Chapitre 3 - Programmation orientée objet

Objectifs :

- ▷ Connaître le vocabulaire de la programmation objet : classes, attributs, méthodes, objets.
- ▷ Ecrire la définition d'une classe
- ▷ Accéder aux attributs et méthodes d'une classe.

1 Introduction

Jusqu'à présent nous avons utilisé des objets dont le type est prédéfini : `int`, `float`, `bool`, `str`, `list`, `tuple`, `dict`. Sans le savoir, nous avons découvert les concepts de la **programmation orientée objet** ou **POO**. Ainsi, en classe de Première, nous avions déjà appris à utiliser des méthodes associées à ces types, comme :

```
ma_liste.append(valeur)
```

Comme de nombreux autres langages tels que *Java* ou *C++*, *Python* est un **langage orienté objet**. On peut donc dire que tout y est **objet**. Une variable de type `int` est en fait un objet de type `int` donc construit à partir de la classe `int`. Idem pour les `float` et `str`. Mais également pour les `list`, `tuple`, `dict`, ..., *etc.*

On peut donc créer ses propres objets qui auront leurs **méthodes**, ce qui aura pour effet :

- ▷ de faciliter l'implémentation de données
- ▷ d'augmenter la lisibilité du programme.

2 Définition

A retenir !

Une **classe** définit des objets qui sont des **instances** (des représentants) de cette classe.

On utilisera le mot **objet** ou **instance** pour désigner la même chose.

Les objets peuvent posséder des **attributs** (variables associées aux objets) et des **méthodes** (qui sont des fonctions associées aux objets et qui peuvent agir sur ces derniers ou encore les utiliser).

3 Découverte par des exemples

Imaginons que nous ayons un objet `Eleve`, qui permettrait pour un élève de stocker dans une variable `eleve1` des données comme son *nom*, son *prénom*, sa *date de naissance* et ses *notes* dans différentes matières, ainsi qu'une méthode qui calculerait sa *moyenne*.

3.1 Création de la classe

- ▷ Pour créer une classe, on utilise le mot-clé : `class` suivi par le nom de la classe et les `":"`
- ▷ Par convention, le nom de la classe commence par une **majuscule** et ne contient que des caractères alphanumériques.

```
1 #creation de la classe Eleve
2 class Eleve:
```

Ajoutons l'instruction `pass` et créons une variable `eleve1` de type `Eleve` (une instance de la classe `Eleve`).

```
class Eleve:
    pass                    # signifie qu'on ne fait rien...

eleve1 = Eleve()
print(type(eleve1))
```

Le résultat signifie que `eleve1` est une instance de la classe `Eleve` (une variable de type `Eleve`).

3.2 Les attributs

Les attributs sont des variables associées à la classe.

Il y en a de deux types :

- ▷ **Les attributs de classe :** un attribut de classe (ou variable de classe) est un attribut qui sera identique pour chaque instance et n'a pas vocation à être changé.
- ▷ **Les attributs d'instance :** une variable ou attribut d'instance est une variable accrochée à une instance et qui est spécifique à cette instance. Et d'une instance à l'autre, il ne prendra pas forcément la même valeur.

Pour notre exemple, nous choisirons les matières comme attributs de classe et les *nom*, *prénom*, *date de naissance* et les *notes* comme attributs d'instance.

Le constructeur

L'endroit le plus approprié pour déclarer un attribut est à l'intérieur d'une méthode appelée le **constructeur**. S'il est défini, il est implicitement exécuté lors de la création de chaque instance.

Le constructeur d'une classe se présente comme une méthode et suit la même syntaxe sauf que son nom est imposé : `__init__` (**Attention, il faut deux underscores de chaque côté !**).

Hormis le premier paramètre `self`, il n'existe pas de contrainte concernant la liste des paramètres excepté que le constructeur ne doit pas retourner de résultat.

Exemple :

```
class Eleve:
    #attributs de classe
    matiere1 = "Programmation"
    matiere2 = "Algorithmique"
    matiere3 = "Projet"

    #constructeur
    def __init__(self, Nom, Prenom, Date, Note1, Note2, Note3):
        #attributs d'instance
        self.nom = Nom
        self.prenom = Prenom
        self.date = Date
        self.note_mat1 = Note1
        self.note_mat2 = Note2
        self.note_mat3 = Note3

#creation d'un eleve
eleve1 = Eleve('Terieur','Alain','01/01/2005',12,10,15)
```

Comment accéder aux valeurs des attributs d'instance créés ?

On utilise la syntaxe : `instance.attribut` (*notez bien le point*)

```
print(eleve1.prenom, eleve1.nom)
print(eleve1.matiere1,':',eleve1.note_mat1)
```

```
print(eleve1.matiere2,':',eleve1.note_mat2)
print(eleve1.matiere3,':',eleve1.note_mat3)
```

Noter ci-dessous l'affichage obtenu :

3.3 Les méthodes

Il serait intéressant de pouvoir obtenir la moyenne de l'élève avec l'instruction `eleve1.moyenne()`, c'est-à-dire en appliquant la méthode `moyenne()` à l'instance `eleve1`.

Pour cela on crée une fonction à l'intérieur de la classe `Eleve` qui retourne la moyenne de l'élève :

```
1     def moyenne(self):
2         return ((self.note_mat1 + self.note_mat2 + self.note_mat3)/3)
```

A faire :

- ▷ Ajouter cette **méthode** et faire afficher la moyenne de l'élève.
- ▷ Ajouter ces trois élèves et faire afficher leurs résultats et leurs moyennes.

Nom : Oma Prénom : Modeste Date : 01/03/2006 Programmation : 17 Algorithmique : 6 Projet : 16	Nom : Neymar Prénom : Jean Date : 01/07/2006 Programmation : 7 Algorithmique : 14 Projet : 11	Nom: Duff Nom: John Date : 01/11/2007 Programmation : 13 Algorithmique : 8 Projet : 17

- ▷ Ecrire une **fonction** (hors de la classe bien sûr !) qui prend en paramètres une liste constituée de ces quatre élèves et qui retourne les moyennes par matière.

  On attend le rendu suivant :

```
Programmation : 12.25
Algorithmique : 9.5
Projet : 14.75
```

3.4 La documentation

Notre classe `Eleve` est une structure de données qui peut être utilisée dans différents programmes par différents programmeurs, il est donc important voire primordial de bien la documenter.

Cette documentation doit montrer à minima comment sont construites les instances, quels sont les attributs et les méthodes disponibles.

Cette documentation est accessible via l'instruction : `help(Eleve)`

Par exemple, voici une documentation possible :

```
1 class Eleve:
2     '''
3     Creation d'une instance eleve:
4     eleve = eleve(nom(str), prenom(str), date(str),
          note1(float),note2(float),note3(float))
5
```

```
    attributs d'instance : nom,prenom,date,note_mat1,note_mat2,note_mat3
    attributs de classe : matiere1, matiere2, matiere3
    methode : moyenne() retourne la moyenne de l'eleve
    '''
```

C'est d'autant plus important lorsque si cette classe se trouve dans un autre fichier.

A faire :

- Copier-coller la classe `Eleve` dans un fichier que vous enregistrerez sous le nom `Maclasse.py`
- Créer un autre fichier `mon_programme.py`, dans lequel vous écrirez :

```
from Maclasse import Eleve
eleve5 = Eleve('Onette','Camille','01/08/2006',18,17,19)
print(eleve5.prenom,eleve5.nom,':',eleve5.moyenne())
```

Attention, les deux fichiers doivent être dans le même dossier !

La documentation est toujours très importante car si vous donnez cette classe à un autre programmeur, il faut qu'il puisse savoir comment l'utiliser sans avoir à en décortiquer le code...

Remarques :

Nous avons construit une structure de données répondant à un cahier des charges qui n'est pas trop ambitieux.

On pourrait se poser les questions suivantes :

- Que faire si un élève n'a pas de notes dans une des matières ?
- Comment rendre impossible la saisie d'une note supérieure à 20 ?
- Comment rendre impossible la saisie d'une note inférieure à 0 ?
- Que faire lors de la saisie d'une note non numérique, ..., *etc.*

4 Exercices

Exercice 1 : *Dominos*

1. Écrire une classe `Domino` pour représenter une pièce de domino. Les objets sont initialisés avec les valeurs des deux faces, A et B.
2. Ajouter une méthode `affichePoints(self)` qui affiche les valeurs des deux faces, et une méthode `total(self)`qui retourne la somme des deux valeurs.

Exercice 2 : *Compte bancaire*

1. Écrire une classe `CompteBancaire`. Les objets sont initialisés avec le nom du titulaire et le solde. L'argument solde doit être facultatif et avoir une valeur prédéfinie à zéro.
2. Ajouter deux méthodes `depot(self,somme)` et `retrait(self,somme)` pour changer le solde.
3. Ajouter une méthode `affiche(self)` qui montre le solde courant.

Exercice 3 : *Rectangle*

1. Écrire une classe `Rectangle`, permettant de construire un rectangle doté d'attributs longueur et largeur.
2. Ajouter deux méthodes `perimetre(self)` et `surface(self)`.

Exercice 4 : *Personnage*

1. Écrire une classe `Personnage`, avec son nom et ses points de vie comme attribut.
2. Ajouter une méthode `combat(self,other)` qui diminue de façon aléatoire les points de vie de l'un des personnages.

Exercice 5 : *Robot*

1. Écrire une classe `Robot`, avec ses coordonnées comme attributs et une direction (None par défaut).
2. Ajouter la méthode `avancer(self)`, qui permet au robot d'avancer d'une case dans la direction choisie.

Exercice 6 : *Personnage 2*

On considère une classe `Personnage` représentant un personnage de jeu. Le plateau de jeu est représenté par un repère orthonormé à trois axes. La position du joueur dans le plateau est repérée par ses attributs `x`, `y` et `z`.

1. Ecrire un constructeur initialisant les mesures.
2. Ecrire les méthodes `avance`, `droite` et `saute` permettant respectivement de faire avancer, aller à droite et sauter le personnage, c'est-à-dire d'augmenter de 1 respectivement x, y et z.
3. Implémenter une autre méthode coord renvoyant les coordonnées sous forme d'un triplet.
4. Essayer avec : `Laura = Personnage(0, 0, 0)`

Exercice 7 : *Mystère*

Voici un programme en Python :

```
import random

class Piece :

    def alea(self) :
        return random.randint(0,1)

    def moyenne(self,n):
        tirage = [ ]
        for i in range(n) :
            tirage.append(self.alea())
        return sum(tirage)/n

p = Piece()
print(p.moyenne(100))
```

Expliquer en détail ce que ce programme permet d'afficher.

Exercice 8 : *Voitures*

1. Créer une classe `Voiture` avec deux attributs d'instance :
   - `couleur`, qui stocke la couleur de la voiture sous forme de chaîne de caractères
   - `kilometrage`, qui stocke le nombre de kilomètres sur la voiture sous forme d'entier.

2. Instancier deux objets `Voiture` :
   - une voiture bleue de 20 000 kilomètres
   - une voiture rouge de 30 000 kilomètres.

3. Afficher leurs couleurs et leur kilométrage.

   Votre sortie devrait ressembler à ceci :

```
La voiture bleue a 20 000 kilometres.
La voiture rouge a 30 000 kilometres.
```

Exercice 9 : *Cinéma*

On considère les définitions de classe suivantes :

```
class Personne:
    """Objet representant une personne"""

    def __init__(self, nom: str, annee_naissance: int, lieu_naissance:
        str):
         self.nom = nom
         self.annee_naissance = annee_naissance
         self.lieu_naissance = lieu_naissance


class Film:
    """Objet representant un film"""

    def __init__(self, titre: str, realisateur: Personne):
         self.titre = titre
         self.realisateur = realisateur
```

1. Comment créer une instance de la classe `Personne` appelée `lautner` pour le réalisateur Georges Lautner né en 1926 à Nice ?
2. Proposer la définition d'une méthode `__str__` dans la classe `Personne` qui afficherait `"Georges Lautner est une personne née à Nice en 1926"` lors de l'appel `print(lautner)`.
3. On crée une instance de la classe `Film` avec l'instruction suivante :
   `tonton = Film("Les tontons flingueurs", lautner)`.

   - Qu'affiche l'instruction : `print(tonton.titre)` ?

   - Qu'affiche l'instruction : `print(tonton.réalisateur.nom)` ?
4. - Proposer la définition d'une méthode `__str__` dans la classe `Film` qui afficherait `"Les tontons flingeurs est un film réalisé par Georges Lautner originaire de Nice"` lors de l'appel `print(tonton)`.

Exercice 10 : *Jeu de dominos*

Le domino est un jeu très ancien composé de 28 pièces toutes différentes. Sur chacune de ces pièces, il y a deux côtés constitués de 0 (blanc) à 6 points. Lorsque deux côtés possèdent le même nombre de points, on l'appelle domino double.

1. Proposer une classe `Domino` permettant de représenter une pièce. Les objets seront initialisés par les valeurs portées par les des deux côtés (gauche et droite). On définit des méthodes `est_double` et `est_blanc` pour tester si le domino est double ou blanc.
2. Ajouter une méthode `affiche` qui affiche les valeurs des deux faces de manière horizontale pour un domino classique et de manière verticale pour un domino double.
3. Proposer une classe `JeuDeDomino` permettant de manipuler le jeu de domino complet. On créera une méthode pour mélanger le jeu et pour une autre distribuer selon 2 joueurs.

   On pourra utiliser la méthode `random.shuffle(mylist)`.

   Rappel :

```
>>> import random
mylist = ['apple', 'banana', 'cherry']
random.shuffle(mylist)
>>> mylist
['banana', 'apple', 'cherry']
```

En utilisant cette classe, on devrait obtenir le tirage au sort de 7 dominos pour chacun des 2 joueurs.

Par exemple :

```
>>> mon_jeu=JeuDeDomino()
>>> mon_jeu.montre()
Joueur 1
- - - - - - - - - -
|                 |
|   6   |   3   |
|                 |
- - - - - - - - - -
- - - - - - - - - -
|                 |
|   6   |   5   |
|                 |
- - - - - - - - - -
- - - -
|       |
|   0   |
|- - - -|
|   0   |
|       |
- - - -
- - - - - - - - - -
|                 |
|   5   |   4   |
|                 |
- - - - - - - - - -
- - - - - - - - - -
|                 |
|   3   |   1   |
|                 |
- - - - - - - - - -
```

```
- - - - - - - - -
|               |
|   5   |   0   |
|               |
- - - - - - - - -
- - - - - - - - -
|               |
|   3   |   2   |
|               |
- - - - - - - - -

------------------------------
Joueur 2
- - - - - - - - -
|               |
|   1   |   0   |
|               |
- - - - - - - - -
- - - - - - - - -
|               |
|   2   |   1   |
|               |
- - - - - - - - -
- - - - - - - - -
|               |
|   6   |   4   |
|               |
- - - - - - - - -
- - - -
|       |
|   2   |
|- - - -|
|   2   |
|       |
- - - -
- - - - - - - - -
|               |
|   3   |   0   |
|               |
- - - - - - - - -
- - - - - - - - -
|               |
|   6   |   2   |
|               |
- - - - - - - - -
- - - - - - - - -
|               |
|   2   |   0   |
|               |
- - - - - - - - -
```

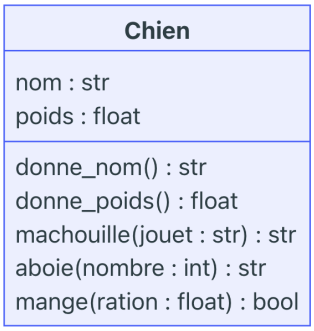
Exercice 11 : *Chiens*

On souhaite dans cet exercice créer une classe `Chien` ayant deux attributs :

- un nom `nom` de type `str`,
- un poids `poids` de type `float`.

Cette classe possède aussi différentes méthodes décrites ci-dessous (`chien` est un objet de type `Chien`) :

- `chien.donne_nom()` qui renvoie la valeur de l'attribut `nom` ;
- `chien.donne_poids()` qui renvoie la valeur de l'attribut `poids` ;
- `chien.machouille(jouet)` qui renvoie son argument, la chaine de caractères `jouet`, privé de son dernier caractère ;
- `chien.aboie(nb_fois)` qui renvoie la chaine `'Ouaf' * nb_fois`, où `nb_fois` est un entier passé en argument ;
- `chien.mange(ration)` qui modifie l'attribut `poids` en lui ajoutant la valeur de l'argument `ration` (de type `float`).



On ajoute les contraintes suivantes concernant la méthode `mange` :

- on vérifiera que la valeur de `ration` est comprise entre 0 (exclu) et un dixième du poids du chien (inclus),
- la méthode renverra `True` si `ration` satisfait ces conditions et que l'attribut `poids` est bien modifié, `False` dans le cas contraire.

Exemples :

```
>>> medor = Chien('Medor', 12.0)
>>> medor.donne_nom()
'Medor'
>>> medor.donne_poids()
12.0
>>> medor.machouille('baton')
'bato'
>>> medor.aboie(3)
'OuafOuafOuaf'
>>> medor.mange(2.0)
False
>>> medor.mange(1.0)
True
>>> medor.donne_poids()
13.0
>>> medor.mange(1.3)
True
```

Compléter le code de la classe `Chien` ci-dessous :

```
class Chien:
    def __init__(self, nom, poids):
        self.... = nom
        self.... = poids

    def donne_nom(self):
        return self....

    def ...(self):
        return self....

    def machouille(self, jouet):
        resultat = ""
        for i in range(...):
            resultat += jouet[...]
        return ...

    def ...(self, ...):
        ...

    def ...(self, ration):
        if ...:
            ...
            return True
    else:
        return ...


# Tests
medor = Chien('Medor', 12.0)
assert medor.donne_nom() == 'Medor'
assert medor.donne_poids() == 12.0
assert medor.machouille('baton') == 'bato'
assert medor.aboie(3) == 'OuafOuafOuaf'
assert not medor.mange(2.0)
assert medor.mange(1.0)
assert medor.donne_poids() == 13.0
assert medor.mange(1.3)
```

Exercice 12 : *Trains*

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

Un objet de type `Wagon` possède deux attributs :

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

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

Un objet de la classe `Train` possède deux attributs :

- `premier` contient son premier wagon (de type `Wagon`) ou `None` si le train est vide (il n'y a que la locomotive),

- `nb_wagons` (de type int) contient le nombre de wagons attachés à la locomotive.

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

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

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

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

Exemples :

- Création d'un train vide :

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

- Ajout de wagons :

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

- Description du train :

```
>>> train.est_vide()
False
>>> train.donne_nb_wagons()
3
>>> train.transporte_du('ble')
True
>>> train.transporte_du('materiel')
False
```

- Suppression de wagon :

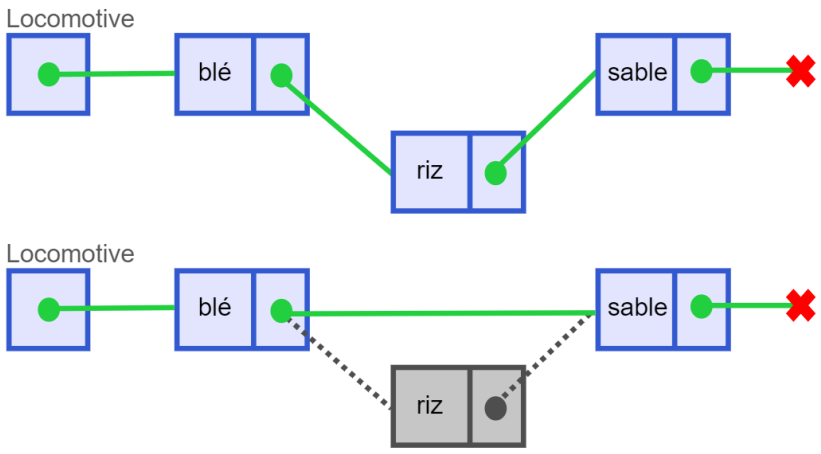
```
>>> train.supprime_wagon_de('riz')
True
>>> train
'Locomotive - Wagon de ble - Wagon de sable'
>>> train.supprime_wagon_de('riz')
False
```

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

```
wagon = self.premier
while wagon is not None:
    wagon = wagon.suivant
```

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

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



Compléter le code ci-dessous :

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

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

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

    def transporte_du(self, contenu):
        """Determine si ce train transporte du {contenu} (une chaine de
           caracteres).
        Renvoie True si c'est le cas, False sinon
        """
        wagon = self.premier
```

```
        while wagon is not None:
            if wagon.contenu == ...:
                return ...
            ... = wagon....
        return ...

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

    def supprime_wagon_de(self, contenu):
        """Supprime le premier wagon de {contenu}
        Renvoie False si ce train ne contient pas de {contenu},
        True si la suppression est effectuee
        """
        # On parcourt le train afin de trouver le contenu
        precedent = None
        wagon = self.premier
        while wagon is not ... and wagon.contenu != ...:
            precedent = wagon
            wagon = wagon....

        if wagon is ...:  # on a parcouru tout le train sans trouver le
            contenu
            return ...
        if precedent is ...:  # le wagon supprime est le premier du
            train
            self.premier = wagon....
        else:  # le wagon supprime n'est pas le premier
            precedent.... = wagon....
        self.nb_wagons -= ...
        return ...

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

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


# Tests
train = Train()
w1 = Wagon("ble")
train.ajoute_wagon(w1)
w2 = Wagon("riz")
```

```
train.ajoute_wagon(w2)
train.ajoute_wagon(Wagon("sable"))
assert str(train) == 'Locomotive - Wagon de ble - Wagon de riz - Wagon
    de sable'
assert not train.est_vide()
assert train.donne_nb_wagons() == 3
assert train.transporte_du('ble')
assert not train.transporte_du('materiel')
assert train.supprime_wagon_de('riz')
assert str(train) == 'Locomotive - Wagon de ble - Wagon de sable'
assert not train.supprime_wagon_de('riz')
```