Programmation objet

Avec `Python` *tout est objet : un entier, une chaîne, une liste, une fonction... Nous allons voir comment définir nos propres objets.*

Cours 1 (Programmation objet : la classe !).
Un ***objet*** est une entité qui regroupe à la fois des variables et des fonctions. Le premier intérêt est qu'un objet est indépendant et auto-suffisant puisqu'il contient tout ce qu'il faut pour être utilisé, il permet d'éviter le recours aux variables globales par exemple.
Un objet est défini comme une ***instance*** d'une ***classe***, c'est-à-dire un élément d'une catégorie. Voici un exemple de la vie courante : on considère la classe *Chien*, alors mon chien *Médor* est un objet, appartenant à la classe *Chien*. Note que *Chien* est un concept, mais que *Médor* est bien réel. Mon autre chien *Foulcan* est aussi une instance de *Chien*.

Voici comment définir le début d'une classe `Vecteur()` afin de modéliser des vecteurs de l'espace :

```
class Vecteur:
    def __init__(self,x,y,z):
        self.x = x
        self.y = y
        self.z = z
```

Pour l'instant un vecteur est un concept auquel sont rattachés trois nombres (x, y, z). Le mot `self` fait référence à l'objet lui-même mais dont on ne connaît pas encore le nom (ce sera `V` ou bien `V1`, `V2`...).
Et voici un objet défini à partir de cette classe :

`V = Vecteur(1,2,3)`

Cet objet possède trois ***attributs*** :

`V.x`   `V.y`   `V.z`

qui valent ici respectivement 1, 2 et 3. Autre exemple, le calcul `V.x + V.y + V.z` renvoie 6. Je peux changer une de ces valeurs comme pour une variable classique (même si ce n'est pas la manière recommandée), par exemple :

`V.x = 7`

Maintenant `V.x + V.y + V.z` vaut 12.
Tu peux définir plusieurs objets qui seront indépendants les uns des autres :

`V1 = Vecteur(1,2,3)`   `V2 = Vecteur(1,0,0)`

Ainsi par exemple `V1.y` vaut 2, `V2.y` vaut 0.

Cours 2 (Programmation objet : de la méthode.).
On a vu comment attribuer des variables à un objet. Nous allons voir comment lui associer des fonctions.

Pour un objet, une fonction associée s'appelle une ***méthode***.
Si on reprend l'exemple de la classe *Chien*, on pourrait lui associer une méthode *Viens_ici_!*. On peut donc demander *Médor.Viens_ici_!* ou bien *Foulcan.Viens_ici_!* pour appeler chacun de nos chiens.

Complétons notre classe `Vecteur()` pour lui associer trois nouvelles méthodes :

```
class Vecteur:
    def __init__(self,x,y,z):
        self.x = x
        self.y = y
        self.z = z

    def norme(self):
        N = sqrt(self.x**2 + self.y**2 + self.z**2)
        return N

    def produit_par_scalaire(self,k):
        W = Vecteur(k*self.x,k*self.y,k*self.z)
        return W

    def addition(self,other):
        W = Vecteur(self.x+other.x,self.y+other.y,self.z+other.z)
        return W
```

- La méthode `norme()` renvoie la norme $\sqrt{x^2+y^2+z^2}$ d'un vecteur (x,y,z) (il faut importer le module `math`). Par exemple pour `V = Vecteur(1,2,3)`, on demande sa norme par la commande :
  `V.norme()`
  qui renvoie ici une valeur approchée de $\sqrt{14} = 3.74\ldots$
  La méthode `norme()` est définie comme une fonction classique, le paramètre prend le nom de `self` et correspond à l'objet (le vecteur `V` pour notre exemple). On récupère les coordonnées par `self.x`, `self.y`, `self.z` (pour notre exemple cela correspond à `V.x`, `V.y`, `V.z`).
- La méthode `produit_par_scalaire(self,k)` multiplie les coordonnées d'un vecteur par un réel k. Par exemple pour `V = Vecteur(1,2,3)` alors la commande :
  `W = V.produit_par_scalaire(7)`
  définit un nouvel objet `Vecteur()`, noté `W`, représentant le vecteur $\vec{w} = (7,14,21)$. Maintenant `W` est un objet de classe `Vecteur()` comme les autres et on peut par exemple calculer sa norme par `W.norme()`. La méthode `produit_par_scalaire()` est définie à l'aide de deux paramètres. Le premier est obligatoirement `self` et fait toujours référence à l'objet traité. Le second est ici le facteur k. Lorsque l'on appelle la méthode cela devrait être `produit_par_scalaire(V,7)` mais la syntaxe des objets est `V.produit_par_scalaire(7)` (le premier argument passe devant le nom de la méthode, les autres arguments sont décalés).
- La méthode `addition(self,other)` renvoie le vecteur somme de deux vecteurs, cela correspond à l'opération
  $$(x,y,z)+(x',y',z') = (x+x',y+y',z+z')$$
  Voici un exemple d'utilisation :

```
V1 = Vecteur(1,2,3)
V2 = Vecteur(1,0,-4)
V3 = V1.addition(V2)
```

On définit deux vecteurs $\vec{v}_1$ et $\vec{v}_2$, leur somme $\vec{v}_3$ vaut ici $(2,2,-1)$.
La méthode `addition()` est définie à l'aide de deux paramètres : le premier est toujours `self` et le second se nomme ici `other` pour signifier qu'il s'applique à un autre objet de la même classe. Pour notre exemple le paramètre `self` correspond à l'argument `V1` et le paramètre `other` à l'argument `V2`.

Cours 3 (Programmation objet : convivialité).
Complétons notre classe `Vecteur()` afin de permettre un joli affichage et d'additionner les vecteurs à l'aide de l'opérateur « + ».

```
class Vecteur:
    def __init__(self,x,y,z):
        ...

    def __str__(self):
        ligne = "("+str(self.x)+","+str(self.y)+","+str(self.z)+")"
        return ligne

    def __add__(self,other):
        W = Vecteur(self.x+other.x,self.y+other.y,self.z+other.z)
        return W
```

- La méthode `__str__()` (le nom est réservé) renvoie ici un bel affichage du vecteur. Par exemple avec `V = Vecteur(1,2,3)` alors :

  `print(V.__str__())` affiche `(1,2,3)`.

  Mais ce n'est pas comme cela qu'on l'utilise car une fois que la méthode `__str__()` est définie alors la commande :

  `print(V)` affiche aussi `(1,2,3)`.

  C'est très pratique !
- La méthode `__add__()` a exactement la même définition que la méthode `addition()` définie précédemment. Avec `V1 = Vecteur(1,2,3)` et `V2 = Vecteur(1,0,-4)` on pourrait l'utiliser par :

  `V3 = V1.__add__(V2)`

  Mais comme on a utilisé le nom réservé `__add__()` alors cela a défini l'opérateur « + » et il est beaucoup plus agréable d'écrire simplement :

  `V3 = V1 + V2`

Cours 4 (Programmation objet : résumé.).
Voici la définition complète de la classe `Vecteur()` accompagnée d'un résumé des explications.

mot reservé `class`
nom de la classe

```
class    Vecteur:
```

méthode d'initialisation `__init__()`
`self` correspond à l'objet en cours
définition des attributs x, y, z

```
    def __init__(   self    ,x,y,z):
        self.x = x
        self.y = y
        self.z = z
```

méthode pour l'affichage par `print()`

```
    def __str__   (self):
        ligne = "("+str(self.x)+","+str(self.y)+","+str(self.z)+")"
        return ligne
```

`self` : objet en cours
`self.x` : valeur de l'attribut x de l'objet en cours

```
    def  norme   (self):
        N = sqrt(   self.x **2 + self.y**2 + self.z**2 )
        return N
```

renvoie un nombre

```
    def produit_par_scalaire(self,k):
        W =   Vecteur  (k*self.x,k*self.y,k*self.z)
        return W
```

définition d'un objet de la classe `Vecteur`
renvoie un objet

`self` : objet en cours
`other` : un autre objet

```
    def addition(   self,   other):
        W = Vecteur(self.x+other.x,self.y+other.y,self.z+other.z)
        return W
```

renvoie un nouvel objet

méthode `__add__()` pour l'addition par `"+"`

```
    def  __add__   (self,other):
        W = Vecteur(self.x+other.x,self.y+other.y,self.z+other.z)
        return W
```

```
# Exemple 1
V = Vecteur(1,2,3)
print("Valeur de x :",   V.x)
print("Vecteur :", V)
print("Norme :", V.norme())
```

un objet `V` : une instance de la classe `Vecteur` initialisée par des valeurs x, y, z
`V.x` valeur de l'attribut x associé à `V`
affichage de l'objet `V` grâce à la méthode `__str__()`
appel de la méthode `norme()`
l'argument `V` correspond au paramètre `self`

```
# Exemple 2
V1 = Vecteur(1,2,3)
V2 = Vecteur(1,0,-4)

V3 = V1.addition(V2)
print(V3)

V4 = V1 + V2
print(V4)
```

définition de deux objets
appel de la méthode `addition()`
l'argument `V1` correspond au paramètre `self`
l'argument `V2` correspond au paramètre `other`
utilisation de `"+"` par l'appel à la méthode `__add__()`

Cours 5 (Matrice 2×2).

- Une matrice 2×2 est un tableau :

$$M = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$$

- On peut additionner deux matrices et multiplier une matrice par un réel :
$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} + \begin{pmatrix} a' & b' \\ c' & d' \end{pmatrix} = \begin{pmatrix} a+a' & b+b' \\ c+c' & d+d' \end{pmatrix} \qquad k \cdot \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} ka & kb \\ kc & kd \end{pmatrix}$$
- Le produit de deux matrices est défini par la formule suivante :
$$\begin{pmatrix} a & b \\ c & d \end{pmatrix} \times \begin{pmatrix} a' & b' \\ c' & d' \end{pmatrix} = \begin{pmatrix} aa'+bc' & ab'+bd' \\ ca'+dc' & cb'+dd' \end{pmatrix}$$
- La trace et le déterminant sont deux réels associés à une matrice :
$$\operatorname{tr}\begin{pmatrix} a & b \\ c & d \end{pmatrix} = a+d \qquad \det\begin{pmatrix} a & b \\ c & d \end{pmatrix} = ad-bc$$
- Si une matrice $M = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ a son déterminant non nul alors elle admet un inverse :
$$M^{-1} = \frac{1}{ad-bc}\begin{pmatrix} d & -b \\ -c & a \end{pmatrix}$$
de sorte que
$$M \times M^{-1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Activité 1 (Matrices).

Objectifs : définir des matrices comme des objets.

On commence à définir une classe `Matrice()` pour stocker des matrices 2×2 et leurs opérations.

```
class Matrice:
    def __init__(self,a,b,c,d):
        self.a = a
        self.b = b
        self.c = c
        self.d = d
```

Un matrice M sera donc définie par la commande :

```
M = Matrice(1,2,3,4)
```

pour définir la matrice
$$M = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

1. Définis une méthode `__str__(self)` qui permet l'affichage de la matrice. Cette méthode permet aussi d'obtenir l'affichage à l'aide `print()`. Si on a défini `M = Matrice(1,2,3,4)` alors la commande `print(M)` équivaut à la commande `print(M.__str__())` et affiche à l'écran :
```
1  2
3  4
```
2. Définis une méthode `trace(self)` et une méthode `determinant(self)` qui calcule la trace et le déterminant d'une matrice. Pour notre exemple `M.trace()` renvoie 5 et `M.determinant()` renvoie −2.
3. Définis une méthode `produit_par_scalaire(self,k)` qui renvoie la matrice correspondant au produit de chaque coefficient par le réel k. Ainsi, à partir de notre matrice M, on peut définir une

nouvelle matrice M' par la commande `MM = M.produit_par_scalaire(5)` qui correspond à $M' = \begin{pmatrix} 5 & 10 \\ 15 & 20 \end{pmatrix}$.

4. Définis une méthode `inverse(self)` qui calcule l'inverse d'une matrice (et renvoie `None` si le déterminant est nul). Pour notre exemple `M.inverse()` renvoie la matrice qui correspond à

$$M^{-1} = \begin{pmatrix} -2 & 1 \\ \frac{3}{2} & -\frac{1}{2} \end{pmatrix}$$

5. (a) Définis une méthode `addition(self,other)` qui calcule la somme de deux matrices. Par exemple avec :

```
M1 = Matrice(4,3,2,1)      M2 = Matrice(1,0,-1,1)
```

puis :

```
M3 = M1.addition(M2)
```

alors `M3` correspond à la matrice :

$$M_3 = M_1 + M_2 = \begin{pmatrix} 4 & 3 \\ 2 & 1 \end{pmatrix} + \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} = \begin{pmatrix} 5 & 3 \\ 1 & 2 \end{pmatrix}$$

(b) C'est beaucoup mieux de nommer cette méthode `__add__(self,other)` puisque cela permet d'écrire tout simplement :

```
M3 = M1 + M2
```

6. (a) Définis une méthode `multiplication(self,other)` qui calcule le produit de deux matrices. Par exemple avec nos matrices M_1 et M_2 :

```
M4 = M1.multiplication(M2)
```

correspond à la matrice :

$$M_4 = M_1 \times M_2 = \begin{pmatrix} 4 & 3 \\ 2 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 \\ -1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 3 \\ 1 & 1 \end{pmatrix}.$$

(b) C'est beaucoup mieux de nommer cette méthode `__mul__(self,other)` puisque cela permet d'écrire tout simplement :

```
M4 = M1 * M2
```

Vérifie que $M_1 \times M_2$ et $M_2 \times M_1$ ne sont **pas** les mêmes matrices !

(c) Vérifie sur plusieurs exemples qu'une matrice, multipliée par son inverse, vaut la matrice identité $I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$. Par exemple on a bien `M1 * M1.inverse()` qui vaut la matrice identité.

7. *Application.* La suite de Fibonacci est définie par récurrence :

$$F_0 = 1 \qquad F_1 = 1 \qquad F_{n+2} = F_{n+1} + F_n \quad \text{pour } n \geqslant 0.$$

Chaque terme est donc la somme des deux termes précédents. Les premiers termes sont :

$$F_0 = 1 \quad F_1 = 1 \quad F_2 = 2 \quad F_3 = 3 \quad F_4 = 5 \quad F_5 = 8 \quad F_6 = 13 \ldots$$

Une autre façon de calculer F_n est d'utiliser des matrices. Soit

$$M = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$$

Alors

$$M^n = \underbrace{M \times M \times \cdots \times M}_{n \text{ fois}} = \begin{pmatrix} F_{n-2} & F_{n-1} \\ F_{n-1} & F_n \end{pmatrix}$$

Autrement dit F_n est le dernier coefficient de la matrice M^n.

Calcule F_{100} à l'aide des matrices.

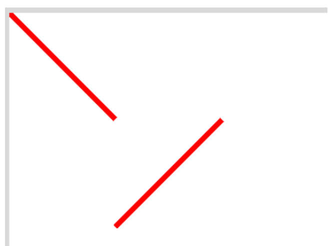
Activité 2 (Tortue basique).

Objectifs : programmer une tortue basique (sur le principe de Scratch*) qui réagit à des instructions simples.*

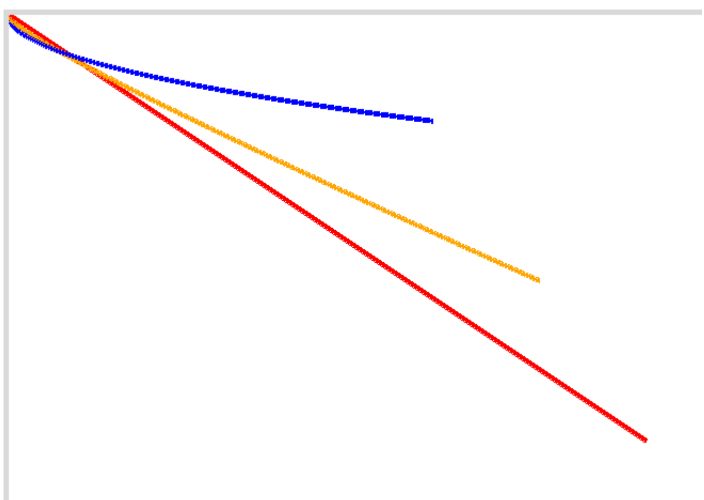
Voici le début de la définition d'une classe `TortueBasique()` qui définit quatre attributs : les coordonnées x et y de la position courante de la tortue (située au départ en $(0,0)$), la position du stylo (`trace` vaut « Vrai » ou « Faux »), la couleur du stylo :

```
class TortueBasique:
    def __init__(self):
        self.x = 0
        self.y = 0
        self.trace = True
        self.couleur = 'red'
```

1. Copie puis complète cette définition avec une méthode `renvoyer_xy(self)` qui renvoie les coordonnées (x, y) de la position courante.
2. Complète avec une méthode `aller_a_xy(self,x,y)` qui déplace la tortue à la position (x, y) indiquée. Si `trace` vaut « Vrai », trace un segment entre l'ancienne et la nouvelle position.
   Indication. Pour le tracé utilise le module `tkinter` (voir plus bas).
3. Complète avec des méthodes `abaisser_stylo(self)`, `relever_stylo(self)` qui changent la valeur de l'attribut `trace` et une méthode `changer_couleur(self,couleur)` puis dessine la figure suivante :



4. Définis une `tortue1` (rouge), une `tortue2` (bleue). Définis une `tortue3` (orange) qui à chaque déplacement de `tortue1` et `tortue2` se place au milieu de ces deux tortues.
   Sur le dessin ci-dessous : `tortue1` se déplace en $(\frac{3}{2}i, i)$ (pour i allant de 0 à 400) ; `tortue2` se déplace successivement en $(i, 5\sqrt{i})$. Pour chaque i, on récupère les positions de ces deux tortues et `tortue3` se place au milieu.



Voici un exemple d'utilisation de notre tortue en utilisant le module `tkinter` pour l'affichage (voir le dessin ci-dessus).

```
from tkinter import *
root = Tk()
canvas = Canvas(root, width=800, height=600, background="white")
canvas.pack(side=LEFT, padx=5, pady=5)

tortue = TortueBasique()

tortue.aller_a_xy(100,100)
tortue.relever_stylo()
tortue.aller_a_xy(200,100)
tortue.abaisser_stylo()
tortue.aller_a_xy(100,200)

root.mainloop()
```

Cours 6 (Programmation objet : héritage)**.**
La programmation objet possède un autre intérêt : à partir d'une classe d'objets on peut en définir d'autres, en récupérant certaines des fonctionnalités et en ajouter de nouvelles. C'est très utile par exemple pour reprendre et compléter du code écrit par d'autres. C'est la notion d'***héritage***.
Voyons un exemple : on veut créer un jeu vidéo où il faut combattre des ennemis. Il y a différents types d'ennemis mais ils ont tous des caractéristiques communes : une position (x, y) et des points de vie. Voici une classe `Ennemi()` avec une méthode qui affiche les points de vie restant et une autre qui diminue les points de vie après avoir été attaqué.

```
class Ennemi():
    def __init__(self,x,y,vie):
        self.x = x
        self.y = y
        self.vie = vie

    def affiche_vie(self):
        print("Vie =",self.vie)

    def perd_vie(self,n):
        self.vie = self.vie - n
```

Voici deux objets immobiles (des tours) définis par cette classe `Ennemi()` et quelques actions.

```
tour = Ennemi(1,2,100)
super_tour = Ennemi(5,3,200)
tour.affiche_vie()
tour.perd_vie(50)
tour.affiche_vie()
```

Pour les ennemis passifs (qui peuvent être attaqués, mais ne peuvent pas attaquer) la classe `Ennemi()` est bien adaptée. Par contre cette classe n'est pas assez évoluée pour des ennemis plus performants.

Prenons l'exemple d'un zombie : en plus des caractéristiques déjà décrites, il est actif (il peut vous attaquer, se déplacer...). Une solution est de programmer une classe `Zombie()` depuis zéro, mais ce serait dommage car une partie du travail a été faite avec la classe `Ennemi()`. Le plus simple est de récupérer les caractéristiques déjà existantes et d'en ajouter de nouvelles. C'est ce qu'on fait avec la classe `Zombie()` qui hérite des propriétés de la classe `Ennemi()` :

```
class Zombie(Ennemi):
    def __init__(self,x,y,vie,force):
        Ennemi.__init__(self,x,y,vie)
        self.force = force

    def affiche_force(self):
        print("Force =",self.force)
```

Voici un exemple d'utilisation :

```
mechant = Zombie(4,4,100,100)
mechant.affiche_force()
mechant.perd_vie(50)
mechant.affiche_vie()
```

Voici les explications :

- la classe `Zombie()` est définie par l'entête « `class Zombie(Ennemi):` » et ainsi hérite des attributs et des méthodes de la classe `Ennemi()`.
- Une instance de la classe `Zombie()` possède les attributs `x`, `y` et `vie` hérités de `Ennemi()` mais possède en plus des points d'attaques stockés dans `force`.
- La méthode `__init__()` initialise un objet de la classe `Zombie()`. Pour les caractéristiques déjà pré-existantes de la classe mère, on les initialise par `Ennemi.__init__(self,x,y,vie)` et il ne reste plus qu'à initialiser `force`.
- On définit une nouvelle méthode `affiche_force()` qui concerne seulement les `Zombie()`.
- On voit dans l'exemple d'utilisation comment définir un objet de la classe `Zombie()` (avec ses quatre attributs). On peut bien sûr utiliser la méthode `affiche_force()` spécifique à cette classe, mais aussi les méthodes `affiche_vie()` et `perd_vie()` héritées de la classe `Ennemi()`.

Activité 3 (Tortue tournante)**.**

Objectifs : définir une tortue plus performante en se basant sur les propriétés de la tortue basique déjà construite.

On veut améliorer la classe `TortueBasique()` en une classe `TortueTournante()` qui permet de diriger une tortue selon une direction. La classe `TortueTournante()` est donc héritée de la classe `TortueBasique()` et un nouvel attribut `direction` est créé. Le début de la définition est donc :

```
class TortueTournante(TortueBasique):
    def __init__(self):
        TortueBasique.__init__(self)
        self.direction = 0
```

L'attribut `direction` correspond à l'angle en degrés vers lequel pointe la tortue. L'angle 0 correspond à la droite, l'angle 90 degrés correspond au Nord.

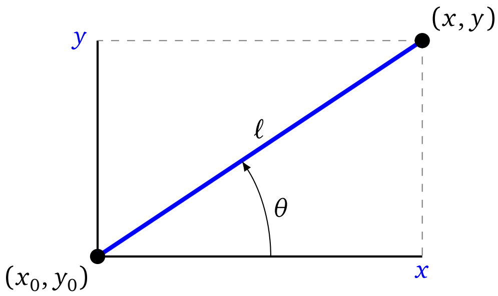
1. Complète la définition de la classe avec une méthode `fixer_direction(self,direction)` qui met à jour la direction courante avec l'angle donné.

Encapsulation. Cette fonction sert juste à éviter d'écrire `tortue.direction = 90` ce qui est déconseillé en dehors de la définition de la classe. Il faut plutôt utiliser `tortue.fixer_direction(90)`. Cette recommandation s'appelle l'***encapsulation***.

2. Complète la définition de la classe avec une méthode `tourner(self,angle)` qui change la direction courante en ajoutant l'angle donné.
3. Complète la définition de la classe avec une méthode `avancer(self,longueur)` qui fait avancer la tortue de la longueur donnée selon sa direction courante.

   Voici les formules pour calculer les coordonnées (x, y) du point d'arrivée en fonction du point de départ (x_0, y_0) de la direction θ (en degrés) et de la longueur ℓ :

$$x = x_0 + \ell \cos\left(\frac{2\pi}{360}\theta\right) \qquad y = y_0 + \ell \sin\left(\frac{2\pi}{360}\theta\right)$$

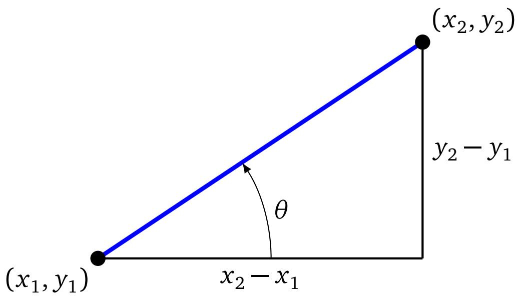


4. Complète la définition de la classe avec une méthode `sorienter_vers(self,other)` qui oriente une tortue `self` en direction de la tortue `other`.

   Indications. Si (x_1, y_1) sont les coordonnées d'une première tortue, (x_2, y_2) les coordonnées d'une seconde tortue alors l'angle formé entre l'horizontale et la droite joignant les deux tortues est donné (en degrés) par la formule :

$$\theta = \frac{360}{2\pi} \operatorname{atan2}(y_2 - y_1, x_2 - x_1)$$

   où $\operatorname{atan2}(y, x)$ est une variante de la fonction arctangente disponible dans le module `math`.



Programme une poursuite de tortue : une tortue bleue descend pas à pas, à chaque étape la tortue rouge s'oriente vers la tortue bleue, puis avance. La courbe tracée par la tortue rouge s'appelle une « courbe de poursuite ».