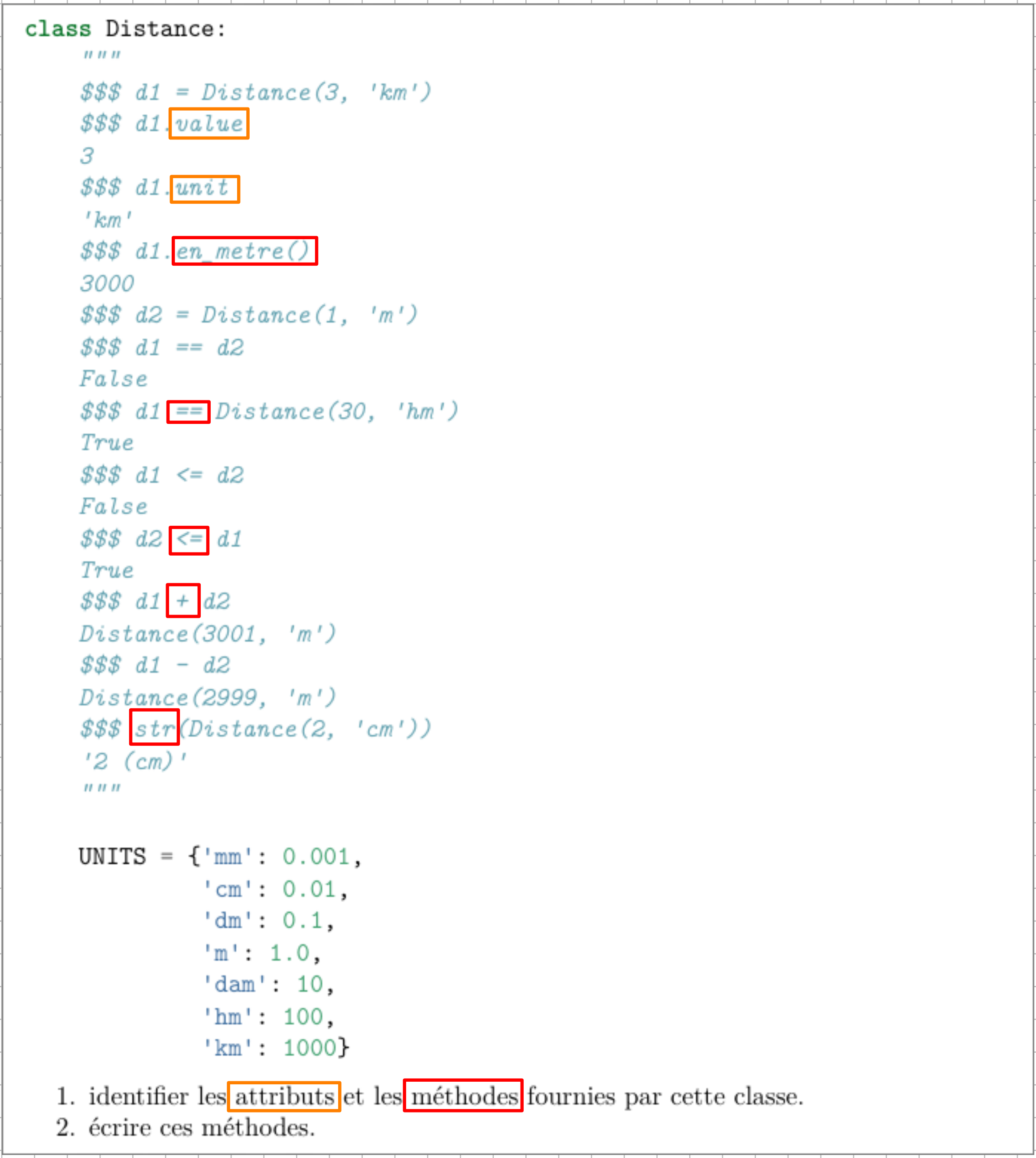
TD4: AP.

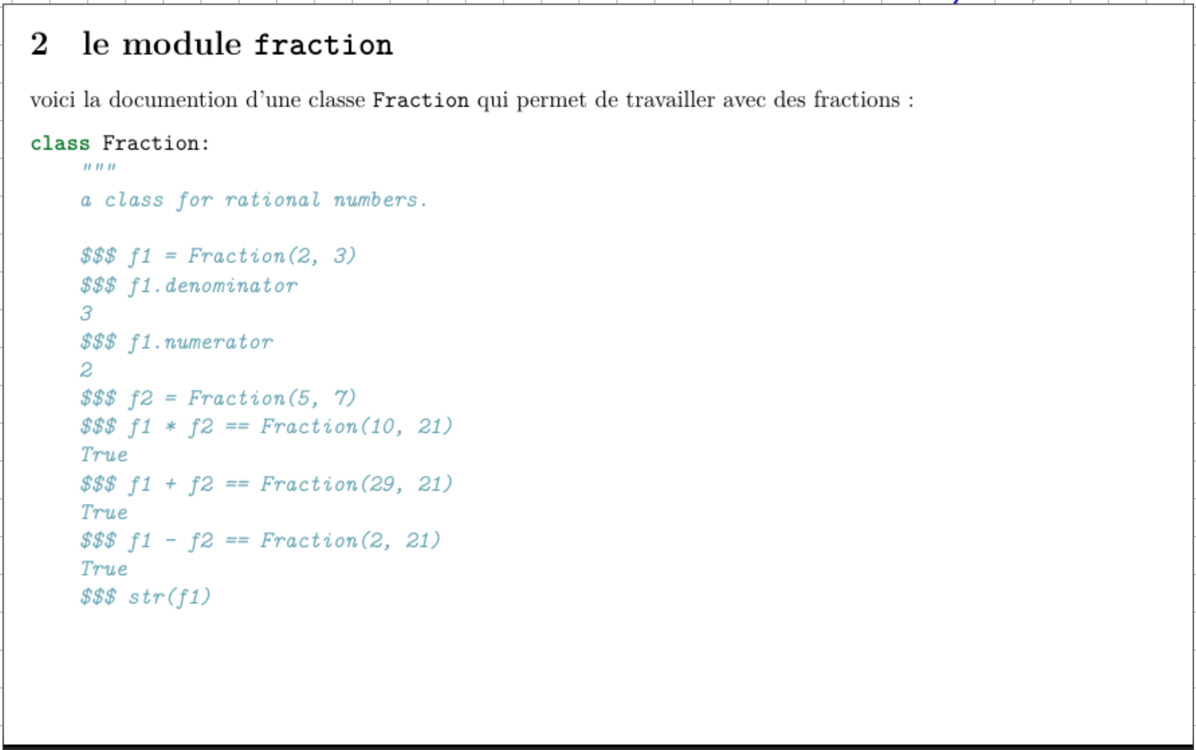


```
class Distance:
    """
    $$$ d1 = Distance(3, 'km')
    $$$ d1.value
    3
    $$$ d1.unit
    'km'
    $$$ d1.en_metre()
    3000
    $$$ d2 = Distance(1, 'm')
    $$$ d1 == d2
    False
    $$$ d1 == Distance(30, 'hm')
    True
    $$$ d1 <= d2
    False
    $$$ d2 <= d1
    True
    $$$ d1 + d2
    Distance(3001, 'm')
    $$$ d1 - d2
    Distance(2999, 'm')
    $$$ str(Distance(2, 'cm'))
    '2 (cm)'
    """

    UNITS = {'mm': 0.001,
             'cm': 0.01,
             'dm': 0.1,
             'm': 1.0,
             'dam': 10,
             'hm': 100,
             'km': 1000}
```

1. identifier les attributs et les méthodes fournies par cette classe.
2. écrire ces méthodes.

```
def __stR__(self):
    return f"{self.value} ({self.unit})"

def en_metre(self)-> float:
    Return self.value * UNITS[self.unit].

def __eq__(self, d2 : Self) -> bool :
    Return self.en_metre() == d2.en_metre()

def __le__(self, d2: Self)-> bool:
    Return self.en_metre() <= d2.en_metre().
```

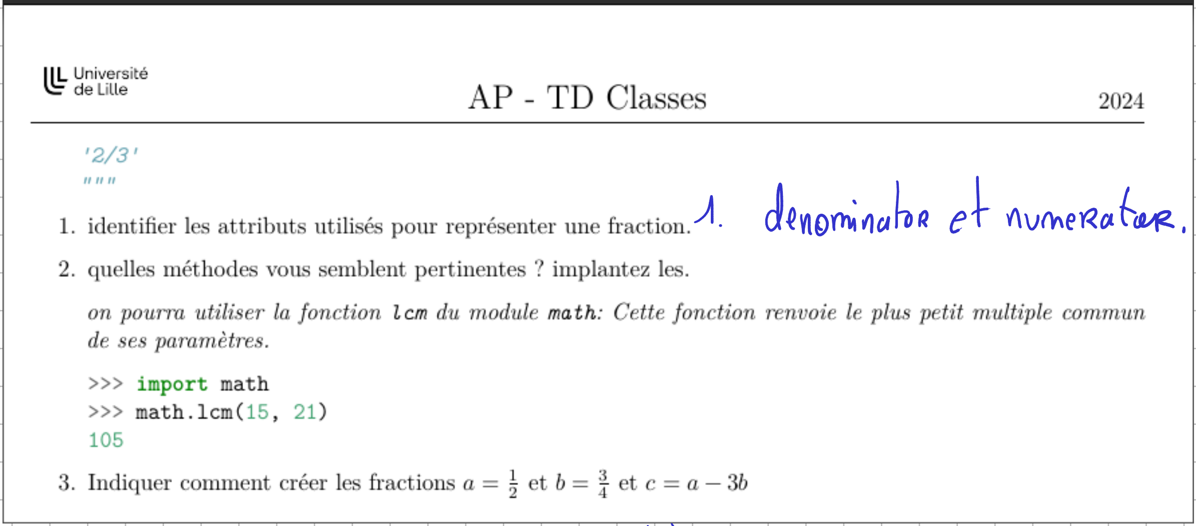
```
def __add__(self, d2: Self) -> Self:
    return Distance(self.en_metre() + d2.en_metre(), 'm').

def __sub__(self, d2: Self) -> Self:
    return Distance(self.en_metre() - d2.en_metre(), 'm').
```



2 le module fraction

voici la documention d'une classe **Fraction** qui permet de travailler avec des fractions :

```
class Fraction:
    """
    a class for rational numbers.

    $$$ f1 = Fraction(2, 3)
    $$$ f1.denominator
    3
    $$$ f1.numerator
    2
    $$$ f2 = Fraction(5, 7)
    $$$ f1 * f2 == Fraction(10, 21)
    True
    $$$ f1 + f2 == Fraction(29, 21)
    True
    $$$ f1 - f2 == Fraction(2, 21)
    True
    $$$ str(f1)
```



```
    '2/3'
    """
```

1. identifier les attributs utilisés pour représenter une fraction. 1. denominator et numerator.
2. quelles méthodes vous semblent pertinentes ? implantez les.

   on pourra utiliser la fonction `lcm` *du module* `math`*: Cette fonction renvoie le plus petit multiple commun de ses paramètres.*

   ```
   >>> import math
   >>> math.lcm(15, 21)
   105
   ```
3. Indiquer comment créer les fractions $a = \frac{1}{2}$ et $b = \frac{3}{4}$ et $c = a - 3b$

```
2. def __mult__(self, f2: Self|float) -> Self
     a = self.numerator; b = self.denominator
     if type(f2) == Self:
        c = f2.numerator; f2.denominator.
        gcd = math.gcd(a*c, b*d)
        return Fraction(a*c/gcd, b*d/gcd)
    else:
        gcd = math.gcd(a*f2, b)
        return Fraction((a*f2)/gcd, b/gcd)
```

```
def __add__ (self, f2:Self)-> Self:
    a= self.numerator; b=self.denominator
    c= f2.numerator; f2.denominator.
    gcd = math.gcd (a*d+b*c, b*d)
    Return Fraction(a*d+b*c/gcd, b*d/gcd).

def __sub__ (self, f2:Self)-> Self:
    Return self + Fraction(-f2.numerator, f2.denominator).

def __eq__(self, f2:Self)-> bool:
    Return (self - f2).numerator == 0.
```

3.
```
a= Fraction(1,2)
b= Fraction(3,4)
c= a - b*3.
```

3 Les cartes

Dans cet exercice on souhaite réaliser un module pour représenter des cartes à jouer. Ce module sera nommé `card` et définira une unique classe nommée `Card`.

Nous considérons ici des cartes à jouer caractérisées par

- une valeur qui est un élément de l'ensemble $\{A, 2, 3, 4, 5, 6, 7, 8, 9, 10, J, Q, C, K\}$ (J pour jack, C pour knight (le cavalier des jeux de tarot), Q pour queen, K pour king et A pour ace) ; vous pourrez supposer que cet ensemble de valeurs est défini dans la classe `Card` par la constante

  `VALUES = ("Ace", "2", ..., "9", "10", "Jack", "Knight", "Queen", "King")` ;

- une couleur qui est un élément de l'ensemble $\{♣, ♢, ♡, ♠, \}$; vous pourrez supposer que cet ensemble de couleurs est défini dans la classe par la constante

  `COLORS = ("spade", "heart", "diamond", "club")`.

1. On suppose déjà réalisé constructeur `__init__` qui à partir d'une valeur et d'une couleur valides renvoie une carte de cette valeur et de cette couleur. Ce constructeur initialise les attributs `color` et `value` représentant respectivement la couleur et la valeur d'une carte, couleur et valeur étant l'une des chaînes de caractères des constantes `Card.COLORS` et `Card.VALUES`

   ```
   >>> c1 = Card('Ace', 'heart')
   >>> c1.color
   'heart'
   >>> c1.value
   'Ace'
   ```

   Réalisez pour cette classe les méthodes suivantes :

   - `random` (sans paramètre) de création aléatoire d'une carte ;
   - `compare` qui renvoie un entier négatif si la carte passée en paramètre est inférieur à l'objet propriétaire de la méthode, un entier positif si elle est supérieure et un entier nul si elles sont égales (même couleur et même valeur).
     L'ordre total utilisé sur les cartes s'appuie d'abord sur l'ordre des valeurs tel que défini par la liste `VALUES` des valeurs (ainsi Ace < 2 < 3 < ... < King) puis en cas d'égalité sur l'ordre défini par la liste `COLORS` des couleurs (ainsi on a spade < heart < diamond < club).

```
def Random (self):
    Return Card( choice(VALUES), choice (COLORS)).

def compare (self, c2: Self)-> int :
    v1 = VALUES.index(self.value) ; v2 = VALUES.index(c2.value)
    c1 = COLORS.index(self.color); c2 = COLORS.index(c2.color)
    if v1 == v2
        Return 0 if c1 == c2 else 1 if c1 > c2 else -1.
    Return 1 if c1 < c2 else -1 .
```

2. Réalisez maintenant les méthodes spéciales permettant de
   - utiliser les opérateurs de comparaison usuels du langage Python,

```
>>> c1 == c2
False
>>> c1 != c2
True
>>> c1 < c2
True
>>> c1 > c2
False
```

   - obtenir une représentation externe plus lisible des cartes.

```
>>> Card.random()
Card("2", "spade")
>>> print(c1)
Card("Ace", "heart")
```

3. En utilisant le module **card**, réalisez une fonction **deck** sans paramètre qui renvoie un jeu complet des 56 = 4 × 14 cartes sous forme d'une liste dans laquelle les cartes sont rangées par couleur d'abord et au sein d'une couleur par valeur.
4. Réalisez une fonction **deck** avec un paramètre indiquant si on veut un paquet de 56 cartes, un paquet de 52 cartes (poker), ou un paquet de 32 cartes (belote, manille ou piquet).
5. Proposez une réalisation du constructeur `__init__`.

```
def __eq__(self, c2: Self) -> bool :
    Return self.compare(c2) == 0 .

def __neq__(Self, c2: Self) -> bool:
    return not self == c2.
```

```
def __gt__(self, c2) -> bool:
    Return self.compare(c2) < 0

def __lt__(self, c2) -> bool:
    Return self.compare(c2) > 0

3. def deck():
       Res: list[Card] = []
         for c in COLORS:
           for v in VALUES:
             Res.append(Card(c, v))
       Return Res.

4. def deck(num_cards: int) -> list [Card].
       Res: list[Card] = []
         for c in COLORS:
           for v in VALUES[:num_cards//len(COLORS)].
             Res.append(Card(c, v))
       Return Res.

           values.pop(values.index('Jack')).
```