

4.2

Création de classe sur un exemple

NSI TLE - JB DUTHOIT

- Il s'agit ici de créer l'objet "carte" tirée dans un jeu de 52 cartes.

4.2.1 Identification des attributs, des méthodes, des relations et de construction de l'interface

Dans cette section, on va s'intéresser à un jeu de 52 cartes, que l'on va créer en informatique.

Dans le monde réel, un jeu de cartes est défini par :

- Sa couleur : Carreau, Cœur, Pique ou Trèfle
- Sa valeur : On prendra ici la valeur indiquée sur la cartes pour les cartes de 2 à 10. On prendra 11 pour le valet, 12 pour la Dame, 13 pour le Roi et 14 pour l'As.
- Sa figure : aucune, Roi, Dame, Valet

On va ainsi créer une classe Carte.

Remarque

| Par convention, une classe s'écrit toujours avec une majuscule.

Code couleur :

- En rouge, le constructeur, qui est ici aussi une méthode publique.
- En vert, les attributs privés
- En bleu, les méthodes publiques
- En orange, la méthode privée

Carte
Attributs :
<ul style="list-style-type: none"> - Valeur - Couleur - Figure
Méthodes
<ul style="list-style-type: none"> - Construire(val,coul) - Obtenir_valeur() - Obtenir_couleur() - Obtenir_figure() - Attribuer_valeur(val) - Attribuer_couleur(coul) - Attribuer_figure(val)

Les attributs privés signifient qu'ils sont accessibles uniquement par les méthodes publiques. En principe, ils sont initialisés dans la méthode de construction.
 Les méthodes publiques seront utilisées pour manipuler notre objet.
 Les méthodes privées ne sont utilisables qu'en interne (au sein de la classe).

4.2.2 Implémentation en Python

```
class Carte:
    """ Classe définissant une carte , et
    caractérisé par
    - sa valeur
    - sa couleur
    - sa figure """
    pass
```

Remarque

On a documenté notre classe. On pourra accéder à ces informations avec la commande "`__doc__`". Ce commentaire est appelé *docstring*.

Pour créer une carte, il suffit de taper :

```
>>> carte1 = Carte()
```

On dit que `carte1` est une *instance* de la *classe* `Carte`.

Pour l'instant, cette classe est une "coquille vide", nous allons la construire.

Le constructeur Pour déterminer et initialiser les attributs d'un objet que l'on crée, on utilise la méthode appelée **constructeur**.

En Python, son nom est imposé : `__init__`

La variable `self`, dans les méthodes d'un objet, désigne l'objet auquel s'applique la méthode. Elle représente l'objet dans la méthode en attendant qu'il soit créé.

On retrouve les 3 attributs, précédés d'un double underscore.

⚠️ Les underscore servent à "sécuriser" les attributs. Sans eux, ils peuvent être modifiés facilement en dehors de toute méthode, et cela peut poser des problèmes de cohérence.

☞ Essai sans le double underscore

```
class Carte:
    """ Classe définissant une carte , et
    caractérisé par
    - sa valeur
    - sa couleur
    - sa figure """
    def __init__(self, val, coul): #notre méthode constructeur
        """constructeur de cartes"""
        self.__valeur = val
        self.__couleur = coul
        if val == 11:
            self.__figure = "Valet"
        elif val == 12:
            self.__figure = "Dame"
```

```

        elif val == 13:
            self.__figure = "Roi"
        elif val == 14:
            self.__figure = "As"
        else:
            self.__figure = "aucune"
    
```

Nous pouvons maintenant *instancier* notre classe :

```
ma_carte = Carte(11,"Trèfle")
```

Tester ensuite les instructions suivantes :

```

>>> print(ma_carte)
>>> print(ma_carte.__doc__)
>>> print(ma_carte.__init__.__doc__)

```

Savoir-Faire 4.4

SAVOIR CRÉER LE CONSTRUCTEUR D'UN OBJET

Supposons que l'on ait envie de manipuler des triplets d'entiers représentant des temps mesurés en heures, minutes et secondes. On appellera la structure correspondante *Chrono*. Créer la classe Chrono avec uniquement le constructeur (dans un premier temps). Ne pas oublier de documenter.

Créer ensuite un objet (ou une instance de la classe).

Faire des essais en utilisant ou non les doubles underscore.

 : Dans l'état actuel des choses (avec les doubles underscores) , on ne peut accéder ni modifier les attributs puisqu'ils sont privés !

On va le faire au travers de méthodes publiques.

4.2.3 Les méthodes dédiées

Il existe deux familles de méthodes :

- Les accesseurs : pour obtenir la valeur d'un attribut
- Les mutateurs : pour modifier un attribut

Les Accesseurs (getters)

```

def GetValeur(self):
    "retourne la valeur de la carte"
    return self.__valeur
def GetCouleur(self):
    "retourne la couleur de la carte"
    return self.__couleur
def GetFigure(self):
    "retourne la figure de la carte"
    return self.__figure

```

On accède aux valeurs des attributs :

```
>>> ma_carte.GetValeur()
```

Les mutateurs (setters)

Pour les mutateurs, on souhaite rendre publique la modification de la valeur et de la couleur. Par contre on souhaite garder la main sur la façon d'attribuer une figure à notre carte (pour des questions de cohérence, et ainsi évité d'avoir un Roi d'une valeur 4 par exemple).

Le mutateur correspondant à l'attribut figure sera en accès privé : son nom sera précédé d'un double underscore.

```
def SetValeur(self, val):
    if 2 <= val <= 14:
        self.__valeur = val
        self.__SetFigure(val)
        return True
    else:
        return False
def Setcouleur(self, coul):
    liste = ["Trèfle", "Carreau", "Pique", "Coeur"]
    if coul in liste:
        self.__coul = coul
        return True
    else:
        return False
def __SetFigure(self, val):
    """ méthode pour changer la figure en fonction de
    la nouvelle valeur de val"""
    if val == 11:
        self.__figure = "Valet"
    elif val == 12:
        self.__figure = "Dame"
    elif val == 13:
        self.__figure = "Roi"
    elif val == 14:
        self.__figure = "As"
    else:
        self.__figure = "aucune"
```

Affichage de la carte

```
def affichage(self):
    return ('La carte est le ' + str(self.__valeur) + ' de ' +
           str(self.__couleur))
```

Un petit mot sur les exceptions

Pour contraindre l'utilisateur à entrer des valeurs pertinentes par rapport à l'objet créé, on peut utiliser *raise* pour lever des exceptions :

```
class Carte:
    """ Classe définissant une carte , et
    caractérisé par
    - sa valeur
```

```
- sa couleur
- sa figure """
def __init__(self, val, coul): #notre méthode constructeur
    """constructeur de cartes"""
    if type(val) != int:
        raise TypeError ("Entrer un entier comme valeur")
    if val >= 15 or val <=0:
        raise ValueError ("Entrer un entier entre 1 et 14
comme valeur")
    liste = ["Trèfle","Carreau","Pique","Coeur"]
    if coul not in liste:
        raise ValueError ("La couleur doit être Trèfle,
Carreau,Pique ou Coeur")
    self.__valeur = val
    self.__couleur = coul
```

4.2.4 Utilisation de la classe Carte comme module Python

- Enregistrer le fichier précédent sous le nom carte.py
- Créer un nouveau fichier python et écrire :

```
from carte import *
ma_carte = Carte(12,"Pique")
print(ma_carte.GetFigure())
```

Remarque

| Cela montre bien que l'on peut utiliser la classe Carte sans savoir comment elle est conçue !

Savoir-Faire 4.5

SAVOIR CRÉER DES MÉTHODES GETTERS ET SETTERS

On reprend le savoir-faire précédent, et on ajoute des méthodes afin de pouvoir utiliser et manipuler l'objet :

- Accès aux heures, minutes et secondes
- Possibilité de changer heure, minutes et secondes
- Possibilité d'ajouter un certain nombre de seconde , de minutes ou d'heures.
- Possibilité d'afficher le chrono

Exercice 4.56

Définir une classe Fraction pour représenter un nombre rationnel. Cette classe possède deux attributs : num et denom, qui sont des entiers, et désigne respectivement le numérateur et le dénominateur. On veut que le dénominateur soit un entier strictement positif.

- Ecrire le constructeur de cette classe, qui doit lever une ValueError si le dénominateur fourni n'est pas strictement positif.
- Ajouter une méthode qui simplifie la fraction sous forme irréductible.
- Ajouter une méthode qui renvoie une chaîne de caractère de la forme "45/3", ou de la forme "12" quand le dénominateur vaut 1, avec la fraction sous forme irréductible.
- Ajouter une méthode *Ajout* qui accepte en argument une fraction (sous forme de deux entiers avec dénom non nul) et qui ajoute cette fraction à l'objet.

Exercice 4.57

Écrire une classe Vector qui permet de représenter un vecteur dans le plan, composé de deux coordonnées x et y . Elle propose une méthode *norm* permettant de calculer la norme du vecteur, qui vaut, pour rappel, $\sqrt{x^2 + y^2}$. Créer aussi un affichage de l'objet.

Exercice 4.58

Écrire une classe Music qui représente une musique décrite par un titre, une liste d'artistes et une durée en secondes. Proposez une méthode *hasAuthor* qui teste si un artiste spécifié fait partie des artistes de la musique en question. Prévoyez également un affichage du contenu : "Si demain" par Bonnie Tyler, Karen Antonn".

Exercice 4.59

Définir une classe Intervalle représentant des intervalles de nombres. Cette classe possède deux attributs a et b représentant respectivement la borne inférieure et la borne supérieure. Les deux bornes sont incluses dans l'intervalle. Tout intervalle avec $b < a$ représente un intervalle vide.

- Écrire le constructeur de la classe Intervalle et une méthode *est_vide* renvoyant True si l'objet représente un intervalle vide, et False sinon.
- Créer une méthode *long* qui retourne la longueur de l'intervalle.
- Créer une méthode *appartient* qui accepte un nombre en argument et qui renvoie True si ce nombre est dans l'intervalle, et False sinon.
- Ajouter une méthode qui prend en argument un objet intervalle et qui détermine si cet objet entré en argument est inclus dans l'objet.

```
>>> I = Intervalle(1,10)
>>> I.Est_vide()
False
>>> I.Long()
9
>>> I.Appartient(5)
True
>>> I.Appartient(12)
False
>>> J= Intervalle(4,5)
>>> I.Est_inclus(J)
True
>>> K = Intervalle (4,13.2)
>>> I.Est_inclus(K)
False
```

4.2.5 Opérations sur deux objets

Approche générale

On reprend l'exemple de l'exercice avec la classe Vector. On aimera additionner deux objets vecteurs. Le résultat de l'opération est un nouveau vecteur dont les composantes sont les sommes des composantes des vecteurs additionnés.

```
def add(self, other):
    return Vector(self.x + other.x, self.y + other.y)
```

La méthode renvoie donc un nouvel objet de type Vector dont les composantes x et y s'obtiennent en faisant la somme des composantes de l'objet représenté par self (le vecteur de l'objet cible) avec celles de l'objet représenté par other (le vecteur reçu en paramètre).

```
u = Vector(1, -1)
v = Vector(2, 2)
u.add(v)
```

Surcharge d'opérateurs

En Python, il suffit de définir une méthode add qui accepte un paramètre qui est le vecteur à additionner. On remplace donc simplement la méthode add précédemment définie par :

```
def __add__(self, other):
    return Vector(self.x + other.x, self.y + other.y)
```

```
u = Vector(1, -1)
v = Vector(2, 2)
print(u + v)
```

Cette capacité du langage Python s'appelle la surcharge d'opérateur.

⚠ Ceci est spécifique au langage Python. Vous trouverez sur le net la liste des opérateurs de surcharge !