



PYTHON & ALGORITHMIQUE

Bastien Gorissen & Thomas Stassin

READY ?

Press start!

JEU DU JOUR

“Nuru”

NURU



BLENDINGJAM12

JEU DU JOUR

“The writer will do something”

<https://mrwasteland.itch.io/twwds>

LEVEL 1-9

Re-Rappel

LES DÉFINITIONS IMPORTANTES

Classe: Description d'un type d'objet.

Objet: Représentant d'une classe.

Instance: Un objet est une instance d'une classe.

Attribut: Variable de classe.

Méthode: Fonction de classe.

self: Référence à l'objet lui-même (à usage interne à la classe)

OU ENCORE, NIVEAU SYNTAXE :

```
class MyClass:  
    def __init__(self, arg1):  
        self.attribute = arg1  
  
    def my_method(self):  
        print(self.attribute)
```

Et création d'un objet:

```
my_obj = MyClass("test")
```

LEVEL 1-10

Please, make it stop!

CLASSE ET JEU DE RÔLE, LE RETOUR

On peut faire le parallèle entre les **classes** d'un langage de programmation et celles d'un jeu de rôle, et pareil entre les **objets** et les personnages.

CLASSE ET JEU DE RÔLE, LE RETOUR

Une **classe** regroupe toutes les caractéristiques qui définissent un **type**, de la même façon que dans un RPG, la classe "mage" définit ce qu'est un mage, quels types de magie il peut utiliser et quelles sont ses caractéristiques.



CLASSE ET JEU DE RÔLE, LE RETOUR

Et de la même façon que dans un groupe de personnages de RPG vous pouvez avoir plusieurs mages étant différents les uns des autres, dans votre code, vous aurez plusieurs **objets** partageant la même **classe**.



LEVEL 1-11

One. Last. Time.

```
class Carrot:

    def __init__(self):

        self.sale_price = 5

        self.growth_max = 7

        self.growth = 0

    def grow(self):

        self.growth += 1
```

```
carrot = Carrot()
```

```
carrot.grow()
```

```
class Carrot:
```

```
    def __init__(self):
```

```
        self.sale_price = 5
```

```
        self.growth_max = 7
```

```
        self.growth = 0
```

```
    def grow(self):
```

```
        self.growth += 1
```

Classe

Une classe est un type d'objet.

Dans le code d'une classe on trouve tout ce qui définit un type, comme les méthodes et les attributs.

```
carrot = Carrot()
```

```
carrot.grow()
```

Instance
de la
classe

Une instance de la class est un objet (ou variable) du type de la classe.

L'instance possède les attributs et les méthodes de sa classe.


```
class carrot:
```

```
    def __init__(self):
```

```
        self.sale_price = 5.
```

```
        self.growth_max = 7.
```

```
        self.growth = 0.
```

```
    def grow(self):
```

```
        self.growth += 1.
```

```
carrot = Carrot()
```

```
carrot.grow()
```

Attribut
de la
classe

**Un attribut est une
variable de classe.**

Les attributs sont les
variables propres à la
classe.

```
class carrot:
```

```
    def __init__(self):
```

```
        self.sale_price = 5.
```

```
        self.growth_max = 7.
```

```
        self.growth = 0.
```

```
    def grow(self):
```

```
        self.growth += 1.
```

Méthode de
la classe

```
carrot = Carrot()
```

```
carrot.grow()
```

Appel de la
méthode

**Une méthode est une
fonction de classe.**

La méthode est une
fonction propre à la
classe n'agissant que sur
la classe et ses attributs

**Une instance de classe peut
faire appel aux méthodes de
sa classe.**

```
class carrot:
```

```
    def __init__(self):
```

```
        self.sale_price = 5.
```

```
        self.growth_max = 7.
```

```
        self.growth = 0.
```

```
    def grow(self):
```

```
        self.growth += 1.
```

```
carrot = Carrot()
```

```
carrot.grow()
```

Constructeur
de la classe

**Le constructeur est la
méthode qui crée la classe**

Cette méthode est appelée
lors de la l'instanciation
de la classe.

Instanciation de
la classe

```
class carrot:
```

```
    def __init__(self):
```

```
        self.sale_price = 5.
```

```
        self.growth_max = 7.
```

```
        self.growth = 0.
```

```
    def grow(self):
```

```
        self.growth += 1.
```

“self” le
paramètre
fantôme

**self est le paramètre
obligatoire qui contient
l'instance de la classe**

Ce paramètre fait le lien
entre la méthode et le
reste de la classe.

```
carrot = Carrot()
```

```
carrot.grow()
```

LEVEL 1-12

Fighting Vegetables II - Electric Boogaloo
End of the line

LÉGUMES BAGARREURS

Vous allez reprogrammer un combat de légumes (si, si, on y tiens VACHEMENT à notre thème...)

Chaque légume fera l'objet d'une classe différente:

Nous avons :

- la tomate tueuse
- le brocoli cogneur
- la carotte castagneuse

LÉGUMES BAGARREURS

Les légumes se battent sur un ring. Celui-ci sera aussi représenté par une classe.

On va y aller étape par étape, afin d'être sûrs de ne rien louper.

LÉGUMES BAGARREURS

La tomate tueuse:

Elle a:

- 10 points de vie
- 1 point de défense



Elle peut attaquer un adversaire, lui retirant entre 4 et 6 points de vie moins la défense de son opposant (minimum 0).

Faisons ensemble le code de la classe Tomate Tueuse ou KillerTomato...


```
class KillerTomato:
```

```
    def __init__(self):
```

```
        self.hp = 10
```

```
        self.hp_max = 10
```

```
        self.defense = 1
```

Pour commencer, écrivons
la classe et son
constructeur.

La Tomate tueuse:

Elle a:

- 10 points de vie
- 1 point de défense

```
class KillerTomato:

    def __init__(self):

        self.hp = 10

        self.hp_max = 10

        self.defense = 1
```

```
killer_tomato = KillerTomato()

print (killer_tomato.hp)
```

A ce stade on peut
afficher les points de vie
de la tomate tueuse pour
constater qu'elle est bien
en vie!

```
from random import randint
```

```
class KillerTomato:
```

```
    def __init__(self):
```

```
        self.hp = 10
```

```
        self.hp_max = 10
```

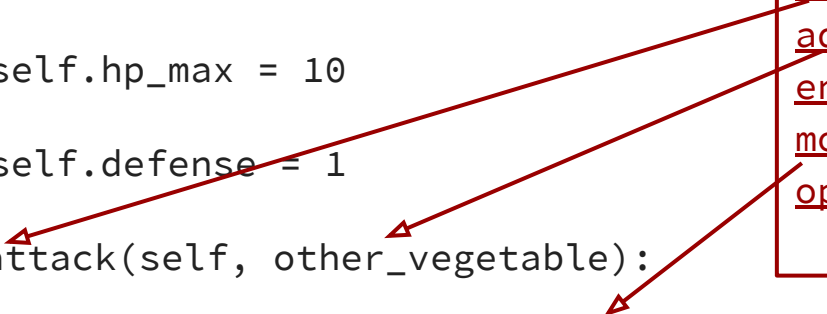
```
        self.defense = 1
```

```
    def attack(self, other_vegetable):
```

```
        damage = max(0, randint( 4, 6) - other_vegetable.defense)
```

```
        other_vegetable.hp -= damage
```

Elle peut attaquer un
adversaire lui retirant
entre 4 et 6 points de vie
moins la défense de son
opposant (minimum 0).



...

```
tomato_1 = KillerTomato()
```

```
tomato_2 = KillerTomato()
```

```
tomato_1.attack(tomato_2)
```

```
print(tomato_2.hp)
```

Si on fait attaquer une
tomate tueuse avec une
congénère, on constate que
les points de vie de la
tomate victime ont baissé.

VOILÀ LE CODE DE LA CLASSE DE LA TOMATE TUEUSE

```
from random import randint
```

```
class KillerTomato:
```

```
    def __init__(self):
```

```
        self.hp = 10
```

```
        self.hp_max = 10
```

```
        self.defense = 1
```

```
    def attack(self, other_vegetable):
```

```
        damage = max(0, randint( 4, 6) - other_vegatable.defense)
```

```
        other_vegetable.hp -= damage
```



LÉGUMES BAGARREURS

Le brocoli cogneur:

Il a:

- 10 points de vie
- 2 points de défense



Il peut attaquer un adversaire, lui retirant entre 1 et 4 points de vie moins la défense de son opposant (minimum 0).

Il retire deux fois plus de points de vie si il est en dessous de la moitié de ses points de vie originaux.

Je vous laisse faire le code du brocoli cogneur ou PuncherBroccoli

```
class PuncherBroccoli:
```

```
    def __init__(self):
```

```
        self.hp = 10
```

```
        self.hp_max = 10
```

```
        self.defense = 2
```

```
    def attacks(self, other_vegetable):
```

```
        factor = 1
```

```
        if self.hp < self.hp_max/2:
```

```
            factor = 2
```

```
        damage = max(0, randint(1, 4)) * factor
```

```
        damage -= other_vegetable.defense
```

```
        other_vegetable.hp -= damage
```

Le brocoli cogneur:

Il a:

- 10 points de vie
- 2 point de défense

Il peut attaquer un adversaire, lui retirant entre 1 et 4 points de vie moins la défense de son opposant (minimum 0).

Il retire deux fois plus de points de vie si il est en dessous de la moitié de ses points de vie originaux.

LÉGUMES BAGARREURS



La carotte castagneuse:

elle a:

- 8 points de vie
- 1 point de défense

Elle peut attaquer un adversaire lui retirant entre 3 et 5 points de vie moins la défense de son opposant (minimum 0).

De plus, si elle fait le maximum de dégâts (5 donc), elle récupère 1 point de vie (en ne dépassant pas 8).

Je vous laisse faire le code du carotte castagneuse ou BrawlerCarrot


```
class BrawlerCarrot:
```

```
    def __init__(self):
```

```
        self.hp = 8
```

```
        self.hp_max = 8
```

```
        self.defense = 1
```

```
    def attacks(self, other_vegetable):
```

```
        damage = randint(3, 5)
```

```
        if damage == 5 and self.hp < self.hp_max:
```

```
            self.hp += 1
```

```
        damage -= other_vegetable.defense
```

```
        damage = max(0, damage)
```

```
        other_vegetable.hp -= damage
```

La carotte castagneuse:

elle a:

- 8 points de vie
- 1 point de défense

Elle peut attaquer un adversaire lui retirant entre 3 et 5 points de vie moins la défense de son opposant (minimum 0).

De plus, si elle fait le maximum de dégâts (5 donc), elle récupère 1 point de vie (en ne dépassant pas 8).

LÉGUMES BAGARREURS

Normalement à ce stade
vous avez les classes des
trois légumes.



Il ne manque plus que le ring pour qu'ils se battent.

LÉGUMES BAGARREURS

Le ring permet à deux légumes de se battre.

Le combat ne concerne que deux légumes.

Chaque round, le premier légume attaque le second et vice-versa.

A la fin de chaque round, on affiche les points de vie de chaque combattant.

A la fin d'un round, le ring donne le gagnant si il y en a un. Le gagnant étant le premier qui met l'autre à 0 point de vie.

Voici à quoi devrait représenter la structure de la classe.

```
class Ring:
```

```
    def __init__(self, veg_1, veg_2):
```

```
        pass
```

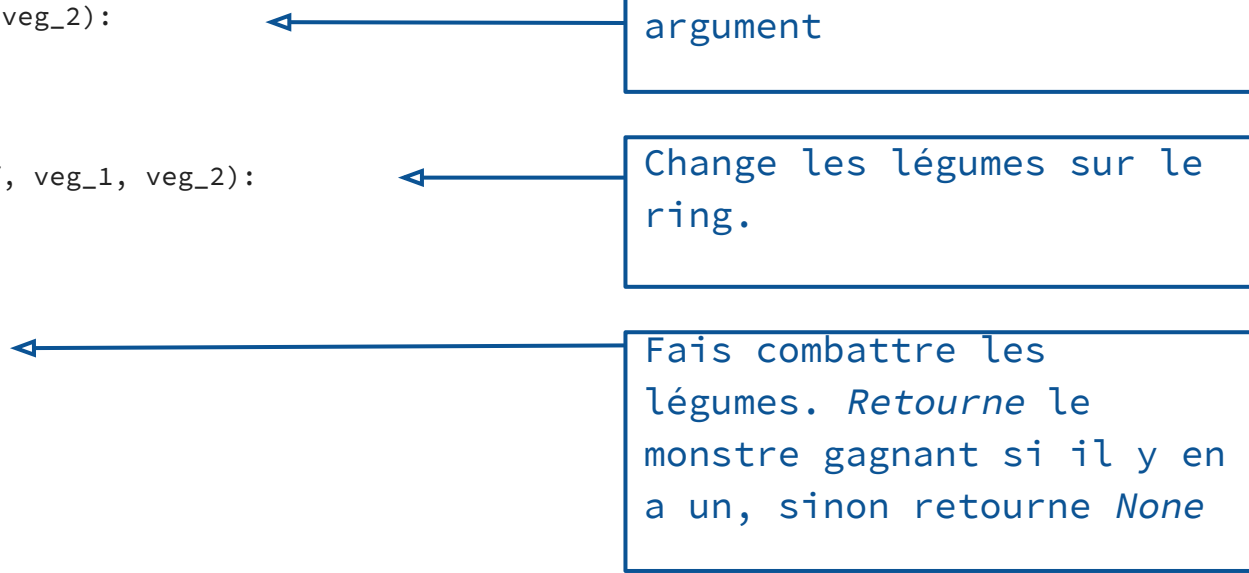
```
    def change_vegetables(self, veg_1, veg_2):
```

```
        pass
```

```
    def fight_round(self):
```

```
        pass
```

Constructeur.
Il prend 2 légumes en
argument



```
graph LR
    subgraph Methods
        direction TB
        M1["def __init__(self, veg_1, veg_2):  
    pass"]
        M2["def change_vegetables(self, veg_1, veg_2):  
    pass"]
        M3["def fight_round(self):  
    pass"]
    end
    D1["Constructeur.  
Il prend 2 légumes en  
argument"]
    D2["Change les légumes sur le  
ring."]
    D3["Fais combattre les  
légumes. Retourne le  
monstre gagnant si il y en  
a un, sinon retourne None"]
    D1 --> M1
    D2 --> M2
    D3 --> M3
```

Change les légumes sur le
ring.

Fais combattre les
légumes. *Retourne* le
monstre gagnant si il y en
a un, sinon retourne *None*

```
class Ring:
```

```
    def __init__(self, veg_1, veg_2):
```

```
        self.veg_1 = veg_1
```

```
        self.veg_2 = veg_2
```

```
    def change_vegetables(self, veg_1, veg_2):
```

```
        self.veg_1 = veg_1
```

```
        self.veg_2 = veg_2
```

Constructeur.
Il prend 2 légumes en arguments

A blue arrow points from the right side of the text box to the right side of the `def __init__` line in the code.

Change les légumes sur le ring.

A blue arrow points from the right side of the text box to the right side of the `def change_vegetables` line in the code.

```
class Ring:
```

```
...
```

```
def fight_round(self):
```

```
    self.veg_1.attack(self.veg_2)
```

```
    if self.veg_2.hp <= 0:
```


```
        return self.veg_1
```

```
    self.veg_2.attacks(self.veg_1)
```

```
    if self.veg_1.hp <= 0:
```

```
        return self.veg_2
```

```
    return None
```



Fais combattre les légumes. *Retourne* le légume gagnant si il y en a un, sinon retourne *None*

LÉGUMES BAGARREURS

Il ne reste plus qu'à faire le script qui fait combattre les légumes entre eux.

Créez un légume de chaque type.

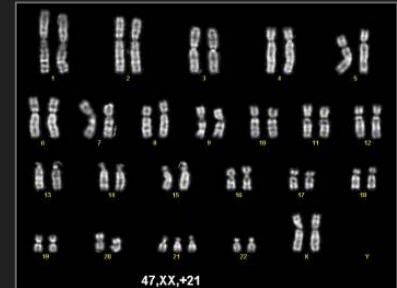
Faites deux combats par paire de légumes (un match aller et match retour) et comptez les points. N'oubliez pas de remettre les points de vie aux légumes avant de les refaire combattre.

Le légume qui a gagné le plus de matchs a gagné.

A la fin, affichez le gagnant.

LEVEL 2-1

L'héritage,
La génétique 2.0



L'HÉRITAGE :

Une classe peut récupérer le comportement d'une autre.

C'est ce qu'on appelle l'héritage.

Sous ce comportement d'apparence simple se cache l'une des bases du langage orienté objet.

ET À QUOI ÇA SERT:

Il arrive souvent que deux classes aient des comportements similaire... comme si elles étaient soeurs.

Comme par nature le développeur est fainéant et n'aime pas répéter plusieurs fois la même choses, il a créé un mécanisme pour faire en sorte de ne coder qu'une seule fois les comportements génériques entre deux ou plusieurs classes pour qu'il ne lui reste plus qu'à écrire le spécifique.

LEVEL 2-2

Telle mère, telles filles

MISE EN CONTEXTE :

Imaginons que l'on doivent coder deux classes.

La première représente des Vélos, “Bike”:

- Elle possède un attribut “speed” qui est à 15
- Elle possède un attribut “distance” qui est à 0 de base
- Elle a une méthode “ride” qui prend une durée en paramètre et qui lorsqu’elle est appelée ajoute à distance l’attribut “distance”, la durée passée en paramètre multipliée par l’attribut “speed”.

MISE EN CONTEXTE :

```
class Bike:
    def __init__(self):
        self.speed = 15
        self.distance = 0

    def ride(self, duration):
        travel = duration * self.speed
        Self.distance += travel
```

MISE EN CONTEXTE :

La deuxième représente des Voitures, “Car”.

- Elle possède un attribut “speed” qui est à 100
- Elle possède un attribut “distance” qui est à 0 de base
- Elle a une méthode “ride” qui prend une durée en paramètre et qui lorsqu’elle est appelée ajoute à distance l’attribut “distance”, la durée passée en paramètre multipliée par l’attribut “speed”.

MISE EN CONTEXTE :

```
class Car:
    def __init__(self):
        self.speed = 100
        self.distance = 0

    def ride(self, duration):
        travel = duration * self.speed
        Self.distance += travel
```

COMME DEUX SOEURS...

Nous sommes tous d'accord pour dire que les deux classes sont super similaires.

Il serait bon de pouvoir un peu agréger tout ça... non?

Voyons comment faire.

COMME DEUX SOEURS...

D'abord il faut créer une classe avec le comportement commun:

```
class Vehicle:
    def __init__(self, speed):
        self.speed = speed
        self.distance = 0

    def ride(self, duration):
        travel = duration * self.speed
        self.distance += travel
```

Speed devient un paramètre car il varie d'un véhicule à l'autre

ENSUITE VIENT L'HÉRITAGE...

```
class Bike(Vehicle):  
    def __init__(self):  
        super().__init__(15)
```

Le code suivant est dès lors suffisant pour décrire un vélo.

Décodons un peu tout ça.

ENSUITE VIENT L'HÉRITAGE...

```
class Bike(Vehicle):  
    def __init__(self):  
        super().__init__(15)
```

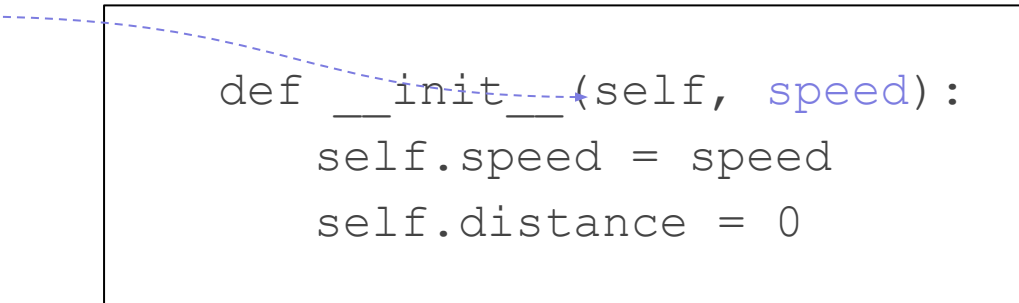
Entre parenthèses après la classe,
on déclare le parent

Le `super()` sert à exprimer qu'on
fait appelle à l'ancêtre c'est à
dire à la classe parent, dans ce cas
`Vehicle`.

Dans ce cas, on joue simplement le
constructeur de la classe parent
avec 15 comme paramètre (ce
paramètre symbolise la vitesse
souvenez-vous).

ENSUITE VIENT L'HÉRITAGE...

```
class Bike(Vehicle):  
    def __init__(self):  
        super().__init__(15)
```



```
def __init__(self, speed):  
    self.speed = speed  
    self.distance = 0
```

En gros on appelle le constructeur du parent avec le paramètre 15. Ce qui aura pour effet de créer l'attribut "speed" avec 15 dedans et l'attribut distance avec la valeur 0.

UN EXERCICE QUI VA ROULER...

Créer la classe “Car” avec l’héritage.

Une fois que vous avez fait les trois classes (Vehicle, Car, Bike) instancier un vélo et une voiture et faite les rouler tout les deux pendant 2 heures.

Puis afficher le résultat.

LEVEL 2-3

Redéfinissons tout ça

COMPLÉTONS NOTRE VOITURE

Changeons un peu notre voiture.

Ajoutons lui deux attributs:

- Un qui contient la réserve d'essence et qui est commence à 100. Appelons la “fuel”
- Et un autre qui contient la consommation de la voiture. Cette variable sera initialisé à 0.05 et se nommera “consumption”

COMPLÉTONS NOTRE VOITURE

```
class Car(Vehicle):  
    def __init__(self):  
        super().__init__(100)  
        self.fuel = 100  
        self.consumption = 0.05
```

En générale le code qui diverge du comportement initial de la classe est mis après l'appel à la méthode ancêtre.

L'ajout de code après l'appel du parent modifie le comportement de la classe par rapport à son parent.

ÇA ROULE TOUJOURS?...

Faisons en sorte de prendre en compte la consommation de la voiture dans la méthode “ride”.

Pour ce faire, il faut diminuer le fuel en fonction de la distance parcourue...

ÇA ROULE TOUJOURS

```
class Car(Vehicle):  
    def __init__(self):  
        super().__init__(100)  
        self.fuel = 100  
        self.consumption = 0.05  
  
    def ride(self, duration):  
        super().ride(duration)  
        fuel_loss = self.consumption * duration * self.speed  
        self.fuel -= fuel_loss
```

Le code après l'appel
à la méthode ancêtre
Change le comportement
de la méthode.

De la même façon que pour le constructeur le code après le `super()` modifie le comportement de la méthode

ON PEUT AUSSI COMPLÈTEMENT RÉÉCRIRE...

```
class Car(Vehicle):  
    def __init__(self):  
        super().__init__(100)  
        self.fuel = 100  
        self.consumption = 0.05
```

La méthode ici est complètement réécrite, car l'ancêtre n'est pas appelé.

```
def ride(self, duration):  
    travel = duration * self.speed  
    travel_max = self.fuel / self.consumption  
    travel = min(travel, travel_max)  
    self.distance += travel  
    self.fuel -= travel * self.consumption
```

C'EST ÇÀ LA REDÉFINITION

Le fait de réécrire une méthode d'une classe est appelé redéfinition (ou overwrite en anglais).


Grâce à cela on peut faire en sorte de coder le comportement générique dans une classe parent et le code spécifique dans les enfants.

Il est évident qu'en dehors de la redéfinition on peut aussi ajouter des méthodes supplémentaires à un enfant.

ON PEUT AUSSI COMPLÈTEMENT RÉÉCRIRE...

```
class Car(Vehicle):  
    def __init__(self):  
        super().__init__(100)  
        self.fuel = 100  
        self.consumption = 0.05  
  
    def ride(self, duration):  
        ...  
  
    def fill_tank(self, fuel_volume):  
        Self.fuel += fuel_volume
```

Cette méthode n'existe
que pour l'enfant.



LEVEL 2-4

Exercice... qui ne manque pas d'en...train

ON EST SUR LES RAILS

1.

Ecrivez la classe `Train` qui hérite de `Vehicle`

Un train, à une vitesse de 150.

ON EST SUR LES RAILS

2.

Ajouter à la classe `Train` un attribut `passengers` qui est initialisé à 0.

Ajouter ensuite un variable `max_capacity` qui représente le nombre de passager maximum, faites en sorte que cette attribut soit un paramètre du constructeur.

ON EST SUR LES RAILS

3.

Ajouter à la classe `Train` une méthode `take_on_board` qui prend un entier en paramètre et qui ajoute cette entier à l'attribut `passengers`.

Attention cette attribut ne peut pas dépasser la capacité maximum du train.

ON EST SUR LES RAILS

4.

Créer un nouvelle classe `TrainInterCity` qui peut prendre maximum 100 passagers.

Et une autre classe `FreightTrain` qui ne prend que 4 passagers maximum

ON EST SUR LES RAILS

5.

La classe `FreightTrain` récupère un attribut `chargement` qui commence à 0 et une méthode `add_chargement` qui prend un entier en paramètre et qui l'ajoute à l'attribut `chargement`

ON EST SUR LES RAILS

7.

La classe `TrainInterCity` récupère un attribut `profit_by_kilometer` qui est à 2.5 et un autre nommé `profit` qui lui commence à 0.

Ensuite redéfinissez la méthode `ride` pour qu'elle calcule le profit du train.

LEVEL 2-5

Là où l'on plante sans se planter.

ON VA MODÉLISER UN JARDIN...

Donc, j'ai un jardin dans lequel je plante trois sortes de fleur:

- Des tulipes
- Des roses
- Des lilas

ATTRIBUTS

Chaque fleur possède un attribut qui indique sa croissance.

Chaque fleur possède un facteur de croissance

Chaque fleur sait si elle a été arrosée. (donc un attribut *booléen* à *True* si elle a été arrosée). Cette attribut est à *False* de base

Chaque plante a une croissance maximum égale à 10.

Chaque plante a un prix.

Codez la classe avec les attributs décrit ci-dessus.

Notez que le facteur de croissance et le prix sont des paramètres du constructeur.

MÉTHODES

Chaque plante peut grandir:

La fleur grandit comme suit:

Si elle a été arrosée ajoutez son facteur de croissance à sa croissance

Sinon

Ajoutez seulement la moitié du facteur de croissance.

Attention ne pas dépasser la croissance maximum

MÉTHODES

Chaque plante peut dire si elle est arrivée à maturité:

Cette méthode renvoie *True* si la croissance est égale à croissance maximum et *False* dans les autres cas.

ROSES

Les roses ont un facteur de croissance de 1.1 et un prix de 5 euros.

De plus à chaque fois qu'elles grandissent sans être arrosée elles perdent 0.2 euros à leur valeur.

TULIPES

Les tulipes ont un facteur de croissance de 1.2 et un prix de 6 euros.

Si les tulipes ne sont pas arrosée elle ne grandissent pas.

LILA

Les lilas ont une valeur de 3 euros et un facteur de croissance de 0.6

LE JARDIN

Il nous reste à coder le jardin.

Celui-ci à deux attributs:

- Une liste de plantes qui est vide à la base.
- Et les gains rapportés par la vente de plante.

Il possède aussi quatre méthodes:

- Une pour planter
- Une pour arroser
- Une pour couper (et vendre) les plantes matures.
- Faire pousser toutes les plantes

PLANTER

Cette méthode prend une fleur en paramètre et l'ajoute au jardin. Si il y a déjà 5 plantes dans celui-ci, aucune plante n'est ajoutée.

ARROSER

Cette méthode arrose chaque plante du jardin.

(Ce qui consiste à passer la variable qui checke si elles ont été arrosé à *True*)

COUPER

Pour chaque plante mature du jardin (rappelez-vous les plantes ont une méthodes pour dire si elles sont mature), ajouter son prix à l'attribut de gain et puis retirer la plante de la liste des plantes du jardin.

FAIRE POUSSER LES PLANTES

Pour chaque plante du jardin faites les pousser.

Une fois que la plante à pousser, repassez la variable qui gère l'arrosage à False.

SCRIPT

Faites un petit script qui instancie un jardin.

Ce qui va suivre les instructions suivantes tant que le jardin n'a pas rapporté 100€:

- Planter une plante au hasard
- Arroser
- Faire pousser toutes les plantes
- Couper les plantes

VARIANTE

Faites en sorte de ne pas arroser un jour sur 4.

LEVEL 2-6

Combat de légumes III, la vengeance du fils.

ENCORE EUX?

Donc maintenant que vous savez faire hériter une classe, tenter de refaire le combat de légumes avec l'héritage.