**PYTHON\_ POO**

<https://openclassrooms.com/fr/courses/4302126-decouvrez-la-programmation-orientee-objet-avec-python>

**Partie 1**

Python est un des langages les + utilisés dans l'univers scientifique et + spécifiquement dans la manipulation de données

**1\_ PROJET DU COURS**

Données : réseau social factice de x100.000 de personnes, pour peupler notre monde <http://pplapi.com/>

* Une personne définie par :
  + âge, date de naissance, lieu de un lieu de résidence, une nationalité, une langue...
  + 5 traits de personnalité : Ouverture |Conscienciosité | Extraversion |Agréabilité | Neuroticisme

(utilisés dans un des modèles les plus connus en psychologie : le modèle des [Big Five](https://fr.wikipedia.org/wiki/Mod%C3%A8le_des_Big_Five_(psychologie)))

* **chaque trait a une valeur** : 2=très supérieur | 1=assez supérieur | 0=moyenne | -1=assez inf | -2=très inf
* **données au format JSON**

|  |
| --- |
| {"neuroticism": 0.9011914234209494, "language": "English", "latitude": -26.25297800501783, "country\_tld": "au", "age": 34, "income": 59956, "longitude": 132.6457795091038, "sex": "Male", "religion": "Catholic", "extraversion": -0.8392218706596577, "date\_of\_birth": "1983-10-25", "agreeableness": 3.1053208332787023, "id\_str": "ilV-COu", "conscientiousness": 1.737208020264794, "internet": true, "country\_name": "Australia", "openness": -0.20570297449617833, "id": 6352373083} |

Programme : 1-ouvre JPON(100000agents) | 2-calculs | 3-agréabilitéVSdensité population | 4-revenuVSage

Étapes (sommaire) :

* Découverte de la Programmation Orientée Objet
* Dessin d'un diagramme de classe
* Transformation des agents JSON en agents que nous pourrons réutiliser.
* Agents : ajout des attributs (chaque agent a plusieurs "propriétés" : agréabilité, revenu, ...)
* Création de la grille qui nous permettra de situer les personnages dans le monde.
* Création des zones composant cette grille.
* Ajout du premier habitant dans sa zone. Et le monde fut !
* Peuplement du monde avec les 100 000 agents.
* Création du graphique Agréabilité versus densité de population.
* Création du graphique Revenu versus âge

**2\_ PRINCIPES DE LA PROGRAMMATION ORIENTEE OBJET**

alternative : non pas comme un ensemble de fonctions mais en tant qu'ensemble d'objets qui interagissent

définition :

* + **objet** = entité qui sert de conteneur à des données et qui contrôle l'accès à ces dernières.
  + **c**haque objet a un ensemble **d'attributs** (variables spécifiques) et de **méthodes** (fonctions propres)

AOO (analyse orientée objet) : 1/ analyse du problème en termes d’objets

#### DOO (Design Orienté Objet) : 2/ définition des actions et interactions pour chaque objet

🡪 **diagramme de séquences** pour matérialiser les interactions.

Source : [*http://agilemodeling.com/images/models/sequenceDiagramBasicCourse.jpg*](http://agilemodeling.com/images/models/sequenceDiagramBasicCourse.jpg)

#### POO (Programmation Orientée Objet) : transformer le schéma en code

**🡪diagramme de classe**. sert-à bien comprendre la structure de chaque objet.

🡪une classe a : un nom, des attributs, et des méthodes, chacun est représenté par un **rectangle**.

Source : [*http://agilemodeling.com/images/models/classDiagramInheritance.jpg*](http://agilemodeling.com/images/models/classDiagramInheritance.jpg)

**3\_ STRUCTURE DU PROGRAMME**

Objets : selon modélisation du monde envisagée

* **agent** : représenté par un pion rouge, plusieurs attributs, dont 1 position
* **position** représentée par le placement d'un pion sur le plateau.
* **zone** représentée par une zone minimale de jeu
* **graphiques** : pas dans le jeu de bataille navale mais il faut quand même y songer !

#### Attributs :

#### agent : un agent a plusieurs attributs (âge, sexe, ...) dont l'agréabilité et sa position sur Terre.

#### zone : une Zone est un rectangle (2 positions : coin inférieur gauche & coin supérieur droit.

#### position : une position (latitude, longitude) est un objet utilisé à la fois par un agent et par une zone.

#### graph : un graphique a un ensemble de données (1 abscisse (axe x) et 1 ordonnée (axe y)).

Méthodes :

* **constructeur** : une méthode qui lui sert à générer de nouveaux objets sur même modèle.
* **calculer densité de population** pour une zone doit calculer la densité de population => 3 méthodes

**1/ population**=valeur **2/ area**=périmètre+surface en km2 **3/population\_density**

Diagramme de classe : à compléter avec l’outil [GenMyModel](https://app.genmymodel.com/) par exemple

**Partie 2**

**4\_ PREMIER AGENT**

Ressources**:** télécharger[**https://github.com/OpenClassrooms-Student-Center/la\_poo\_avec\_python/tree/00\_setup**](https://github.com/OpenClassrooms-Student-Center/la_poo_avec_python/tree/00_setup)

* agents-100k.zip  : 100 000 agents provenant de PPLAPI en format JSON
* model.py  : ce fichier vide hébergera votre script

Création : chaque objet est une "classe" en programmation.

* + Définition : class Agent:
  + intanciation : first\_agent = Agent()
  + Appartenance : print(first\_agent) => *<\_\_main\_\_.Agent object at 0x10e16a160>*

Github : [**https://github.com/OpenClassrooms-Student-Center/la\_poo\_avec\_python/tree/01\_first\_class**](https://github.com/OpenClassrooms-Student-Center/la_poo_avec_python/tree/01_first_class)

**5 \_ CUSTOMISATION**

**ajouter une méthode**

* création

|  |
| --- |
| class Agent:  def say\_hello(self, first\_name):  return "Bien le bonjour " + first\_name + "!" |

* utilisation

|  |
| --- |
| agent = Agent()  print(agent.say\_hello("Céline"))  "Bien le bonjour Céline !" |

**ajouter un attribut**

* données JSON d’un agent

|  |
| --- |
| {"neuroticism": -0.0739192627121713, "language": "Shona", "latitude": -19.922097800281783, "country\_tld": "zw", "age": 12, "income": 333, "longitude": 29.798455535838603, "sex": "Male", "religion": "syncretic", "extraversion": 1.051833688742943, "date\_of\_birth": "2005-01-10", "agreeableness": 0.1441229877537559, "id\_str": "LB3-3Cl", "conscientiousness": 0.2419104411765549, "internet": false, "country\_name": "Zimbabwe", "openness": -0.024607605122172617, "id": 6636726630} |

* creation : à partir du constructeur = méthode \_\_init\_\_()
* par défaut : le constructeur crée une instance **sans attribut**.
* 2 attributs : degré d'agréabilité, position

|  |
| --- |
| class Agent:    def \_\_init\_\_(self):  self.agreeableness = 0 # self pour l’associer à l’instance |

Remarque : ces deux underscores sont **dunders**.

* Utilisation : mais ici tous la même valeur 0 !

|  |
| --- |
| first\_agent = Agent()  print(first\_agent.agreeableness) # python my\_model.py renvoie 0 |

**modifier un attribut**

* Constructeur

|  |
| --- |
| class Agent:    def \_\_init\_\_(self, agreeableness):  self.agreeableness = agreeableness |

* Initialisation

|  |
| --- |
| first\_agent = Agent(0)  print(first\_agent.agreeableness) |

**6\_ AJOUT ATTRIBUTS**

Question : comment "transformer" dynamiquement un élément JSON en instance de notre classe Agent ?

|  |
| --- |
| {"neuroticism": -0.0739192627121713, "language": "Shona", "latitude": -19.922097800281783, "country\_tld": "zw", "age": 12, "income": 333, "longitude": 29.798455535838603, "sex": "Male", "religion": "syncretic", "extraversion": 1.051833688742943, "date\_of\_birth": "2005-01-10", "agreeableness": 0.1441229877537559, "id\_str": "LB3-3Cl", "conscientiousness": 0.2419104411765549, "internet": false, "country\_name": "Zimbabwe", "openness": -0.024607605122172617, "id": 6636726630} |

Idée : dictionnaire en paramètre à l'initialisation 🡪 chaque clé se transforme en nouvel attribut

(mieux que de fait les attributs les uns après les autres)

**passer un dictionnaire en paramètre**

création :

|  |
| --- |
| class Agent:    def \_\_init\_\_(self, agent\_attributes):  self.agreeableness = agent\_attributes['agreeableness']  agent\_attributes = {"neuroticism": -0.0739192627121713, "language": "Shona", "latitude": -19.922097800281783, "country\_tld": "zw", "age": 12, "income": 333, "longitude": 29.798455535838603, "sex": "Male", "religion": "syncretic", "extraversion": 1.051833688742943, "date\_of\_birth": "2005-01-10", "agreeableness": 0.1441229877537559, "id\_str": "LB3-3Cl", "conscientiousness": 0.2419104411765549, "internet": "false", "country\_name": "Zimbabwe", "openness": -0.024607605122172617, "id": 6636726630}  first\_agent = Agent(agent\_attributes)  print(first\_agent.agreeableness) |

Rem : false  comme une chaine et non comme une variable Python

méthode items() : tous les éléments d'un dictionnaire :

|  |
| --- |
| def \_\_init\_\_(self, agent\_attributes):  print(agent\_attributes.items())  # -------  def \_\_init\_\_(self, agent\_attributes):  for attr\_name, attr\_value in agent\_attributes.items():  # set attribute |

**méthode setattr()**

|  |
| --- |
| def \_\_init\_\_(self, agent\_attributes):  for attr\_name, attr\_value in agent\_attributes.items():  setattr(self, attr\_name, attr\_value)  # méthode setattr() équivalente ‘my\_object.attribute = value’ |

**varier de nombre d’arguments :** possibilité d'avoir un **nombre variable d'arguments d**ans une fonction.

#### passer une suite d'éléments sans clé : ajoutez ****une étoile**** avant la définition du paramètre

🡪les éléments passés en paramètre sont transformés en **tuples**

|  |
| --- |
| def cities(country, \*cities):  print(country, cities)  print("Type is ", type(cities))  cities("France")  # France ()  # Type is <class 'tuple'>  cities("France", "Paris", "Mollégès", "Bourg la Reine")  # France ('Paris', 'Mollégès', 'Bourg la Reine')  # Type is <class 'tuple'> |

passer en paramètre des éléments avec une clé (**dictionnaire)** : ajouter **deux étoiles**

|  |
| --- |
| def list\_songs(\*\*songs):  print(songs)  print("Type is ", type(songs))  list\_songs()  # {}  # Type is <class 'dict'>  list\_songs(adele\_songs = ["Hello", "Someone like you"], backstreet\_boys\_songs = ["Larger than life", "I want it that way"])  # {'adele\_songs': ['Hello', 'Someone like you'], 'backstreet\_boys\_songs': ['Larger than life', 'I want it that way']}  # Type is <class 'dict'> |

**application à notre projet**

|  |
| --- |
| class Agent:    def \_\_init\_\_(self, \*\*agent\_attributes):  for attr\_name, attr\_value in agent\_attributes.items():  setattr(self, attr\_name, attr\_value)  ...  first\_agent = Agent(\*\*agent\_attributes)  print(first\_agent.agreeableness) |

**module JSON** comment récupérer les agents JSON et les transformer en agents Python

|  |
| --- |
| import json  class Agent:    def \_\_init\_\_(self, \*\*agent\_attributes):  for attr\_name, attr\_value in agent\_attributes.items():  setattr(self, attr\_name, attr\_value)  for agent\_attributes in json.load(open("agents-100k.json")):  agent = Agent(\*\*agent\_attributes)  print(agent.agreeableness) |

version2 : rem : plusieurs commandes 🡪 les regrouper en une fonction, main() lancée par le script

|  |
| --- |
| import json  class Agent:    def \_\_init\_\_(self, \*\*agent\_attributes):  for attr\_name, attr\_value in agent\_attributes.items():  setattr(self, attr\_name, attr\_value)  def main():  for agent\_attributes in json.load(open("agents-100k.json")):  agent = Agent(\*\*agent\_attributes)  print(agent.agreeableness)    main() |

Github : <https://github.com/OpenClassrooms-Student-Center/la_poo_avec_python/tree/02_class_attributes>

**7\_ CLASSE POSITION**

Constructeur :

|  |
| --- |
| class Position:  def \_\_init\_\_(self, longitude, latitude):  self.latitude = latitude  self.longitude = longitude  # nouvelle instance Position avec latitude & longitude de chaque agent  def main():  for agent\_attributes in json.load(open("agents-100k.json")):  latitude = agent\_attributes.pop("latitude")  longitude = agent\_attributes.pop("longitude")  position = Position(longitude, latitude)  agent = Agent(position, \*\*agent\_attributes)  # MAJ de la classe Agent pour associer une position à chaque nouvelle instance  class Agent:    def \_\_init\_\_(self, position, \*\*agent\_attributes):  self.position = position  for attr\_name, attr\_value in agent\_attributes.items():  setattr(self, attr\_name, attr\_value)    ...      def main():  ... |

Test : ok

Propriétes : des degrés aux radians ([cet excellent article](https://mathwithbaddrawings.com/2013/05/02/degrees-vs-radians/))

* Formule : longitude = longitude\_degrees \* PI / 180
* Principe : calcul dans une nouvelle méthode en y accédant comme s'il s'agissait d'un attribut.

🡪 solution python : **propriétés** = une méthode qui est accessible comme un attribut (#renvoie une valeur)

Etape1 : méthode qui effectue le calcul

|  |
| --- |
| class Position:  def \_\_init\_\_(self, longitude, latitude):  self.latitude = latitude  self.longitude = longitude  def longitude(self):  # Longitude in radians  pass  # return self.longitude\_degrees \* pi / 180 |

Etape 2 : pour plus de clarté, changer le nom des attributs

latitude => latitude\_degrees / longitude => longitude\_degrees

Etape 3 : pour accéder à la longitude : position.longitude(), mais on veut écrire position.longitude #attribut

🡪utiliser une propriété : syntaxe : ajouter le mot-clé @property avant

|  |
| --- |
| class Position:  def \_\_init\_\_(self, longitude\_degrees, latitude\_degrees):  self.latitude\_degrees = latitude\_degrees  self.longitude\_degrees = longitude\_degrees  @property  def longitude(self):  return self.longitude\_degrees \* pi / 180  def main():  for agent\_attributes in json.load(open("agents-100k.json")):  ...  print(agent.position.longitude) |

Le nombre PI : module mathématique donne la valeur de Pi = math.pi

|  |
| --- |
| import math  return self.longitude\_degrees \* math.pi / 180  #idem pour latitude  @property  def latitude(self):  # Latitude in radians  return self.latitude\_degrees \* math.pi / 180 |

Github : [**https://github.com/OpenClassrooms-Student-Center/la\_poo\_avec\_python/tree/03\_properties**](https://github.com/OpenClassrooms-Student-Center/la_poo_avec_python/tree/03_properties)

**Partie 3**

**8\_ CLASSE ZONE**

On a : les agents, la position de chaque agent et globalement la possibilité de calculer une position

On va : créer le plateau de jeu (=ensemble de zones qui constituent notre monde)

Constructeur : la classe Zone a 3 attributs (position bas-gauche, position haut-droit, habitants)

Rem : par défaut, la zone est considérée comme vide à sa création

|  |
| --- |
| class Zone:  def \_\_init\_\_(self, corner1, corner2):  self.corner1 = corner1  self.corner2 = corner2  self.inhabitants = 0 |

Rem : créer toutes les zones qui constituent le monde = comme un quadrillage

Attributs de classe : variable qui appartiennent à une classe plutôt qu’aux instances

Longitude : entre -180 et 180 degrés & latitude entre -90 et 90 degrés

#### Création : par convention, un attribut de classe s'indique en majuscules et en début de la classe

|  |
| --- |
| class Zone:    MIN\_LONGITUDE\_DEGREES = -180  def \_\_init\_\_(self, corner1, corner2):  self.corner1 = corner1  self.corner2 = corner2  self.inhabitants = 0 |

#### Accès :

#### À partir de la classe : print(Zone.MIN\_LONGITUDE\_DEGREES)

#### À partir d’une instance :

|  |
| --- |
| def \_\_init\_\_(self, corner1, corner2):  ...  longitude = self.MIN\_LONGITUDE\_DEGREES |

* Attributs nécessaires la première ligne

|  |
| --- |
| def \_\_init\_\_(self, corner1, corner2):  ...  longitude = self.MIN\_LONGITUDE\_DEGREES |

* Méthode pour initialiser la grille

|  |
| --- |
| def initialize\_zones(self): |

#### 1ere boucle : créer les lignes avec la longitude : 1ère ligne : latitude = 1

🡪 1ère boucle = 1 zone tout nombre entre MIN\_LONGITUDE\_DEGREES et MAX\_LONGITUDE\_DEGREE

🡪 méthode range() permet justement de créer une liste avec 3vameurs (min, max, intervalle)

range(minimal\_value, maximal\_value, added\_value)

|  |
| --- |
| class Zone  ...  def initialize\_zones(self):  for longitude in range(self.MIN\_LONGITUDE\_DEGREES, self.MAX\_LONGITUDE\_DEGREES, self.WIDTH\_DEGREES):  # zone = Zone(bottom\_left\_corner, top\_right\_corner) |

* nous avons déjà accès à la longitude de depart dans la boucle

|  |
| --- |
| class Zone  ...  def initialize\_zones(self):  for longitude in range(self.MIN\_LONGITUDE\_DEGREES, self.MAX\_LONGITUDE\_DEGREES, self.WIDTH\_DEGREES):  bottom\_left\_corner = Position(longitude, 1) |

* position du coin supérieur droit = ajouter 1 intervalle (déjà stocké dans WIDTH\_DEGREE)

|  |
| --- |
| class Zone  def initialize\_zones(self):  for longitude in range(self.MIN\_LONGITUDE\_DEGREES, self.MAX\_LONGITUDE\_DEGREES, self.WIDTH\_DEGREES):  bottom\_left\_corner = Position(longitude, 1)  top\_right\_corner = Position(longitude + self.WIDTH\_DEGREES) |

* on ajoute l'intervalle utilisé pour la latitude

|  |
| --- |
| class Zone  MIN\_LONGITUDE\_DEGREES = -180  MAX\_LONGITUDE\_DEGREES = 180  WIDTH\_DEGREES = 1 # degrees of longitude  HEIGHT\_DEGREES = 1 # degrees of latitude  ... |

* calcul alors de la latitude du coin supérieur droit et créer la zone correspondante :

|  |
| --- |
| class Zone  ...  def initialize\_zones(self):  for longitude in range(self.MIN\_LONGITUDE\_DEGREES, self.MAX\_LONGITUDE\_DEGREES, self.WIDTH\_DEGREES):  bottom\_left\_corner = Position(longitude, 1)  top\_right\_corner = Position(longitude + self.WIDTH\_DEGREES, 1 + self.HEIGHT\_DEGREES)  zone = Zone(bottom\_left\_corner, top\_right\_corner) |

#### 2ème boucle : Créer les colonnes avec la latitude

Ajoutons les attributs de classe :

class Zone

...

MIN\_LATITUDE\_DEGREES = -90

MAX\_LATITUDE\_DEGREES = 90

WIDTH\_DEGREES = 1 # degrees of longitude

HEIGHT\_DEGREES = 1 # degrees of latitude

Enfin ajoutons une autre boucle concernant la latitude :

class Zone

...

def initialize\_zones(self):

for latitude in range (self.MIN\_LATITUDE\_DEGREES, self.MAX\_LATITUDE\_DEGREES, self.HEIGHT\_DEGREES):

for longitude in range(self.MIN\_LONGITUDE\_DEGREES, self.MAX\_LONGITUDE\_DEGREES, self.WIDTH\_DEGREES):

bottom\_left\_corner = Position(latitude, longitude)

top\_right\_corner = Position(longitude + self.WIDTH\_DEGREES, latitude + self.HEIGHT\_DEGREES)

zone = Zone(bottom\_left\_corner, top\_right\_corner)

YEAH !! Si nous lançons notre programme, nos remarquons que toutes les zones se créent ! :)

### Toutes les zones de notre grille

Pourquoi avons-nous créé cette grille ? Pour positionner nos habitants à l'intérieur. Il nous faut donc un moyen de parcourir toutes les zones, d'une manière ou d'une autre, afin de trouver la zone d'habitation de chaque agent. Or pour le moment, nous ne stockons les instances nulle part. Elles sont créées et seul l'ordinateur sait où elles sont.

#### Agréger toutes les zones

Nous allons donc créer un nouvel attribut de classe pour stocker chaque nouvelle zone qui est créée.

class Zone:

ZONES = []

...

Enfin, à l'intérieur de notre méthode qui initialise les zones, nous allons ajouter une à une chaque zone à notre liste ZONES.

class Zone:

...

def initialize\_zones(self):

...

self.ZONES.append(zone)

Afin de connaître combien de zones ont été créées, j'ajoute un print() à la fin :

class Zone:

...

def initialize\_zones(self):

for latitude in range (self.MIN\_LATITUDE\_DEGREES, self.MAX\_LATITUDE\_DEGREES, self.HEIGHT\_DEGREES):

for longitude in range(self.MIN\_LONGITUDE\_DEGREES, self.MAX\_LONGITUDE\_DEGREES, self.WIDTH\_DEGREES):

bottom\_left\_corner = Position(latitude, longitude)

top\_right\_corner = Position(longitude + self.WIDTH\_DEGREES, latitude + self.HEIGHT\_DEGREES)

zone = Zone(bottom\_left\_corner, top\_right\_corner)

self.ZONES.append(zone)

print(len(self.ZONES))

#### Les méthodes de classe

Exécutons notre script afin de vérifier que tout fonctionne...

...

Mille milliards de mille sabords ! Rien ne se passe ! Pourquoi donc ?

Réfléchissons un peu... Ah, mais bien sûr ! À aucun moment je n'exécute la méthode. Pour l'instant je l'ai juste définie ! Forcément, cela ne risque pas de fonctionner...

J'aimerais pouvoir lancer ma méthode maintenant ! Mais si j'essaie de la lancer, je suis prise moi-même dans une boucle infinie. Le seul moyen que nous connaissions pour lancer une méthode, jusqu'à maintenant, est de créer une instance et de l'utiliser pour exécuter la méthode. Comme ceci :

zone = Zone(1, 1)

zone.initialize\_zones()

Mais... Comment lancer une méthode sur une instance alors que cette même méthode est celle qui, justement, est censée les créer ? Aaaaah !

Il faudrait une méthode qui soit globale, au niveau de la classe, et non de l'instance. En fait nous l'avons déjà fait pour les attributs d'instance et de classe ! Alors, comment le faire pour les méthodes ? Vous ajoutez  @classmethod  juste avant.

class Zone:

...

@classmethod

def initialize\_zones(self):

...

Bien. Étant donné que nous ne sommes plus au niveau de l'instance mais au niveau de la classe, nous allons remplacer self par cls (afin de ne pas confondre).

En soi, vous pourriez mettre n'importe quel mot : caillou, boubou, chouchou... Mais bon. On aime bien les conventions quand même, elles nous permettent de lire le code des autres !

Nous allons donc remplacer par cls :

class Zone:

@classmethod

def initialize\_zones(cls):

for latitude in range (cls.MIN\_LATITUDE\_DEGREES, cls.MAX\_LATITUDE\_DEGREES, cls.HEIGHT\_DEGREES):

for longitude in range(cls.MIN\_LONGITUDE\_DEGREES, cls.MAX\_LONGITUDE\_DEGREES, cls.WIDTH\_DEGREES):

bottom\_left\_corner = Position(latitude, longitude)

top\_right\_corner = Position(longitude + cls.WIDTH\_DEGREES, latitude + cls.HEIGHT\_DEGREES)

zone = Zone(bottom\_left\_corner, top\_right\_corner)

cls.ZONES.append(zone)

print(len(cls.ZONES))

Nous pouvons désormais l'exécuter hors de la classe ainsi :  MaClasse.MaMethodeDeClasse()

Zone.initialize\_zones()

Bravo bravo ! Nous avons notre grille ! Nous allons dans le prochain chapitre ajouter les habitants.

📜

Une **méthode de classe** est une fonction dont le champ d'action s'étend à l'ensemble de la classe. Elles sont souvent utilisées pour créer de nouvelles instances à travers des boucles ou pour modifier des attributs de classe.

Vous souhaitez en savoir plus sur les méthodes de classe et découvrir une autre manière de les créer ? Je vous conseille le chapitre [Première approche des classes](https://openclassrooms.com/courses/apprenez-a-programmer-en-python/premiere-approche-des-classes#/id/r-2232992) du cours Apprenez à programmer en Python de Vincent le Goff.

### Code de ce chapitre

Retrouvez le code de ce chapitre en cliquant ici : <https://github.com/OpenClassrooms-Student-Center/la_poo_avec_python/tree/04_class_methods>

**9\_ AJOUTER UN HABITANT**

Nous avons la grille, les zones, les positions et les agents ! Youhouhou !! Mais pour l'instant les habitants sont hors des zones, un peu comme s'ils lévitaient... Faisons-les redescendre sur terre en les plaçant sur la grille !

### Trouver la zone d'habitation

Nous allons commencer par créer une méthode qui trouve dans quelle zone "habite" l'agent. Puis nous mettrons à jour la zone pour calculer la population !

C'est parti !

Premièrement, trouvons les zones dans lesquelles habitent chacun de nos agents. Étant donné que nous allons parcourir les zones, nous créons une méthode de classe :

class Zone:

...

@classmethod

def find\_zone\_that\_contains(cls):

La première solution serait de parcourir toutes les zones et de comparer la position des coins à la position de la zone.

Cela veut dire que Python va réaliser beaucoup d'opérations ! En fait, pour chaque agent il va parcourir toutes les zones ! On multiplie le nombre de zones par le nombre d'habitants et on obtient le nombre d'opérations effectuées. Ainsi :

 operations = (nombre de zones) \* (nombre d'habitants)

Le problème est que cela va demander beauuuuuuuucoup de calculs et prendra très longtemps !

La seconde solution est de trouver l'index de la zone à partir de l'index de la position.

class Zone:

...

def contains(self, position):

return position.longitude >= min(self.corner1.longitude, self.corner2.longitude) and \

position.longitude < max(self.corner1.longitude, self.corner2.longitude) and \

position.latitude >= min(self.corner1.latitude, self.corner2.latitude) and \

position.latitude < max(self.corner1.latitude, self.corner2.latitude)

@classmethod

def find\_zone\_that\_contains(cls, position):

# Compute the index in the ZONES array that contains the given position

longitude\_index = int((position.longitude\_degrees - cls.MIN\_LONGITUDE\_DEGREES)/ cls.WIDTH\_DEGREES)

latitude\_index = int((position.latitude\_degrees - cls.MIN\_LATITUDE\_DEGREES)/ cls.HEIGHT\_DEGREES)

longitude\_bins = int((cls.MAX\_LONGITUDE\_DEGREES - cls.MIN\_LONGITUDE\_DEGREES) / cls.WIDTH\_DEGREES) # 180-(-180) / 1

zone\_index = latitude\_index \* longitude\_bins + longitude\_index

# Just checking that the index is correct

zone = cls.ZONES[zone\_index]

assert zone.contains(position)

return zone

Puis nous allons modifier la fonction  main()  pour lancer cette méthode.

def main():

Zone.initialize\_zones()

for agent\_attributes in json.load(open("agents-100k.json")):

...

zone = Zone.find\_zone\_that\_contains(position)

### Ajoutez un habitant !

Commençons par ajouter une méthode d'instance  add\_inhabitant(agent)  :

def main():

for agent\_attributes in json.load(open("agents-100k.json")):

...

zone.add\_inhabitant(agent)

Puis créons la méthode associée dans la classe. Cette méthode va ajouter l'agent à une liste. Il nous faut donc également mettre à jour la méthode \_\_init\_\_() pour que les habitants par défaut ne soient pas 0 mais une liste vide :

class Zone:

...

def \_\_init\_\_(self, corner1, corner2):

self.corner1 = corner1

self.corner2 = corner2

self.inhabitants = []

def add\_inhabitant(self):

pass

Comment ajouter un élément à une liste ? En utilisant la méthode append() !

class Zone

...

def add\_inhabitant(self, inhabitant):

self.inhabitants.append(inhabitant)

Dernière ligne droite à présent ! Nous souhaitons connaître la population d'une zone, c'est-à-dire le nombre total d'habitants, que nous diviserons par le nombre de kilomètres carrés afin d'obtenir la densité de population. Nous allons donc créer une nouvelle méthode qui renvoie le nombre total d'éléments dans la liste de population.

class Zone

...

def population(self):

return len(self.inhabitants)

Étant donné que cette méthode existe uniquement pour me donner une information, et non vraiment pour effectuer une action, je vais la transformer en propriété :

class Zone

...

@property

def population(self):

return len(self.inhabitants)

Testons ! J'ajoute  print(zone.population)  dans la fonction  main() ‌ pour vérifier que les habitants sont bien ajoutés.

def main():

Zone.initialize\_zones()

for agent\_attributes in json.load(open("agents-100k.json")):

latitude = agent\_attributes.pop("latitude")

longitude = agent\_attributes.pop("longitude")

position = Position(longitude, latitude)

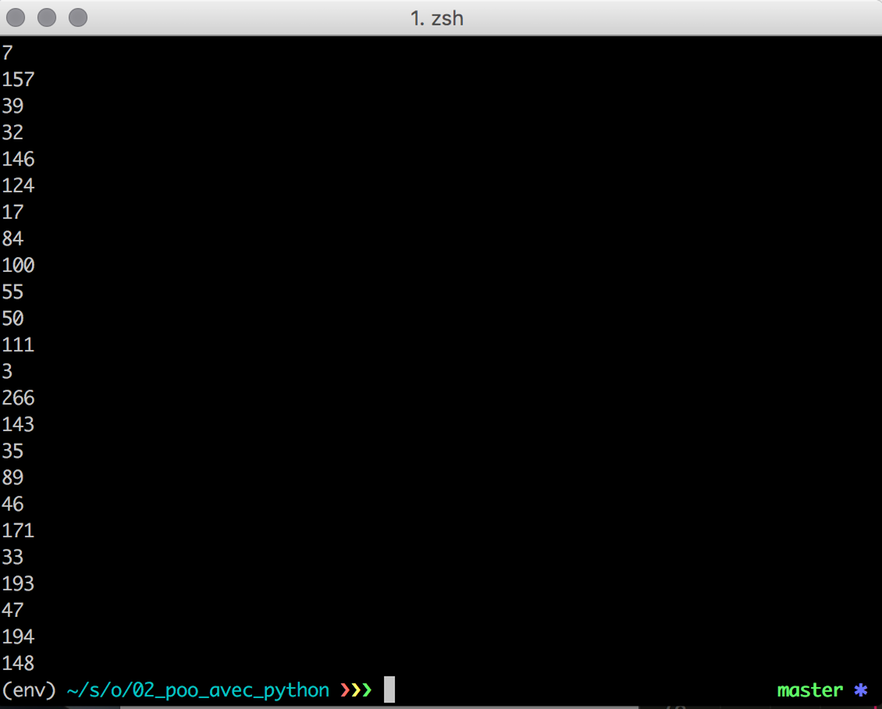
agent = Agent(position, \*\*agent\_attributes)

zone = Zone.find\_zone\_that\_contains(position)

zone.add\_inhabitant(agent)

print("Zone population: ", zone.population)

Et ça maaaaaaaarche ! \o/



### Code du chapitre‌

Le code complet du chapitre est disponible ici : <https://github.com/OpenClassrooms-Student-Center/la_poo_avec_python/blob/05_add_inhabitants/model.py>

**10\_ ENCAPSULATION**

Analogie :capsules à café ;-) # une classe est comme une capsule

* elle contient en elle-même tout ce dont elle a besoin pour fonctionner
* une partie de son fonctionnement est caché afin de ne donner accès qu'aux fonctions essentielles

**Définition : encapsulation** est un mot qui illustre deux concepts :

* les attributs et les méthodes d'un objet lui sont spécifiquement associés
* le champ d'action des attributs et des méthodes est par défaut l'objet lui-même et non tout autre objet

#### Programmation : classe CoffeeMachine + bouton Tasse de Café => allume machine, chauffer eau, verser café

#### 🡪 méthode \_\_init\_\_() appelle 3 autres méthodes : demarrer\_machine(), bouillir\_eau() et faire\_cafe()

#### pas de bouton bouillir l'eau, méthode bouillir\_eau() accessible uniquement interne = éléménts privés

* un bouton demarrer\_machine() si besoin mais plutôt sur le bouton Tasse de Café = éléments **protégés**. = on peut y accéder en dehors de la classe, mais on l'utilise plutôt à l'intérieur de la classe.

Application :

Étape 1 : classe CoffeeMachine qui a 3 méthodes :  start\_machine(), boil\_water() et make\_coffee()

|  |
| --- |
| class CoffeeMachine():  def start\_machine(self):  # Start the machine  def boil\_water(self):  # Boil water  def make\_coffee(self):  # Make a new coffee! |

Étape 2 : niveau d'eau maximum de 100, ne démarre que si son niveau d'eau est supérieur à 20

|  |  |
| --- | --- |
| class CoffeeMachine():  WATER\_LEVEL = 100  def start\_machine(self):  # Start the machine  if self.WATER\_LEVEL > 20:  return True  else:  print("Please add water!")  return False  def boil\_water(self):  return "boiling..."  def make\_coffee(self):  # Make a new coffee!  if self.start\_machine():  self.WATER\_LEVEL -= 20  print(self.boil\_water())  print("Coffee is ready!")  machine = CoffeeMachine()  for i in range(0, 5):  machine.make\_coffee() | boiling...  Coffee is ready!  boiling...  Coffee is ready!  boiling...  Coffee is ready!  boiling...  Coffee is ready!  Please add water! |

#### Méthodes privées ou protégées : on ne dois pas pouvoir exécuter la méthode boil\_water() hors de la classe.

#### 🡪 méthodes protégées : préfixer d’1 underscore

|  |  |
| --- | --- |
| class CoffeeMachine:  ...  def \_start\_machine(self):  # Start the machine.  if self.water\_level > 20:  return True  else:  print("Please add water!")  return False | 🡪sans le underscore, cela ne fonctionnera plus :  Traceback (most recent call last):  File "private.py", line 26, in <module>  print("Start Machine: Protected", machine.start\_machine())  AttributeError: 'CoffeeMachine' object has no attribute 'start\_machine' |
| class CoffeeMachine:  ...  def make\_coffee(self):  # Make a new coffee!  if self.\_start\_machine():  self.water\_level -= 20  print(self.boil\_water())  print("Coffee is ready!") | 🡪 ajoutez le underscore pour l’utiliser encore |

🡪méthodes privées = préfixer de 2 underscores

|  |
| --- |
| class CoffeeMachine  ...  def make\_coffee(self):  # Make a new coffee!  if self.\_start\_machine():  self.water\_level -= 20  print(self.\_\_boil\_water())  print("Coffee is ready!") |

#### accès tout public : vraiment privées, protégées ?

|  |  |
| --- | --- |
| machine = CoffeeMachine()  # for i in range(0, 5):  # machine.make\_coffee()  print("Make Coffee: Public", machine.make\_coffee())  print("Start Machine: Protected", machine.\_start\_machine())  print("Boil Water: Private", machine.\_\_boil\_water()) | Résultat attendu :  1. devrait afficher "boiling..." et "Coffee is ready"  2. devrait afficher "Start Machine"  3. devrait renvoyer une erreur.  Résultat :  boiling...  Coffee is ready!  Make Coffee: Public None  Start Machine: Protected True  Traceback (most recent call last):  File "private.py", line 28, in <module>  print("Boil Water: Private", machine.\_\_boil\_water())  AttributeError: 'CoffeeMachine' object has no attribute '\_\_boil\_water' |

Rem : en écrivant \_MaClasse\_\_methode\_privee(), je peux quand même l'exécuter la méthode privée

|  |  |
| --- | --- |
| print("Boil Water: Private", machine.\_CoffeeMachine\_\_boil\_water()) | Résultat :  boiling...  Coffee is ready!  Make Coffee: Public None  Start Machine: Protected True  Boil Water: Private boiling... |

Rem : certains langages sont très stricts au sujet de l'encapsulation, Python est plus laxiste.

= philosophie du langage est plus axée sur la confiance : Guido van Rossum, "We are all consenting adults here"

**Par défaut tout est public en Python,** notion d'encapsulation se fait à travers une convention de syntaxe

Réf : livre [Fluent Python](https://books.google.fr/books?id=kYZHCgAAQBAJ&lpg=PA272&ots=irCiYxyRQk&dq=fluent%20python%20private&hl=es&pg=PA272#v=onepage&q&f=false) par Luciano Ramalho.

Encapsulation pour notre programme

initialisation des zones devrait se faire de l'intérieur de la classe, et non de l'extérieur sauf si besoin

=> "protéger" la méthode de la classe initialize\_zones(cls) et l'enlever de notre fonction main().

|  |
| --- |
| class Zone  ...    @classmethod  def \_initialize\_zones(cls):  for latitude in range (cls.MIN\_LATITUDE\_DEGREES, cls.MAX\_LATITUDE\_DEGREES, cls.HEIGHT\_DEGREES):  for longitude in range(cls.MIN\_LONGITUDE\_DEGREES, cls.MAX\_LONGITUDE\_DEGREES, cls.WIDTH\_DEGREES):  bottom\_left\_corner = Position(longitude, latitude)  top\_right\_corner = Position(longitude + cls.WIDTH\_DEGREES, latitude + cls.HEIGHT\_DEGREES)  zone = Zone(bottom\_left\_corner, top\_right\_corner)  cls.ZONES.append(zone)  def main():  for agent\_attributes in json.load(open("agents-100k.json")):  latitude = agent\_attributes.pop("latitude")  longitude = agent\_attributes.pop("longitude")  position = Position(longitude, latitude)  agent = Agent(position, \*\*agent\_attributes)  zone = Zone.find\_zone\_that\_contains(position) |

Étant donné la fonction find\_zone\_that\_contains(position), vérifier si les zones existent déjà au tout début :

|  |
| --- |
| class Zone  ...    @classmethod  def find\_zone\_that\_contains(cls, position):  if not cls.ZONES:  # Initialize zones automatically if necessary  cls.\_initialize\_zones()  # Compute the index in the ZONES array that contains the given position  longitude\_index = int((position.longitude\_degrees - cls.MIN\_LONGITUDE\_DEGREES)/ cls.WIDTH\_DEGREES)  latitude\_index = int((position.latitude\_degrees - cls.MIN\_LATITUDE\_DEGREES)/ cls.HEIGHT\_DEGREES)  longitude\_bins = int((cls.MAX\_LONGITUDE\_DEGREES - cls.MIN\_LONGITUDE\_DEGREES) / cls.WIDTH\_DEGREES) # 180-(-180) / 1  zone\_index = latitude\_index \* longitude\_bins + longitude\_index  # Just checking that the index is correct  zone = cls.ZONES[zone\_index]  assert zone.contains(position)    return zone |

Github : <https://github.com/OpenClassrooms-Student-Center/la_poo_avec_python/tree/06_encapsulation>

**Partie 4**

**11\_ AGREBILITE MOYENNE D’UNE ZONE**

## Calculez l'agréabilité moyenne d'une zone

Nous y sommes presque ! Comment calcule-t-on la densité de population ? Nous avons la population par zone. Il ne nous manque plus qu'à calculer la surface en kilomètres carrés.

### Conversion en kilomètres

Actuellement nous n'avons que la longitude et la latitude en radians. Il nous faut les convertir en kilomètres avant de pouvoir les utiliser pour calculer une surface.

Ne vous inquiétez pas, nous n'allons pas faire de maths très avancés !

En cherchant un peu, j'ai trouvé que nous pouvons calculer la largeur d'une surface à partir de sa longitude (en radians). Il suffit de multiplier un angle, en radian, par le rayon de la Terre (en kilomètres).

Créons une nouvelle méthode  width(). Etant donné que ce sera une valeur qui sera renvoyée et que je ne veux pas particulièrement faire une action, transformons-la en une propriété :

class Zone

...

@property

def width(self):

Etant donné que le rayon de la Terre ne sera pas amené à changer, je crée un nouvel attribut de classe :

class Zone:

...

EARTH\_RADIUS\_KILOMETERS = 6371

Et je mets à jour la propriété  width() :

class Zone

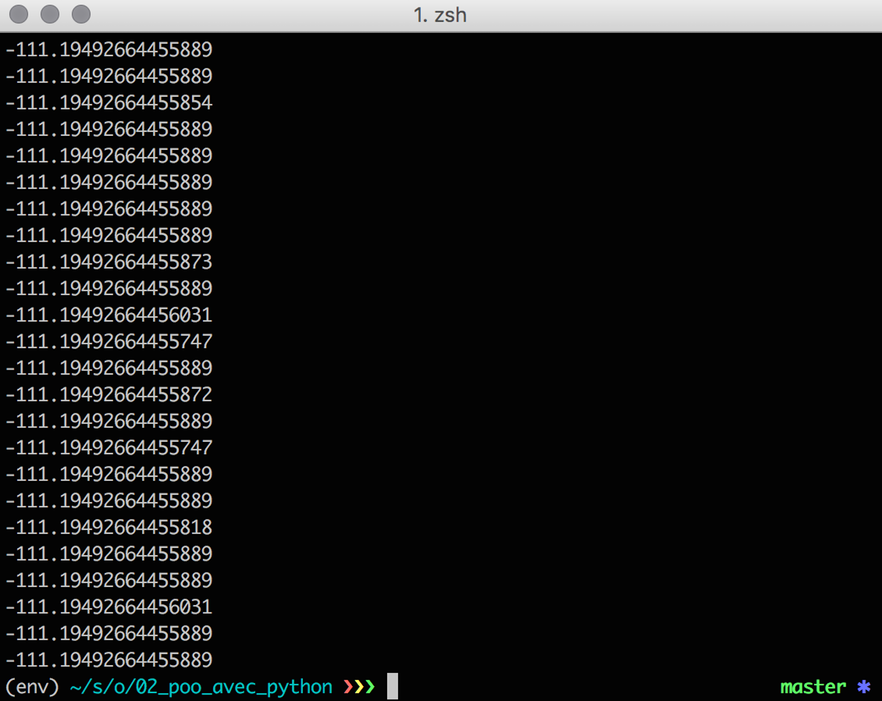
...

@property

def width(self):

return (self.corner1.longitude - self.corner2.longitude) \* self.EARTH\_RADIUS\_KILOMETERS

Je vais vérifier que ça fonctionne en ajoutant un print() à ma fonction  main().



C'est plutôt pas mal... sauf que les largeurs sont parfois négatives ! Pourquoi ? Car la longitude de corner1 est inférieure à la longitude de  corner2. Si vous écrivez  "3 - 4", le résultat sera -1 ! Il en est de même ici.

Pour éviter cela, transformons la longitude en valeur absolue.

Une valeur absolue est un nombre qui est soit nul, soit positif. La valeur absolue de -1 est donc... 1.  
La librairie standard de Python contient une méthode qui s'appelle abs() et qui prend en paramètre un nombre. Il transforme ce dernier en valeur absolue. Utilisons-la !

class Zone

...

@property

def width(self):

return abs(self.corner1.longitude - self.corner2.longitude) \* self.EARTH\_RADIUS\_KILOMETERS

@property

def height(self):

return abs(self.corner1.latitude - self.corner2.latitude) \* self.EARTH\_RADIUS\_KILOMETERS

### Kilomètres carrés

Ajoutons une nouvelle méthode pour calculer la surface d'une zone puis trouvons la densité de population :

class Zone

...

@property

def area(self):

return self.height \* self.width

def population\_density(self):

return self.population / self.area

À noter que nous aurons une erreur si la surface est égale à zéro car nous essayerions de diviser par 0 ! Mais bon, ça ne devrait pas arriver.

### Agréabilité moyenne

Maintenant que nous avons la densité de population, il nous manque l'agréabilité moyenne d'une zone. Comment faire ? Nous allons ajouter l'agréabilité de toutes les personnes de la zone et diviser cette somme par le nombre de personnes. Une simple moyenne en fait ! :)

Etant donné que la division par zéro cause l'arrêt du programme, nous allons ajouter une condition : s'il n'y a pas d'habitant dans une zone, l'agréabilité sera de zéro.

class Zone:

...

def average\_agreeableness(self):

if not self.inhabitants:

return 0

agreeableness = []

for inhabitant in self.inhabitants:

agreeableness.append(inhabitant.agreableness)

return sum(agreeableness) / self.population

### List comprehension

Que diriez-vous si je vous disais que vous pouvez **condenser** les 4 lignes de la boucle et celle de création de liste en une seule ? Vous sauteriez au plafond, n'est-ce pas ?! Hé bien oui ! Nous allons le faire !   
Cela porte même le nom super sexy de "list comprehension".

Une list comprehension est une liste dans laquelle vous effectuez une opération pour créer cette même liste. Oui, je sais, nous repartons encore dans Inception... Reprenons notre exemple.

Nous commençons par écrire le résultat de l'action dans la liste. Dans notre cas, nous voulons que la liste contienne l'agréabilité d'un habitant. Puis nous indiquons la condition ou la boucle à effectuer. Dans notre cas, c'est une boucle for :

[inhabitant.agreeableness for inhabitant in self.inhabitants]

Enfin, nous pouvons nous passer de la création d'une variable. Nous n'en avons pas besoin plus tard ! Nous allons donc effectuer l'opération directement dans la méthode sum() :

class Zone

...

def average\_agreeableness(self):

if not self.inhabitants:

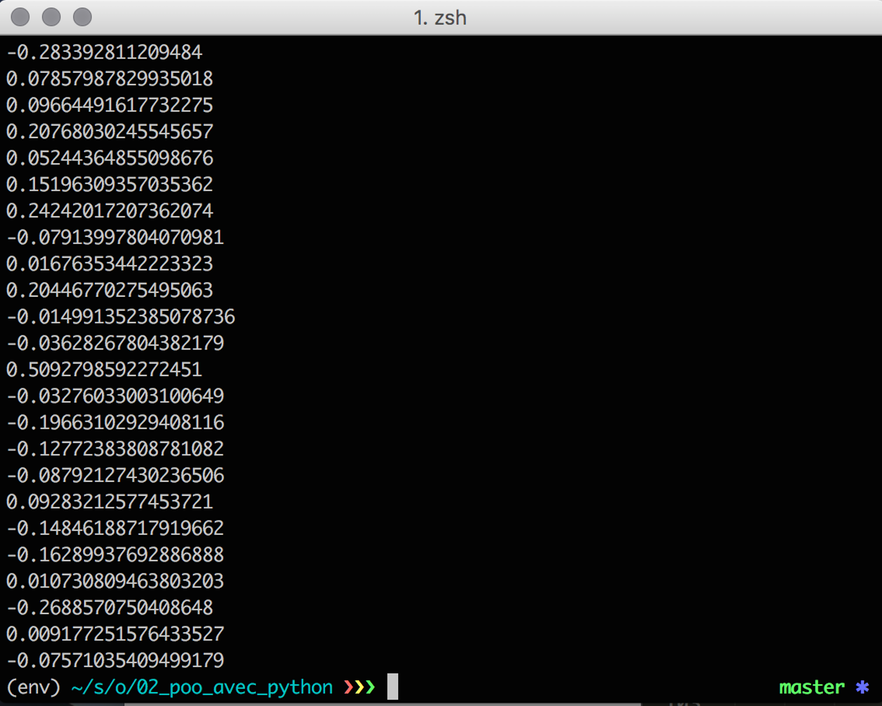
return 0

return sum([inhabitant.agreeableness for inhabitant in self.inhabitants]) / self.population

Et voilà le travail !

Vous pouvez évidemment créer des list comprehensions bien plus poussées ! Lisez cet article de Trey Hunner pour aller plus loin : [Python list comprehensions: explained visually](http://treyhunner.com/2015/12/python-list-comprehensions-now-in-color/) 🇺🇸.

Testons ! Je modifie la fonction main() pour afficher l'agréabilité moyenne. Et ça fonctionne ! \o/



Nous avons tous les éléments pour créer nos graphiques désormais ! Youpi ! Voyons comment faire dans le prochain chapitre.

### Code du chapitre

Retrouvez l'intégralité du code de ce chapitre ici : <https://github.com/OpenClassrooms-Student-Center/la_poo_avec_python/blob/07_list_comp/model.py>

**11\_ GRAPHIQUE**

## Créez un graphique !

C'est parti pour la création de notre graphique !

### Options du graphique de base

Avant tout, commençons par écrire la manière dont nous souhaitons que se lance notre graphique. Dans la fonction main(), ajoutons quelques lignes de pseudo code :

# Initialisation du graphique

# agreeableness\_graph = AgreeablenessGraph()

# Affichage du graphique. On passe en paramètre la liste de nos zones pour y avoir accès à l'intérieur de notre classe AgreeablenessGraph.

# agreeableness\_graph.show(Zone.ZONES)

Passons désormais à la définition de la classe.

Créons la classe Graph et ajoutons la méthode \_\_init\_\_(), comme nous l'avons déjà fait plusieurs fois jusqu'à maintenant. A l'intérieur, indiquons les attributs.

Je sais que mes graphiques auront chacun une abscisse, une ordonnée et un titre. Je veux également une grille en fond afin que la granularité du graphique soit plus évidente.

class BaseGraph:

def \_\_init\_\_(self):

self.title = "Your graph title"

self.x\_label = "X-axis label"

self.y\_label = "X-axis label"

self.show\_grid = True

### Le module MatPlotLib

[Matplotlib](https://matplotlib.org) est une librairie Python utilisée pour représenter des données en deux dimensions. Concrètement, elle vous permet de créer des schémas, des graphiques et même des spectres de couleurs. Elle est assez vaste et nous n'aurons pas besoin de toutes ses fonctionnalités.

En fait, nous allons utiliser un de ses modules qui s'appelle Pyplot et qui permet de créer des graphiques en couleur. Exactement ce dont nous avons besoin !

Commencez par installer le module. Dans votre console, écrivez pip install matplotlib.

Puis importez-le dans votre script en indiquant, tout en haut, import matplotlib.pyplot as plt.

Si vous utilisez Linux ou Mac, vous pouvez avoir un bug quand vous lancez le script. Afin de l'éviter, ajoutez quelques lignes AVANT l'import de Matplotlib. Comme ceci :

import matplotlib as mil

mil.use('TkAgg')

import matplotlib.pyplot as plt

[Je vous laisse lire la documentation](https://matplotlib.org/api/pyplot_api.html#matplotlib.pyplot.plot) et réfléchir à la manière dont vous pourrez créer un graphique.

...

Vous êtes prêt·e ? 😎

La création d'un graphique se fait en plusieurs phases :

* Initialisation du graphique : méthode plot() qui prend en argument les valeurs de l'abscisse, celles de l'ordonnée et le type de graphique voulu.  .plot(x\_values, y\_values, graph\_type)
* Ajout de la légende de l'abscisse : .xlabel(label)
* Ajout de la légende de l'ordonnée : .ylabel(label)
* Ajout du titre : .title()
* En option, nous pouvons indiquer si nous désirons une grille dans le fond : .grid()
* Création du graphique et ouverture d'une nouvelle fenêtre : .show()

Nous allons utiliser tout cela dans une nouvelle méthode : .show()

class Graph:

...

def show(self, zones):

# x\_values = gather only x\_values from our zones

# y\_values = gather only y\_values from our zones

plt.plot(x\_values, y\_values, '.')

plt.xlabel(self.x\_label)

plt.ylabel(self.y\_label)

plt.title(self.title)

plt.grid(self.show\_grid)

plt.show()

### Comprendre l'héritage

Nous avons parlé de la classe Graph qui allait avoir les attributs x (données en abscisse) et y (données en ordonnée). Pourtant, nous n'aurons pas besoin des mêmes données pour nos deux graphiques et, surtout, le calcul sera différent. Comment faire pour que nos graphiques partagent certaines méthodes et en adaptent d'autres ?

Laissez-moi vous présenter, sous un tonnerre d'applaudissements, **l'héritage** !

L'héritage permet de créer des "sous-classes" qui vont étendre ou modifier les méthodes de la classe "commune".   
En programmation, nous parlerons plutôt de "classe parent" pour désigner la classe commune et de "classes enfant" pour désigner les classes qui héritent de la classe parent.

**📜**

L'**héritage** est un mécanisme qui nous permet d'**étendre** les fonctionnalités d'une classe parent à travers une classe enfant. Il nous permet de réutiliser du code de manière simple et efficace.

De manière générale, nous disons qu'une classe enfant **hérite** d'une classe parent et que la classe parent **étend** la classe enfant.

La classe **enfant** a accès à tous les attributs et à toutes les méthodes de sa classe parent mais l'inverse n'est pas vrai. ;-)

Commençons par la classe parent que nous appellerons BaseGraph. Son but est d'intégrer tous les éléments dont nous aurons besoin, a minima, pour créer un graphique. Quelles méthodes ? Quels attributs ?

Mais nous ne créerons jamais d'instance directement à partir de cette classe, nous utiliserons les classes enfant pour cela. La classe parent existe uniquement pour donner une ligne de conduite, une base. Les classes enfant sont, elles, chargées d'appliquer cette ligne de conduite.

En résumé, la classe parent peut être vue comme une loi et les classes enfant comme les décrets d'application.   
Si vous n'êtes pas très sensible à ma métaphore législative, ce n'est pas grave. Reprenons l'exemple du sens de la conduite. Si nous avions à réaliser le programme correspondant, voici ce que nous écririons certainement.

#### Classe parent : TypeDeConduite

sens\_de\_la\_conduite  : attribut "droite" ou "gauche"  
depasser() : méthode pour faire avancer la voiture  
klaxonner() : méthode pour savoir dans quel cas il faut klaxonner.

Nous savons que, peu importe le pays, il y aura toujours, a minima, un sens de la conduite et que chacun devra dépasser ou klaxonner un jour. Pour autant, nous ne voulons pas encore dire comment. Ce sera à chaque classe enfant de définir les règles applicables dans le pays.

#### Classes enfant : ConduiteAnglaise et ConduiteFrancaise

Nous indiquons alors les éléments demandés. Si nous oublions de le faire, le programme nous renverra une erreur nous indiquant que nous avons oublié un élément obligatoire. Si nous avons à ajouter des règles pour la France, la priorité à droite par exemple, nous pouvons le faire dans la classe  ConduiteFrancaise.

La relation entre une classe enfant et une classe parent est souvent représentée par l'expression "est un (ou une)" ("is a" en anglais 🇺🇸). Dans notre cas, nous pouvons dire que la ConduiteAnglaise "est un" TypedeConduite. Cela crée une sorte de hiérarchie entre les classes qui aide à mieux comprendre l'héritage.

### L'héritage dans notre programme

Maintenant réfléchissons un peu. Notre classe parent sera  BaseGraph. Elle aura deux classes enfant qui correspondent aux deux types de graphique que nous voulons créer :  AgreeablenessGraph et IncomeGraph.

Quels vont être les éléments qui vont vraiment varier dans ces classes enfant ? Les attributs (le titre et les légendes) ainsi que les valeurs. Nous allons donc utiliser, dans les classes enfant, la méthode \_\_init\_\_ et nous devrons trouver le moyen de faire varier les données.

Renommons la classe parent (BaseGraph) puis créons la première classe enfant. Nous allons dessiner le graphique qui met en relation le taux d'agréabilité et la densité de population :  AgreeablenessGraph.

Hériter d'une classe en Python est très simple : vous l'indiquez entre parenthèses lorsque vous créez la classe enfant. Comme ceci :

class AgreeablenessGraph(BaseGraph):

pass

Ajoutons une méthode  \_\_init\_\_()  et les attributs que nous souhaitons changer :

class AgreeablenessGraph(BaseGraph):

def \_\_init\_\_(self):

self.title = "Nice people live in the countryside"

self.x\_label = "population density"

self.y\_label = "agreeableness"

Parfait. De cette façon, nous remplaçons la méthode d'origine et indiquons au programme d'utiliser nos valeurs et pas celles de la classe mère.

Une classe enfant peut utiliser une méthode héritée de la classe parent comme s'il s'agissait de la sienne. Elle peut néanmoins également la redéfinir entièrement si cela est nécessaire.

📜

On "override" une méthode lorsqu'on la redéfinit entièrement.

Je ne sais pas si vous vous en rendez compte, mais il manque un élément. Vous l'avez trouvé ? Oui, il s'agit bien de la grille !

Parlons-en, justement, de la grille. Nous l'aurons à la fois pour ce graphique et pour le suivant. C'est un prérequis. Je devrais donc être en mesure de l'indiquer dans ma classe parent et de l'importer, automatiquement, dans mes classes enfant.

Vous allez utiliser pour cela la méthode super() et la méthode parent que vous voulez utiliser. Dans notre cas :

class AgreeablenessGraph(BaseGraph):

def \_\_init\_\_(self):

super().\_\_init\_\_() # executes the parent's \_\_init\_\_() method

self.title = "Nice people live in the countryside"

self.x\_label = "population density"

self.y\_label = "agreeableness"

La méthode  super()  appelle la méthode correspondante de la classe parent. Cela vous permet d'en changer une partie sans pour autant tout remplacer !

### Intégrer des données

Pour rappel, voici où en est notre code :

class BaseGraph:

def \_\_init\_\_(self):

self.title = "Your graph title"

self.x\_label = "X-axis label"

self.y\_label = "X-axis label"

self.show\_grid = True

def show(self, zones):

# x\_values = gather only x\_values from our zones

# y\_values = gather only y\_values from our zones

plt.plot(x\_values, y\_values, '.')

plt.xlabel(self.x\_label)

plt.ylabel(self.y\_label)

plt.title(self.title)

plt.grid(self.show\_grid)

plt.show()

class AgreeablenessGraph(BaseGraph):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.title = "Nice people live in the countryside"

self.x\_label = "population density"

self.y\_label = "agreeableness"

Bien bien bien ! A présent, comment faire varier les données ? Plutôt que de modifier la méthode   
 show(), nous allons créer une nouvelle méthode xy\_values() dans la classe AgreeablenessGraph  dont le but sera de calculer ces valeurs. J'utilise les list comprehensions que vous connaissez déjà. ;-)

def AgreeablenessGraph(BaseGraph):

...

def xy\_values(self, zones):

x\_values = [zone.population\_density() for zone in zones]

y\_values = [zone.average\_agreeableness() for zone in zones]

return x\_values, y\_values

J'ai accès aux zones puisque je les ai passées en paramètre en lançant ma fonction. Rappelez-vous :

agreeableness\_graph.show(Zone.ZONES)

Mais... Ah mais en fait non. Je ne devrais pas mettre cette méthode dans cette classe mais plutôt dans la classe parent. Je pourrai ainsi l'appeler depuis la méthodeshow(). Ce serait malin non ?!

Je déplace :

class BaseGraph:

...

def xy\_values(self, zones):

x\_values = [zone.population\_density() for zone in zones]

y\_values = [zone.average\_agreeableness() for zone in zones]

return x\_values, y\_values

Sauf que... Vous vous en doutez, je ne calculerai pas les données de la même manière en fonction des graphiques que je souhaite générer. Je vais donc créer la même méthode mais dans la classe enfant :

class AgreeablenessGraph(BaseGraph):

...

def xy\_values(self, zones):

x\_values = [zone.population\_density() for zone in zones]

y\_values = [zone.average\_agreeableness() for zone in zones]

return x\_values, y\_values

Enfin, j'aimerais qu'une erreur apparaisse si j'oublie d'implémenter cette méthode dans ma classe enfant. En effet, tous nos graphiques auront forcément une abscisse et une ordonnée.

Je vais donc, dans la classe parent, ajouter une erreur :

class BaseGraph:

...

def xy\_values(self, zones):

raise NotImplementedError

Quand vous ne pouvez pas utiliser une classe parent pour vraiment créer une instance, nous disons qu'il s'agit d'une classe abstraite. Pourquoi ? Tout simplement car son objectif n'est pas de créer un objet mais de donner les règles communes, abstraites, qui seront partagées par tous ses enfants. Les classes enfant, qui partagent toutes les mêmes règles de la classe parent, sont appelées des classes concrètes car elles concrétisent des notions plus abstraites.

📜

Une classe **abstraite** est une classe qui ne peut pas être instanciée sans renvoyer une erreur. Elle doit être héritée par une classe enfant qui elle-même créera des instances.

L'inverse est appelé une classe **concrète** : on ne s'attend pas à ce qu'elle étende d'autres classes.

Lançons le programme... et tout fonctionne ! Youpi !

### L'héritage multiple

Python permet à une classe enfant d'hériter de plusieurs classes parent (comme dans une famille recomposée !). Cela s'appelle l'héritage multiple ou **Multiple Inheritance**.

Par exemple, une classeVampire pourrait hériter de la classeHumain et de la classe [Hématophage](http://www.larousse.fr/dictionnaires/francais/h%C3%A9matophage/39450?q=h%C3%A9matophage#39369) (et dePersonnageNiaissi nous sommes dans Twilight :-° ).

Tout comme dans les familles recomposées, cette fonctionnalité est sujette à controverse et ouvre la porte à quelques problématiques. C'est pourquoi je n'entre pas dans le détail de cette fonctionnalité.

Si vous désirez en savoir plus, je vous conseille de lire le chapitre [Inheritance: for good and for worse](https://books.google.fr/books?id=bIZHCgAAQBAJ&lpg=PT503&ots=mNhyIqNT6V&dq=fluent%20python%20multiple%20inheritance&hl=es&pg=PT479#v=onepage&q&f=false) (du livre Fluent Python)

### Challenge

Console de code

##### À votre tour de coder

**Mettez immédiatement en application ce que vous avez appris.**  
Vous êtes libre de vous tromper et de réessayer autant que vous le souhaitez.

Pensez à vous entraîner avant de terminer ce chapitre.

### Code de ce chapitre

Retrouvez le code de ce chapitre ici : <https://github.com/OpenClassrooms-Student-Center/la_poo_avec_python/tree/08_inheritance>

**12 \_ CONCLUSION**

## Conclusion

### Conclusion

Nous avons tout fini ! BRAVO !! Je vous félicite ! Si vous le désirez, vous pouvez aller plus loin en créant le second graphique par vous-même. Vous trouverez la proposition de Régis Behmo et moi sur Github, ici : <https://github.com/OpenClassrooms-Student-Center/la_poo_avec_python/blob/master/model.py>

Vous verrez que le code est légèrement différent de celui que nous avons fait ensemble. Nous y avons ajouté de bonnes pratiques dont nous n'avons pas encore parlé, notamment les Docstrings (les trois guillemets). Hé oui, cela fera l'objet d'un prochain cours !

### Aller plus loin

Vous venez de terminer ce cours et vous pouvez être fier ou fière de vous ! Mais votre apprentissage de l'orienté objet ne s'arrête pas là.

En fait, vous n'avez fait qu'ouvrir une porte vers un monde très vaste. Rappelez-vous : l'orienté objet est un paradigme, une manière de résoudre les problèmes. Il est composé de plusieurs domaines extrêmement intéressants mais trop étendus pour être résumés ici ! Ce cours le présente de manière extrêmement succincte et, par conséquent, incomplète.

Afin d'approfondir votre apprentissage, je vous conseille de vous former au Design Orienté Objet et notamment à UML ([Unified Modeling Language](https://fr.wikipedia.org/wiki/UML_(informatique)" \t "_blank)). Il s'agit d'un langage de modélisation graphique qui aide à concevoir un système informatique. Vous y découvrirez comment réaliser de véritables diagrammes de classe et de séquence qui suivent des conventions bien particulières.

Mon cours favori est celui de Laurent Audibert, [UML 2](http://laurent-audibert.developpez.com/Cours-UML/), sur Développez.com.  Une version plus récente est disponible en [format papier](http://livre.fnac.com/a7882988/L-Audibert-UML-2-de-l-apprentissage-a-la-pratique).

Un des lives de référence sur le sujet est [Applying UML and Patterns](https://www.amazon.es/Applying-UML-Patterns-Introduction-Object-Oriented/dp/0131489062). Très complet et volumineux, il s'utilise comme une Bible dans laquelle vous pourrez venir trouver des solutions au fur et à mesure de votre évolution professionnelle.

Puis intéressez-vous de plus près à l'héritage multiple et au polymorphisme, deux notions que nous avons survolées dans ce cours. Je vous recommande vivement la lecture de l'excellent ouvrage [Fluent Python](http://shop.oreilly.com/product/0636920032519.do).

### Bonus

Voici comment Steve Jobs a expliqué la programmation orientée objet à un journaliste du magazine Rolling Stone en 1994 :

***Jeff Goodell*** : Pourriez-vous expliquer, dans des termes simples, ce que signifie exactement "programme orienté objet" ?

***Steve Jobs*** : Les objets sont comme des gens. Ce sont des choses vivantes qui respirent et qui portent avec eux différentes connaissances qui leur permettent de faire des choses. Ils ont également une mémoire donc ils se souviennent des choses. Et plutôt que d'interagir avec eux à un bas niveau, vous interagissez avec eux à un très haut niveau d'abstraction, comme nous sommes en train de le faire actuellement.

Voici un exemple : si je suis votre objet "pressing", vous pouvez me donner vos affaires sales et m'envoyer un message qui dit : "Pourriez-vous envoyer mes affaires chez le pressing, s'il-vous-plait ?". Il se trouve que je sais exactement quel est le meilleur pressing de tout San Francisco. Et je parle anglais, et j'ai des dollars dans ma poche. Donc je dors, j'appelle un taxi et je demande au chauffeur de me laisser en face de ce pressing à San Francisco. Je fais laver vos affaires, je saute dans le taxi et je reviens ici. Je vous donne vos vêtements propres et je vous dit : "Voici vos vêtements propres".

Vous n'avez aucune idée de la manière dont j'ai fait tout ça. Vous ne savez pas où est le pressing. Peut-être même que vous parlez français et que vous ne pouvez pas appeler un taxi ici, à San Francisco. Vous ne pouvez pas le payer car vous n'avez pas de dollars en poche. Mais moi je sais comment faire tout ça. Et vous n'avez pas eu besoin de savoir quoi que ce soit sur la manière dont je l'ai fait. Toute la complexité était cachée à l'intérieur de moi et nous avons pu interagir à un très haut niveau d'abstraction. Voici ce que sont les objets. Ils "encapsulent" la complexité et les interfaces avec cette complexité sont de très haut niveau.