Programmation Orientée Objet (POO) à destination de la robotique

Travaux Pratique : Prise en main de la POO

Enseignant : Adam GOUGUET
Mail : adam.gouguet@imt-nord-europe.fr
Année : 2025–2026

Table des matières

1 Objectifs du TP **1**

2 Implémentation de la classe `RobotMobile` **2**
2.1 Mise en place de la structure du projet 2
2.2 Attributs 2
2.3 Méthodes 3

3 Encapsulation **4**

4 Héritage et polymorphisme **5**
4.1 Introduction de la classe abstraite `Moteur` 5
4.2 Deux types de moteur 6
4.3 Polymorphisme par composition 7

5 Attributs et méthodes statiques **9**

6 Mots clés et méthodes spéciales en Python **10**

7 Exercice Bonus : Diagramme UML **10**

1 Objectifs du TP

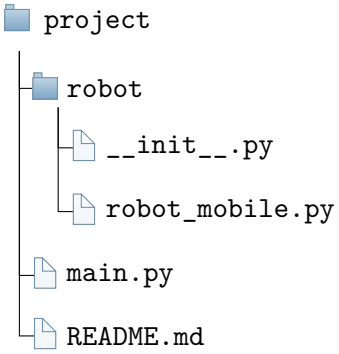
Ce TP a pour objectif de vous donner les bases sur la programmation orienté objet en Python, en ayant une thématique d'application robotique. À l'issue de ce TP, vous devrez être capables de :

- Structurer un projet Python en programmation orientée objet.
- Implémenter les notions vues lors du cours :
  - Définition de classes
  - Encapsulation
  - Héritage et polymorphisme
- Modéliser un système simple à l'aide d'un diagramme de classes UML.

2 Implémentation de la classe `RobotMobile`

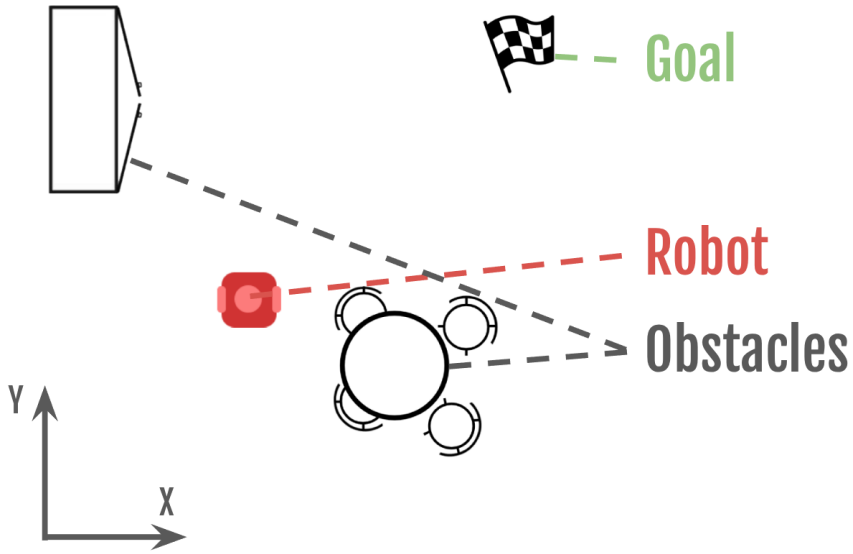
2.1 Mise en place de la structure du projet

Mettez en place l'arborescence suivante dans votre dépôt :



> Le dossier `robot` correspond à un module Python. Le fichier `main.py` servira de point d'entrée du programme. Pour bien faire, vous pouvez déjà transformer ce dossier en projet Git.

La classe `RobotMobile` représente un robot se déplaçant dans un plan bidimensionnel (x, y).



2.2 Attributs

La classe devra contenir les attributs suivants :

— une position (`x`, `y`)

— une orientation (en radians)

```
class RobotMobile:
    def __init__(self, ...):
        ...
```

On a créer la classe `RobotMobile`, on va maintenant créer un objet `RobotMobile`, une instance de la classe. Pour ce faire, dans le `main.py`, importer votre classe comme un module et créer un objet de type `RobotMobile` :

```
from robot.robot_mobile import RobotMobile

robot = RobotMobile(...)
```

Pour l'instant notre classe contient uniquement des attributs donc on ne peut pas en faire grand chose...

2.3 Méthodes

La classe devra proposer une méthode `avancer(distance)`. La méthode `avancer` devra mettre à jour la position du robot en fonction de la distance voulue et son orientation, comme ceci :

$$x = x + distance * cos(orientation)$$

$$y = y + distance * sin(orientation)$$

Importer et utiliser le module `math` pour le cosinus et sinus.

Dans le fichier `main.py`, testez le comportement du robot :

```
from robot.robot_mobile import RobotMobile

robot = RobotMobile(...)
robot.avancer(1.0) # avance de 1 metre
```

Que constatez vous ? Normalement rien.

Si on veut un affichage, on définit une méthode `afficher` dans la classe :

```
def afficher(self):
    print("A changer, afficher les attributs du robot ! ")
```

Puis dans le `main.py` :

```
from robot.robot_mobile import RobotMobile

robot = RobotMobile(...)
robot.afficher()
robot.avancer(1.0) # avance de 1 metre
robot.afficher()
```

Vous devriez avoir dans le terminal quelque chose comme :

```
(x=0.00, y=0.00, orientation=0.00)
(x=1.00, y=0.00, orientation=0.00)
```

Maintenant on fait la méthode `tourner(angle)` du robot en mettant à jour son orientation :

$$orientation = (orientation + angle) \ \% \ 2\pi$$

Faite avancer le robot d'un metre, puis le faire tourner de 45° et ensuite avancer de 3 metre. Quel est sa position final ?

3 Encapsulation

En programmation orientée objet, l'encapsulation consiste à **protéger l'état interne d'un objet** et à contrôler la manière dont il peut être consulté ou modifié. La position et l'orientation du robot ne doivent pas être modifiées arbitrairement depuis l'extérieur de la classe. Elles doivent uniquement être modifiées par les méthodes internes de la classe.

Utiliser les décorateurs `@property` et `@<attribut>.setter` pour implémenter des accesseurs (*getter*) et mutateurs (*setter*) contrôlés. Comme ci-dessous, faites la même chose pour tous les attributs.

```python
# Getter : Permet d'acceder a l'attribut depuis l'exterieur de la classe.
@property
def x(self) -> float:
    return self.__x

# Setter : Permet la modification de l'attribut depuis l'exterieur de la classe.
@x.setter
def x(self, value: float):
    self.__x = value
```

Quelle est la différence entre un attribut *public* et *private* ? Et donc après modification, les attributs du robot ont quels visibilité ?

Donc si essayer d'acceder à un attribut directement dans le main.py (par exemple : `print(robot.__x)`) :

```
AttributeError: 'RobotMobile' object has no attribute '__x'
```

Par contre, faire `print(robot.x)` fonctionne !

Vous devez bien réflechir à l'architecture de votre code et au niveau de visibilité que vous definissez à vos attributs et méthodes.

4 Héritage et polymorphisme

Le robot que vous avez coder juste avant suppose qu'il :

— avance d'une distance
— strictement dans la direction de son orientation
— sans glissement latéral
— sans changement d'orientation pendant le déplacement

Ce qui n'est pas très réaliste, on a considéré le robot comme un point se déplacant dans un plan 2D. Dans la réalité, un robot :

— possède des roues et des moteurs
— peut tourner pendant qu'il avance
— a des contraintes mécaniques liées à son type de locomotion

4.1 Introduction de la classe abstraite `Moteur`

Pour modéliser cette réalité, nous introduisons une nouvelle classe : `Moteur`.
Le moteur représente l'**actionneur** du robot :

— il reçoit des *commandes de vitesse*,
— il applique ces commandes pendant un certain temps,
— il met à jour la position et l'orientation du robot.

Le robot ne connaît donc pas les équations de déplacement : il délègue cette responsabilité à son moteur.
Tous les moteurs partagent une interface commune :

— recevoir une commande,
— appliquer une mise à jour cinématique.

Donc dans un fichier `moteur.py` dans le dossier `robot`, on implémente la classe abstraite `Moteur` :

```
from abc import ABC, abstractmethod
from math import cos, sin

class Moteur(ABC):

    @abstractmethod
    def commander(self, *args):
        pass

    @abstractmethod
    def mettre_a_jour(self, robot, dt):
        pass
```

En Python, une classe abstraite hérite de `ABC` et utilise le décorateur `@abstractmethod` pour indiquer les méthodes obligatoires.

Pourquoi definit-on la classe `Moteur` comme abstraite ? Quels sont les conséquences ?

4.2 Deux types de moteur

Il existe plusieurs types de robots mobiles, chacun possédant des capacités de déplacement différentes. Dans ce cours, nous nous concentrerons sur deux architectures courantes :

- `MoteurDifferentiel`,
- `MoteurOmnidirectionnel`.

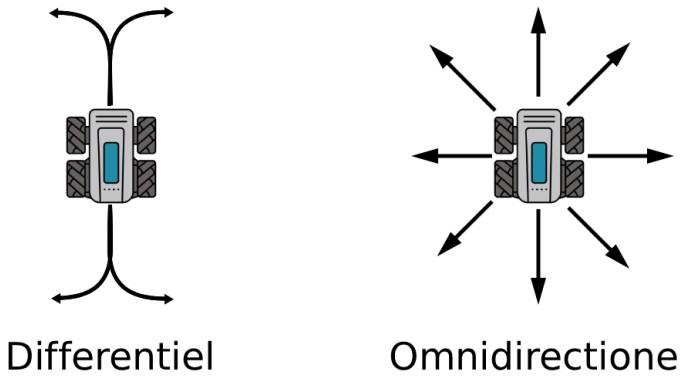


FIGURE 1 – Différence de déplacement entre robot différentiel et omnidirectionnel

Chaque sous-classe implémente sa propre cinématique, correspondant à la réalité physique du robot (voir figure 1).

Moteur différentiel Un robot différentiel est commandé par :

- une vitesse linéaire v,
- une vitesse angulaire ω.

La mise à jour de la position du robot pendant un pas de temps Δt est donnée par :

$$\theta_{k+1} = \theta_k + \omega\,\Delta t$$

$$x_{k+1} = x_k + v\cos(\theta_k)\,\Delta t$$

$$y_{k+1} = y_k + v\sin(\theta_k)\,\Delta t$$

Le robot avance donc uniquement dans la direction de son orientation et peut tourner simultanément.

```python
class MoteurDifferentiel(Moteur):
    def __init__(self, v=0.0, omega=0.0):
        self.v = v          # vitesse linaire
```

```
        self.omega = omega # vitesse angulaire

    def commander(self, v, omega):
        ...

    def mettre_a_jour(self, robot, dt):
        ...
```

Moteur omnidirectionnel Un robot omnidirectionnel est commandé par :

- une vitesse v_x dans la direction avant du robot,
- une vitesse v_y dans la direction latérale,
- une vitesse angulaire ω.

Les vitesses (v_x, v_y) sont exprimées dans le repère du robot. La mise à jour de la position pendant un pas de temps Δt est donnée par :

$$\theta_{k+1} = \theta_k + \omega\,\Delta t$$

$$x_{k+1} = x_k + (v_x \cos(\theta_k) - v_y \sin(\theta_k))\,\Delta t$$

$$y_{k+1} = y_k + (v_x \sin(\theta_k) + v_y \cos(\theta_k))\,\Delta t$$

Ce modèle permet au robot de se déplacer dans toutes les directions, indépendamment de son orientation.

```
class MoteurOmnidirectionnel(Moteur):
    def __init__(self, vx=0.0, vy=0.0, omega=0.0):
        self.vx = vx    # vitesse avant
        self.vy = vy    # vitesse latrale
        self.omega = omega

    def commander(self, vx, vy, omega):
        ...

    def mettre_a_jour(self, robot, dt):
        ...
```

Travail demandé Pour chaque type de moteur, ajouter la classe dans le fichier `moteur.py` et :

- implémenter la méthode `commander` pour stocker les vitesses,
- implémenter la méthode `mettre_a_jour` en utilisant les équations précédentes.

4.3 Polymorphisme par composition

La classe `RobotMobile` possède un moteur, mais ne dépend pas de son type concret. Elle manipule uniquement une référence de type `Moteur`.
Ainsi :

— le même robot peut être équipé de moteurs différents,
— le comportement du robot change sans modifier sa classe,
— le choix du moteur détermine le type de déplacement.

Ce mécanisme illustre le **polymorphisme** : le robot appelle les mêmes méthodes, mais le comportement dépend du moteur utilisé. On peut ajouter à la définition de la classe `RobotMobile` les méthodes suivantes : `commander` et `mettre_a_jour`.

```
def commander(self, **kwargs):
    if self.moteur is not None:
        self.moteur.commander(**kwargs)

def mettre_a_jour(self, dt):
    if self.moteur is not None:
        self.moteur.mettre_a_jour(self, dt)
```

En Python, *args et **kwargs servent à rendre une fonction plus flexible sur le nombre d'arguments qu'elle peut recevoir. *args permet de récupérer un nombre variable d'arguments positionnels sous forme de tuple, tandis que **kwargs récupère des arguments nommés sous forme de dictionnaire (clé : valeur).

On met à jour le fichier `main.py` pour prendre en compte notre nouvelle architecture :

```
import math
from robot.robot_mobile import RobotMobile
from robot.moteur import *

...

moteur_diff = MoteurDifferentiel()
robot = RobotMobile(moteur=moteur_diff)

dt = 1.0 # pas de temps (s)

robot.afficher()
# On doit nommer les arguments (v = ..., omega = ...) car on utilise **kwargs !
robot.commander(v = 1.0, omega = 0.0)  # avance
robot.mettre_a_jour(dt)
robot.afficher()
```

Essayer de déplacer le robot en position x=3 et y=1, faite la même chose avec le moteur omnidirectionnel.

5 Attributs et méthodes statiques

Jusqu'à présent, nous avons utilisé des attributs et des méthodes liés à une instance d'objet. Il existe cependant des éléments qui sont communs à toutes les instances d'une même classe.

Un **attribut statique** (ou attribut de classe) est partagé par toutes les instances. Une **méthode statique** appartient à la classe et ne dépend d'aucun objet particulier.

Attribut statique On souhaite connaître le nombre total de robots mobiles créés pendant l'exécution du programme.
Ajouter un attribut statique à la classe `RobotMobile` permettant de compter le nombre d'instances créées :

```
class RobotMobile:

    _nb_robots = 0

    def __init__(self, ...):
        ...
        RobotMobile._nb_robots += 1
        ...

    ...

    @classmethod
    def nombre_robots(cls) -> int:
        """
        Retourne le nombre total de robots crees.
        """
        return cls._nb_robots
```

Méthode statique La vérification du type du moteur ne dépend pas d'un robot particulier. Il est donc pertinent de la regrouper dans une méthode statique.
Ajouter une méthode statique chargée de vérifier si un objet peut être utilisé comme moteur.

```
class RobotMobile:

    @staticmethod
    def moteur_valide(moteur):
        return ...
```

L'utilisation de `isinstance` pourrait être pertinent ici ;)

Quels sont les différences entre les `@staticmethod` et les `@classmethod` ?

6 Mots clés et méthodes spéciales en Python

Au début du TP, nous avons définit une méthode `afficher` pour afficher les informations du robot dans le terminal.

Mais que se passe-t-il si on print l'objet robot directement (`print(robot)`) ?

Le terminal vous affiche quelque chose comme :

```
<robot.robot.RobotMobile object at 0x7d78e9d5ad40>
```

Cela correspond au chemin du module + le nom de la classe + l'adresse mémoire. print(obj) appelle obj.___str___().

Il est possible de surcharger la méthode `__str__` de la classe `RobotMobile` pour changer le résultat d'un print :

```
def __str__(self):
    return str((self.x, self.y, self.orientation))
```

Il existe un certain nombre de méthode spécifique comme ceci (___str___, ___eq___, ___iter___, ...) qui font chacune appel à des fonctions connu de python que vous avez déjà utilisé.

7 Exercice Bonus : Diagramme UML

Réaliser le diagramme UML de tout votre projet jusqu'a maintenant, cela sera un plus dans la documentation de votre code plus tard :)

Diagramme de classes UML Un diagramme de classes UML permet de représenter la structure d'un programme orienté objet. Il décrit les **classes**, leurs **attributs**, leurs **méthodes** ainsi que les **relations** entre les classes (association, héritage, composition, etc.). Chaque classe est représentée par un rectangle contenant son nom, puis ses attributs et ses méthodes.

Vous pouvez le faire sur papier, mais faites en sorte qu'il soit lisible. Sinon il existe des outils en ligne pour réaliser des diagrammes UML.