1 En autonomie

Les questions suivantes sont le minimum à savoir après le cours, à réaliser en autonomie.

1. En python, quel est le nom du constructeur ?
2. Qu'est-ce qu'une méthode ?
3. Quel est (habituellement) le nom du premier paramètre des méthodes ?

Voici la définition d'une classe :

```
class A:
    def __init__(self, x: int, y: str=""):
        """Initialise un objet de classe A.
        Précondition : \
        """
        self.num = x
        self.chaine = y
    def incremente(self):
        self.num = self.num + 1
    def __str__(self) -> str:
        return self.chaine * self.num
```

Donner le nom d'un attribut et d'une méthode.

4. On considère le code suivant

```
a = A(1, "De")
b = A(3, "Da")
c = A(1, "Do")
c.incremente()
c.incremente()
print(a, c, a, b)
```

qu'affiche le code ? Quelle valeur est associée à `c.num` ?

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
    $$$ f1 - f2 == Fraction(-1, 21)
    True
    $$$ str(f1)
    '2/3'
    """
```

1. Identifier les attributs utilisés pour représenter une fraction.
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 $a = \frac{1}{2}$, $b = \frac{3}{4}$ et $c = a - 3b$, puis afficher `c`.

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 $\{\spadesuit, \heartsuit, \diamondsuit, \clubsuit\}$; vous pourrez supposer que cet ensemble de couleurs est défini dans la classe par la constante

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

1. Le 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`. Ces attributs sont passés en paramètre du constructeur :

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

   Réaliser pour cette classe le constructeur et 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).

```
>>> c2 = Card.random()
>>> c2.value, c2.color
('Jack', 'heart')
>>> c2.compare(c1)
10
>>> c1.compare(c2)
-10
>>> c1.compare(c1)
0
```

2. Réaliser maintenant les méthodes spéciales permettant

   - d'utiliser les opérateurs de comparaison usuels du langage Python,

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

- d'obtenir une représentation externe plus lisible des cartes.

```
>>> Card.random()
Card("King", "diamond")
>>> print(c1)
Card("Ace", "heart")
```

3. En utilisant le module `card`, réaliser une fonction `deck` sans paramètre qui renvoie un jeu complet des $56 = 4 \times 14$ 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. Réaliser 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).

indications :

- les valeurs d'un paquet de 52 cartes sont `("Ace", "2", ..., "9", "10", "Jack", "Queen", "King")`;
- les valeurs d'un paquet de 32 cartes sont `("Ace", "7", ..., "9", "10", "Jack", "Queen", "King")`.

4 Ampoules et guirlandes

Le module `guirlandes.py` contient une classe `Ampoule` (dont on a omis la documentation) :

```
class Ampoule:
    def __init__(self, etat: str="ok", est_allume: bool=False):
        self.etat = etat
        self.est_allume = est_allume
    def __str__(self) -> str:
        if self.etat == "ok":
            if self.est_allume:
                return "O"
            return "X"
        return "R"
```

1. Qu'affiche le code suivant ?

```
from guirlandes import Ampoule
amp1 = Ampoule("ok")
amp1.est_allume = True
amp2 = Ampoule("ko", False)
print(amp1, amp2)
```

On admet que

- si l'état de l'ampoule n'est pas `"ok"`, alors elle affiche toujours "R" (pour Recycle-moi) ;
- sinon elle affiche "O" si elle est allumée et "X" dans le cas contraire.

La classe `GuirlandeParallele` dispose d'un attribut `ampoules` de type `list[Ampoule]`. Cet attribut est initialisé par le constructeur qui prend en paramètre le nombre d'ampoules contenu dans la guirlande.

2. Écrire le constructeur de la classe `GuirlandeParallele`

2. Écrire une méthode `bascule` de la classe `GuirlandeParallele` dont voici la documentation :

```
def bascule(self, power: bool):
    """Allume ou éteint la guirlande.

        Si power vaut True, alors les ampoules sont allumées ;
        si power vaut False, alors elles sont éteintes.

        précondition:          """
```

3. Écrire une méthode permettant l'affichage de la guirlande :

```
>>> guirl = GuirlandeParallele(10)
>>> print(guirl)
XXXXXXXXXX
```

```
>>> guirl.ampoules[3].etat="ko"
>>> guirl.bascule(True)
>>> print(guirl)
OOOROOOOOO
```

4. On souhaite écrire une classe `GuirlandeSerie` modélisant une guirlande dans laquelle les guirlandes sont branchées en série : la guirlande peut s'allumer si, et seulement si toutes les ampoules peuvent s'allumer (c'est-à-dire que tous les attributs `etat` valent `"ok"`).

Écrire une classe `GuirlandeSerie` possédant les mêmes méthodes que celles de la classe`GuirlandeParallele` et implantant ce comportement.

5. La classe `GuirlandeClignotante` est une guirlande qui fonctionne de la manière suivante : lorsqu'elle est allumée, seule une ampoule sur deux est allumée. De plus, elle possède une méthode `clignote` :

```
def clignote(self):
    """Fait clignoter la guirlande.

        Lorsque la guirlande est éteinte, cette fonction ne fait rien.
        Lorsque la guirlande est allumée, allume les ampoules éteintes et éteint celles allumées.
        """
```

Lors du premier appel à la méthode `bascule`, les ampoules d'indice pair sont allumées (et les autres éteintes).

```
>>> guirl = GuirlandeClignotante(20)
>>> print(guirl)
XXXXXXXXXXXXXXXXXXX
>>> guirl.clignote()
>>> print(guirl)
XXXXXXXXXXXXXXXXXXX
>>> guirl.bascule(True)
>>> print(guirl)
OXOXOXOXOXOXOXOXOXO
>>> guirl.clignote()
>>> print(guirl)
XOXOXOXOXOXOXOXOXOX
```

Proposer une implantation de la classe `GuirlandeClignotante`