# [Découvrez la programmation orientée objet avec Python](https://openclassrooms.com/fr/courses/4302126-decouvrez-la-programmation-orientee-objet-avec-python)

Table des matières

[Découvrez la programmation orientée objet avec Python 1](#_Toc57218119)

[**Tirez pleinement parti de ce cours** 4](#_Toc57218120)

[**Notre programme** 4](#_Toc57218121)

[**Les habitants de notre monde** 4](#_Toc57218122)

[**Fonctionnalités du programme** 7](#_Toc57218123)

[**Etapes** 7](#_Toc57218124)

[**Quelques précisions** 7](#_Toc57218125)

[**Challenges** 8](#_Toc57218126)

[Découvrez la programmation orientée objet 9](#_Toc57218127)

[La programmation orientée objet 9](#_Toc57218128)

[Analyse, design et programmation 9](#_Toc57218129)

[Structurez votre programme 13](#_Toc57218130)

[Objets 13](#_Toc57218131)

[Attributs 13](#_Toc57218132)

[Méthodes 14](#_Toc57218133)

[Structurez votre programme 16](#_Toc57218134)

[Compétences évaluées 16](#_Toc57218135)

[ Question 1 16](#_Toc57218136)

[ Question 2 16](#_Toc57218137)

[ Question 3 16](#_Toc57218138)

[ Question 4 16](#_Toc57218139)

[ Question 5 17](#_Toc57218140)

[**Créez votre premier agent** 18](#_Toc57218141)

[**Démarrer le projet** 18](#_Toc57218142)

[**Créer un agent** 18](#_Toc57218143)

[**Challenge** 20](#_Toc57218144)

[**Code du chapitre** 20](#_Toc57218145)

[Customisez votre agent ! 21](#_Toc57218146)

[Ajouter une méthode 21](#_Toc57218147)

[Ajouter un attribut 21](#_Toc57218148)

[Modifier des attributs 23](#_Toc57218149)

[Challenge 24](#_Toc57218150)

[Ajoutez des attributs à partir d'un dictionnaire 25](#_Toc57218151)

[Passer un dictionnaire en paramètre 25](#_Toc57218152)

[Varier de nombre d'arguments 26](#_Toc57218153)

[Module json 28](#_Toc57218154)

[Challenge 30](#_Toc57218155)

[Code du chapitre 30](#_Toc57218156)

[**Créez la classe Position** 31](#_Toc57218157)

[**La classe Position** 31](#_Toc57218158)

[**Des degrés aux radians : les propriétés** 32](#_Toc57218159)

[**Le nombre Pi** 34](#_Toc57218160)

[**Code du chapitre** 34](#_Toc57218161)

[Concepts clés 35](#_Toc57218162)

[Compétences évaluées 35](#_Toc57218163)

[ Question 1 35](#_Toc57218164)

[ Question 2 35](#_Toc57218165)

[ Question 3 35](#_Toc57218166)

[ Question 4 36](#_Toc57218167)

[ Question 5 36](#_Toc57218168)

[Créez votre première Zone ! 37](#_Toc57218169)

[La classe Zone 37](#_Toc57218170)

[Toutes les zones de notre grille 41](#_Toc57218171)

[Code de ce chapitre 44](#_Toc57218172)

[Ajoutez un habitant ! 46](#_Toc57218173)

[Trouver la zone d'habitation 46](#_Toc57218174)

[Ajoutez un habitant ! 47](#_Toc57218175)

[Code du chapitre‌ 50](#_Toc57218176)

[Comprenez l'encapsulation 51](#_Toc57218177)

[What else? 51](#_Toc57218178)

[L'encapsulation appliquée à notre programme 57](#_Toc57218179)

[Code du chapitre 59](#_Toc57218180)

[Concepts avancés 60](#_Toc57218181)

[Compétences évaluées 60](#_Toc57218182)

[ Question 1 60](#_Toc57218183)

[ Question 2 60](#_Toc57218184)

[ Question 3 60](#_Toc57218185)

[ Question 4 60](#_Toc57218186)

[ Question 5 61](#_Toc57218187)

[Calculez l'agréabilité moyenne d'une zone 62](#_Toc57218188)

[Conversion en kilomètres 62](#_Toc57218189)

[Kilomètres carrés 64](#_Toc57218190)

[Agréabilité moyenne 64](#_Toc57218191)

[List comprehension 65](#_Toc57218192)

[Code du chapitre 66](#_Toc57218193)

[**Créez un graphique !** 67](#_Toc57218194)

[**Options du graphique de base** 67](#_Toc57218195)

[**Le module MatPlotLib** 67](#_Toc57218196)

[**Comprendre l'héritage** 69](#_Toc57218197)

[**L'héritage dans notre programme** 70](#_Toc57218198)

[**Intégrer des données** 72](#_Toc57218199)

[**L'héritage multiple** 75](#_Toc57218200)

[**Challenge** 75](#_Toc57218201)

[Code de ce chapitre 75](#_Toc57218202)

[Conclusion 76](#_Toc57218203)

[Conclusion 76](#_Toc57218204)

[Aller plus loin 76](#_Toc57218205)

[Bonus 76](#_Toc57218206)

[La programmation orientée objet - conclusion 78](#_Toc57218207)

[Compétences évaluées 78](#_Toc57218208)

[ Question 1 78](#_Toc57218209)

[ Question 2 78](#_Toc57218210)

[ Question 3 78](#_Toc57218211)

[ Question 4 79](#_Toc57218212)

[ Question 5 79](#_Toc57218213)

**Tirez pleinement parti de ce cours**

**Bienvenue !**

Dans ce cours nous allons parler de Programmation Orientée Objet. Vous n’y connaissez rien ? C’est normal ! Nous allons voir tout cela ensemble.

**Notre programme**

Afin d'illustrer les concepts que je vais vous présenter, nous allons créer ensemble un programme. Mais pas n'importe lequel ! Nous réaliserons... (roulements de tambours) un monde parallèle !

Oui oui, vous avez bien lu. Vous avez toutes les compétences pour le faire !

Nous allons manipuler les données sur les habitants qui peuplent ce monde parallèle pour répondre à deux questions existentielles :

* À partir de quelle densité de population est-on moins agréable que la moyenne ? Cela nous permettra de répondre à notre question d'introduction : "***l'enfer, c'est les autres ?***"
* Est-ce que les vieux gagnent vraiment plus d’argent que les jeunes ?

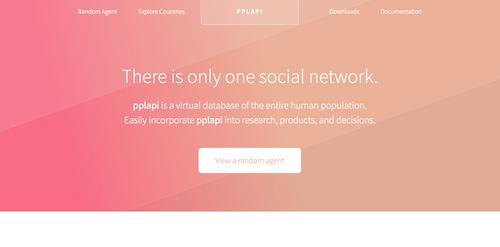
Mais je ne vais pas y arriver… On ne pourrait pas plutôt faire des camemberts dans Excel ?

Vous allez très bien y arriver. Nous pourrions bien sûr utiliser un fichier Excel qui regroupe toutes les données de tous les habitants de notre monde parallèle, mais il serait vite limité. Et puis, à quoi bon s'évertuer à cliquer sur 15 000 boutons quand une ligne de commande suffit ? En plus, votre but est d'apprendre à programmer en Python, pas vrai ?

Python est un des langages les plus utilisés dans l'univers scientifique et plus spécifiquement dans la manipulation de données. Ce cours est un (tout) petit aperçu de ce que vous pouvez faire ! J'ai axé la modélisation de notre monde sur deux questions mais libre à vous d'aller plus loin à la fin de ce cours.

**Les habitants de notre monde**

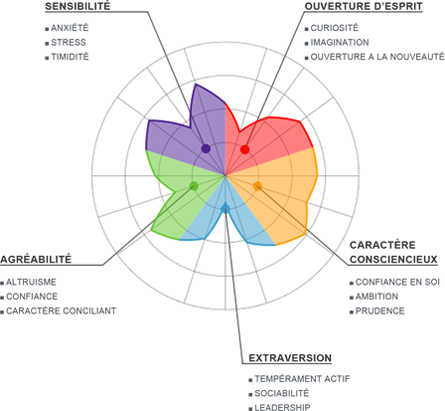
Nous utiliserons la base [PPLAPI](http://pplapi.com/), un réseau social factice composé de centaines de milliers de personnes, pour peupler notre monde.

[](http://pplapi.com/)

Regardons cette base d'un peu plus près. Chaque personne a un âge, une date de naissance, un lieu de résidence, une nationalité, une langue... Mais aussi cinq traits de personnalité :

* Ouverture
* Conscienciosité
* Extraversion
* Agréabilité
* Neuroticisme ou névocisme

Ces traits sont 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)). Vous le connaissez peut-être déjà ! Plusieurs grandes entreprises aiment donner des questionnaires de personnalité aux candidats afin de mieux cerner leur personnalité. Si vous répondez à un questionnaire qui reprend le modèle des Big Five, votre personnalité sera évaluée en 5 tendances séparées.



*Source : Vadequa*

Prenons par exemple **l'agréabilité**. Si votre personnalité est très "agréable", il sera important pour vous d'être serviable, généreux, "gentil". Vous pouvez faire passer autrui avant vous-même. Si vous êtes peu "agréable", vous serez plutôt fermé à la collaboration. Votre caractère peut être vu comme dur ou froid, peu enclin à la négociation.  
Ce sont ces tendances entre elles qui vont leur permettre de mieux vous cerner. Par exemple, si vous êtes peu ouvert, très consciencieux, peu extraverti, peu agréable et assez névrosé.

Dans la base PPLAPI, **chaque trait de personnalité est associé à un numéro**.

|  |  |
| --- | --- |
| 2 | Très supérieur à la moyenne |
| 1 | Assez supérieur à la moyenne |
| 0 | Moyenne |
| -1 | Assez inférieur à la moyenne |
| -2 | Très inférieur à la moyenne |

Les données de chaque agent sont disponibles sous différents formats, dont le 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}

Si cela ne vous dit rien, je vous invite à refaire [le cours en prérequis](https://openclassrooms.com/courses/demarrez-votre-projet-avec-python) !

**Fonctionnalités du programme**

Comment notre programme va-t-il nous aider à répondre à nos interrogations ? Voici les étapes qu'il devra suivre :

* Il ouvre un fichier JSON qui contient 100 000 agents,
* Il réalise des calculs,
* Il affiche un premier graphique affichant le degré d'agréabilité en fonction de la densité de population
* Quand je ferme ce graphique, un second apparaît avec les revenus moyens en fonction de l’âge.

**Etapes**

Nous allons réaliser ce programme en plusieurs étapes :

* 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

Puis on sort faire la fête pour célébrer la sortie de notre programme ! 🎉

**Quelques précisions**

Pour tirer pleinement parti de ce cours, vous devez être à l'aise avec les notions suivantes :

* syntaxe de base de Python, notamment les listes et les dictionnaires
* import de modules
* définition de fonctions, exécution et ajout de paramètres,
* lancement d'un script

Afin de faciliter votre apprentissage, j'utiliserai les notations suivantes :

class Agent

...

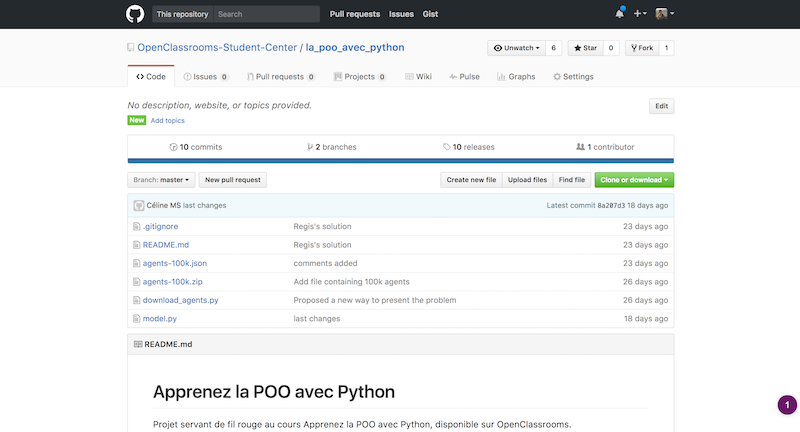
def \_\_init\_\_():

agreeableness = 0

Une partie du code peut être remplacée par trois petits points quand nous l'avons déjà vue précédemment. Dans ce cas, vous ne voyez que le code qui est essentiel.

📜 : Illustre un point de vocabulaire. Vous trouverez ici une définition sympa d'une notion un peu compliquée.

Le code que nous réaliserons ensemble est disponible librement sur un dépôt Github. Je vous indiquerai le lien en fin de chapitre.

[](https://github.com/OpenClassrooms-Student-Center/la_poo_avec_python)

**Challenges**

Vous pouvez lire le cours sans pratiquer… mais ce serait un peu comme si vous appreniez à faire du vélo en lisant un manuel. Vous avez beau connaître la théorie, vous ne serez pas plus avancé·e.

Chaque chapitre sera ponctué de petits exercices pratiques que vous pourrez réaliser directement dans notre console interactive. Entrez votre réponse et cliquez sur Run Code. Votre exercice est corrigé instantanément !

C'est le moment de vous entrainer !

## Découvrez la programmation orientée objet

La première manière de créer ce programme serait de créer des fonctions qui interagiraient les unes avec les autres. Nous pourrions, par exemple, commencer par créer une fonction qui récupèrerait les données du JSON, une autre qui les transformerait en agents, une troisième pour calculer les moyennes, et ainsi de suite. En fait, c'est déjà la démarche que nous avons adoptée dans le cours [Démarrez un projet avec Python](https://openclassrooms.com/courses/demarrez-votre-projet-avec-python).

Cette approche est très bien mais elle peut vite devenir très dense et difficile à maintenir. Il est très facile de se perdre entre les fonctions et de créer du code spaghetti : plein de variables dans tous les sens, de grandes fonctions qui font trop d'actions, et ainsi de suite. Lorsque vous ouvrirez votre projet dans 6 mois ou un an, vous risquez de ne plus vous y retrouver et de perdre du temps inutilement.

### La programmation orientée objet

Une autre approche est la Programmation Orientée Objet (communément appelée POO). Nous allons concevoir notre programme non pas comme un ensemble de fonctions mais en tant qu'ensemble d'objets qui interagissent les uns avec les autres. Chaque objet a un comportement bien à lui et des propriétés spécifiques.

Prenons un exemple que nous connaissons tous : la bataille navale ! Dans mon jeu de bataille navale, mon plateau est standard et tous mes bateaux font deux cases. Je sais donc que j'ai déjà deux grands types d'objets : une zone de bataille et un bateau.

Chaque bateau a une taille : il fait deux cases de long. Il a aussi une couleur (gris) et un poids. Ces caractéristiques sont appelés des **attributs**.

A un moment du jeu, je vais vouloir déplacer ce bateau. Dans la vraie vie, les matelots allumeraient certainement les moteurs du bateau pour le déplacer. C'est donc à l'ensemble du bateau de faire l'action et ils sont responsables du bon déroulement de la procédure. Chaque action réalisée par un objet est appelée une **méthode**.

Si je résume, la programmation orientée objet vous permet de découper un programme en concepts, appelés Objets, qui ont chacun des attributs et des méthodes.

📜

Un **objet** est une entité qui sert de conteneur à des données et qui contrôle également l'accès à ces dernières.

Chaque objet a un ensemble **d'attributs** (des variables spécifiques à l'objet) et de **méthodes** (des fonctions qui lui sont propres). Ces attributs et ces méthodes sont spécifiques à l'objet et ne fonctionneront pas s'ils sont appelés sur un autre objet.

### Analyse, design et programmation

#### L'Analyse Orientée Objet

L'orienté objet étant [un paradigme](https://fr.wikipedia.org/wiki/Paradigme), on ne commence pas par coder de but en blanc. Nous allons commencer par analyser notre programme : que va-t-il faire ? Quelles en sont les grandes composantes ? Quels sont les objets de notre programme et quels sont leurs caractéristiques ?

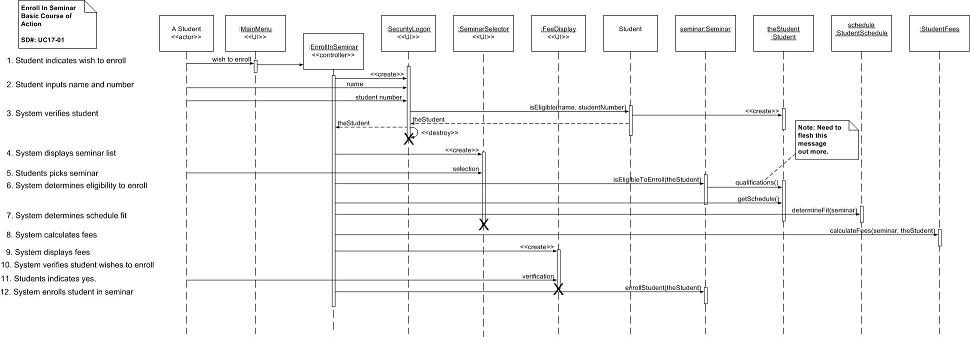
Cette phase est communément appelée l'Analyse Orientée Objet. Dans le cas de notre bataille navale, nous avons un premier objet Zone et un autre objet Bateau. Cela peut paraître tout simple pour notre jeu de bataille navale... Mais imaginez si vous deviez faire la même chose pour le site des impôts !

#### 

#### Le Design Orienté Objet

Quand nous avons déterminé les objets qui composent notre programme, nous nous intéressons à leurs interactions. Quelles sont les actions possibles dans notre programme ? Nous écrivons ainsi les méthodes relatives à chaque objet.

Un bateau va avoir, lui, plein de méthodes ! Une méthode  bouger , pour le déplacer, une méthode  toucher , une méthode  couler .  
Nous allons alors dessiner un diagramme de séquence pour matérialiser les interactions.



Exemple de diagramme de séquence

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

#### 

#### La programmation orientée objet

Vient enfin l'étape de code ! Nous allons transformer nos schémas en texte. 😎

Avant, il convient de refaire un petit schéma ! Mais cette fois-ci, nous ne nous concentrons plus sur les interactions mais bien la structure de notre programme. C'est le moment de définir les attributs de chaque objet et ses méthodes.

Dans le cas de notre bataille navale, une  Zone  a certainement bien des attributs différents : une largeur en nombre de cases, une hauteur en nombre de cases et certainement un index qui liste la position de chaque case. Je reviendrai sur ce point dans un chapitre ultérieur.

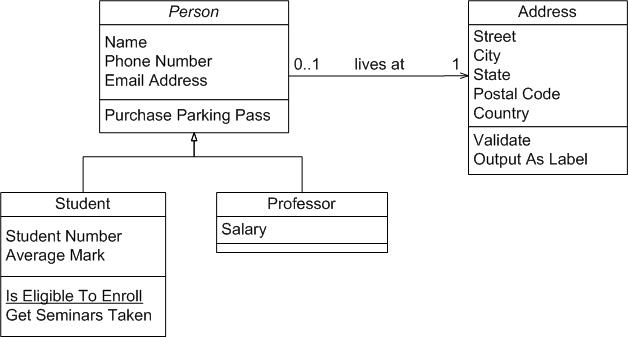
C'est en général à ce moment-là que nous réalisons un**diagramme de classe**. A quoi cela sert-il ? A bien comprendre la structure de chaque objet.

Pourquoi appelons-nous ce schéma un diagramme de classe ?

Nous avons vu qu'un objet est générique. C'est un peu le schéma d'un moule à gaufre sur papier. Quand le schéma est prêt, nous voulons produire des gaufres identiques, donc créer un moule. La classe est ce moule : comme une usine, elle permet la production d'objets qui partagent les mêmes attributs et les mêmes méthodes.

Nous verrons comment en créer une en Python un peu plus loin dans ce cours. En attendant, il est important de retenir qu'une classe a un nom, des attributs (variables qui lui sont propres) et des méthodes (fonctions qui lui sont propres).

Dans un diagramme de classe, chacune est représentée par un **rectangle**. La première case est le nom de la classe, la seconde liste ses attributs et la troisième les méthodes.



Un exemple de diagramme de classe

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

Maintenant que vous avez une connaissance générale de l'univers de la programmation orientée objet, structurons notre programme en dessinant un premier diagramme de classe.

En bref...

**Orienté Objet** : approche en programmation qui consiste à créer un programme composé d'éléments indépendants (les objets) qui sont chacun responsables de leurs propres attributs et méthodes.

**Classe** : "moule" permettant de dupliquer un objet. Chaque classe a ses propres attributs et méthodes. Exemple : la classe  Bateau ou la classe  Zone .

**Attribut** : caractéristique d'un objet. En programmation, il s'agit d'une variable qui n'existe qu'au sein d'une classe. Exemple :  largeur  pour le nombre de cases d'un bateau.

**Méthode** : action réalisée par un objet. En programmation, il s'agit d'une fonction qui n'existe qu'au sein d'une classe. Exemple :  toucher  pour attaquer un bateau ennemi.

## Structurez votre programme

Notre programme est comme un monde tout à fait vierge dans lequel il n'y aurait absolument rien. Nous devons apprendre à notre programme les concepts qui nous semblent, à nous, basiques afin qu'il puisse lui aussi les manipuler.

Imaginons que vous rencontriez [un extraterrestre](https://www.youtube.com/watch?v=Gdi50ZJnE9Y) et que vous vouliez jouer à la bataille navale avec lui. Vous lui montrez le jeu et il rêve d'en jouer (évidemment !). Avant de lui en expliquer les règles, vous allez commencer par lui expliquer ce que représentent les différentes pièces du jeu.

* Un pion rouge est un obus.
* Le plateau est la mer. Il est composé de plusieurs zones de jeu. Le plus petit bateau fait deux points de long et il est possible de le positionner en diagonale, donc on peut dire qu'une zone minimale de jeu est un carré de deux points de largeur et de deux points de hauteur. Indiquer la position d'une zone est simple : nous donnons l'abscisse et l'ordonnée.
* Un bateau est un objet qui avance sur l'eau, donc sur le plateau. Il a une longueur en nombre de zones.
* Un obus et un bateau ont une position. Ils partagent ce concept de position. Une position est composée de deux coordonnées : une abscisse, représentée par des nombres, et une ordonnée, représentée par des lettres.

### Objets

Bien bien ! A ce point de la discussion, notre visiteur venu d'ailleurs vous écoute avec attention. Vous continuez donc à discourir sur le jeu mais en divergeant sur votre programme. En fait, vous vous dites que ce jeu de bataille navale illustre parfaitement la manière dont vous modéliserez votre monde :

* un Agent est représenté par un pion rouge. Il a plusieurs attributs, dont une position.
* une Position est représentée par le placement d'un pion sur le plateau.
* une Zone est représentée par une zone minimale de jeu dans ma bataille navale.
* Les Graphiques ne sont pas dans le jeu de bataille navale mais il faut quand même y songer !

Nous avons nos objets ! Parlons maintenant des attributs de chacun.

### Attributs

#### Agent

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

#### Zone

Une Zone est un rectangle, elle a donc deux positions complémentaires : celle de son coin inférieur gauche et celle de son coin supérieur droit.

#### Position

Une position est un objet utilisé à la fois par un agent et par une zone. Elle a une latitude et une longitude.

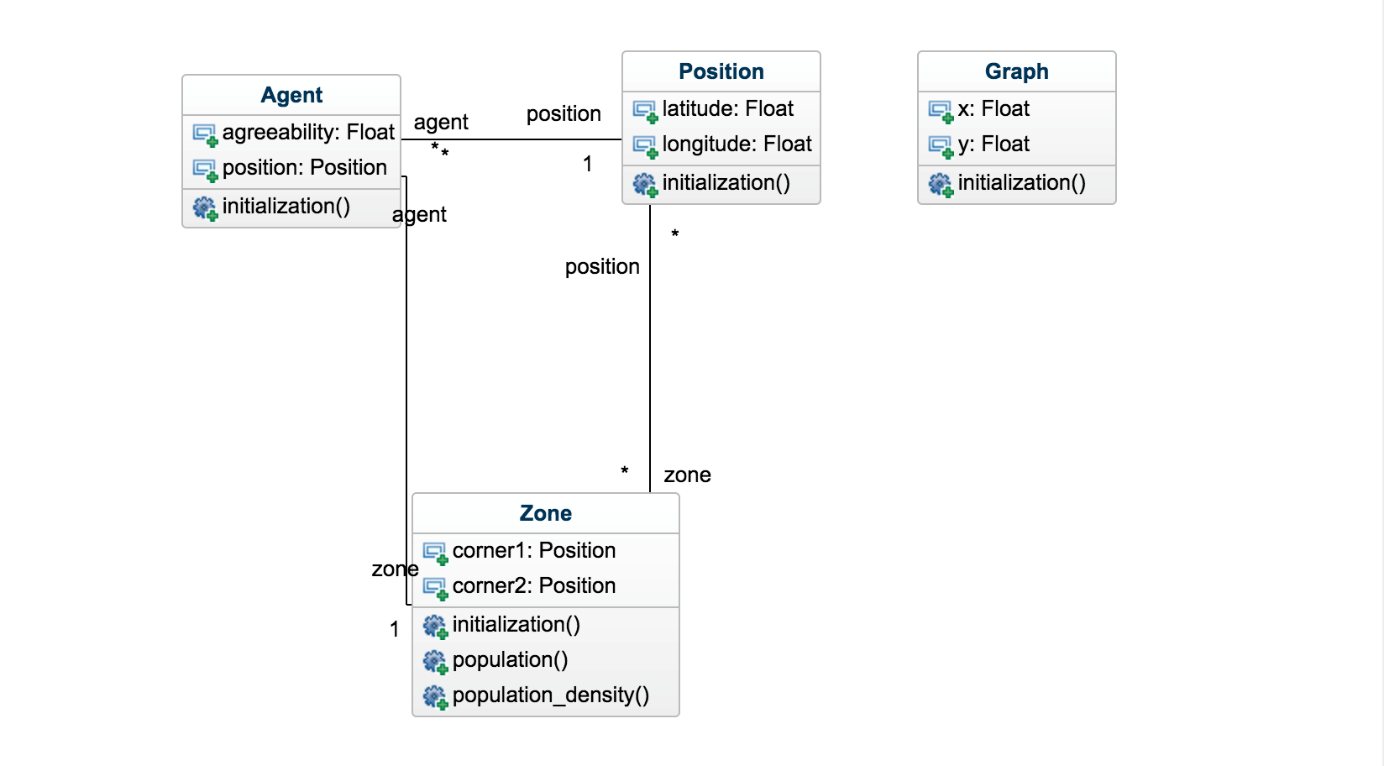
#### Graph

Enfin, un graphique a un ensemble de données ayant chacun une abscisse (axe x) et une ordonnée (axe y).

### Méthodes

Chaque objet a une méthode qui lui sert à générer de nouveaux objets sur même modèle. Cela peut paraitre abstrait pour le moment mais nous reviendrons sur cette notion un peu plus tard.

Une Zone doit calculer la densité de population. Pour cela, elle doit connaitre la population qui vit sur son périmètre ainsi que la surface en kilomètres carrés qui la compose. Nous pouvons donc déjà y déceler trois méthodes importantes :  population ,  area  et  population\_density .



Notre diagramme de classe

Ce diagramme de classe est incomplet. Dans une visée pédagogique, je n'y ai pas inclus toutes les méthodes et tous les attributs que nous utiliserons au fur et à mesure de ce cours.

Je vous invite à le compléter petit à petit ! J'ai utilisé l'outil [**GenMyModel**](https://app.genmymodel.com/) mais d'autres tout aussi efficaces existent.

Nous avons désormais nos objets bien en tête ! Passons à une étape essentielle : celle de la programmation !

# Structurez votre programme

Bravo ! Vous avez réussi cet exercice !

### Compétences évaluées

* Appréhender la programmation orientée objet

### Question 1

**Un diagramme de classe sert à...**

* + 

visualiser, en un schéma synthétique, les **attributs** et les **méthodes** de chacun de nos objets.

* + 

visualiser, en un schéma synthétique, les **interactions** entre nos objets.

### Question 2

**Un diagramme de séquence sert à...**

* + 

visualiser les **attributs** et les **méthodes** de chacun de nos objets.

* + 

visualiser les **interactions** entre nos objets.

### Question 3

**Un attribut est...**

* + 

une caractéristique d'un objet, par exemple la taille d'un bateau.

* + 

une fonctionnalité de mon programme, par exemple la possibilité de télécharger les résultats d'un jeu de bataille navale.

### Question 4

**Une méthode est...**

* + 

un guide commun à toutes les actions réalisées par mon programme et qui les aide à se coordonner.

* + 

une action réalisée par un objet, par exemple le déplacement d'un bateau de la case A6 à la case B12.

### Question 5

**En programmation orientée objet, un objet est...**

* + 

un ensemble d'atomes que je peux, moi, humain, déplacer de mes propres mains.

* + 

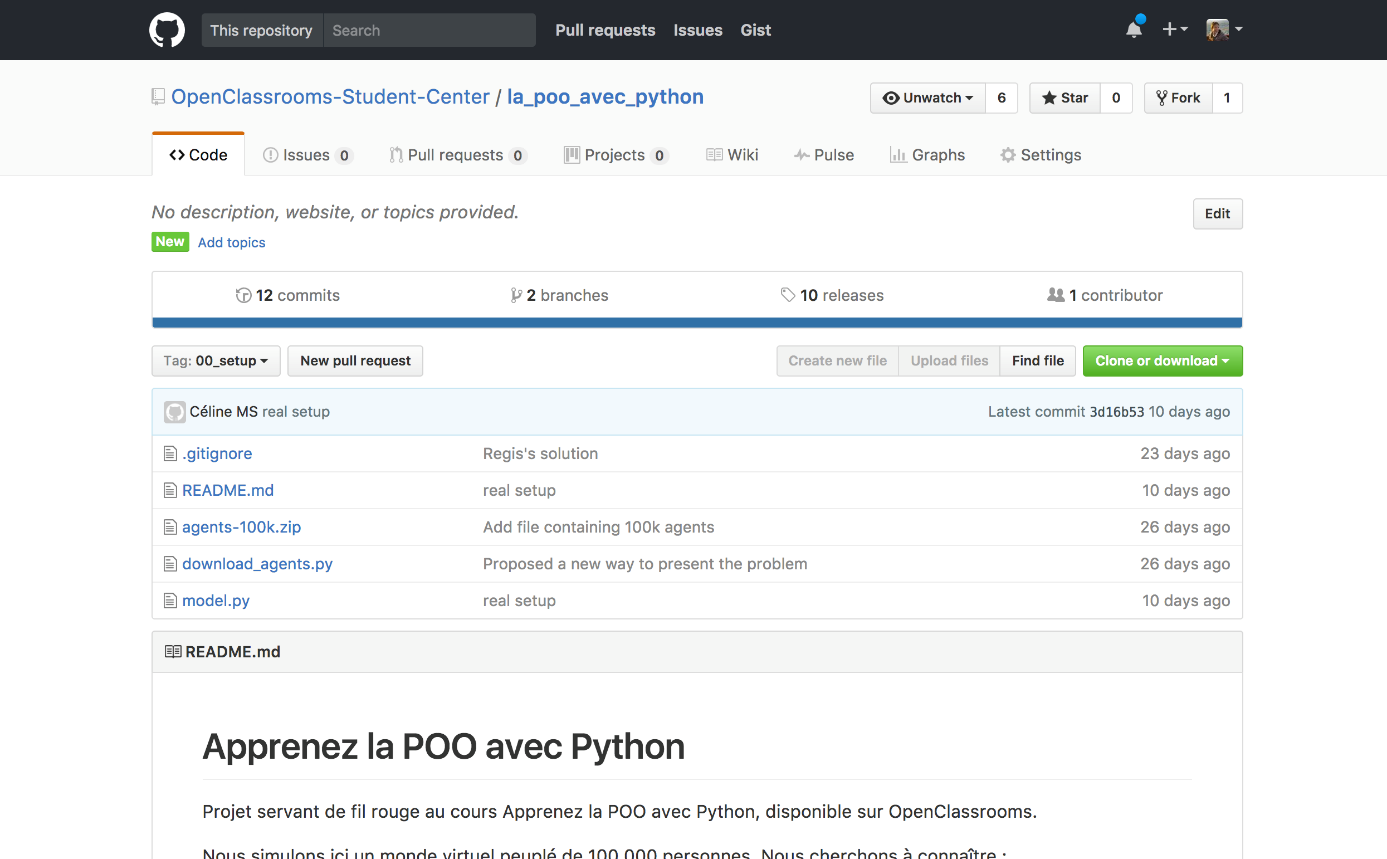
un concept qui représente un ensemble de données et qui en contrôle l'accès.

**Créez votre premier agent**

Nous y sommes ! Il est temps de coder ! Avant de vous montrer comment créer votre premier objet, il est essentiel que vous téléchargiez un dossier que Régis Behmo et moi avons préparé spécialement pour vous.

**Démarrer le projet**

Cliquez sur [ce lien](https://github.com/OpenClassrooms-Student-Center/la_poo_avec_python/tree/00_setup) (ou sur la photo), puis sur Clone or Download et enfin Download ZIP :

[](https://github.com/OpenClassrooms-Student-Center/la_poo_avec_python/tree/00_setup)

Ce dossier contient :

* agents-100k.zip  : 100 000 agents provenant de PPLAPI en format JSON (pensez à le dézipper !)
* model.py  : ce fichier vide hébergera votre script

Ouvrez l'ensemble avec votre éditeur de texte préféré et placez-vous dans model.py.

Bien, vous êtes prêts !

**Créer un agent**

Chaque objet est une "classe" en programmation.

Comment créer une classe ? En écrivant le mot-clé  class, son nom puis deux points. Comme ceci :

class Agent:

pass

pass  est un mot-clé qui permet d'indiquer à Python que vous souhaitez qu'il ne fasse rien. [**Un petit rappel ici**](https://openclassrooms.com/courses/demarrez-votre-projet-avec-python/ajoutez-un-peu-de-logique-avec-les-conditions#/id/r-4332072) ;-)

C'est tout ! Vous venez de créer une classe ! Facile, n'est-ce pas ?!

La classe est bien définie mais, pour le moment, nous n'en faisons rien. Créons un premier agent ! Pour cela, rien de plus simple : vous créez une variable puis vous l'assignez à votre classe. Comme ceci :

first\_agent = Agent()

Il existe plusieurs moyens de vérifier à quelle classe appartient une instance. Vous en connaissez d'ailleurs déjà une :  print()  !

print(first\_agent)

Si vous lancez le script, vous verrez que first\_agent appartient bien à la classe Agent.

<\_\_main\_\_.Agent object at 0x10e16a160>

Chaque objet créé à partir d'une classe est appelé une **instance de cette classe**. Chaque instance d'une classe est créée sur le même modèle, exactement comme les gaufres qui ont été faites dans le même gaufrier ! 😋

📜

Une **instance** est un objet créé à partir d'une classe. Elle partage avec les autres instances de sa classe les mêmes attributs et méthodes. Les valeurs de ces attributs varient en fonction des instances.

Vous pouvez créer autant d'instances que nécessaire.

Par exemple :

second\_agent = Agent()

print(first\_agent, second\_agent)

<\_\_main\_\_.Agent object at 0x10ca66160> <\_\_main\_\_.Agent object at 0x10ca66208>

**Challenge**

class Waffle:

    pass

sugar\_waffle=Waffle()

**Code du chapitre**

Retrouvez le code ici : <https://github.com/OpenClassrooms-Student-Center/la_poo_avec_python/tree/01_first_class>

## Customisez votre agent !

Dans ce chapitre nous allons "transformer" nos agents JSON en agents Python. C'est parti !

### Ajouter une méthode

Nous l'avons vu, une classe comprend des méthodes (fonctions d'une classe) et des attributs (variables d'une classe). Voyons tout de suite comment ajouter une méthode.

Nous allons faire en sorte qu'un agent nous dise bonjour. Hé oui, les gens sont polis dans notre monde !

 Modifions notre classe  en ajoutant une fonction à l'intérieur. Pensez bien à indenter !

class Agent:

def say\_hello(self, first\_name):

return "Bien le bonjour " + first\_name + "!"

Exécutons ensuite la méthode.

Si nous essayons de l'exécuter en écrivant son nom, comme s'il s'agissait d'une simple fonction, Python renverra une erreur. **Nous devons l'exécuter sur une instance.**

Kézako ?

Pour exécuter la méthode d'une classe, vous devez auparavant instancier une classe. Autrement dit, vous allez créer un nouvel objet à partir de cette dernière.

Puis vous exécutez la méthode en ajoutant un point, le nom de votre méthode, les parenthèses et d'éventuels paramètres. Comme ceci :

agent = Agent()

print(agent.say\_hello("Céline"))

"Bien le bonjour Céline !"

Et voilà ! Nous avons notre première méthode !

### Ajouter un attribut

Ouvrons le JSON et intéressons-nous aux données 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}

De quels attributs avons-nous vraiment besoin pour le moment ? Commençons par agreeableness.

Comment ajouter un attribut à notre instance ? Pour l'instant, en fait, nous n'avons aucune manière de le faire. Pour ajouter, ou modifier, un attribut, nous devons avant tout modifier légèrement la méthode responsable de la création d'une instance.

Je reprends. :)

Lorsque vous écrivez variable = nomdelaclasse(), Python utilise une **méthode spéciale** pour créer votre instance. On l'appelle un **constructeur** car, comme son nom l'indique, la méthode a pour responsabilité de construire une nouvelle instance à partir de votre classe.  Heureusement, vous n'avez pas à créer cette méthode. Elle est déjà écrite en Python et vous n'avez qu'à l'appeler !

Cette méthode est \_\_init\_\_(). Un petit moyen mnémotechnique pour s'en souvenir est que cette méthode sert à initialiser une instance, d'où le nom "init"...

Par défaut, le constructeur crée une instance **sans attribut**. C'est-à-dire qu'il va créer un agent qui pourra bel et bien utiliser les autres méthodes de la classe mais qui n'aura aucun attribut. Pour les ajouter, dans ce cas, il faut le faire une fois que l'instance est créée.

Sauf que nous, nous voulons que tous nos agents aient, par défaut, certains attributs obligatoires comme le degré d'agréabilité ou leur position. Comment faire ? Nous allons modifier le constructeur en l'écrivant dans notre classe, comme nous l'avons déjà fait pour notre première méthode.

Commençons !

class Agent

...

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

agreeableness = 0

Ces deux underscores sont utilisés assez souvent ! Ils sont appelés **dunders**.

Néanmoins, si nous faisons cela, nous ajoutons une variable à notre classe mais nous ne l'associons pas vraiment à l'instance que nous sommes en train de créer.

L'instance est représentée par le mot-clé self que nous passons en paramètre. Si nous reprenons la métaphore de la gaufre, la classe est le moule et  self  est la gaufre qui est en train de cuire dans le moule. Pour y associer un attribut, nous utiliserons un point, comme ceci :

def init(self):

self.agreeableness = 0

Si nous relançons notre script, nous voyons que nous avons bien accès à cet attribut :

class Agent:

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

self.agreeableness = 0

first\_agent = Agent()

print(first\_agent.agreeableness)

~/ python my\_model.py

0

### 

### Modifier des attributs

C'est un bon début !

Le problème est que nous avons un attribut par défaut (une agréabilité de 0), et que nous ne pouvons pas la faire varier d'une instance à l'autre. Si nous restons ainsi, tous nos agents auront la même agréabilité. Or, ce serait bien de faire passer la valeur de l'attribut en paramètre, non ? Par exemple, en écrivant  first\_agent = Agent(0).

Comment faire ? De la même manière que vous le feriez avec une fonction classique : en créant un paramètre et en y associant une valeur lors de l'exécution la fonction. Démonstration :

class Agent:

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

self.agreeableness = agreeableness

first\_agent = Agent(0)

print(first\_agent.agreeableness)

### Challenge

class Waffle:

  def \_\_init\_\_(self):

      self.sugar=2

sugar\_waffle = Waffle()

print(sugar\_waffle.sugar)

Nous savons désormais comment ajouter et modifier des attributs. Bien ! Mais vous vous doutez bien que nous n'allons pas réaliser la même opération pour chaque agent de notre JSON. Comment faire ? Voyons cela dans le prochain chapitre !

## Ajoutez des attributs à partir d'un dictionnaire

Nous avons fait une grande partie. Bravo ! Maintenant, comment "transformer" dynamiquement un élément JSON en instance de notre classeAgent ?

Si nous ouvrons notre JSON, nous voyons que chaque élément est structuré comme un dictionnaire, c'est-à-dire qu'il a des valeurs associées à chaque clé :

{"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}

Ce serait bien de donner ce dictionnaire en paramètre lors de l'initialisation d'une instance pour que chaque clé se transforme automatiquement en nouvel attribut. Nous pourrions le faire manuellement, autrement dit indiquer un à un le nom des attributs, mais cela va être fastidieux. Je vais vous montrer comment le faire automatiquement.

Ne faites pas de copier-coller de ce code avant de lire la suite. ;-)

### Passer un dictionnaire en paramètre

Pour l'instant, créons un nouveau dictionnaire et passons-le en paramètre. Nous verrons plus tard comment utiliser les données de notre fichier JSON.

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)

false  est à indiquer entre guillemets ! Sinon il sera interprété comme une variable par Python et non comme une chaîne de caractères.

Ça fonctionne ! \o/

Afin de connaître tous les éléments d'un dictionnaire, utilisons la méthode items() :

def \_\_init\_\_(self, agent\_attributes):

print(agent\_attributes.items())

Puis utilisons cette même méthode à l'intérieur d'une boucle afin de récupérer le nom et la valeur de chaque clé :

def \_\_init\_\_(self, agent\_attributes):

for attr\_name, attr\_value in agent\_attributes.items():

# set attribute

Enfin, utilisons une méthode qui s'appelle  setattr()  et qui prend en paramètre l'instance, donc self, le nom de l'attribut et la valeur de l'attribut. Comme ceci :

def \_\_init\_\_(self, agent\_attributes):

for attr\_name, attr\_value in agent\_attributes.items():

setattr(self, attr\_name, attr\_value)

La méthode setattr()est équivalente au code suivant :

my\_object.attribute = value

Il s'agit simplement d'une autre notation.

Dans notre cas, utiliser cette méthode est bien plus utile car nous nous situons à l'intérieur d'une boucle (nous souhaitons utiliser les variablesattr\_nameetattr\_value).

### Varier de nombre d'arguments

Python vous donne la possibilité d'avoir un **nombre variable d'arguments** dans une fonction. C'est très pratique lorsque vous passez à cette dernière une suite d'éléments (liste, dictionnaire, ...).

#### Passer une suite d'éléments sans clé

Imaginons que vous souhaitez passer en paramètre un nombre variable d'éléments (une suite de villes, de températures, ...). Vous ne savez pas à l'avance combien de paramètres il y aura mais vous voulez tout de même les utiliser dans une fonction. Vous ajoutez dans ce cas **une étoile** avant la définition du paramètre.

Les éléments que vous passerez en paramètre seront transformés en **tuples**.

Par exemple :

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'>

Et si nous souhaitons passer en paramètre des éléments qui ont une clé afin de les utiliser en tant que **dictionnaire** dans une fonction ? Nous allons dans ce cas ajouter **deux étoiles**. Par exemple :

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'>

Cette notion n'est pas évidente à comprendre. N'hésitez pas à lancer votre console et à tester jusqu'à ce que vous ayez bien compris !

Comment allons-nous utiliser ce nouveau concept dans notre code ? De la manière suivante :

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

À présent, voyons comment récupérer les agents JSON et les transformer en agents Python. Nous allons utiliser le module JSON que vous connaissez déjà puisque nous l'avons utilisé dans le cours Démarrez votre projet avec Python !

Dézippons le fichier agents-100k.zip en tapant dans la console la commande suivante :

unzip agents-100k.zip

Importons le module :

import json

Puis chargeons les agents et transformons-les :

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)

Nous y sommes presque !

Je commence à avoir beaucoup de commandes différentes qui s'exécutent quand je lance mon script. Vous aussi ? C'est bien ce que je me disais… Une bonne pratique est de les regrouper en une grande fonction, main(), qui sera lancée par le script. Ainsi nous savons exactement ce qui sera lancé !

Créons donc une nouvelle fonction et exécutons-la à la toute fin de notre programme :

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()

### Challenge

class Waffle:

  def \_\_init\_\_(self, \*\*toppings):

    for cle, valeur in toppings.items():

        setattr(self, cle, valeur)

toppings = {"sugar": 2, "maple\_syrup": 1, "whipped\_cream": 1}

fat\_waffle = Waffle(\*\*toppings)

print(fat\_waffle.sugar)

### Code du chapitre

Retrouvez le code ici : <https://github.com/OpenClassrooms-Student-Center/la_poo_avec_python/tree/02_class_attributes>

**Créez la classe Position**

Pour calculer la densité de population, nous devons connaître la position d'un agent sur une carte. Cela suppose que notre programme connaît déjà ces concepts ! Or pour le moment il ne connaît que le concept Agent (le pion rouge de notre jeu de touché/coulé).

**La classe Position**

Créons donc une nouvelle classe qui nous servira à calculer les positions à la fois des Agents et des coins de nos futures zones.

class Position:

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

Notre classePosition a deux attributs : une longitude et une latitude.

class Position:

def \_\_init\_\_(self, longitude, latitude):

self.latitude = latitude

self.longitude = longitude

Ensuite, créons une nouvelle instancePosition composée de la latitude et de la 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)

Enfin, nous mettons à jour 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():

...

Si nous lançons notre programme, nous constatons qu'il fonctionne. Nickel !

**Des degrés aux radians : les propriétés**

Sauf que... si la vie était simple, cela se saurait ! Les coordonnées de chaque agent sont en degrés. Or, en mathématiques, il est préférable d'utiliser des radians. Il s'agit tout simplement d'une unité de mesure différente qui sera plus pratique à utiliser.

Pour en savoir plus sur ce choix, je vous conseille de lire [**cet excellent article**](https://mathwithbaddrawings.com/2013/05/02/degrees-vs-radians/) !

La longitude en radians est égale à sa longitude en degrés \* pi divisé par 180. Soit le calcul suivant :

longitude = longitude\_degrees \* pi / 180

Pourquoi ne pas effectuer ce calcul dans notre méthode \_\_init\_\_() ? Car cela reviendrait à y conserver à la fois les données en radians et en degrés ce qui, vous l'avouerez, n'est pas très pratique pour les utiliser ailleurs.

Dans un monde idéal, j'aimerais effectuer ce calcul dans une nouvelle méthode mais y accéder comme s'il s'agissait d'un attribut. Le [dictateur bienveillant à vie](https://fr.wikipedia.org/wiki/Guido_van_Rossum) ayant pensé à tout, Python nous offre une manière très pratique de faire cela : les **propriétés** !

Une propriété est une méthode qui est accessible comme un attribut ! Vous ne voyez pas bien ce que je veux dire ? On va faire une démo !

Commençons par écrire une méthode qui effectue le calcul en question.

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

Nous avons besoin de changer le nom des attributs pour les adapter aux besoins de notre nouvelle méthode. L'attribut latitude devient latitude\_degrees afin que ce soit plus clair et l'attribut longitude devient longitude\_degrees.

Si nous voulons accéder à la longitude, nous devons écrire position.longitude(). Or nous voulons plutôt écrire position.longitude, afin que la longitude soit utilisée comme un attribut et non comme une méthode ! Cela peut vous paraître anodin mais, croyez-moi, c'est important ! En effet, vous ne voulez pas que votre instance fasse une action, ce qui correspond plutôt à une méthode, mais qu'elle vous renvoie une valeur qui lui est propre. Nous utilisons donc une propriété.

Pour créer une propriété nous ajoutons, juste avant la définition de la méthode, le mot-clé @property.

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

Nous pouvons maintenant y accéder en écrivant position.longitude :

def main():

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

...

print(agent.position.longitude)

Bien bien !

**Le nombre Pi**

Intéressons-nous maintenant au calcul en lui-même. Le nombre pi étant très, très, très long à écrire, nous allons utiliser le module math de Python qui contient la variable pi.

Nous importons le module :

import math

Puis utilisons la variable math.pi :

return self.longitude\_degrees \* math.pi / 180

Nous pouvons en faire de même pour la latitude :

@property

def latitude(self):

# Latitude in radians

return self.latitude\_degrees \* math.pi / 180

**Code du chapitre**

Retrouvez le code de ce chapitre ici : <https://github.com/OpenClassrooms-Student-Center/la_poo_avec_python/tree/03_properties>

# Concepts clés

Bravo ! Vous avez réussi cet exercice !

### Compétences évaluées

* Maîtriser les concepts de la programmation orientée objet

### Question 1

**Une classe a des attributs et des méthodes.**

* + 

Vrai

* + 

Faux

### Question 2

**Une classe est :**

* + 

une salle dans laquelle se réunissent les élèves apprenant à coder.

* + 

la représentation d'un objet en programmation.

### Question 3

**Pour créer la classe Fruit, vous écrivez :**

* + 
  + class Fruit:

* + 
  + classe fruit():

* + 
  + classe Fruit:

* + 
  + class Fruit()
  + end

### Question 4

**Créer l'instance d'une classe s'écrit ainsi :**

* + 

banane = fruit

* + 
  + banane = Fruit

* + 
  + banane = Fruit()

### Question 5

**Vous pouvez créer autant d'instances que vous le souhaitez.**

* + 

Vrai

* + 

Faux

## Créez votre première Zone !

Nous avons les agents, la position de chaque agent et globalement la possibilité de calculer une position. Bien bien ! Il nous faut maintenant créer le plateau de jeu, donc l'ensemble de zones qui constituent notre monde.

### La classe Zone

Créons la classe Zone qui a trois attributs : la position de son coin inférieur gauche, la position de son coin supérieur droit et ses habitants.

Lorsque nous créerons chaque zone, nous déterminerons sa position et disons que, par défaut, la zone est vide.

class Zone:

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

self.corner1 = corner1

self.corner2 = corner2

self.inhabitants = 0

Parfait. Maintenant, comment allons-nous créer toutes les zones qui constituent le monde ? Nous allons le voir comme un quadrillage qui est composé de plusieurs lignes verticales et horizontales.

#### Les attributs de classe

Si nous reprenons notre jeu de bataille navale, nous voyons que nous avons plusieurs lignes. Une ligne a une abscisse minimale, une abscisse maximale et un intervalle distinguant chaque zone minimale de jeu.

Autrement dit, toujours dans le jeu, nous avons 9 zones comprises entre 0 et 9 avec un intervalle de 1 pour cette première ligne. Nous allons utiliser la même logique dans notre grille mais pas les mêmes valeurs, puisque la longitude est comprise entre -180 et 180 degrés et que la latitude, elle, est comprise entre -90 et 90 degrés.

Pour créer la première ligne de notre grille, nous devons donc définir certaines variables qui, a priori, ne changeront pas. Nous voulons qu'elles soient associées à la classe et non pas aux instances. En effet, si nous changeons l'abscisse minimale de nos zones, nous voulons le faire pour toutes les instances en une fois. Nous pouvons raisonnablement nous dire que le champ d'action de ces variables sera plus large que celui des attributs d'instance.

Nous appelons ces variables des **attributs de classe** car elles appartiennent à la classe dans son ensemble, et non à l'instance. C'est notre zone de manière générale qui a une latitude minimale et une latitude maximale, et non l'instance !

De manière générale, nous faisons référence aux attributs d'instance lorsque nous parlons d'attributs. Mais nous utilisons toujours l'expression "attribut de classe" quand l'attribut s'applique sur la classe.

#### 

#### Ajouter un attribut de classe

Par convention, un attribut de classe s'indique en majuscules, avant les méthodes :

class Zone:

MIN\_LONGITUDE\_DEGREES = -180

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

self.corner1 = corner1

self.corner2 = corner2

self.inhabitants = 0

Vous pouvez par la suite y accéder en écrivant MaClasse.MON\_ATTRIBUT\_DE\_CLASSE. Exemple :

print(Zone.MIN\_LONGITUDE\_DEGREES)

Vous pouvez également y accéder via une instance de la classe. Par exemple :

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

...

longitude = self.MIN\_LONGITUDE\_DEGREES

Indiquons tous les attributs dont nous aurons besoin pour notre première ligne :

class Zone:

MIN\_LONGITUDE\_DEGREES = -180

MAX\_LONGITUDE\_DEGREES = 180

WIDTH\_DEGREES = 1 # degrees of longitude

Nous allons ensuite créer une méthode qui va initialiser la grille, en commençant par la longitude.

def initialize\_zones(self):

📜

Un **attribut de classe** est une variable dont le champ d'action s'étend à l'ensemble d'une classe. Ils sont très utilisés pour compter le nombre d'instances d'une classe par exemple.

Par convention, ils sont définis en majuscules et au début de la classe.

#### Créer les lignes avec la longitude

Nous créons une première boucle qui crée une zone pour chaque nombre compris entre MIN\_LONGITUDE\_DEGREES et MAX\_LONGITUDE\_DEGREE.  
La méthode range() permet justement de créer une liste à partir d'une valeur minimale, d'une valeur maximale et d'un 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)

Chaque instance de type Zone prend en paramètre la position de son coin inférieur gauche et la position de son coin supérieur droit. Dans notre cas, il s'agit de la toute première ligne. La latitude est donc toujours égale à 1 ! Quant à la longitude de départ, nous y avons déjà accès dans notre 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)

Comment calculer la position du coin supérieur droit ? En ajoutant un intervalle ! Nous l'avons d'ailleurs déjà stocké dans l'attribut de classe 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)

Néanmoins, il nous manque l'intervalle utilisé pour la latitude. Ajoutons-le :

class Zone

MIN\_LONGITUDE\_DEGREES = -180

MAX\_LONGITUDE\_DEGREES = 180

WIDTH\_DEGREES = 1 # degrees of longitude

HEIGHT\_DEGREES = 1 # degrees of latitude

...

Nous pouvons désormais calculer également 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)

Paaaaarfait ! Nous allons ensuite créer une seconde boucle, sur le même principe, pour effectuer la même opération mais au niveau de la latitude ! Une boucle à l'intérieur d'une boucle, comme dans [Inception](https://www.youtube.com/watch?v=B4nIVh1yvvc) :)

#### 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(longitude, latitude)

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(longitude, latitude)

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(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)

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>

## Ajoutez 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)

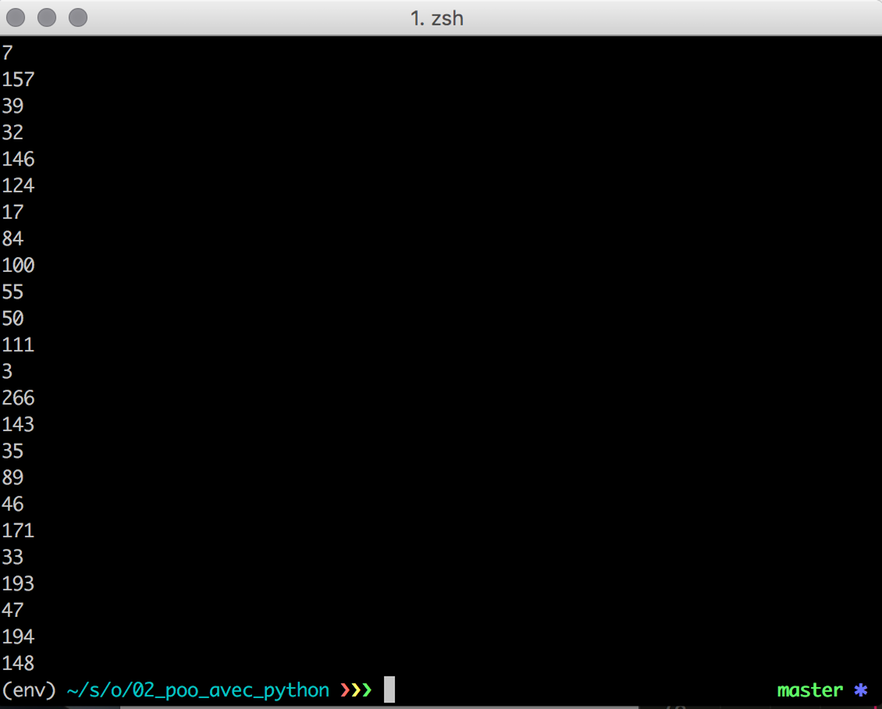
agent = Agent(position, \*\*agent\_attributes)

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>

## Comprenez l'encapsulation

Je ne vous ai pas encore parlé d'un principe fondamental en POO : celui de **l'encapsulation**.

### What else?

Pour bien le comprendre, prenons l'exemple des capsules à café. (Oui oui, nous allons bien parler d'encapsulation grâce à des capsules... J'ai un humour bien particulier mais je me soigne.)

#### Prenez un café, Emile

Comment faisait-on le café dans les temps anciens quand l'électricité n'était pas si répandue ? On prenait du café en grain, on en faisait de la poudre grâce à un moulin à café, puis on le mettait dans la cafetière, on tassait bien, on ajoutait l'eau et on posait sur le feu.

Que les amateurs de café me pardonnent : cela demande du temps que la plupart des consommateurs préfèrent allouer à d'autres activités. La majorité des gens n'a en réalité pas envie de passer du temps à faire un café. En fait, ils ne veulent même pas savoir comment il est fait. Connaître vaguement le principe des machines leur suffit. Ce qu'ils veulent est le résultat : un café pour se réveiller.

C'est pourquoi une entreprise, incarnée par un ancien acteur d'Urgences, a inventé le principe des capsules et de la machine. C'est tout simple : pour obtenir un café, vous n'avez qu'à insérer une capsule et à appuyer sur un bouton. Et à mettre de l'eau froide. Point.

Toute la logique de la conception du café est déléguée à la machine. Cette capsule renferme du café déjà moulu, conservé à bonne température, prêt à être dégusté.

C'est la même logique avec les classes. Une classe est comme une capsule : elle contient en elle-même tout ce dont elle a besoin pour fonctionner. Lorsque nous l'utilisons, une partie de son fonctionnement nous est caché afin de ne nous donner accès qu'aux fonctions essentielles pour nous.

📜

**L'encapsulation** est un mot qui illustre deux concepts :

* le fait que 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.

#### Si notre machine était un programme informatique...

Continuons notre exemple. Imaginons une classe CoffeeMachine. Lorsque nous appuyons sur le bouton Tasse de Café, la machine s'allume, fait chauffer l'eau puis nous verse un café. Nous pouvons donc dire que la méthode \_\_init\_\_() va faire appel à trois autres méthodes : demarrer\_machine(), bouillir\_eau() et faire\_cafe().

Notre machine ne nous donne pas de bouton pour faire bouillir l'eau. La méthode bouillir\_eau() est donc complètement **secrète**, accessible uniquement de l'intérieur du programme et pas par nous, pauvres mortels que nous sommes. Nous appelons cela des éléments **privés**.

Si nous regardons derrière la machine, nous voyons un bouton pour l'allumer. Je suppose qu'en appuyant dessus cela va déclencher la méthode demarrer\_machine(). Il a fallu que je cherche un peu pour la trouver, elle n'est pas évidente. En soi, on ne l'utilise pas vraiment de l'extérieur puisqu'on appuie plutôt sur le bouton Tasse de Café pour faire un café. Mais la machine nous donne quand même la possibilité de le faire, au cas où. Nous appelons cela des éléments **protégés**. Vous pouvez y accéder en dehors de la classe, mais on l'utilise plutôt à l'intérieur de la classe.

Démonstration !

Créons une classe CoffeeMachine qui a trois 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!

Son niveau d'eau maximum est de 100. Imaginons que la machine 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()

La console affiche :

boiling...

Coffee is ready!

boiling...

Coffee is ready!

boiling...

Coffee is ready!

boiling...

Coffee is ready!

Please add water!

#### Méthodes privées ou protégées

Bien. Vous serez d'accord avec moi pour dire que je ne dois pas pouvoir exécuter la méthode boil\_water() hors de la classe. Comment faire ? Nous allons ajouter des underscores pour transformer nos méthodes en méthodes privées ou protégées.

Une méthode protégée est accessible à l'intérieur d'une classe mais ne doit pas être aisément accessible de l'extérieur. Vous ajoutez pour cela un underscore au début du nom.

Ainsi :

class CoffeeMachine:

...

def \_start\_machine(self):

# Start the machine.

if self.water\_level > 20:

return True

else:

print("Please add water!")

return False

Si vous l'appelez 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'

Pour continuer à utiliser cette méthode, ajoutez le underscore.

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!")

Et pour les méthodes privées ? Vous ajoutez deux underscores ! Ainsi :

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!")

#### Tout est public

Je ne sais pas pour vous, mais lorsque je lis cela, je me dis : si j'essaie d'accéder à ces méthodes hors de la classe, je devrais avoir une erreur. Mon ordi va exploser ou Python va me dire de rentrer chez moi et de me faire une tisane. Mais j'aime le danger ! Je vais essayer.

Je vais commenter la boucle et ajouter trois nouvelles lignes :

...

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())

Normalement, la première devrait bien afficher "boiling..." et "Coffee is ready". La seconde devrait afficher "Start Machine" et la troisième ligne 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'

Seule la troisième ligne renvoie bien une erreur ! La méthode "protégée" est bien accessible de l'extérieur. Top. :)

Mais je vais vous montrer une petite astuce : on peut bel et bien accéder à cette méthode privée. Tout est question de syntaxe ! En écrivant \_MaClasse\_\_methode\_privee(), je peux quand même l'exécuter. Démonstration :

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...

Certains langages sont très stricts au sujet de l'encapsulation. Vous ne pouvez vraiment, vraiment pas accéder à des méthodes privées en dehors de la classe. Python est plus laxiste. La philosophie du langage est plus axée sur la confiance : si nous voulons accéder à un élément protégé ou privé c'est que nous avons une bonne raison de le faire.

D'ailleurs, une phrase du fondateur de Python, Guido van Rossum, est fréquemment citée à ce propos : "We are all consenting adults here". Elle signifie : "Nous sommes tous des adultes consentants". Sous entendu : si vous voulez vous tirer une balle dans le pied, allez-y, vous êtes adulte après tout.

C'est pourquoi Python utilise cette notion d'encapsulation et d'éléments privés mais à travers une convention de syntaxe. Tout est public en Python, néanmoins il vous sera plus difficile d'accéder à des éléments considérés comme privés. Il faut vraiment, vraiment le vouloir !

Sans entrer dans le détail, cette fonctionnalité du langage est très pratique lorsque vous importez un module et que vous souhaitez légèrement l'adapter à vos besoins.

Je vous conseille la lecture de ce chapitre de l'excellent 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.

### L'encapsulation appliquée à notre programme

À présent que vous avez bien compris le principe d'encapsulation, revenons à notre programme d'origine et modifions-le légèrement.

L'initialisation des zones devrait se faire de l'intérieur de la classe, et non de l'extérieur. Néanmoins, je dois pouvoir quand même le faire en cas de besoin. Je vais donc "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)

Après tout, nous n'avons pas envie de savoir s'il faut initialiser la grille ou pas hors de la zone. C'est une vérification qui devrait être faite à l'intérieur. Étant donné que nous exécutons la fonction find\_zone\_that\_contains(position), je vais 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

Bravo ! ‌

Réfléchissons dans le prochain chapitre aux différents moyens de calculer l'agréabilité d'une zone.

### Code du chapitre

Retrouvez tout le code de ce chapitre ici : <https://github.com/OpenClassrooms-Student-Center/la_poo_avec_python/tree/06_encapsulation>

# Concepts avancés

Bravo ! Vous avez réussi cet exercice !

### Compétences évaluées

* Maîtriser les concepts de la programmation orientée objet

### Question 1

**Une méthode est...**

* + 

une fonction spécifique à une classe.

* + 

une fonction comme une autre.

### Question 2

**Un attribut est :**

* + 

une variable comme les autres.

* + 

une variable spécifique à une classe.

### Question 3

**Une propriété est...**

* + 

une méthode que vous pouvez exécuter en écrivantinstance.methode(propriete).

* + 

une méthode que vous pouvez exécuter en écrivantinstance.propriete.

### Question 4

**Pour convertir des degrés en radian, il faut utiliser la formule suivante :**

* + 

radian = degrés \* 4 / 180

* + 

radian = degrés \* pi / 180

* + 

radians = degrés / pi / 180

### Question 5

**Selon le principe de l'encapsulation, un objet ne présente aux autres objets du programme que les attributs et les méthodes nécessaires à leurs interactions. Certaines méthodes restent privées et ne doivent être utilisées qu'à l'intérieur d'une classe.**

* + 

Vrai

* + 

Faux

*Les méthodes privées sont pensées pour une utilisation en interne, autrement dit dans une classe et non pas au niveau d'une instance. En Python, il n'est pas recommandé d'accéder à une méthode privée ou protégée au niveau de l'instance, même si le langage le permet.*

## 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

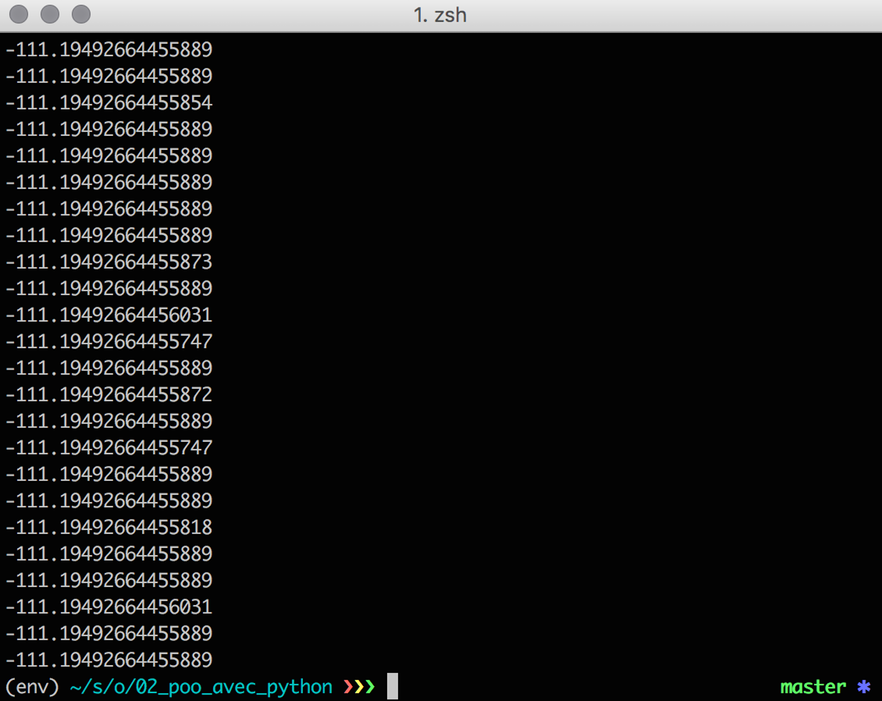
...

@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:

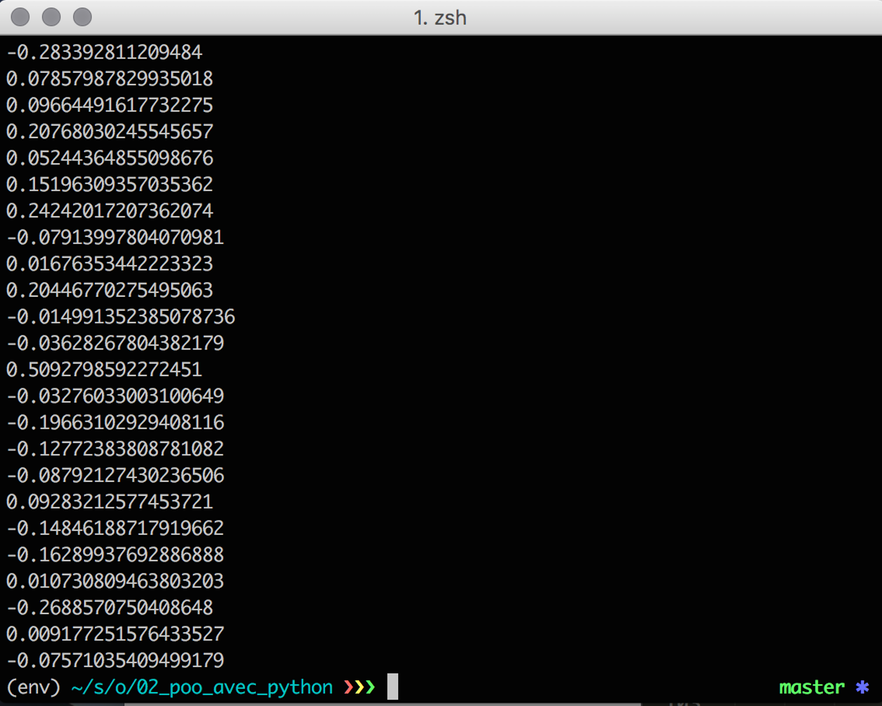
return 0

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>

**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 classe Vampire pourrait hériter de la classe Humain et de la classe [Hématophage](http://www.larousse.fr/dictionnaires/francais/h%C3%A9matophage/39450?q=h%C3%A9matophage#39369) (et de PersonnageNiaissi 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**

[Réalisez le challenge de ce chapitre en cliquant sur ce lien.](https://www.codevolve.com/api/v1/publishable_key/2A9CAA3419124E3E8C3F5AFCE5306292?content_id=9c28d9bc-1405-4254-96e3-3a2f99e9c810)

### Code de ce chapitre

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

## 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))). 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.

# La programmation orientée objet - conclusion

Bravo ! Vous avez réussi cet exercice !

### Compétences évaluées

* Maîtriser les concepts de la programmation orientée objet

### Question 1

**JSON est :**

* + 

l'abréviation de Jason Sollicite la toison d'Or Numérique.

* + 

l'abréviation de Javascript Object Notation. Un format de données qui permet de représenter des informations de manière structurée.

* + 

l'abréviation de Javascript Open Notation. Une **plateforme** pour partager des projets Open Source et les commenter.

### Question 2

**Une valeur absolue est :**

* + 

Une valeur qui est toujours négative.

* + 

Une valeur qui est toujours positive ou égale à zéro (donc jamais négative).

* + 

Une valeur qui est toujours positive (donc jamais négative ou égale à zéro).

### Question 3

**Une list comprehension s'utilise de la manière suivante :**

* + 
  + my\_list = [human.height for human in humans]
  + 
  + my\_list = (human.height for human in humans).to\_list
  + 
  + my\_list = [human.height, human for all humans]
  + 
  + my\_list = [for human in humans, human.height]

### Question 4

**Un attribut de classe est...**

* + 

un attribut associé à la classe et utilisable par toutes les instances de cette dernière.

* + 

un attribut associé à une instance et utilisable par les autres instances de la même classe.

* + 

un super-attribut, utilisable par toutes les classes de mon programme.

### Question 5

**Une méthode de classe est...**

* + 

une super-fonction, utilisable par toutes les classes de mon programme.

* + 

une méthode associée à une classe. Elle s'appelle sur la classe et non sur l'instance, comme ceci :

France.population\_totale()

* + 

une méthode associée à une instance. Elle s'appelle sur l'instance et non sur la classe, comme ceci :

france = France()

france.population\_totale()