Programmation orientée objet

1 Vocabulaire de la programmation orientée objet

Nous avons déjà utilisé sans le savoir des objets.

Par exemple en Python lorsqu'on ajoute un nouvel élément à une liste donnée en utilisant la fonction `append()` on écrit par exemple à la console et dès que l'on a écrit `liste.` une fenêtre nous propose plusieurs fonctions possibles parmi lesquelles on retrouve `append()`

```
>>> liste = [1,2,3]

>>> liste.
```

append
clear
copy
count
extend
index
insert
pop

Si on écrit une fonction autre que celles proposées ci-dessus il y aura un message d'erreur

```
>>> liste = [1,2,3]

>>> liste.split(" ")
Traceback (most recent call last):
  File "<console>", line 1, in <module>
AttributeError: 'list' object has no attribute 'split'

>>> type(liste)
<class 'list'>
```

Nous voyons apparaître le mot 'object' dans le message d'erreur, de même lorsque nous demandons le type de la variable, nous voyons apparaître le mot 'class'

Nous avons aussi utilisé le module `turtle` qui contient plu-

sieurs **classes** et nous avons déjà utilisé l'une d'entre elles la classe `Turtle`

Qu'est ce qu'une **classe** ? qu'est ce qu'un **objet** ?

Par analogie, une **classe** ressemble au plan de construction d'une voiture à partir duquel on peut créer plusieurs voitures appelés **objets**

On dit aussi que les objets sont des **instances** de la classe

Du module turtle on importe la classe Turtle et on créé trois instances de cette classe

```
from turtle import Turtle
a = Turtle()
b = Turtle()
c = Turtle()
```

On appelle **attribut** une caractéristique d'un objet, ainsi une instance de la classe Turtle a plusieurs attributs comme une position relativement à un repère du plan (abscisse et ordonnée) une direction (un angle) une forme (triangle, etc...) une couleur, etc...

Nous pouvons connaître l'état d'une tortue et la faire évoluer grâce à des fonctions appelée **méthodes**

Toutes les méthodes de la tortue sont documentées sur le site officiel de Python à l'adresse ci-dessous

`https://docs.python.org/fr/3/library/turtle.html#overview-o`

Par exemple la méthode `position()` ou encore la méthode `towards()` sont définies ci-dessous dans la documentation officielle de Python, où `turtle` est une instance de la class `Turtle`

La syntaxe est `objet.méthode(paramètres)` pour exécuter une méthode sur un objet avec éventuellement des paramètres

Connaître l'état de la tortue

turtle.**position**()
turtle.**pos**()

Renvoie la position actuelle de la tortue (x,y) (en tant qu'un vecteur Vec2d).

```
>>> turtle.pos()
(440.00,-0.00)
```

turtle.**towards**(*x*, *y=None*)

Paramètres:
- **x** -- un nombre, ou une paire / un vecteur de nombres, ou une instance de tortue
- **y** -- un nombre si *x* est un nombre, sinon `None`

Renvoie l'angle entre l'orientation d'origine et la ligne formée de la position de la tortue à la position spécifiée par (x,y), le vecteur ou l'autre tortue. L'orientation d'origine dépend du mode — "standard"/"world" ou "logo".

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

En utilisant la méthode goto() on va placer les trois tortues a,b et c en trois points de coordonnées (-200,-150), (200,-150) et (0,250)

```
from turtle import Turtle
SEUIL = 10
#on créé trois instances a,b et c de la classe Turt
a = Turtle()
b = Turtle()
c  = Turtle()

#on place les tortues
a.penup()
a.goto(-200,-150)
a.color("red")
a.pendown()

b.penup()
b.goto(200,-150)
b.color("green")
```

```
b.pendown()

c.penup()
c.goto(0,250)
c.color("blue")
c.pendown()
```

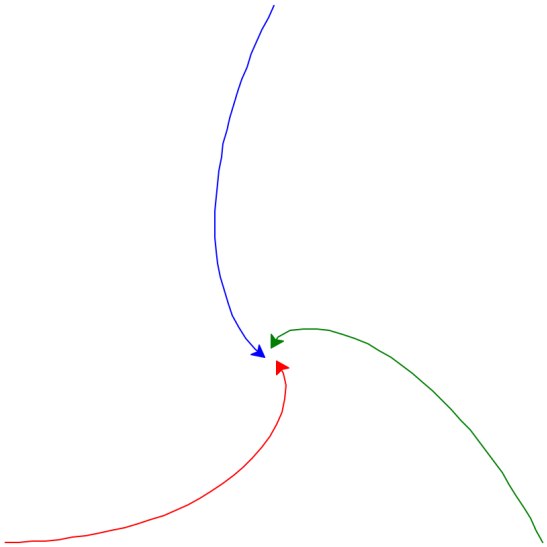
Enfin on va faire avancer la tortue c, dans la direction de la tortue a , de 10 pas, puis la tortue a, dans la direction de la tortue b , de 10 pas, puis la tortue b, dans la direction de la tortue c , de 10 pas, et **ceci tant que chaque tortue et celle qu'elle vise sont éloignées de plus de 10 pas**

`a.distance(b.position())` retourne la distance entre les tortues a et b en pas

```
from turtle import Turtle
SEUIL = 10
#on créé trois instances a,b et c de la classe Turt
a = Turtle()
b = Turtle()
c  = Turtle()

#on place les tortues
a.penup()
a.goto(-200,-150)
a.color("red")
a.pendown()

b.penup()
b.goto(200,-150)
b.color("green")
b.pendown()
```

```
c.penup()
c.goto(0,250)
c.color("blue")
c.pendown()

while a.distance(b.position()) > SEUIL and\
      b.distance(c.position()) > SEUIL and\
      c.distance(a.position()) > SEUIL:
    # c avance de 10 pas dans la direction de a
    c.setheading(c.towards(a.position()))
    c.forward(10)

     # b avance de 10 pas dans la direction de c
    b.setheading(b.towards(c.position()))
    b.forward(10)

    #a avance de 10 pas dans la direction de b
    a.setheading(a.towards(b.position()))
    a.forward(10)
```

On observe alors une **courbe de poursuite**



1.1 Exercice

L'utilisateur entre un entier n entre 3 et 6 compris

Créer n instances de la classe Turtle (et les insérer dans une liste)

Placer les n instances aux sommets d'un polygone régulier de n côtés

Créer n courbes de poursuites à la manière de ce qui a été fait précédemment

2 Définition d'une classe Vecteur

On a pu un jour **définir** nos propres fonctions, maintenant on va pouvoir **définir** notre propre type plus précisément une classe

Ci après nous allons créer la classe `Vecteur` (à deux dimensions) qui existe déjà en Python sous le nom de classe Vecteur2D, dont voici un extrait de la documentation

class `turtle`.**Vec2D**(*x*, *y*)

Une classe de vecteur bidimensionnel, utilisée en tant que classe auxiliaire pour implémenter les graphiques *turtle*. Peut être utile pour les programmes graphiques faits avec *turtle*. Dérivé des *n*–uplets, donc un vecteur est un *n*–uplet !

Permet (pour les vecteurs *a*, *b* et le nombre *k*) :

- `a + b` addition de vecteurs
- `a - b` soustraction de deux vecteurs
- `a * b` produit scalaire
- `k * a` et `a * k` multiplication avec un scalaire
- `abs(a)` valeur absolue de a
- `a.rotate(angle)` rotation

On va **définir** ce qu'est un vecteur de deux manières :

1. Par ses **attributs** :

   Un vecteur a deux attributs : une abscisse et une ordonnée

2. Par ses **méthodes**

   Etant donné un vecteur on peut le multiplier par un réel pour obtenir un nouveau vecteur, on peut l'ajouter avec un autre vecteur pour obtenir un autre vecteur etc...

```
#Définition
class Vecteur:
    # Constructeur
    def __init__(self,abscisse,ordonnee):
        self.x = abscisse
        self.y = ordonnee

    #Méthodes
    def mult_reel(self,k):
        return Vecteur(k*self.x,k*self.y)

    def somme(self,other):
        return Vecteur(self.x+other.x,self.y+other.

#Exécution
```

```
v1 = Vecteur(2,-1)
v2 = Vecteur(3,4)
v3 = v1.somme(v2)
```

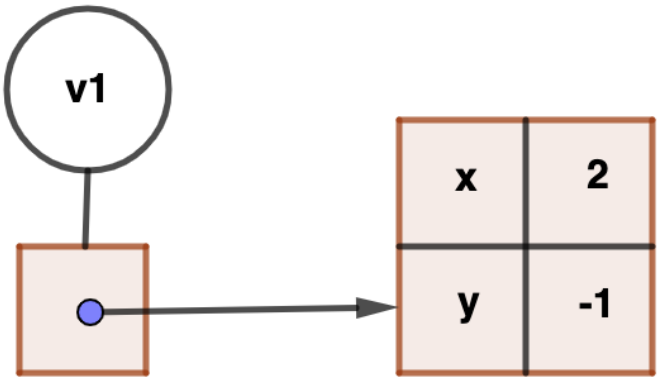
Remarques

1. Une classe est définie avec le mot réservé `class` qui englobe toutes les méthodes
2. Par convention le nom d'une classe commence par une lettre majuscule
3. Une méthode joue un rôle important c'est la méthode `__init__()` qui construit l'objet dénommé par `self` en mémoire

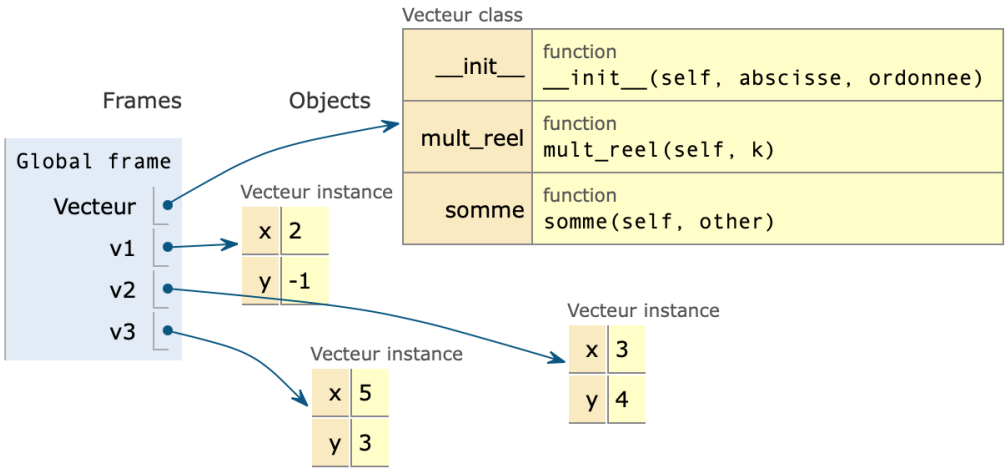
   L'instruction `v1 = Vecteur(2,-1)` créé en mémoire une variable `v1` de type Vecteur et dont la valeur est une référence en mémoire où se trouvent les valeurs 2 et -1

   Cette référence est représentée par une flèche dans le dessin ci-dessous

   La référence de la variable `v1`



Voici une visualisation avec Python Tutor du programme ci-dessus



2.1 Exercice

Chercher dans votre ordinateur le fichier `turtle.py` du module turtle et comprendre le code de la méthode `rotate()` de la classe Vec2D

3 Implémentation d'une liste chainée

Par analogie avec une chaîne de vélo constituée de maillons, une liste chaînée est constituée de cellules

Dans un premier temps on va créer une classe Cellule et une liste ne sera qu'une référence sur la première cellule de la liste

```
class Cellule:
    """
    définit un élément d'une liste chainée"""
    def __init__(self,v,s):
        self.valeur = v
        self.successeur = s

def est_vide(liste):
    return liste is None
```

```
def ajouter_en_fin(l,v):
    """
    ne modifie pas la référence sur
    la première cellule
    fonction itérative
    """
    pred = liste
    ref = liste.successeur
    while ref is not None:
        pred = ref
        ref = ref.successeur
    pred.successeur = Cellule(val,None)

def ajouter_en_fin2(l,v):
    """
    fonction récursive
    """
    if l.successeur is None:
        l.successeur =  Cellule(v,None)
    else:
        ajouter_en_fin2(l.successeur,v)

liste = Cellule(1,Cellule(2,None))
ajoute_en_fin(liste,3)
```

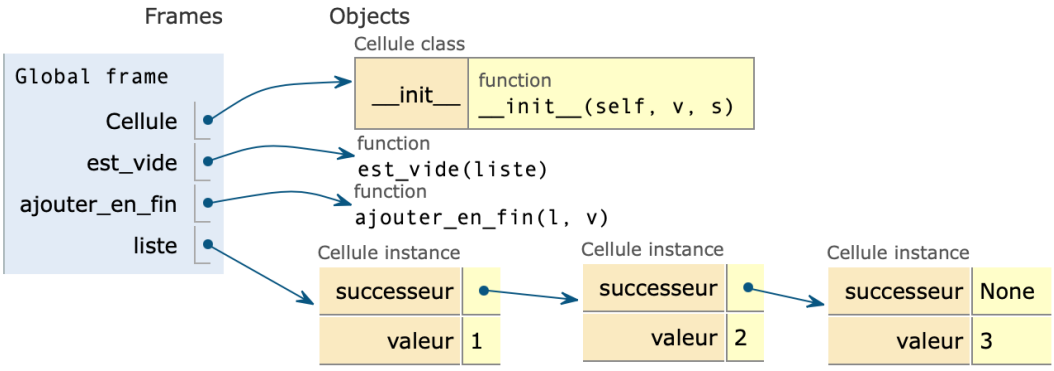
Voici la documentation Python à propos de `is` et `is not`

6.10.3. Comparaisons d'identifiants

Les opérateurs `is` et `is not` testent l'égalité des identifiants des objets : `x is y` est vrai si et seulement si *x* et *y* sont le même objet. L'identifiant d'un objet est déterminé en utilisant la fonction `id()`. `x is not y` renvoie le résultat contraire de l'égalité des identifiants [4].

Voici une visualisation avec Python Tutor du programme ci-

dessus



3.1 Exercice

Définir une fonction `ajouter_en_tete(l,v)` où la valeur v est insérée en tête de liste (la référence de la première cellule change)

4 Une classe Liste (encapsulation)

Une idée importante de la programmation orientée objet est **l'encapsulation** ce qui signifie que l'utilisation d'une classe se fait via une **interface** (on a accès aux attributs via l'interface et on n'a pas accès au contenu des méthodes)

Par exemple voici une partie de l'interface de la classe Turtle où on décrit les méthodes `pos()` et `towards()`

L'utilisateur n'a pas à connaître comment sont définies ces méthodes (on dit aussi comment elles sont **implémentées**)

Connaître l'état de la tortue

turtle.**position**()
turtle.**pos**()

Renvoie la position actuelle de la tortue (x,y) (en tant qu'un vecteur Vec2d).

```
>>> turtle.pos()
(440.00,-0.00)
```

turtle.**towards**(*x*, *y=None*)

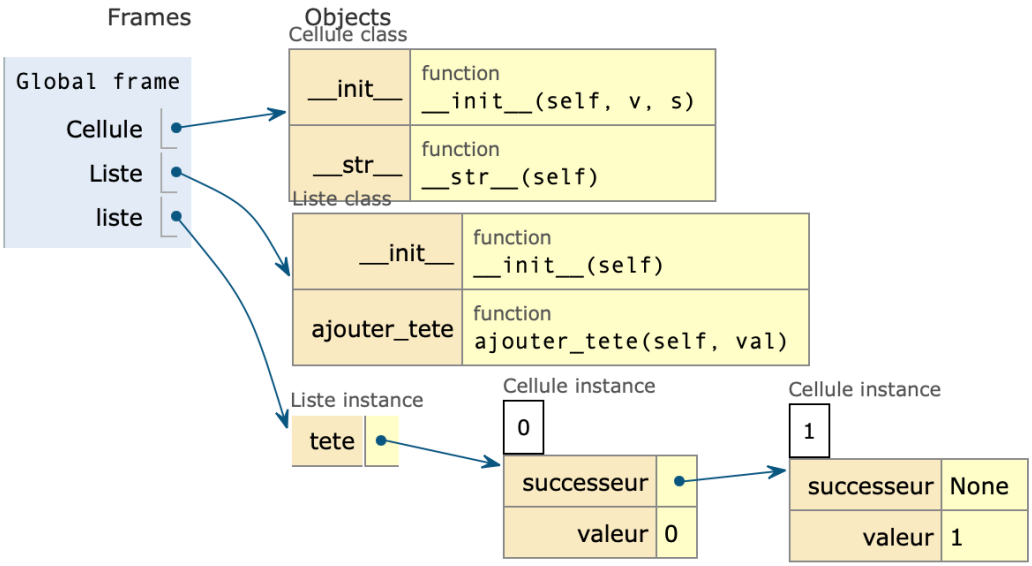
Paramètres:
- **x** -- un nombre, ou une paire / un vecteur de nombres, ou une instance de tortue
- **y** -- un nombre si *x* est un nombre, sinon `None`

Renvoie l'angle entre l'orientation d'origine et la ligne formée de la position de la tortue à la position spécifiée par (x,y), le vecteur ou l'autre tortue. L'orientation d'origine dépend du mode — "standard"/"world" ou "logo".

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

```
class Cellule:
    """
    définit un élément d'une liste chainée"""
    def __init__(self,v,s):
        self.valeur = v
        self.successeur = s

    def __str__(self):
        return str(self.valeur)

class Liste:
    """
    """
    def __init__(self):
        self.tete = None

    def ajouter_tete(self,val):
        self.tete = Cellule(val,self.tete)

if __name__ == "__main__":
```

```
liste = Liste()
liste.ajouter_tete(1)
liste.ajouter_tete(0)
```



Retenir : La programmation objet est un **paradigme de programmation** permettant de :

1. créer des nouveaux **types** autre que `int`,`float`,`tuple`, `list`,`str` par exemple `Vecteur`
2. Une `classe` nommée `Vecteur` par exemple peut être vue comme un ensemble de fonctions dont l'une le constructeur `__init__(self,abscisse,ordonnee)` crée un **objet** de type `Vecteur` en mémoire ayant des **attributs** (caractéristiques) et sur lequel peuvent agir d'autres fonctions de la classe appelées **méthodes**

5 Exercices

Ex 1

Un morceau de musique est défini par les attributs suivants :

1. genre
2. année
3. auteur
4. titre
5. durée

Définir le constructeur pour la classe Morceau_Musique

Ex 2

1. Comment caractériserez vous un élève du Lycée ? (Définir des attributs)
2. Définir un constructeur possible de la classe Eleve
3. Quelles méthodes ajouterez vous à cette classe ?

Ex 3

Définir une méthode `longueur()` pour la classe Liste de deux manières, itérative et récursive

Ex 4

Définir la méthode __str()__ pour la classe Liste

Ex 5

Définir la méthode spécifique __len()__ pour la classe Liste qui retourne la longueur d'une liste l, de telle sorte que l'appel de cette méthode se fait ainsi `len(l)` comme pour les listes en Python

Défi Définir une fonction `inv_liste(l)` qui retourne l'inverse de la liste chaînée l avec une complexité linéaire