ENSG Géomatique

ÉCOLE NATIONALE DES SCIENCES GÉOGRAPHIQUES

Cycle ingénieur
Master géomatique
1ère année

Programmation orientée objet

avec python

Marie-Dominique Van Damme

2023-2024

Table des matières

Planning

mercredi 28/02 matin	~ Paradigme de la POO ~ Classes, objets, constructeurs ~ Exo du triangle
mardi 5/03 matin	~ Surcharge de méthodes ~ Exos
mardi 5/03 après-midi	~ Exo banc de poisson
mardi 12/03 matin	~ Héritage, encapsulation, polymorphisme ~ Exos
vendredi 15/03 matin	~ Classes abstraites ~ Exos
vendredi 22/03 matin	~ Exos stratégies de calcul de parcours
lundi 25/03 matin	~ Qt ~ Dessin, composants graphiques ~ Évènement
vendredi 29/03 après-midi	~ **TP Noté** ~ sans internet - cours et exos autorisés

Table 1 – Programme prévisionnel

1 Exercice de programmation orientée objet

1.1 Point

Définir une classe *Point* (dans un fichier *geom.py*) possédant les éléments suivants :

- ◎ **données** : deux attributs, les coordonnées x et y (flottants) du point
- ◎ **méthodes** :
  - ◇ un constructeur qui permet de créer des objets *Point*
  - ◇ une méthode *plot* qui permet d'afficher l'instance du *Point* dans un graphique. Elle prend en paramètre un attribut $style$ qui spécifie la couleur du tracé (g = green, r = red, b = blue, k = black, etc.) et le symbole d'affichage ('o'= rond, 'x'= croix, '∧ '= triangle, etc.)
  - ◇ une méthode *translate* qui translate l'instance du *Point* suivant un vecteur (dx, dy)
  - ◇ une méthode $duplic\_translate$ qui réplique l'instance du *Point* puis le translate

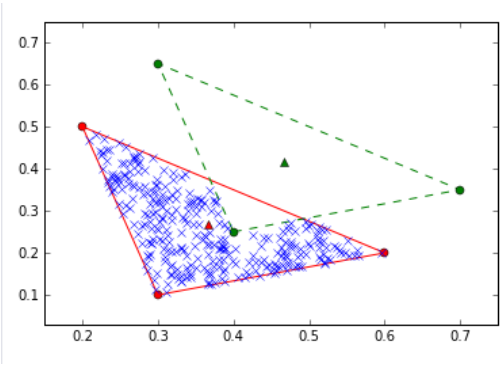
Dans le programme principal (*main.py*), instancier 3 objets de la classe *Point* (par exemple P_1 (0.3, 0.1), P_2(0.6, 0.2), P_3(0.2, 0.5)) et invoquer la méthode *plot* pour chacun d'eux

1.2 Triangle

Définir une classe *Triangle* (dans un fichier *geom.py*) possédant les éléments suivants :

- ◎ **données** : trois attributs de type *Point* : les 3 sommets p1, p2 et p3 du triangle
- ◎ **méthodes** :
  - ◇ un constructeur qui permet de créer des objets *Triangle* en fonction de 3 points (vérifier que les 3 points ne sont pas alignés)
  - ◇ une méthode *plot* qui permet d'afficher l'instance du *Triangle* dans un graphique. Cette méthode prend en paramètre un attribut $style$ défini de la même façon que celui de la méthode *plot* du *Point*.
  - ◇ une méthode *translate* qui translate l'instance du *Triangle* suivant un vecteur (dx, dy). (*Indication :* penser à utiliser la méthode *translate* de la classe *Point*).
  - ◇ une méthode $duplic\_translate$ qui réplique l'instance du *Triangle* puis le translate. (*Indication :* penser à utiliser la méthode *duplic_translate* de la classe *Point*).
  - ◇ une méthode *centroid* qui retourne la position du centre du triangle sous le format *Point*.
  - ◇ une méthode *contains* permettant de savoir si un point est inclus dans le triangle au sens géometrique. Cette méthode est une fonction qui retourne *True* si inclusion, *False* sinon. (*Indication :* penser au produit vectoriel)

Dans le programme principal (*main.py*), instancier un objet t_1 de la classe *Triangle* dont les 3 sommets correspondent aux points définis dans la question précédente. Construire un 2ème objet t_2 de la classe *Triangle* en translatant t_1 suivant le vecteur (0.10, 0.15). Tracer les points associés à t_2. Afficher t_1 et t_2 dans le même graphique. Représenter les centres de ces 2 triangles.



Tirer au hasard un nombre N (par exemple 5000) et construire N objets de type *Point* dans l'emprise [0,1] x [0,1]. Afficher tous les points qui sont à la fois contenus dans le premier triangle et à l'extérieur du second triangle.

1.3 Quelques opérations supplémentaires

1.3.1 Affichage

Écrire la méthode qui permet d'afficher dans la console, les informations pertinentes d'un objet *Point* et d'un objet *Triangle* de façon agréable.

1.3.2 Distances

Écrire une méthode, non statique, *distance*, qui calcule la distance euclidienne en dimension 2 de l'objet de type *Point* avec un autre objet de type *Point*.

Ajouter une méthode statique, *distancePointToPoint*, qui calcule la distance euclidienne en dimension 2 entre deux objets de type *Point*.

1.3.3 Égalité

Écrire une méthode qui permet de dire quand est-ce que deux objets de type *Point* ont les mêmes coordonnées.

1.3.4 Opération *In*

Actuellement, pour afficher uniquement les points tirés au sort qui sont inclus dans le triangle T1 mais pas dans le triangle T2, les tests utilisaient la méthode *contains* du triangle comme illustré dans le code ci-dessous.

```
if t2.contains(p):
    continue

if t1.contains(p):
    p.plot('bx')
```

Question : modifier le code pour que les instructions des tests correspondent dorénavant à celles-ci :

```
if p not in t2:
    continue

if p in t1:
    p.plot('bx')
```

2 Simulation d'un banc de poissons

Le paradigme de la programmation orientée objet est tout à fait adaptée pour des systèmes multi-agents. Dans ces systèmes, des agents, aux comportements individuels simples, vont travailler de concert pour résoudre des problèmes plus complexes. Nous les verrons dans un cas simple : le cas d'un banc de poissons.

> Vous pourrez remarquer qu'on ne va pas modéliser un banc de poisson, mais le comportement d'un seul poisson vis à vis des autres ! On fera une simulation pour observer dans une animation quand il y a plein de poissons en même temps et qui obéissent tous aux mêmes règles.

Les poissons ont des comportements différents liés à la présence d'autres individus suivant trois cercles concentriques : la zone de répulsion (évitement), la zone d'alignement et la zone d'attraction.

Le modèle s'appuie sur 3 règles comportementales :

- si un poisson dans le banc a un poisson très proche de lui, il va changer de direction pour essayer de l'éviter
- s'il y a un autre poisson dans un distance raisonnable, il va essayer de s'aligner dans la direction de cet autre poisson
- s'il y a un autre poisson qui est encore plus loin, il va essayer de s'en approcher.

2.1 Classe *Poisson* : les caractéristiques de base

Définir une classe *Poisson* (dans un fichier *banc.py* au même niveau que les deux précédents fichiers créés lors du précédent TP) possédant les éléments suivants :

- **données** : deux attributs :
  - sa position de type *Point*
  - un vecteur vitesse sous la forme d'un tuple de dimension 2
- **méthodes** :
  - un constructeur qui permet de créer des objets *Poisson*. Pour tester, on construira une fonction qui tirera au hasard les coordonnées et la vitesse.
  - une méthode, non statique, *distance*, qui calcule la distance euclidienne en dimension 2 de l'objet de type *Poisson* avec un autre objet de type *Poisson*
  - une méthode qui permet de comparer si deux poissons sont positionnés au même endroit.
  - une méthode qui met à jour la position du poisson.

- ◇ une méthode qui permet d'indiquer si deux poissons sont alignés, c'est à dire s'ils sont tous deux situés à proximité mais sans être dans une zone d'évitement ni dans la zone de cohésion.
- ◇ une méthode qui normalise la vitesse pour faire en sorte que les vitesses des poissons soient constantes dans le temps.

Pensez à utiliser des attributs statiques pour stocker certaines informations

2.2 Classe *Banc*

Définir une classe *Banc* (dans le fichier *banc.py*) possédant les éléments suivants :

- ◎ **données** : un attribut :
  - ◇ liste de poissons
- ◎ **méthodes** :
  - ◇ un constructeur qui permet d'initialiser la liste des poissons. Assurez-vous que vous n'ajoutez pas deux poissons identiques.
  - ◇ ajouter les méthodes qui permet de rendre la classe *Banc* comme un conteneur de *Poisson*

Pensez à utiliser des attributs statiques pour stocker certaines informations

2.3 Classe *Poisson* : les comportements

Ajouter les méthodes qui modélisent le comportement des objets *Poisson*.

- ◎ La méthode qui permet aux poissons d'éviter les murs. Pour cela, il faut tout d'abord s'arrêter au mur. Ensuite on modifie la direction du poisson en fonction du mur : les murs horizontaux modifient la vitesse horizontale sans modifier la vitesse verticale, de manière à faire tourner le poisson tout en conservant globalement sa direction actuelle. On termine la méthode en normalisant le nouveau vecteur. La méthode retourne *Vrai* si on a détecté un mur.
- ◎ Pour éviter les poissons trop proches, on calcule le vecteur unitaire entre le poisson et le poisson le plus proche, que l'on retranche à sa propre direction (en réalité, on retranche le quart de la différence). On termine en normalisant le vecteur vitesse et on retourne *Vrai* si un poisson a été évité.
- ◎ Le dernier comportement est le comportement d'alignement. Pour cela, il faut identifier tous les poissons dans la zone d'alignement. La nouvelle direction du poisson est une moyenne entre la direction des autres poissons et sa direction actuelle. On termine en normalisant le vecteur vitesse.
- ◎ La dernière méthode est celle permettant de mettre à jour les poissons et qui correspond à leur comportement global. Pour cela, on cherche d'abord si on doit éviter un mur et sinon un autre poisson.S'il n'y a pas eu d'évitement, c'est le comportement d'alignement qui est appliqué. Enfin quand la nouvelle direction est calculée, on l'applique en calculant la nouvelle position.

2.4 Classe *Simulation*

Pour visualiser le banc de poisson, la classe *Simulateur* appelle la fonction d'animation de la librairie Pyplot. La classe *Simulateur* contient un attribut représentant le banc de poissons. Compléter le code.

```
from matplotlib.animation import FuncAnimation

class Simulateur:

    def __init__(self):
        ?????

    def update(self, i):
        COORDS = [] # Liste des positions [xi, yi]

        ?????? remplir COORDS

        self.scatter.set_offsets(COORDS)

        self.fig.gca().relim()
        self.fig.gca().autoscale_view()

        return self.scatter,


    def start_simulation(self):
        self.fig = plt.figure(figsize=(8, 5))
        ax = self.fig.add_axes([0.0, 0.0, 1.0, 1.0], frameon=True)

        ????? dessiner les positions initiales des poissons
        self.scatter = ax.scatter(X, Y, s=30, facecolor="red", alpha=0.8)

        self.ani = FuncAnimation(self.fig, self.update, interval=100)

        ax.set_xlim(0, Poisson.SIZE)
        ax.set_ylim(0, Poisson.SIZE)
        plt.show()
```

3 Exos sur l'héritage

3.1 Ville et Capitale

Soit le programme suivant qui implémente des objets de type *Ville* et *Capitale* :

```
v1 = Ville ("Toulouse")
v2 = Ville ("Strasbourg", 272975)

print(v1)
print(v2)
print("")

c1 = Capitale("Paris", "France")
c2 = Capitale("Rome", "Italie", 2700000)

c1.nbHabitants = 2181371
print(c1)
print(c2)
print("")

print("catégorie de la ville de " + v1.getNom() + " : " +
 ↪  v1.categorie());
print("catégorie de la ville de " + v2.getNom() + " : " +
 ↪  v2.categorie());
print("catégorie de la ville de " + c1.getNom() + " : " +
 ↪  c1.categorie());
print()
```

qui lors de l'exécution affichera le résultat suivant :

> Ville de TOULOUSE.
> Ville de STRASBOURG ; 272975 habitants.
>
> Ville de PARIS ; 2181371 habitants. Capitale de FRANCE.
> Ville de ROME ; 2700000 habitants. Capitale de ITALIE.
>
> catégorie de la ville de TOULOUSE : ?
> catégorie de la ville de STRASBOURG : A
> catégorie de la ville de PARIS : C

Écrire les instructions dans le langage Python afin de définir les classes *Ville* et *Capitale* en respectant les spécifications suivantes :

1/ La classe ***Ville*** a les propriétés suivantes :

- ~ Une ville est décrite par deux informations : son nom et son nombre d'habitants. Les variables seront respectivement nommées nom et nbHabitants ; elles sont d'accès protégé (protected).
- ~ Le nom d'une ville ne peut varier ; il doit être connu dès l'instanciation de l'objet. Il est toujours représenté en majuscules. Rappel : dans la classe String, la méthode d'instance String toUpperCase() convertit tous les caractères de la chaîne en majuscule.
- ~ Le nombre d'habitants peut être inconnu ; sa valeur est alors 0. S'il est connu, il est toujours supérieur à zéro. Il peut varier pour un même objet Ville (suite par exemple à un nouveau recensement).

et les méthodes :

- ~ Un constructeur ayant pour argument le nom de la ville (en majuscules ou minuscules).
- ~ Un constructeur ayant pour argument le nom de la ville (en majuscules ou minuscules) ainsi que le nombre d'habitants.
- ~ Les accesseurs getNom et getNbHabitants qui renvoient respectivement le nom et le nombre d'habitants de la ville. Si le nombre d'habitants est inconnu, getNbHabitants renvoie 0.
- ~ Une méthode setNbHabitants qui permet de mettre à jour le nombre d'habitants. La nouvelle valeur est transmise en argument. Si elle est négative, la valeur de nbHabitants reste à 0.
- ~ Une méthode nbHabitantsConnu qui renvoie un booléen de valeur vrai si le nombre d'habitants est connu et faux sinon.
- ~ Une redéfinition de la méthode *str*.

On suppose que les données transmises en argument sont correctes. On ne demande pas de test et de gestion de situations d'erreur.

2/ Définir la classe Capitale. Une capitale est une ville particulière. Elle a pour attribut supplémentaire le nom du pays dont elle est la capitale. Le nom du pays est en majuscules.

Définir les constructeurs et redéfinir la méthode *str*.

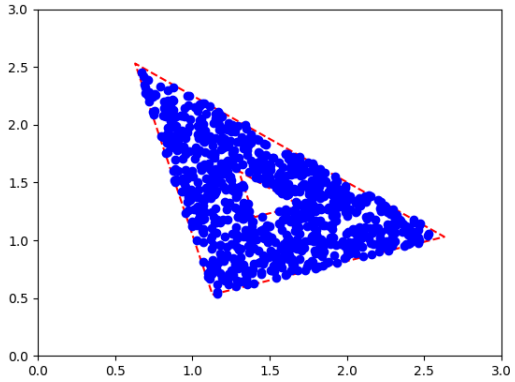
3/ On est amené à classer les villes en 4 catégories :

- ~ catégorie A : celles de moins de 500 000 habitants,
- ~ catégorie B : celles de 500 000 habitants et plus,
- ~ catégorie ? : celles dont le nombre d'habitants est inconnu,
- ~ catégorie C : celles qui sont capitales d'un pays, quel que soit le nombre d'habitants.

Dire comment et ou définir la méthode *categorie* qui renvoie l'un des 4 caractères ' ?', 'A', 'B' ou 'C'. Utiliser les liaisons dynamiques. Votre code ne doit pas comporter de test sur le type des variables.

3.2 Triangle à trou

Le but de cet exercice est de construire un anneau (*Ring*) de forme triangulaire comme étant un triangle dont on a évidé une zone triangulaire définie par une marge (distance entre les bordures du triangle externe et interne). Un *Ring* est avant tout un *Triangle* et a deux attributs supplémentaires : le triangle interne et la marge.



Question : Écrire la classe *Ring* dans le fichier *shapes.py* qui hérite de la classe *Triangle*. Écrire un constructeur de la classe *Ring* prenant en paramètre 3 points de type *Point* et une marge.

Question : On souhaite maintenant dessiner l'anneau dans un graphique. Ajouter une méthode *plot* afin de réaliser l'affichage.

Question : Implanter une méthode dans la classe *Ring* qui permet de tester si un point est contenu dans l'anneau.

4 Héritage et abstraction

4.1 Animaux d'un zoo

Dans cet exercice nous allons modéliser différents animaux et leurs comportements différents :

- Un *Animal* possède un poids et une couleur. Il peut se déplacer (méthode deplacement()) et s'exprimer (méthode moyenExpression()) (cri, meuglement, etc.).
- Un *Animal* peut être un *AnimalDomestique* ou un *AnimalSauvage*. Dans le premier cas, il porte un nom (String). Les animaux domestiques se déplacent tous dans un appartement et les animaux sauvages dans la nature.

Question 1 : Peut-on définir le mode de déplacement général d'un *Animal* ? Sa manière de s'exprimer ? Est-il pertinent de pouvoir instancier un objet de type *Animal* ? *AnimalDomestique* ? *AnimalSauvage* ? Que peut-on en déduire sur ces classes et leurs méthodes deplacement() et moyenExpression() ?

Question 2 : Implémentez ces classes ainsi que les sous classes :

- *Chat*, animal domestique,
- *Chien*, animal domestique
- *Lapin*, animal domestique
- *Tigre*, animal sauvage
- *Vache*, animal sauvage.
- *Sanglier*, animal sauvage.

L'implémentation des méthodes deplacement() et moyenExpression se traduira simplement par un *print* (ex : « Je me déplace en appartement » ; « Je meugle », etc.).

Question 3 : Redéfinissez la méthode ***str()*** de ces classes pour qu'elle affiche par exemple dans le cas d'un *Chien* :

« Je suis un Animal. Je suis un Animal Domestique. Je suis un Chien ».

4.2 Régime alimentaire

On souhaite maintenant considérer que certains animaux (pas forcément tous) sont des carnivores (ils possèdent la méthode void manger avec comme paramètre un autre animal) ou des herbivores (ils possèdent la méthode void manger qui ne prend pas de paramètres) ou les deux (carnivore et herbivore).

Question 4 : Créez deux classes pour décrire ces comportements.

- ~ Modifiez les classes des animaux pour qu'elles héritent de ces classes :
  - Les carnivores : quand ils mangent, les Chiens affichent "Je dévore X", les Tigres affichent "Je déchire X" et les Chats affichent « Je mange X » .
  - Les herbivores : les vaches affichent "Je broute", et les lapins "Je grignotte".
- ~ Les Sangliers sont à les fois carnivores et herbivores.
- ~ Créez un tableau d'animaux. Créez un lapin, et faites le manger par tous les carnivores du tableau.

Question 5 : On voudrait interdire qu'un carnivore mange un animal s'il est le dernier animal du zoo. Quelle solution envisageriez-vous ?

4.3 Collection de géométries

Depuis le début du cours, nous avons différentes géométries : les *Point*, les *Triangle*, les *Ring*. Proposez un modèle optimum qui permet de gérer des collections de géométries, puis implémenter le. Par exemple on voudrait construire une collection composée de :

- ~ un triangle à trou
- ~ une composition d'un triangle et d'un point
- ~ une collection de points

5 Stratégies de profils de niveau

Sujet

L'un des objectifs de cet exercice est de proposer un service de comparaison automatique de niveau d'une randonnée à partir d'une trace GPS. La pratique d'un sport se déroule dans la majorité du temps sur des itinéraires dont les routes sont établies par des guides. Ces routes sont définies spatialement.

En tenant compte de la pente, des modèles (ou profils) de vitesse existants permettent d'évaluer le temps de parcours théorique d'un itinéraire, alors que la trace GPS fournira le temps réel de parcours du même itinéraire.

5.1 Environnement de développement

Nous allons utiliser la librairie **Shapely**. Cette librairie objet python Open Source s'appuie sur un modèle standard pour gérer les géométries ainsi que des algorithmes permettant de les manipuler.

5.2 Package trace

Ce package vise à fournir les méthodes nécessaires pour charger des données GPS représentant des sorties de pratiquants de randonnée. Nous proposons d'implémenter deux classes : la classe *Observation* caractérisée par une position et un timestamp et la classe *TraceGPS* composée d'un ensemble d'observation.

- **Observation** représente une observation d'une trace GPS. Chaque observation contient une coordonnée en 3D et un timestamp.
  - un attribut "temps". Pour calculer le temps à partir d'une date passée en format texte :

```
from datetime import datetime

txtdate = ...;
ts = datetime.strptime(txtdate, '%Y-%m-%d %H:%M:%S')
its = ts.timestamp()
```

  - un attribut "coord" de type Coordinate. Le z de la trace sera ajoutée comme 3ème coordonnée de la classe Coordinate
  - un attribut "pente". Rappel pour le calcul de la pente :

```
pente = math.atan( deniv / d) * 180 / math.pi
```

    avec d = distance entre les deux observations, deniv la différence des altitudes.

  - un attribut "absCurv" qui stocke l'abscisse curviligne de l'observation
  - un constructeur *Observation(double x, double y, double z, timestamp temps)*

- **TraceGPS** représente la trace faite par un sportif qui a suivi un itinéraire. La trace est composée d'une liste d'observations GPS, où chaque observation contient une coordonnée en 3D et un timestamp.
  - un attribut "" de type Liste d'observations représentant les points de passage du sportif le long de l'itinéraire
  - un constructeur *Trace (String filename)* qui charge un fichier de points GPS. A la fin du chargement du fichier on calculera la pente et l'abscisse de chaque point d'observation de la trace.
  - une méthode *double vitesseReelParcours()* qui calcule la vitesse moyenne réelle effectué par le sportif sur sa trace
  - la méthode *str()* qui retourne un résumé des informations de la trace : le nombre de points, l'heure de départ, d'arrivée, le temps de la randonnée ainsi que sa longueur.
  - une méthode qui retourne vrai si la trace est une boucle.

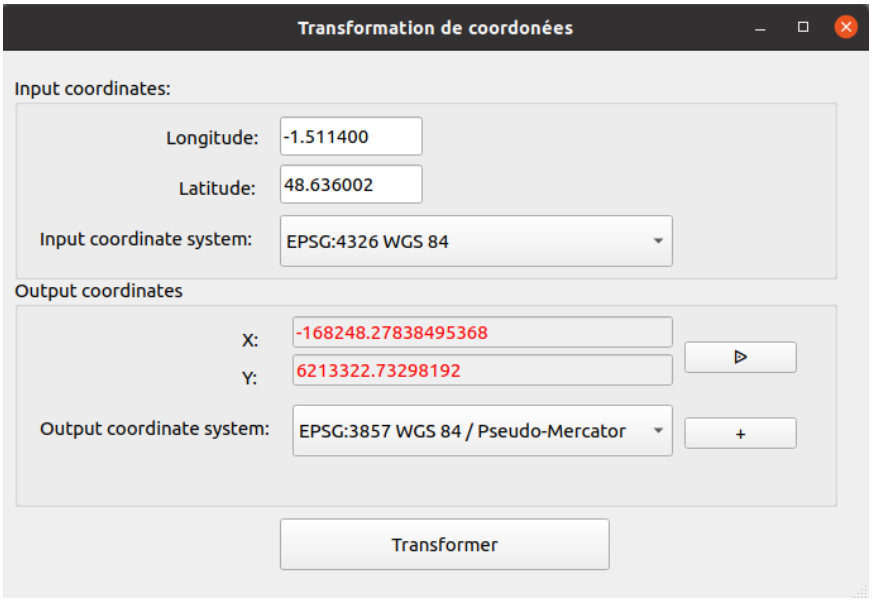
Vous pouvez tester avec le fichier *abbaye-de-valcroissant.dat*.

5.3 Package profils de niveau

Implémenter plusieurs stratégies de temps de parcours d'une trace en fonction de la pente. Proposer un modèle et implémentez-le.

6 Exos interface graphique

Dans cet exercice, on va construire un petit programme qui propose à l'utilisateur une interface graphique afin de transformer des coordonnées géographiques définies dans le système de référence WGS84, dans un autre système.



6.1 Fenêtre principale

Question 1 Créer l'interface graphique avec QT Designer. Quand la conception de l'interface graphique est terminée, enregistrer votre fichier dans le même répertoire que vos fichiers python. L'extension du fichier est : ".ui".

On va maintenant compiler notre fichier ".ui" en un fichier python. PyQt fournit un compilateur **pyuic5** : il se charge de générer un fichier python en traduisant en instructions les éléments dessinés dans Qt Designer. On accède à la commande pyuic5 dans une console de l'OSGeo4W shell.

1. Lancer la console OSGeo4W shell depuis les programmes de Windows
2. Déplacer l'emplacement de votre curseur afin d'arriver dans le répertoire de travail. Pour cela tapez :
   - D :
   - Puis taper le chemin du répertoire où vos fichiers sont stockés
3. Dans la console, lancer la commande :

   pyuic5 MonFichierUI.ui -o MonFichierUI.py

   en remplaçant *MonFichierUI* par le nom de votre fichier créé dans QtDesigner.

Écrire le programme principal qui permet de tester votre interface graphique. Voici le début :

```
import sys
from ... import Ui_MainWindow
from PyQt5.QtWidgets import QMainWindow, QApplication

class XXXXWindow(QMainWindow, Ui_MainWindow):

    def __init__(self, parent=None):
        super(XXXXWindow, self).__init__(parent)
        self.setupUi(self)

        # ...

if __name__ == "__main__":

    def run_app():
        app = QApplication(sys.argv)
        mainWin = XXXXXWindow()
        mainWin.show()
        app.exec_()

    run_app()
```

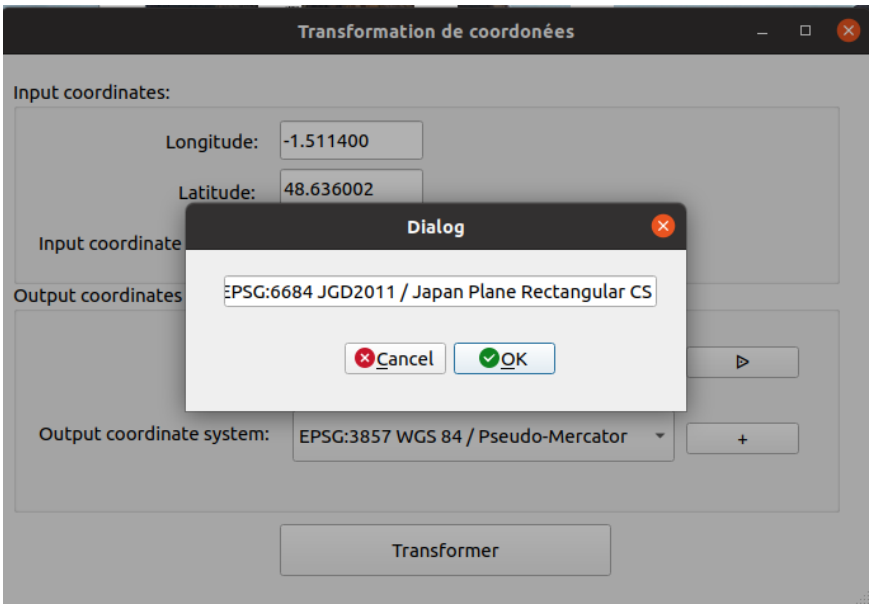
6.2 Transformons !

Question 2 : Personnalisation de l'interface graphique :

- Alimenter la liste déroulante avec des systèmes de référence. Par exemple :
  - EPSG :3857 WGS 84 / Pseudo-Mercator
  - EPSG :2154 RGF93 v1 / Lambert-93
  - EPSG :4326 WGS 84
- Connecter un slot sur le bouton « Transformer » qui permet de lancer le calcul.

6.3 Ajout d'un nouveau système de référence

Question 3 : ajouter une nouvelle fonctionnalité : à partir d'un clic sur un bouton, une **boite de dialogue** s'ouvre et permet de saisir le nom du système de référence.



Pour utiliser la boite de dialogue créée dans QT Designer, vous devez créer une nouvelle classe qui hérite de la classe *QDialog* :

```
from addCRS import Ui_Dialog
from PyQt5.QtWidgets import QDialog

class XXXXDlg(QDialog):

    """CRS dialog."""
    def __init__(self, parent=None):
        super().__init__(parent)
        # Create an instance of the GUI
        self.ui = Ui_Dialog()
        # Run the .setupUi() method to show the GUI
        self.ui.setupUi(self)

    def accept(self):
        # On a cliqué sur OK dans la boite de dialogue
        # ....

        super().accept()
```

La classe *XXXXDlg* hérite de *QDialog*. Pour son constructeur, elle appelle le constructeur de sa superclasse. Ensuite elle crée un objet de type *Ui_Dialog* afin d'appeler la méthode *setupUi*. Ainsi les widgets et les layouts définis avec Designer sont ajoutés dans notre boite de dialogue.

On redéfinit également la méthode *accept* qui est une méthode héritée de la classe *QDialog*. Par défaut, cette méthode signale que l'utilisateur a accepté (généralement en cliquant sur un bouton Ok). La redéfinition de cette méthode permet de récupérer les valeurs saisies.

Pour ouvrir la boite de dialogue, placer le bout de code ci-dessous dans votre code :

```
dlg = CrsDlg(self)
dlg.exec()
```

6.4 Copier-coller !

Question 4 : ajouter une nouvelle fonctionnalité : à partir d'un clic sur un bouton, les nouvelles coordonnées sont ajoutées dans le presse-papier.