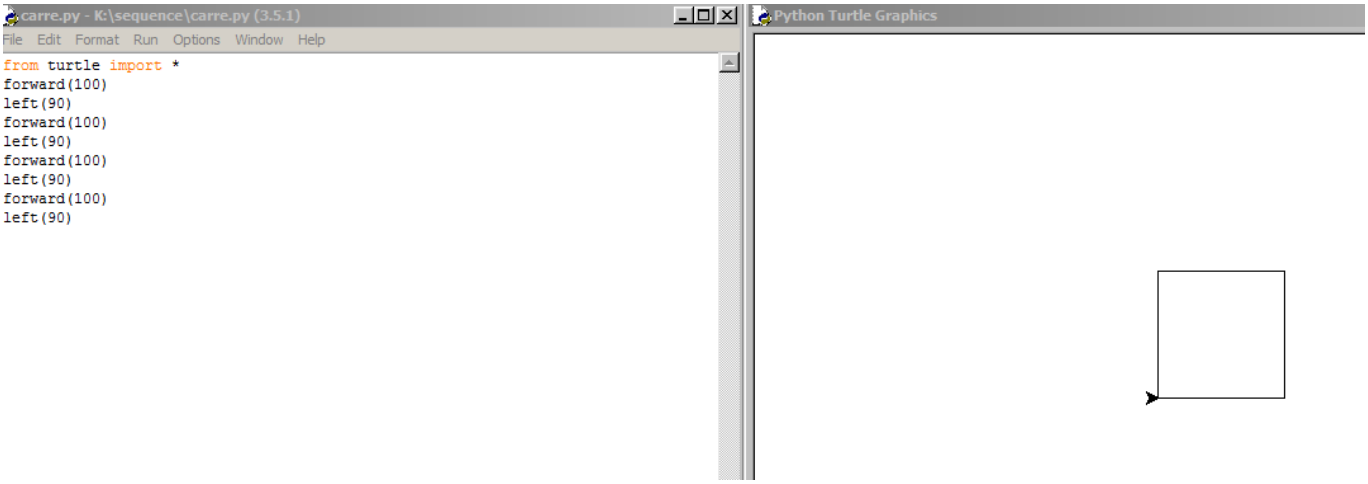
Ensuite on enregistre le programme **dans un fichier avec une extension .py** avec **un nom significatif**

par exemple carre.py, dans le répertoire TP Python.



Une fois enregistré on peut exécuter le programme en tapant F5.

A l'écran vous devez observer ceci :



Si l'interpréteur ne reconnaît pas une instruction il envoie un message d'erreur en rouge, par exemple :

```
forward(100)
NameError: name 'forward' is not defined
```

Dans ce cas précis on avait oublié d'importer la bibliothèque turtle et d'écrire `from turtle import *` au début du programme

Remarque pratique : A partir de maintenant vous devez enregistrer tous les exercices que vous faites sous des noms différents, par exemple, carre.py, hexagone.py, etc.. et ranger tous les fichiers d'un même TP dans le dossier TP Python.

2.2.1 Commentaires

1. A partir de maintenant un programme Python ou une partie d' un programme Python sera écrit sur un fond jaune, par exemple

```
from turtle import *

speed(1)

forward(100)
left(90)

forward(100)
left(90)

forward(100)
left(90)

forward(100)
left(90)
```

2. Chaque ligne du programme correspond à une action de la Tortue, **les actions ou instructions sont exécutées en séquence dans l'ordre**
3. La dernière ligne semble inutile mais elle a deux intérêts : elle remet la tortue dans son état initial, et on voit mieux la répétition
4. On peut régler la vitesse de déplacement de la tortue et la régler au minimum en ajoutant l'instruction `speed(1)`. Prendre le temps de bien comprendre le déplacement de la tortue en lien avec le code, quitte à exécuter plusieurs fois le programme.
5. Peut on changer l'ordre des instructions ?

   Devinez ce que fait ce programme **avant de l'exécuter**

```
from turtle import *

speed(1)

left(90)
forward(100)

left(90)
forward(100)

left(90)
forward(100)

left(90)
forward(100)
```

6. Bien faire la différence entre **l'interpréteur** où il y a >>> et **l'éditeur de texte**

2.3 Exercices

1. **A la maison** :
   (a) Installer Python sur un ordinateur à la maison.
   (b) Ecrire un programme qui trace un triangle équilatéral
   (c) Envoyer par mail via l'E.N.T à votre professeur le programme triangle.py et une copie d'écran du triangle équilatéral sous la forme d'une image PNG
2. Ecrire un programme python qui trace :
   (a) un triangle équilatéral
   (b) un pentagone régulier
   (c) un hexagone régulier
   (d) un heptagone régulier
   (e) un dodécagone régulier

   (Bien réfléchir : De quel angle en degrés dois-je tourner sur la gauche, plusieurs fois pour faire un tour complet sur moi-même, c'est à dire un tour de 360 degrés)

Chapitre 3

Répéter n fois

3.1 Factoriser le code

Nous avons vu qu'il devient fastidieux de réécrire plusieurs fois `forward(100)` et `left(60)` pour écrire le programme de construction d'un hexagone régulier. Heureusement il est possible de n'écrire qu'une fois les instructions que l'on doit répéter et de dire combien de fois on veut que ces instructions soient répétées.

On peut simplifier le programme de la construction d'un carré ainsi :

```
from turtle import *
speed(1)
for i in range(4):
    forward(100)
    left(90)
```

3.1.1 Commentaires

1. **range** signifie intervalle en anglais
2. **range(n)** énumère les entiers de 0 à $n-1$
3. **Les instructions décalées après le : seront répétées n fois** . Ce décalage s'appelle **indentation**, et toutes les instructions répétées doivent avoir la même indentation
4. En français l'idée du programme est :
   répéter 4 fois les instructions suivantes : avancer de 100 pixels puis tourner à gauche de 90 degrés
   On préfère écrire cette idée dans une forme plus proche d'un programme appelée algorithme (ou pseudo-code)

Algorithme 1 : Construction d'un carré de 100 pixels

```
début
    répéter 4 fois
        avancer(100)
        tournerAGauche(90)
fin
```

5. **A partir de maintenant** vous venez en salle informatique avec votre cahier de maths de telle sorte que pour chaque problème "un peu difficile" qui nécessite de la réflexion, vous puissez dans l'ordre, écrire vos idées en français puis le pseudo-code puis le code en Python.

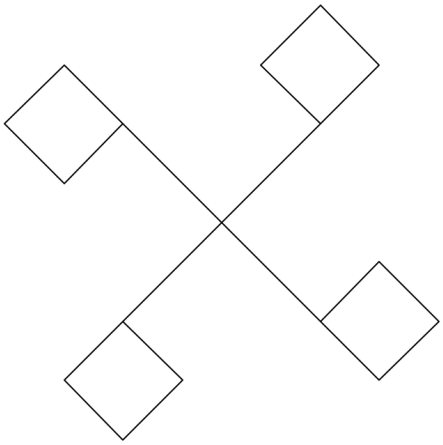
3.2 Exercices

On reprend les exercices de la fois précédente mais cette fois-ci avec la répétition

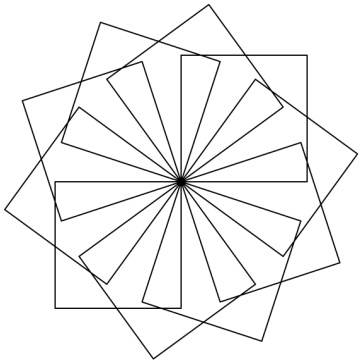
Ecrire un algorithme puis programme python qui trace :

1. un triangle équilatéral
2. un pentagone régulier
3. un hexagone régulier
4. un heptagone régulier
5. un dodécagone régulier

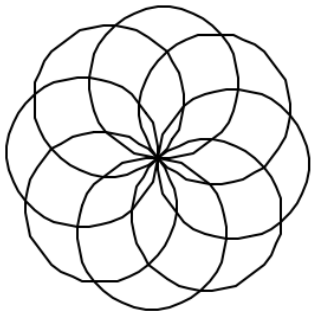
Faire dessiner un "moulin" par la tortue :



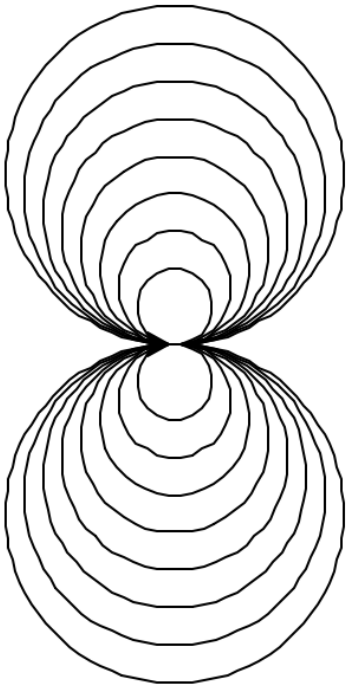
Faire dessiner une "rosace" par la tortue (faire "tourner" un carré sur son coin inférieur gauche) :



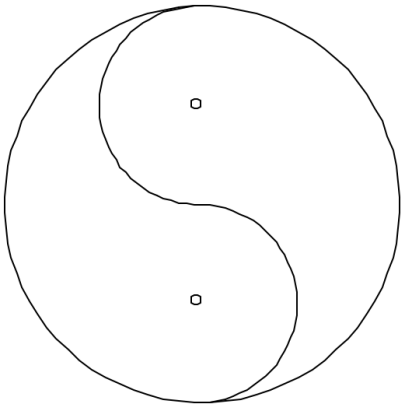
Faire dessiner une vraie "rosace" en faisant tourner un cercle



Faire dessiner cette figure



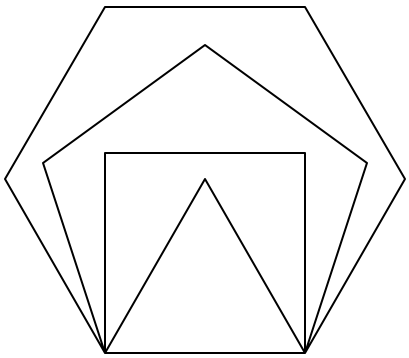
Faire dessiner cette figure



Chapitre 4

Fonction

4.1 Continuons de factoriser en créant nos propres fonctions



Comment tracer à la tortue le dessin ci-dessus ?

En français : Dans le désordre on a tracé un carré, un triangle équilatéral, un hexagone, un pentagone.

Une façon de faire est :

```
for i in range(4):
    forward(100)
    left(360/4)
for i in range(3):
    forward(100)
    left(360/3)
for i in range(6):
   forward(100)
   left(360/6)
for i in range(5):
   forward(100)
   left(360/5)
```

Le programme serait **plus compréhensible** si on pouvait nous aussi **créer des fonctions** pour la tortue et avoir :

```
from turtle import *
dessineCarre()
dessineTriangleEquilateral()
dessineHexagoneRegulier()
dessinePentagoneRegulier()
```

Mieux encore

```
from turtle import *
dessinePolygone(4)
dessinePolygone(3)
dessinePolygone(6)
dessinePolygone(5)
```

Dans un premier temps on va **définir une fonction** appelée `dessinePolygone(nbCotes)` avec comme **paramètre** le nombre de côtés, puis

exécuter cette fonction 4 fois avec des valeurs du paramètre différentes 4 puis 3 puis 6 puis 5 :

```
from turtle import *

def dessinePolygone(nbCotes):
    for i in range(nbCotes):
        forward(100)
        left(360/nbCotes)

dessinePolygone(4)
dessinePolygone(3)
dessinePolygone(6)
dessinePolygone(5)
```

Si on ordonne les appels de la fonction `dessinePolygone(nbCotes)` suivant le nombre de côtés de 3 jusqu'à 6, on fait apparaître une répétition

```
from turtle import *

def dessinePolygone(nbCotes):
    for i in range(nbCotes):
        forward(100)
        left(360/nbCotes)

dessinePolygone(3)
dessinePolygone(4)
dessinePolygone(5)
dessinePolygone(6)
```

On peut donc **factoriser** encore :

```
from turtle import *

def dessinePolygone(nbCotes):
    for i in range(nbCotes):
        forward(100)
        left(360/nbCotes)

for i in range(3,7):
    dessinePolygone(i)
```

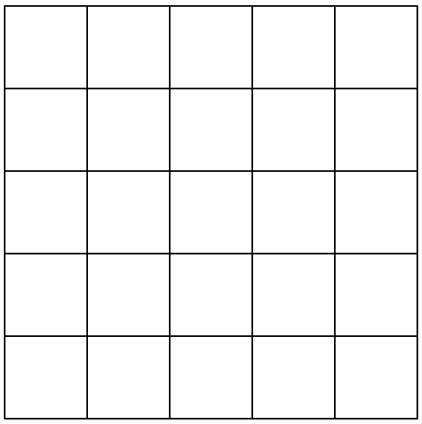
4.1.1 Commentaires

1. Pour **définir** une fonction on utilise le mot réservé `def` qui signifie define en anglais. Ensuite on a donné un nom significatif à la fonction, puis entre parenthèses on écrit les paramètres de la fonction
2. le : marque le début du corps de la fonction et l'indentation comme dans une répétition permet de définir le corps de la fonction.
3. Pour **exécuter** la fonction il faut l'appeler plus loin dans le programme en écrivant par exemple ici `dessinePolygone(3)`. On a donné la valeur particulière 3 au paramètre nbCotes
4. `range(3,7)` permet de parcourir les entiers {3,4,5,6} que l'on note sous la forme d'un intervalle d'entiers $[\![3, 7[\![$
5. En pseudo-code on écrit :

début
  dessinePolygone *(nbCotes)*
    avancer(100)
    tournerAGauche(360/nbCotes)
  dessinePolygone(3)
  dessinePolygone(4)
  dessinePolygone(5)
  dessinePolygone(6)
fin

4.2 Exercices

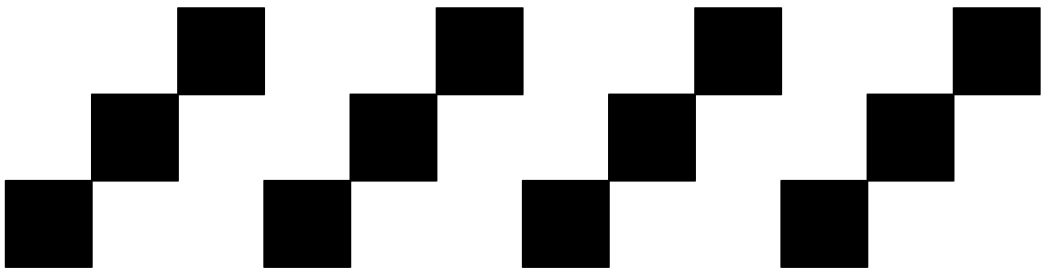
1. Donner un algorithme comportant une fonction `dessineLigne(nbCarres)` qui dessine la grille ci-dessous.
   Puis traduire cet algorithme en Python



Algorithme 2 : Construction d'une grille de 5 par 5

```
début
    dessineLigne (nbCarres)
        .....
    répéter 5 fois
        dessineLigne(5)
        .....
fin
```

2. Ecrire un algorithme puis un programme Python avec des fonctions pour construire le dessin suivant :



Chapitre 5

Variable

5.1 Notion de variable

```
from turtle import *

def dessinePolygone(nbCotes):
    for i in range(nbCotes):
        forward(100)
        left(360/nbCotes)

dessinePolygone(3)
dessinePolygone(4)
dessinePolygone(5)
dessinePolygone(6)
```

La fonction dessinePolygone() est appelée au début avec le paramètre 3 puis par le paramètre 4 puis 5 puis 6.

On aimerait factoriser ces 4 lignes mais autrement que lors du TP précédent . Comment ?

On introduit une **variable** avec un nom significatif, `nbCotes` que l'on initialise avec la valeur 3, par l'instruction d'affectation (pour donner une image, `nbCotes` désigne une case dans la mémoire de l'ordinateur où l'on a mis le nombre 3)

```
nbCotes = 3
```

Par quoi compléter le programme pour que le contenu de la variable `nbCotes` devient 4, puis 5 et finalement 6 ?

```
from turtle import *

def dessinePolygone(nbCotes):
    for i in range(nbCotes):
        forward(100)
        left(360/nbCotes)
nbCotes = 3
dessinePolygone(nbCotes)
```

```
..............
dessinePolygone(nbCotes)
..............
dessinePolygone(nbCotes)
..............
dessinePolygone(nbCotes)
```

Finir de factoriser le programme :

```
from turtle import *

def dessinePolygone(nbCotes):
    for i in range(nbCotes):
        forward(100)
        left(360/nbCotes)
nbCotes = 3
for i in range(........):
    dessinePolygone(nbCotes)
    ..............
```

5.2 Constantes

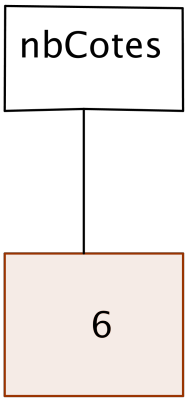
Dans un programme certaines grandeurs sont constantes ou varient peu dans ce cas on évite de les laisser sous forme numérique (**éviter les nombres magiques**) et on préfère introduire des constantes en majuscules au début du programme. Ainsi au lieu de laisser 360 dans le programme précédent on préfère introduire la variable TOUR_COMPLET , et au lieu de laisser 100 on préfère introduire un paramètre supplémentaire à la fonction `dessinePolygone(nbCotes,cote)`.

```
from turtle import *
TOUR_COMPLET = 360

def dessinePolygone(nbCotes,cote):
    for i in range(nbCotes):
        forward(cote)
        left(TOUR_COMPLET/nbCotes)
nbCotes = 3
for i in range(........):
    dessinePolygone(nbCotes,100)
    ..............
```

5.3 Qu'est ce qu'une variable ?

Une variable est un **identificateur** (une étiquette) par exemple **nbCotes** associée à une **référence** (représentée par la boîte ci-dessous dans un souci de simplification)



contenant une **valeur** (par exemple 6). Cette valeur peut changer au cours de l'exécution du programme.

On peut comparer une variable à un nom , la référence à une boîte aux lettres et la valeur au contenu de la boîte aux lettres, **attention ! plusieurs noms peuvent être écrits sur la même boîte aux lettres**

Pour **faire évoluer la référence associée à une variable** on dispose d'une opération importante appelée **l'affectation** :

L'**affectation** correspond à une instruction de la forme :

```
variable = expression
```

une expression est formée à partir des nombres, des variables et des opérations comme $+$, $\times$, etc...

par exemple :

```
nbCotes = 4
nbCotes = 2*nbCotes
```

Dans la référence associée à la variable `nbCotes` il y a eu dans un premier temps la valeur 4 puis cette valeur a été multipliée par 2 puis le résultat, c'est à dire 8 a été remis dans la référence.

En pseudo-code le symbole d'affectation est $\leftarrow$, on voit mieux ainsi dans quel sens lire l'instruction de la droite vers la gauche et le programme précédent en pseudo-code est :

---

début
  nbCotes $\leftarrow$ 4
  nbCotes $\leftarrow$ 2*nbCotes
fin

Allez sur ce site `http://pythontutor.com/visualize.html#mode=display` pour **visualiser** le programme

```
nbCotes = 4
nbCotes = 2*nbCotes
```

5.4 Entrées et Sorties

Une affectation peut prendre en compte une entrée au clavier par l'utilisateur du programme.

Ainsi dans le programme suivant on laisse à l'utilisateur le soin de préciser le nombre de côtés du polygone régulier et la longueur d'un côté. Comment ?

La fonction `input(".....")` affiche à l'écran ce qu'il y a entre guillemets par exemple ici la question Combien de côtés ?

Imaginons que l'utilisateur tape au clavier 4, pour 4 côtés. Pour l'utilisateur 4 est un nombre entier, **mais pour la machine ce qui est tapé au clavier est codé sous la forme d'une chaîne de caractères**, voilà pourquoi il faut convertir cette chaîne de caractères en nombre entier en utilisant la fonction `int()`

(int pour integer qui signifie nombre entier en anglais)

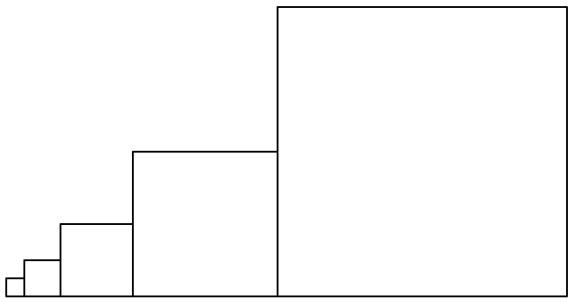
Au final on a l'affectation `nbCotes = int(input("Combien de côtés ?"))`

```
from turtle import *
TOUR_COMPLET = 360
nbCotes = int(input("Combien de côtés ?"))
cote = int(input("Quelle est la longueur d'un  côté?"))
for i in range(nbCotes):
    forward(cote)
    left(TOUR_COMPLET/nbCotes)
```

Si on veut afficher à l'écran le contenu d'une variable ou la valeur d'une expression on utilise la fonction `print()`

5.5 Exercices

1. Dessiner une ligne de 5 carrés de côté variable. Le côté du premier est de 10 pixels, puis le côté double à chaque fois.



2. Dans le programme suivant on répète 4 fois deux instructions et on ne sert pas de la variable `i`

```
from turtle import *
for i in range(4):
    forward(100)
    left(90)
```

dans ce cas l'algorithme correspondant est (répétition) :

début
  répéter 4 fois
    avancer(100)
    tournerAGauche(90)
fin

`range(4)` permet de parcourir les entiers {0,1,2,3} que l'on note sous la forme d'un intervalle d'entiers $[\![0, 4[\![$ qui contient 4 éléments. On aurait pu utiliser `range(14,18)` car ce qui est important ici c'est le nombre d'éléments de l'ensemble sous-jacent et non pas les éléments en eux-même

Cependant on peut utiliser la variable dans le corps de la boucle par exemple si on veut afficher les carrés des entiers de 1 à 10

```
for i in range(1,11):
    print(i**2)
```

Dans ce cas l'algorithme correspondant est (en pseudo-code pour simplifier on inclus la plus grande valeur de l'intervalle, ici 10) :

début
  pour *i de 1 à 10* **faire**
    afficher(i**2)
  fin
fin

Que calcule l'algorithme suivant ?

début
  somme ← 0
  pour *entier de 1 à 4* **faire**
    somme ← somme + entier
  fin
fin

Aide : Compléter le tableau suivant (1 colonne par instruction)

somme	0	0	1	..	..	..	..	..	..
entier	/	1	1	2	2	3	3	4	4

Chapitre 6

Fonction et Variable

6.1 Que "retourne" la fonction ?

Une fonction affine par exemple définie par $x \to 2x + 5$ associe à tout x **réel** son image $2x + 5$ qui est aussi un **réel**. Si en **entrée** on remplace x par 1 en **sortie** on obtient $2 \times 1 + 5 = 7$.

On peut ainsi **composer** les calculs, et ainsi on peut calculer $2 \times f(2)^2 + 1$

Voici une façon de faire en Python qui nous assure que l' on pourra par la suite faire entrer $f(x)$ dans d'autres calculs avec des réels.

```
>>> def f(x):
        return 2*x + 5

>>> f(1)
7
>>> 2*f(2)**2 + 1
163
>>>
```

Le mot réservé `return` en Python signifie retourner.

Par contre on pourrait croire que cette façon de faire aussi est correcte

```
>>> def g(x):
        print(2*x + 5)


>>> g(1)
7
>>>
```

Mais si on veut calculer $2 \times g(2)^2 + 1$ on obtient un **message d'erreur**

```
>>> type(2*g(2)**2 + 1)
9
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    type(2*g(2)**2 + 1)
TypeError: unsupported operand type(s) for ** or pow(): 'NoneType' and 'int'
```

Ce message nous dit que **nous avons voulu faire un calcul avec des objets de type différents**

$g(2)$ est de type `None` ce qui signifie que la fonction g ne retourne rien

```
>>> type(g(2))
9
<class 'NoneType'>
>>>
```

$f(2)$ est de type `int` pour integer (prononcer intejer) pour entier naturel, et $f(2.5) = 10.0$ est de type `float`, pour nombre à virgule flottante (une façon de coder les nombres décimaux en machine) en effet dans la définition de la fonction f il y a un `return` d'un **réel** (entier ou décimal) en fonction de type du paramètre x

```
>>> type(f(2))
<class 'int'>
```

```
>>> f(2.5)
10.0
>>> type(f(2.5))
<class 'float'>
```

Pour la suite du TP on aura besoin d'une fonction `milieu(xA,yA,xB,yB)` dont les paramètres sont les coordonnées du point A et du point B relativement à un repère, et qui **retourne quelque chose du même type**, les coordonnées du milieu de A et de B dont le type pour Python est un **tuple** (pour simplifier un couple de nombres)

Une façon de faire est :

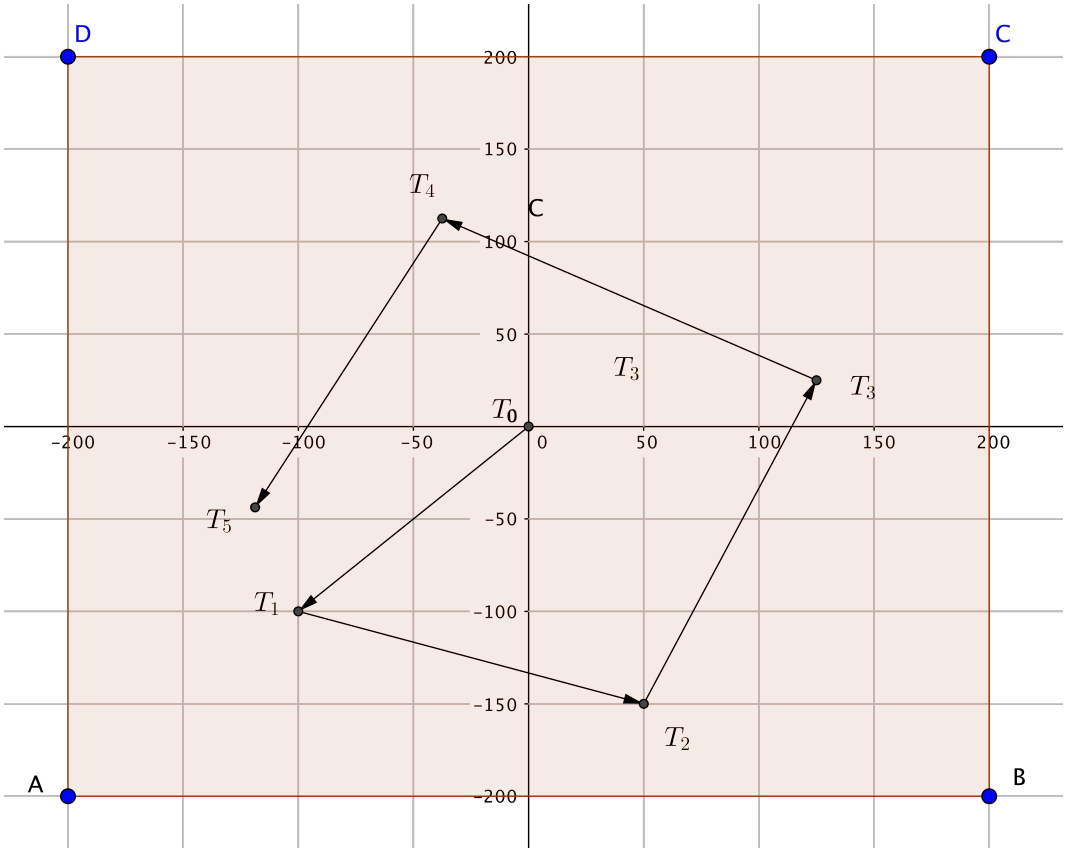
```
>>> def milieu(xA,yA,xB,yB):
        return (xA + xB)/2 , (yA + yB)/2

>>> milieu(1,2,3,4)
(2.0, 3.0)
>>> type(milieu(1,2,3,4))
<class 'tuple'>
```

Par la suite si `xT` et `yT` sont les variables de type `float` pour mémoriser l'abscisse et l'ordonnée de la Tortue on pourra faire une affectation **propre à Python**

```
xT,yT = milieu(xT,yT,xA,yA)
```

6.2 La ronde des milieux



Quatre personnes A,B,C et D sont disposées en carré. Une cinquième personne (vous) se déplace de la manière suivante :

Vous partez de la position T_0 en $(0;0)$, en vous dirigeant ensuite vers A et vous vous arrêtez en T_1, à mi-chemin entre votre position de départ et la position de A.

Ensuite vous faîtes de même de la position T_1, cette fois ci en regardant vers B

Que se passe-t-il si vous continuez ce processus suffisamment "longtemps" ?

Nous allons utiliser la tortue de Python pour **simuler** ce processus

6.3 Exercices

1. Ouvrir le fichier `rondeMilieux.py` contenant le programme suivant et le compléter pour le mettre au point

```
from turtle import *

LENT   = 1
RAPIDE = 0
NB_REPETITIONS = 10

speed(LENT)
#----------------------------------------
def milieu(x1,y1,x2,y2):
    return (x1 + x2)/2 , (y1 + y2)/2
#----------------------------------------
# coordonnées de A
xA = -200
yA = -200
# coordonnées de B
xB = 200
yB = -200
# coordonnées de C
xC = 200
yC = 200
# coordonnées de D
xD = -200
yD = 200

xT = 0
yT = 0

#------DEBUT DE LA RONDE DES MILIEUX----------------------

for i in range(NB_REPETITIONS):
    #la tortue part de sa position (xT,yT)
    #et va au milieu de sa position
    #et de celle de A
    xT,yT = milieu(xT,yT,xA,yA)
    goto(xT,yT)
    #la tortue part de sa position (xT,yT)
    #et va au milieu de sa position
    #et de celle de B
    .......à compléter.....................
    #la tortue part de sa position (xT,yT)
    #et va au milieu de sa position
    #et de celle de C
    .......à compléter.....................
```

```
    #la tortue part de sa position (xT,yT)
    #et va au milieu de sa position
    #et de celle de D
     .......à compléter.....................
```

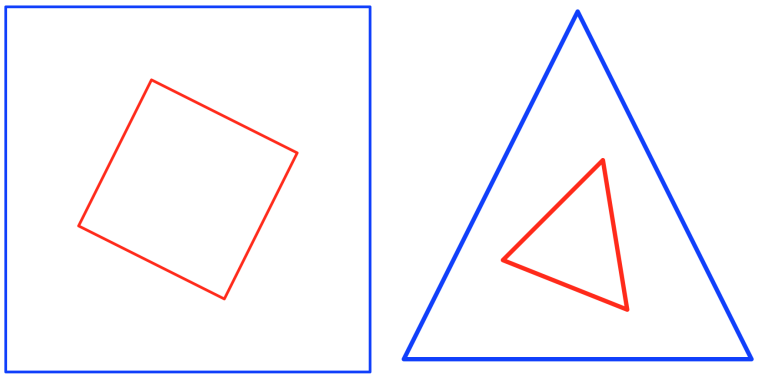
On veut qu'à l'écran le triangle n'apparaisse uniquement la partie "stable" de la trajectoire de la tortue, (voir image ci-dessous)

Une façon de faire est de lever le crayon (`penup()`) pour ne pas tracer la partie "instable" du circuit puis de baisser le crayon ensuite

2. Dessiner dans votre cahier les figures obtenues pour un carré, un triangle et un hexagone

   Avez vous observé des **propriétés géométriques** entre le polygone formé par les points A,B, C, etc... et la trajectoire de la tortue ?

   Faire évoluer votre programme pour confirmer ou infirmer ces conjectures.



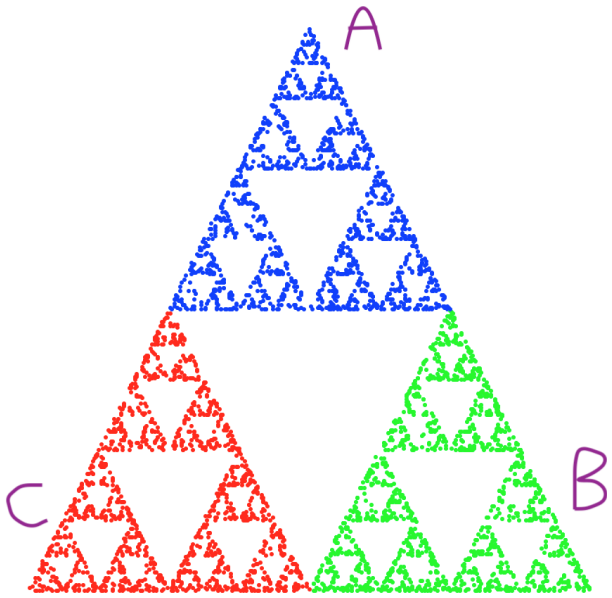
Le programme vous paraît il suffisamment **factorisé** ?

Chapitre 7

Test : Si .... Alors .....Sinon .....

7.1 Jeu du chaos

On va modifier la ronde des milieux dans un triangle ABC pour construire le dessin suivant



Comment ?
Avec la bibliothèque `random` "hasard" en anglais on peut **simuler** le hasard.
On tire au hasard un des trois nombres 1, 2 ou 3.

1. **Si** on obtient 1 la tortue va au milieu de sa position actuelle et du point A et marque cette position d'un point bleu
2. **Si** on obtient 2 la tortue va au milieu de sa position actuelle et du point B et marque cette position d'un point vert
3. **Si** on obtient 3 la tortue va au milieu de sa position actuelle et du point C et marque cette position d'un point rouge

Ouvrir le fichier `chaos.py` , le compléter le et le mettre au point en vous aidant des commentaires ci-dessous :

```
from turtle import *
from random import *

NB_REPETITIONS = 1000
EPAISSEUR   = 3
#-----------------------------------
def milieu(x1,y1,x2,y2):
    return (x1 + x2)/2 , (y1 + y2)/2
#-----------------------------------
#Coordonnées des sommets du triangle ABC
xA = 0
yA = 200

xB = 200
yB = -200

xC = -200
yC = -200
#Position initiale de la tortue
xT = 0
yT = 0
#------DEBUT DU JEU DU CHAOS---------------------

# tracer(NB_REPETITIONS)
penup()
for i in range(NB_REPETITIONS):

    # on choisit un nombre au hasard  entre 1 et 3

    de = randint(1,3)

    # Test pour choisir la position suivante et la couleur

    if de = = 1:
        xT,yT = milieu(xT,yT,xA,yA)
        pencolor("blue")

    if de = = .......  :
    # à compléter

    if    ........       :
    #à compléter

   #Conséquence du test

    goto(xT,yT)
```

```
    pendown()
    dot(EPAISSEUR)
    penup()
```

7.2 Commentaires :

1. **L'interpréteur permet de tester des fonctions**, ainsi vous pouvez tester la fonction `randint()` de la bibliothèque `random`
   D'abord entrer `from random import *` puis entrer par exemple `randint(1,3)`

```
>>> from random import *
>>> randint(1,3)
1
>>> randint(1,3)
2
```

2. `randint(a,b)` retourne un entier au hasard entre a et b entiers compris
3. Le résultat de `randint(1,3)` est mis dans une variable appelé `de`.
4. Très souvent un test permet de **comparer** le contenu d'une variable à d'autres valeurs. Attention le symbole = ne peut pas être utilisé pour le test d'égalité, car il est déjà utilisé pour **l' affectation**. On utilise dans la plupart des langages de programmation le symbole == pour le test d'égalité
5. On dit que `de == 1` est une **expression logique** et cette expression a deux valeurs possibles soit vraie (True) soit fausse (False)

```
>>> de = 1
>>> de == 1
True
>>> de == 2
False
```

```
if de = = 1:
    xT,yT = milieu(xT,yT,xA,yA)
    pencolor("blue")
```

   Si l'expression `de == 1` est vraie alors toutes les instructions situées après le : et indentées sont exécutées, sinon ces instructions ne sont pas exécutées et la prochaine instruction qui sera exécutée est celle située après l'instruction `pencolor("blue")`
6. On découvre une nouvelle fonction de la tortue la fonction `dot(epaisseur)` qui permet de tracer un point de la taille epaisseur où la tortue se trouve
7. Décommenter (c'est à dire enlever le symbole #) devant l'instruction `tracer(1000)` pour la rendre active. Cette commande permet de dessiner "plus rapidement" en ne faisant pas la mise à jour de l'écran pour chaque tracé d'un point mais ici pour 1000 points d'un coup.

8. On ne supprime pas des lignes de code on les commente pour les rendre inactives, lorsque ces lignes de code peuvent servir plus tard.
9. Le test peut être écrit sous cette forme :

```
if de = = 1:
     xT,yT = milieu(xT,yT,xA,yA)
     pencolor("blue")
elif de = = 2:
     xT,yT = milieu(xT,yT,xB,yB)
     pencolor("green")
else:
     xT,yT = milieu(xT,yT,xC,yC)
     pencolor("red")
```

ce qui signifie en Français :

Si on a obtenu 1 alors la tortue va au milieu de sa position actuelle et du point A et la couleur du stylo devient bleu **sinon si on a obtenu 2** alors la tortue va au milieu de sa position actuelle et du point B et la couleur du stylo devient vert (`elif` est la contraction de `else if` pour sinon si) et **sinon** (sous-entendu on a obtenu 3) la tortue va au milieu de sa position actuelle et du point C et la couleur du stylo devient rouge

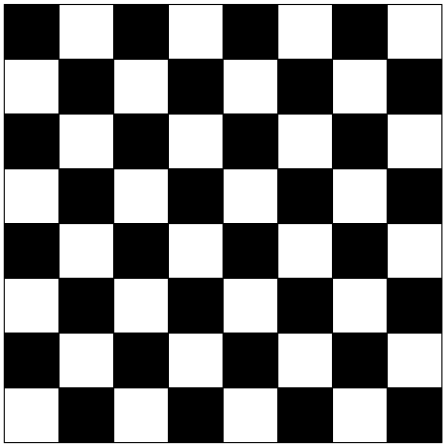
7.3 Exercices

1. Allez sur le site de Pythontutor pour visualiser le programme suivant :

```
from random import *
de = randint(1,3)
if de == 1:
     print(de)
elif de == 2:
    print(de*2)
else:
    print(de*3)
```

2. Comment faire le dessin suivant avec la Tortue ?

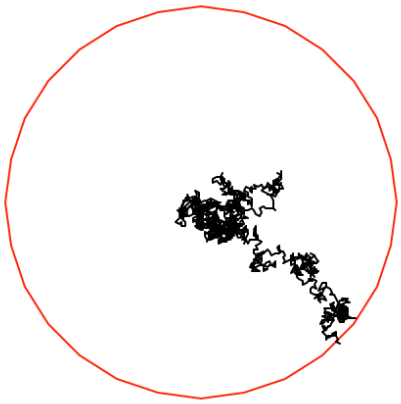
et celui-ci ?



Chapitre 8

La boucle Tant que

8.1 Tortue brownienne



L'image ci-dessus montre un déplacement aléatoire de la tortue à partir de l'origine du repère.

Tant qu'elle reste dans un cercle de centre l'origine et de rayon 100 pixels (cercle en rouge), elle avance par pas de 3 pixels mais change aléatoirement de direction entre deux pas.

Le but est de calculer la distance moyenne parcourue par la tortue pour un "grand" nombre de déplacements

Nous allons découvrir une nouvelle façon de faire des répétitions, **tant qu'une expression logique est vraie**

Voici en **pseudo code** la structure logique de l'algorithme (il manque des variables et certaines fonctions ne sont pas encore définies, cependant on **prévoit** leur rôle **avant** de les définir (Analyse))

Algorithme 3 : Déplacement aléatoire de la Tortue

```
début
    tant que estDansLeCercle(rayon) faire
        changerDirection()
        avancer(3)
    fin
fin
```

Une façon de changer de direction de manière aléatoire est de tirer au hasard un entier entre 1 et 360, et de régler la direction de la tortue à cette valeur.

Puisque cela dépend du langage utilisé on va désormais écrire en Python :

Pour changer de direction :

```
def changerDirection():
    setheading(randint(1,360))
```

Commentaires :

La valeur retournée par la fonction `randint()` est passée en paramètre à la fonction `setheading()` de la tortue qui règle l'angle de direction de la tortue en conséquence.

Ensuite on va définir une fonction `estDansLeCercle(rayon)` qui retourne vrai si la tortue est dans le cercle de centre l'origine et faux sinon.

```
def estDansLeCercle(rayon):
    return xcor()**2+ ycor()**2 < rayon**2
```

Commentaires :

1. Les fonctions `xcor()` et `ycor()` retournent les coordonnées de la tortue.
2. Dans le cours de mathématiques on a vu que la distance entre la tortue T et l'origine O peut être calculée par la formule
   $OT = \sqrt{(xT - xO)^2 + (yT - yO)^2} = \sqrt{(xT)^2 + (yT)^2}$.
   Or $OT < R \iff OT^2 < R^2$
   Le calcul d'une racine carrée par la machine prend plus de temps que le calcul d'un produit de deux nombres par conséquent dans le but d'économiser du temps de calcul on a évité de calculer une racine carrée.
3. $\text{xcor}() * \text{xcor}() + \text{ycor}() * \text{ycor}() < \text{rayon} * \text{rayon}$ est **une expression logique** qui est soit vraie soit fausse, donc ce qui est retourné est **la valeur de vérité** de l'expression logique, c'est à dire vrai ou faux.
   On peut tester la fonction `estDansLeCercle()` dans l'interpréteur :

```
>>> from turtle import *
>>> def estDansLeCercle(rayon):
        return xcor()**2 + ycor()**2 < rayon**2

>>> forward(10)
>>> estDansLeCercle(20)
True
>>> forward(10)
>>> estDansLeCercle(20)
False
```

Ouvrir le fichier `mvtBrownien.py`

Ajouter une variable `nbPas` qui compte le nombre de pas effectué par la tortue au cours d'un déplacement et affiche dans l'interpréteur `nbPas`

Afficher aussi dans l'interpréteur la **distance parcourue** par la tortue au cours d'un déplacement.

(La fonction `circle()` trace un cercle à la manière d'un polygone régulier avec un "grand" nombre de côtés. La position et la direction de la tortue influencent donc le tracé)

```
from turtle import *
from random import *

PAS = 3
RAYON = 100
#-------------------------------------
def changerDirection():
    setheading(randint(1,360))
#-------------------------------------
def estDansLeCercle(rayon):
    return xcor()**2+ ycor()**2 < rayon**2
#-------------------------------------
def tracerCercleLimite(rayon,couleur):
    penup()
    goto((0,-rayon))
    setheading(0)
    pencolor(couleur)
    pendown()
    circle(rayon)
    penup()
    goto(0,0)
    pendown()
#--début d'un déplacement aléatoire----
speed(0)
hideturtle()
tracerCercleLimite(RAYON,"red")
pencolor("black")
while estDansLeCercle(RAYON):
    changerDirection()
    forward(PAS)
```

8.2 Exercices

1. Ce qui nous intéresse maintenant c'est de répéter un grand nombre de fois un déplacement puis de calculer le nombre de pas **moyen** puis la distance moyenne parcourue par la tortue :

   (a) Introduire une constante `NB_REPETITIONS` initialisé à 50 et régler la constante `RAYON` à 20

(b) Introduire une variable `nbPasCumules` initialisé à 0 et et faire afficher la distance moyenne parcourue dans l'interpréteur

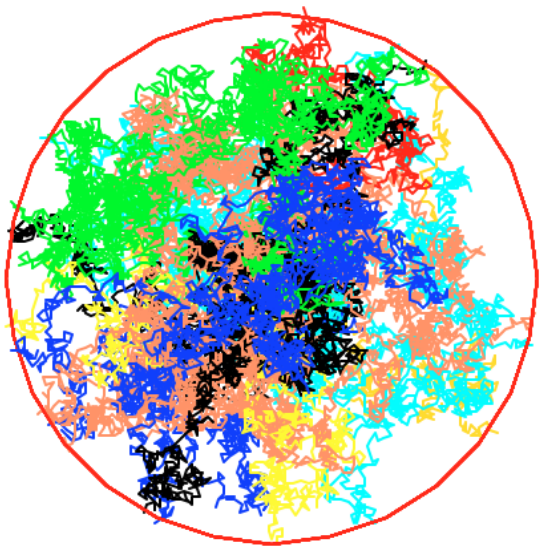
(c) Répéter 5 fois les 50 répétitions et observer que la distance moyenne varie. Calculer la moyenne des 5 valeurs.

2. Pour obtenir une image semblable à celle qui suit, introduire une liste de différentes couleurs

```
COULEURS = ["red","green","blue","cyan","yellow","black"]
```

puis remplacer `pencolor("black")` par `pencolor(choice(COULEURS))`

La fonction `choice()` de la bibliothèque `random` prend une couleur au hasard dans la liste `COULEURS`



Chapitre 9

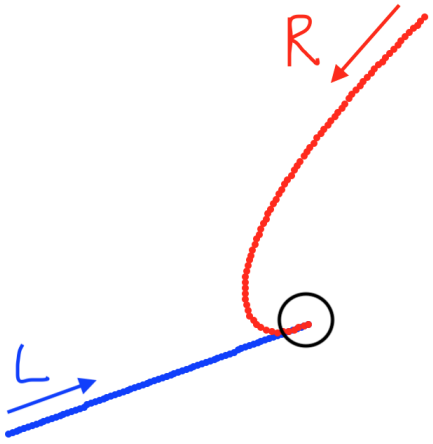
Expressions logiques

Nous avons vu l'importance des expressions logiques pour les tests, les boucles ainsi que les fonctions .

Nous allons voir que certaines contraintes ne peuvent être traduites qu'à l'aide d'expressions logiques obtenues par la connexion d'autres expressions logiques .

Regardons cela sur un exemple

9.1 Courbes de poursuite



Un lièvre suit une trajectoire **rectiligne et à vitesse constante** (en bleu). Un renard le poursuit **en courant toujours dans sa direction et à vitesse constante aussi** (en rouge).

Ils avancent tous les deux **par bonds de différentes longueurs**.

On peut voir sur l'image ci dessus la trajectoire du lièvre en bleue et la trajectoire du renard en rouge **dans un cas où le renard intercepte le lièvre**(à l'intérieur du cercle noir).

Voici comment on va modéliser la situation dans un premier temps

Algorithme 4 : Courbe de poursuite dans un espace illimité

```
début
    Fixer la direction du lièvre
    Fixer la longueur du bond du lièvre avec une constante BOND_LIEVRE
    Fixer la longueur du bond du renard avec une constante BOND_RENARD
    Placer le lièvre
    Placer le renard
    tant que lievreEstHorsDePortee() faire
        Diriger le regard du renard vers la position du lièvre
        Faire avancer le renard d'un bond dans cette direction
        Faire avancer le lièvre d'un bond vers le Nord
    fin
    Le renard capture le lièvre
fin
```

La fonction `lievreEstHorsDePortee()` retourne vrai si le lièvre est à une distance du renard strictement supérieure à `BOND_RENARD`

Mais dans le cas où le lièvre reste toujours hors de portée du lièvre **la boucle tant que ne s'arrêtera pas** et le renard ne capturera pas le lièvre.

Pour éviter cela on va compléter la condition d'arrêt de la boucle en ajoutant une contrainte pour que la poursuite s'arrête (dans le cas où le renard ne rattrapera jamais le lièvre) lorsque le lièvre quittera une zone de chasse délimitée par un carré.

Algorithme 5 : Courbe de poursuite dans un espace limité

```
début
    Fixer la direction du lièvre
    Fixer la longueur du bond du lièvre avec une constante BOND_LIEVRE
    Fixer la longueur du bond du renard avec une constante BOND_RENARD
    Placer le lièvre
    Placer le renard
    tant que lievreEstHorsDePortee() et lievreEstDansCarre() faire
        Diriger le regard du renard vers la position du lièvre
        Faire avancer le renard d'un bond dans cette direction
        Faire avancer le lièvre d'un bond
    fin
    Le renard capture le lièvre ou pas
fin
```

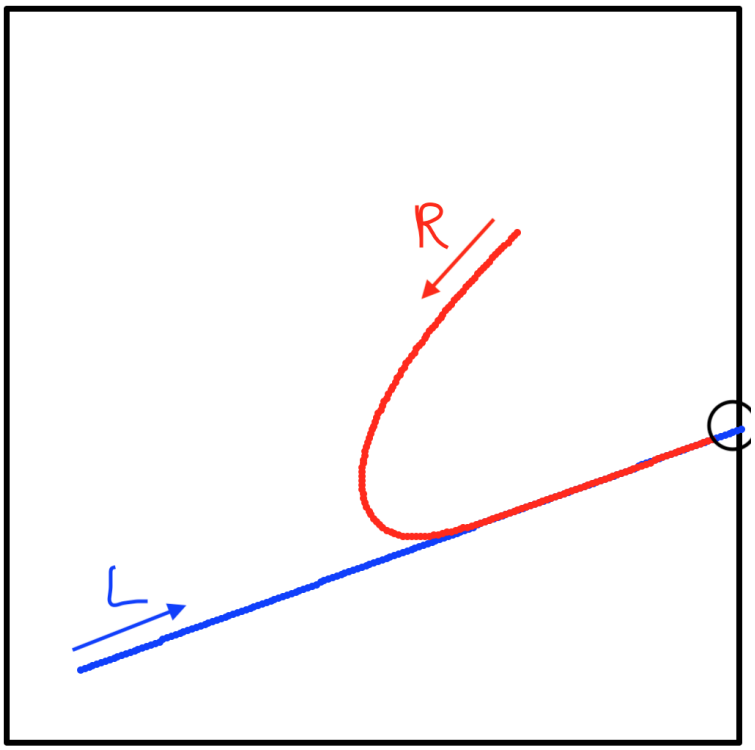
La fonction `lievreEstDansCarre(cote)` retourne vrai si le lièvre est dans le carré dont le coin inférieur gauche est l'origine du repère, de côté `cote`.

La condition `lievreEstHorsDePortee()` **et** `lievreEstDansCarre()` n'est vraie uniquement lorsque les valeurs retournées par les fonctions `lievreEstHorsDePortee()` puis `lievreEstDansCarre()` sont vraies.

Le connecteur logique `et` est une fonction ayant deux paramètres a et b et définie ainsi

et(a,b)	a	b
False	False	False
False	True	False
False	True	False
True	True	True

On peut voir ci-dessous le lièvre sortir du carré (en noir) sans avoir été attrapé par le renard.(cercle noir)



9.2 Exercices

On va mettre au point le programme directement en Python car on va utiliser des fonctions spécifiques pour traduire par exemple "Diriger le regard du renard vers la position du lièvre"

1. Ouvrir le fichier `courbePoursuite.py` et compléter à l'aide des questions suivantes : (ne pas toucher aux valeurs des variables)
   (a) Définir la fonction `lievreEstHorsDePortee()` en s'inspirant du TP sur la tortue brownienne
   (b) Lire en anglais la documentation Python (ci-dessous) sur la fonction `towards()` pour la tortue. Que fait cette fonction ? L'utiliser pour traduire "Diriger le regard du renard vers la position du lièvre"

Return the angle between the line from turtle position to position specified by (x,y), the vector or the other turtle. This depends on the turtle's start orientation which depends on the mode - "standard"/"world" or "logo").

```
>>> turtle.goto(10, 10)
>>> turtle.towards(0,0)
225.0
```

   (c) En ne faisant varier que les variables `BOND_LIEVRE` et `BOND_RENARD` conjecturer quand le renard intercepte le lièvre .
   (d) Définir la fonction `lievreEstDansCarre(cote)` et la fonction `dessineCarre(cote)`
2. Dans l'interpréteur définir une fonction `ou(a,b)` puis compléter le tableau suivant :

```
>>> def ou(a,b):
        return a or b

>>> ou(False,False)
False
```

ou(a,b)	a	b
.....	False	False
.....	True	False
.....	True	False
.....	True	True

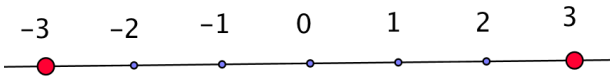
Chapitre 10

Mini-projets

Voici deux mini-projets pour les vacances de Février et Avril à faire par groupe de deux ou trois.

Ces projets permettent de réinvestir ce qui a été vu dans les précédents TP

10.1 Projet 1 : Marche aléatoire avec arrêt



La tortue part de l'origine (0,0) du repère et avance au hasard sur un axe repéré.

Tant qu'elle n'a pas atteint les points d'abscisses 3 ou -3 elle peut avancer d'une unité au hasard, pour cela on lance une pièce, si on obtient FACE son abscisse augmente de 1 sinon son abscisse diminue de 1

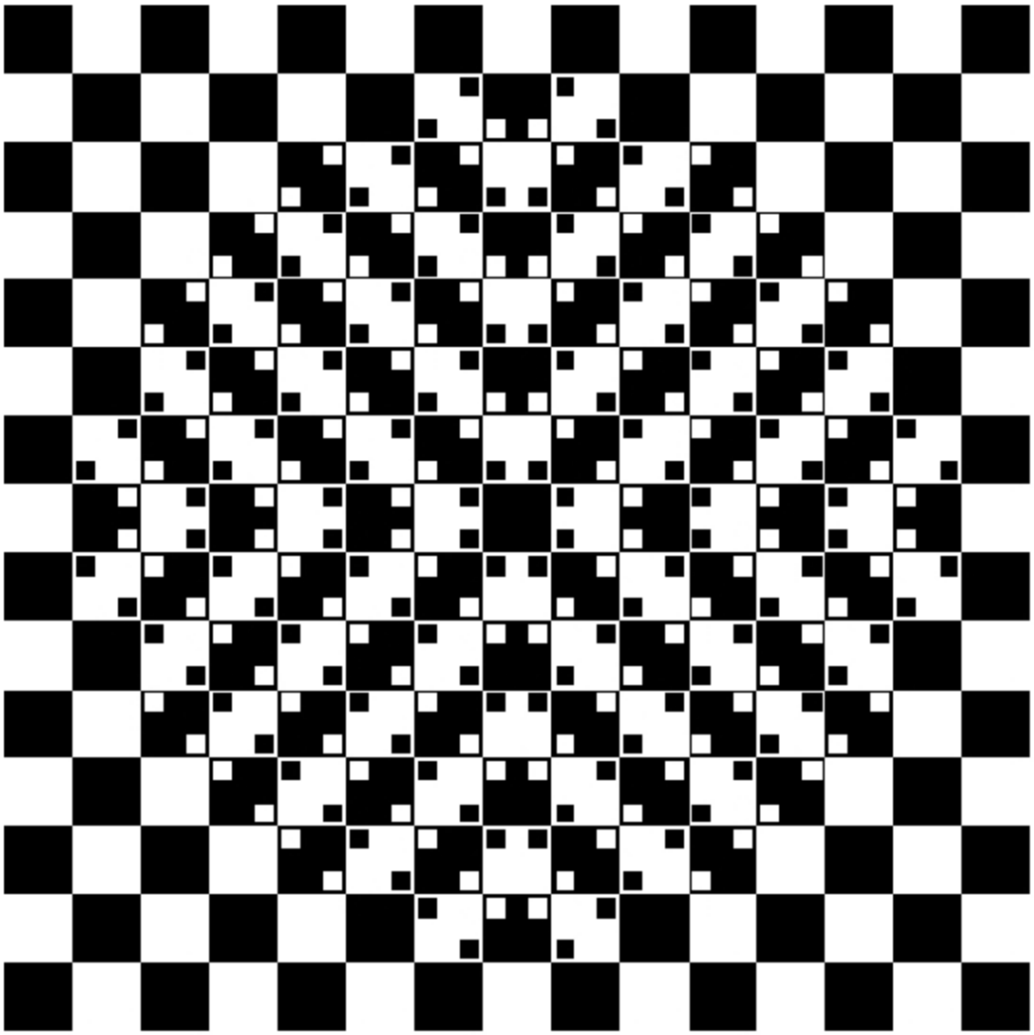
La longueur d'une unité est mémorisé par la variable PAS

Le rayon du cercle "limite" de centre (0,0) est mémorisé par la variable SEUIL, au début SEUIL a pour valeur 3

Ce projet "ressemble" au TP 9 et il s'agit de trouver une relation statistique entre le nombre moyen de lancers de dé et SEUIL

10.2 Projet 2 : Une illusion d'optique

Akiyoshi Kitaoka (voir internet) a réalisé cette illusion d'optique qu'il s'agit de dessiner avec la tortue :

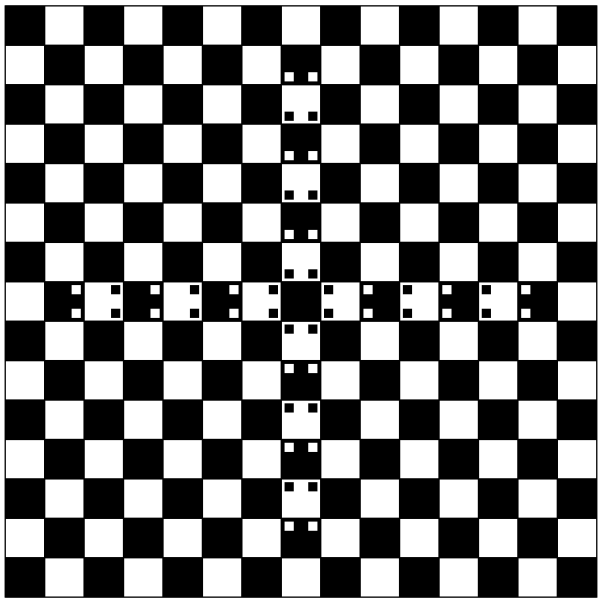


Si on regarde bien il n'y a que des carrés dans cette figure !

1. Analyser dans un premier temps le dessin pour trouver une "logique" pour construire ce dessin :

   Combien de lignes ? Combien de colonnes ? Le carré noir central sera centré en (0 ;0) pour des raisons de symétrie. Utiliser ensuite le programme du chapitre 8 pour dessiner le damier.

2. Comment sont placés les petits carrés noirs et blancs sur la croix centrale ?



3. Comment placer les petits carrés en diagonale ?