Séq. 3 – La Programmation Orientée Objet (POO)

Objectifs

1. Spécifier une structure de données par son interface
2. Écrire la définition d'une classe
3. Accéder aux attributs et méthodes d'une classe

Cette séquence s'appuie sur :

- https://www.lecluse.fr/nsi/NSI_T/langages/paradigmes/
- https://pixees.fr/informatiquelycee/n_site/nsi_term_paraProg_poo.html
- https://isn-icn-ljm.pagesperso-orange.fr/NSI-TLE/res/res_les_classes.pdf
- https://e-nsi.gitlab.io/pratique/N2/700-poo_train/sujet/

1 Introduction

La programmation orientée objet repose, comme son nom l'indique, sur le concept d'objet.
Un objet dans la vie de tous les jours vous connaissez, mais en informatique, c'est un nouveau concept.
Imaginez un objet, par exemple un moteur de voiture : il est évident qu'en regardant cet objet, on est frappé par sa complexité.
Imaginez que l'on enferme cet objet dans une caisse et que l'utilisateur n'ait pas besoin d'en connaître son principe de fonctionnement interne pour pouvoir l'utiliser. L'utilisateur a, à sa disposition, des boutons, des manettes et des écrans de contrôle pour faire fonctionner l'objet, ce qui rend son utilisation relativement simple.
La mise au point de l'objet (par des ingénieurs) a été très complexe, en revanche son utilisation est relativement simple.

Jusqu'à présent nous avons utilisé des objets dont le type est prédéfini : `int, float, bool, str, list, tuple, dict`
Nous avons même appris à utiliser des méthodes associées à ces types, comme *`ma_liste.append(valeur)`*.
Python, comme d'autres langages (Java, C++, ...), est un langage orienté objet. **On peut même dire que tout y est objet.**
Une variable de type `int` est en fait un objet de type `int` donc construit à partir de la classe `int`.
Pareil pour les `float` et `string`. Mais également pour les `list`, `tuple`, `dict`, etc.

2 Créer une classe et des instances

La mot-clé Python est `class`.

A faire vous même 1.

- Ecrivez votre première `class`, la plus simple possible

```
class Voiture:
    pass # signifie ne fait rien...
```

- Cette classe, vous pouvez l'instancier :

```
>>> voiture_de_leo=Voiture()
>>> voiture_de_Anna=Voiture()
```

On a une sorte de "moule" et avec on fabrique des voitures.

- Il n'est plus de type `int`, `float`, `list` ou autre, il est de type `Voiture`

```
>>> voiture_de_leo
<class'__main__.Voiture'>
```

- A noter : Pour poser les bases avant de programmer, il est possible d'utiliser le mot-clé python `pass` qui ne fait rien mais qui évite au code de se mettre en erreur.

3 Les attributs de classe

A faire vous même 2.

- Complétez votre classe

```
class Voiture:
    # attributs de classe
    matière="acier"
    nombre_de_places=5
    longueur=4.5
```

- Cette classe, vous pouvez l'instancier :

```
>>> voiture_de_Peter=Voiture()
```

- Vous avez accès à ses attributs de classe en écrivant le nom de l'instance UN POINT le nom de l'attribut :

```
>>>voiture_de_Peter.matière
"acier"
```

- Toutes les instances de `Voiture` auront ces mêmes attributs
- Affichez les 2 autres attributs de classe

4 Les attributs d'instance et le constructeur

Quand on créé une classe, on écrit une sorte de fonction spéciale (en fait c'est une méthode) appelée le constructeur.
Il est implicitement exécuté lors de la création de chaque instance. Son nom est imposé : `__init__` (2 underscores de chaque côté).
Son premier paramètre est `self`.
Ensuite, vient la liste des autres paramètres.
A noter : Il n'y a pas de `return` (pas de valeur retournée).

A faire vous même 3.

- Complétez votre classe

```
class Voiture:
    # attributs de classe
    matière="acier"
    nombre_de_places=5
    longueur=4.5
    def __init__(self, coul):
        self.couleur = coul
```

- Cette classe, vous pouvez l'instancier :

```
>>> voiture_de_Titouan=Voiture()
```

```
TypeError: __init__() missing 1 required positional argument:
'couleur_utilisateur'
```

Il y a erreur, le constructeur a besoin absolument d'un paramètre.

```
>>> voiture_de_Titouan=Voiture("bleu")
```

- Vous avez accès à ses attributs d'instance :

```
>>> voiture_de_Titouan.couleur
"bleu"
```

- Chaque instance de Voiture peut avoir ses propres attributs d'instance

```
>>> voiture_de_Jackie=Voiture("rouge")
>>> voiture_de_Jackie .couleur
"rouge"
```

5 Les méthodes

Un objet est (rouge, bleu, 15 cm de haut, …) mais aussi FAIT/AGIT/ACTIONNE (démarre, roule, s'arrête, modifie, ...).

A faire vous même 4.

- Complétez votre classe

```
class Voiture:
    # attributs de classe
    matière="acier"
    nombre_de_places=5
    longueur=4.5

    def __init__(self, couleur_utilisateur, nombre_de_km=0,
a_charger=[ ]):
        self.couleur = couleur_utilisateur
        self.km=nombre_de_km
        self.coffre= a_charger[ :]

    def avance(self, distance) :
        self.km+=distance

    def charge(self, a_charger=[ ]) :
        self.coffre+= a_charger
```

- Cette classe, vous pouvez l' instancier et la manipuler :

```
>>> voiture_de_Ines=Voiture('verte')
>>> voiture_de_Alex=Voiture('noir', a_charger=['valise', 'caisse en
carton', 'livres'])
>>> voiture_de_Alex.km
0
>>> voiture_de_Alex.avance(77)
>>> voiture_de_Alex.km
77
>>> voiture_de_Alex.coffre
['valise', 'caisse en carton', 'livres']
>>> voiture_de_Alex.charge(['parapluie', 'poussette'])
>>> voiture_de_Alex.coffre
['valise', 'caisse en carton', 'livres', 'parapluie', 'poussette']
```

A faire vous même 5.

- Ajoutez une méthode `estNeuve` qui retourne `True` si le kilométrage est égal à 0 ou sinon `False`.
- Ajoutez une méthode `videLeCoffre` qui vide la liste correspondant au coffre.
- Il existe des méthodes cachées qui sont disponibles automatiquement pour toutes les classes. Par exemple : `__repr__` et `__str__` . Elles sont toutes précédées et suivies de double underscores
- Ajoutez une méthode `__repr__` Cherchez comment fonctionne cette méthode et écrivez-la.

6 Encore et toujours la documentation

Pour manipuler des objets, une doc. bien rédigée est vraiment nécessaire.

A faire vous même 6.

```
class Voiture:
    """
    Permet de modéliser une voiture
    """
    matière="acier"
    nombre_de_places=5
    longueur=4.5

    def __init__(self, couleur_utilisateur, nombre_de_km=0,
                 a_charger=[ ]):
    """
    Constructeur
    Un arg. pos. - chaine de caractères pour la couleur de la voiture
    Deux arguments nommés :
    * nombre_de_km : Entier pour le nombre de km du véhicule
    * a_charger : Liste pour tous les objets à mettre dans le coffre
    """
        self.couleur = couleur_utilisateur
        self.km=nombre_de_km
        self.coffre= a_charger[ :]
```

- Complétez les autres méthodes de votre classe avec de la documentation
- Ajoutez-y quelques jeux de tests. A vous de voir s'il faut le mettre au niveau de la classe ou au niveau des méthodes.

7 Un autre exemple de POO : La classe Train

On souhaite dans cet exercice créer une classe `Train` permettant de relier des objets de type `Wagon`.

7.1 Objet Wagon

Un objet de type `Wagon` possède deux attributs :

- un contenu `contenu` de type `str`,
- un lien vers le wagon suivant `suivant` de type `Wagon`.

On inclut aussi deux méthodes permettant d'afficher le wagon dans la console ou sous forme d'une chaîne de caractère.

```
class Wagon:
    def __init__(self, contenu):
        "Constructeur"
        self.contenu = contenu
        self.suivant = None

    def __repr__(self):
        "Affichage dans la console"
        return f'Wagon de {self.contenu}'

    def __str__(self):
        "Conversion en string"
        return self.__repr__()
```

A faire vous même 7.

- Recopiez cette classe dans un nouveau script
- Instanciez cette classe pour tester

7.2 Objet Train

Un objet de la classe Train possède deux attributs :

- `premier` contient son premier `wagon` (de type `Wagon`) ou `None` si le train est vide (il n'y a que la locomotive),
- `nb_wagons` (de type `int`) contient le nombre de wagons attachés à la locomotive.

Lors de sa création, un objet de type `Train` sera toujours vide.

Les méthodes de la classe Train sont présentées ci-dessous (train est un objet de type Train) :

- `train.est_vide()` renvoie `True` si `train` est vide (ne comporte aucun wagon), `False` sinon ;
- `train.donne_nb_wagons()` renvoie le nombre de wagons de train ;
- `train.transporte_du(contenu)` détermine si `train` transporte du contenu (une chaîne de caractères). Renvoie `True` si c'est le cas, `False` sinon ;
- `train.ajoute_wagon(wagon)` ajoute un wagon à la fin du train. On passe en argument le wagon à ajouter ;
- `train.supprime_wagon_de(contenu)` prend en argument une chaîne de caractères contenu et supprime le premier wagon de contenu du `train`. Si le train est vide ou ne comporte aucun wagon de contenu, la méthode renvoie `False`. S'il en contient un et que celui-ci est effectivement supprimé, la méthode renvoie `True`.

On inclut là-aussi aussi deux méthodes permettant d'afficher le train dans la console ou sous forme d'une chaîne de caractères.

Voici ce qui devrait fonctionner dans une console python :

- Création d'un train vide :

```
>>> train = Train()
```

- Ajout de wagons :

```
>>> w1 = Wagon('blé')
>>> train.ajoute_wagon(w1)
>>> w2 = Wagon('riz')
>>> train.ajoute_wagon(w2)
>>> train.ajoute_wagon(Wagon('sable'))
>>> train
'Locomotive – Wagon de blé – Wagon de riz – Wagon de sable'
```

- Description du train :

```
>>> train.est_vide()
False
>>> train.donne_nb_wagons()
3
>>> train.transporte_du('blé')
True
>>> train.transporte_du('matériel')
False
```

- Suppression de wagon

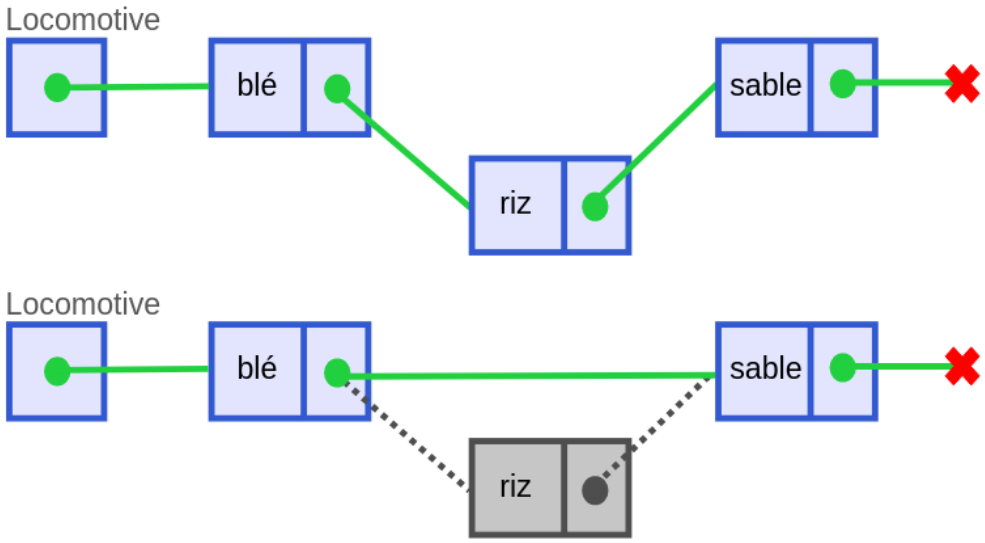
```
>>> train.supprime_wagon_de('riz')
True
>>> train
'Locomotive – Wagon de blé – Wagon de sable'
>>> train.supprime_wagon_de('riz')
False
```

On pourra parcourir tous les wagons du train en utilisant les instructions ci-dessous :

```
wagon = self.premier
while wagon is not None:
    # actions à effectuer
    wagon = wagon.suivant
```

En plusieurs occasions il faudra prendre soin de traiter séparément le cas du premier wagon et celui des suivants.

Enfin, lors de la suppression d'un wagon, on se contentera de l'omettre en liant son wagon précédent à son suivant. La figure ci-dessous illustre ainsi l'instruction `train.supprime_wagon_de('riz')` avant et après la suppression.



A faire vous même 8.

Recopiez et complétez le code ci-après afin que tout ce qui est décrit précédemment puisse fonctionner.

```
class Train:
    def __init__(self):
        "Constructeur"
        self.premier = None
        self.nb_wagons = ...

    def est_vide(self):
        """renvoie True si ce train est vide (ne comporte aucun wagon),
        False sinon
        """
        return ...

    def donne_nb_wagons(self):
        "Renvoie le nombre de wagons de ce train"
        return ...

    def transporte_du(self, contenu):
        """Détermine si ce train transporte du {contenu} (une chaine de caractères).
        Renvoie True si c'est le cas, False sinon
        """
        wagon = self.premier
        while wagon is not None:
            if wagon.contenu == ...:
                return ...
            ... = wagon....
        return ...

    def ajoute_wagon(self, nouveau):
        """Ajoute un wagon à la fin de ce train.
        L'argument est le wagon à ajouter
        """
        if self.est_vide():
            self.premier = ...
        else:
            wagon = self.premier
            while ....suivant is not None:
                wagon = ....suivant
            wagon.suivant = ...
        self.nb_wagons = ...

    def supprime_wagon_de(self, contenu):
        """Supprime le premier wagon de {contenu}
        Renvoie False si ce train ne contient pas de {contenu},
        True si la suppression est effectuée
```

```
        """
        if self.est_vide():
            return ...

        if self.premier.contenu == contenu:
            self.premier = self.premier....
        else:
            precedent = self.premier
            wagon = precedent.suivant
            while wagon.contenu != ...:
                precedent = ...
                wagon = wagon....
                if wagon is None: # pas de "contenu" dans le train
                    ...
            precedent.suivant = wagon.suivant

        # MAJ du nombre de wagons et résultat de la fonction
        self.nb_wagons = ...
        return ...

    def __repr__(self):
        "Affichage dans la console"
        contenus_wagons = ['']
        wagon = self.premier
        while wagon is not None:
            contenus_wagons.append(str(wagon))
            wagon = wagon.suivant
        return "Locomotive" + " - ".join(contenus_wagons)

    def __str__(self):
        "Conversion en string"
        return self.__repr__()


# Tests
train = Train()
w1 = Wagon("blé")
train.ajoute_wagon(w1)
w2 = Wagon("riz")
train.ajoute_wagon(w2)
train.ajoute_wagon(Wagon("sable"))
assert str(train) == 'Locomotive - Wagon de blé - Wagon de riz - Wagon de sable'
assert not train.est_vide()
assert train.donne_nb_wagons() == 3
assert train.transporte_du('blé')
assert not train.transporte_du('matériel')
assert train.supprime_wagon_de('riz')
assert str(train) == 'Locomotive - Wagon de blé - Wagon de sable'
assert not train.supprime_wagon_de('riz')
```

P. 70-73 – Lire le cours

P. 74 ex 1

8 Mini-projet individuel : Une classe Vecteur

Il va falloir vous rappeler ce que vous avez appris sur les vecteurs et leur manipulation. Si besoin, demandez au professeur des précisions sans handicap pour la note.

1. Créez une classe `Vecteur2D` pour un vecteur dans le plan
2. Créez un constructeur avec ses composantes en x et en y
3. Créez une méthode `Norme` qui retourne la norme du vecteur
4. Créez une méthode `Multiplie` qui modifie le vecteur en le multipliant pour un nombre donné en argument
5. Créez une méthode `Ajoute` qui modifie le vecteur en lui ajoutant un autre objet Vecteur donné en argument
6. Créez une méthode `ProduitScalaire` qui calcule le produit scalaire du vecteur avec un autre objet `Vecteur` donné en argument
7. Créez une méthode `EstPerpendiculaire` qui retourne `True` si un autre objet `Vecteur` donné en argument est perpendiculaire et sinon `False`
8. Créez une méthode `EstColinéaire` qui retourne `True` si un autre objet `Vecteur` donné en argument est colinéaire et sinon `False`
9. Modifiez la méthode `__str__` qui permette d'avoir une sortie qui ressemble à ceci :
```
>>> print(mon_vecteur)
vecteur de coordonnées : x=4 et y=5.6
```
10. Etudiez les méthodes `__mul__` et `__add__` et faites-les fonctionner pour les vecteurs
11. Mettez au point la documentation et les jeux de tests

Term NSI **GRILLE D'EVALUATION :** **Noms :** **Note :** **/10**	Rien d'écrit	Ne fonctionne pas	Fonctionne avec aide	Fonctionne sans aide	Fonctionne de manière optimale avec documentation et tests
Structure globale et constructeur (2 points)	I				
Méthode Norme (1 point)					
Méthode Multiplie (1 point)					
Méthode Ajoute (2 points)					
Méthode EstPerpendiculaire (1 point)					
Méthode EstColineaire (1 point)					
Méthode __str__ (1 point)					
Méthodes __mul__ et __add__ (1 point)					

9 Mini-projet de groupe : Le jeu de la vie

Voir vidéo : https://www.youtube.com/watch?v=S-W0NX97DB0

Le jeu de la vie a été inventé par le mathématicien britannique John H. Conway (1937-2020). C'est un exemple de ce qu'on appelle un **automate cellulaire**. Il se déroule sur un tableau rectangulaire $(L \times H)$ de cellules. Une cellule est représentée par ses coordonnées x et y qui vérifient $0 \leqslant x < L$ et $0 \leqslant y < H$.

Une cellule peut être dans deux états : **vivante** ou **morte**. La dynamique du jeu s'exprime par les règles de transition suivantes :
– une cellule vivante reste vivante si elle est entourée de 2 ou 3 voisines vivantes et meurt sinon ;
– une cellule morte devient vivante si elle possède exactement 3 voisines vivantes.

La notion de « voisinage » dans le jeu de la vie est celle des 8 cases qui peuvent entourer une case donnée (on parle de voisinage de Moore). Pour implémenter la simulation, on va tout d'abord donner une modélisation objet du problème, puis procéder à son implémentation.

Partie A Modélisation objet

1. Quelles classes peut-on dégager de ce problème au premier abord ?

2. Quelles sont quelques-unes des méthodes qu'on pourrait leur donner ?

3. Dans quelle classe pouvons-nous représenter simplement la notion de voisinage d'une cellule ? Et le calculer ?

4. Une cellule est au bord si $x = 0$, $x = L - 1$, $y = 0$ ou $y = H - 1$. Combien de voisins possède une cellule qui n'est pas au bord ?
Combien de voisins possède une cellule qui est au bord ?

5. Que pourrions-nous aussi considérer comme voisin de droite de la case en haut à droite de la grille ? Et comme voisin du haut ?

Partie B Implémentation des cellules

1. a. Implémenter tout d'abord une classe `Cellule` avec comme attributs :
- un booléen `actuel` initialisé à `False` ;
- un booléen `futur` initialisé à `False` ;
- une liste `voisins` initialisée à `None`.

CONSEIL

Ces attributs seront ici considérés comme « privés ». La valeur `False` signifie que la cellule est morte et `True` qu'elle est vivante.

b. Ajouter les méthodes suivantes :

- `est_vivant()` qui renvoie l'état actuel (vrai ou faux) ;
- `set_voisins()` qui permet d'affecter comme voisins la liste passée en paramètre ;
- `get_voisins()` qui renvoie la liste des voisins de la cellule ;
- `naitre()` qui met l'état futur de la cellule à `True` ;
- `mourir()` qui permet l'opération inverse ;
- `basculer()` qui fait passer l'état futur de la cellule dans l'état actuel.

c. Ajouter à la classe `Cellule` une méthode `__str__()` qui affiche une croix (un `X`) si la cellule est vivante et un tiret (`-`) sinon. Expliquer brièvement l'utilité d'une telle méthode `__str__()` en Python.

d. Ajouter une méthode `calcule_etat_futur()` dans la classe `Cellule` qui permet d'implémenter les règles d'évolution du jeu de la vie en préparant l'état futur à sa nouvelle valeur.

2. Implémenter la classe `Grille`.

a. Créer la classe `Grille` et y placer les attributs suivants considérés comme « publics » :

- `largeur` ;
- `hauteur` ;
- `matrix` : un tableau de cellules à 2 dimensions (implémenté en Python par une liste de listes).

Fournir une méthode `__init__` permettant l'initialisation d'une `Grille` de `Cellules` avec une `largeur` et une `hauteur` (une nouvelle `Cellule` sera créée par l'appel `Cellule()`).

b. Ajouter les méthodes :

- `dans_grille()` qui indique si un point de coordonnées i et j est bien dans la grille ;
- `setXY()` qui permet d'affecter une nouvelle valeur à la case (i, j) de la grille ;
- `getXY()` qui permet de récupérer la cellule située dans la case (i, j) de la grille ;
- get_largeur() qui permet de récupérer la largeur de la grille ;
- get_hauteur() qui permet de récupérer la hauteur de la grille ;
- `est_voisin()`, une méthode statique qui vérifie si les cases (i, j) et (x, y) sont voisines dans la grille.

c. Ajouter une méthode `get_voisins()` qui renvoie la liste des voisins d'une cellule.

d. Fournir une méthode `affecte_voisins()` qui affecte à chaque cellule de la grille la liste de ses voisins.

e. Donner une méthode `__str__()` qui permet d'afficher la grille sur un terminal.

f. On veut remplir aléatoirement la `Grille` avec un certain taux de `Cellule` vivantes. Fournir à cet effet, une méthode `remplir_alea()` avec le taux (en pourcentage) en paramètre.

3. On joue à présent !

a. Concevoir une méthode `jeu()` permettant de passer en revue toutes les `Cellules` de la `Grille`, de calculer leur état futur, puis une méthode `actualise()` qui bascule toutes les cellules de la `Grille` dans leur état futur.

b. Programme principal : définir enfin un `main` pour terminer l'implémentation du jeu de la vie avec un affichage en console en utilisant les méthodes précédentes. On donne la méthode suivante qui permet d'effacer l'écran dans un terminal `ANSI` :

```
def effacer_ecran():
    print("\u001B[H\u001B[J")
```

AIDES :

Partie A Modélisation objet

1. On peut proposer une classe `Cellule` représentant l'état d'une cellule et son évolution, et une classe `Grille` représentant le plateau de jeu.

2. Dans la classe `Cellule`, il faut pouvoir faire naître ou mourir la cellule, connaître son état actuel et futur et l'afficher sous la forme d'un caractère. Dans `Grille`, il faut pouvoir connaître sa largeur et sa hauteur, récupérer et réaffecter le contenu (la cellule) qui se trouve dans une position (i, j).

3. La notion de voisinage se calcule bien dans la grille mais une cellule doit aussi connaître ses voisins (ou leur nombre) pour calculer son état futur.

4. Une cellule qui n'est pas au bord admet toutes les cases qui l'entourent comme voisins. Il suffit de les compter. Distinguez les cellules dans les angles des autres cellules du bord.

5. Imaginez une grille qui se replie sur elle-même.

Partie B Implémentation des cellules

1. a. et **b.** Les attributs sont donnés et les méthodes sont pour la plupart traitables en 1 seule ligne. Ne pas oublier de mettre `self` en premier argument des méthodes !

c. Simple test : on renvoie le caractère demandé selon l'état de la cellule.

d. On implémente les règles d'évolution du jeu de la vie dans cette méthode en mettant seulement à jour l'attribut `futur`. Si possible ne pas utiliser d'accès aux attributs privés de la classe.

2. a. Implémentation de la grille : la méthode `__init__()` permet de mettre en place les attributs `largeur`, `hauteur` et `matrix` et de les initialiser. Utilisez une compréhension pour initialiser aisément `matrix` qui est un tableau de cellules à deux dimensions.

b. Pour `est_voisin()` qui prend en entrée deux cases de la grille, on souhaite exprimer que ces deux cases diffèrent d'au plus une unité en abscisse ou en ordonnée. Pensez au décorateur `@staticmethod`.

c. On passe en revue toutes les cases contiguës (8 voisins potentiels) de la case (x, y) de la grille et, si elles sont bien dans la grille, on les accumule dans une liste. On renvoie la liste obtenue.

d. Parcourez les cellules de la grille et affectez leurs voisins avec le résultat fourni par la méthode `get_voisins` de `Grille`.

e. Affichez ligne par ligne les cellules de la grille.

f. Utilisez la méthode `random()` qui renvoie un nombre aléatoire entre 0 et 1 puis faites naître et basculer les cellules tirées au sort.

3. a. On passe en revue toutes les cellules de la grille et on fait ce qui est demandé dans les deux cas.

b. Pensez à faire dans l'ordre : l'instanciation, le remplissage aléatoire, le calcul des voisinages, puis jouez en marquant une pause grâce à la méthode `sleep()` du module `time`.

Term NSI **GRILLE D'EVALUATION :** **Noms :** **Note :** **/12**	Rien d'écrit	Ne fonctionne pas	Fonctionne imparfaitement	Fonctionne avec beaucoup d'aide	Fonctionne avec aide	Fonctionne sans aide	Fonctionne de manière optimale et avec les commentaires
Architecture générale du script							
Objet Cellule							
Objet Grille							
Organisation du groupe							
Présentation de 5 min.							