Remarque préliminaire : Jupyter est parfois capricieux pour le téléchargement des images. Si les images n'apparaissent pas dans le notebook, chargez les dans le même dossier que le notebook. Les adresses se trouvent en double cliquant dans les cellules de texte (là où il y a précisé "image", c'est qu'il y a une image normalement...). Puis changez le code comme ceci : `![Image](http://www.maths-info-lycee.fr/images/arbre1.jpg)` devient `![Image](imagearbre1.jpg)` ou même `![](imagearbre1.jpg)`

Introduction à la programmation objet

La programmation objet correspond à une manière peut-être plus naturelle pour les humains, de concevoir le fonctionnement d'un programme. En ce moment où vous lisez le notebook, plusieurs "objets" sont en action. Un écran qui affiche le notebook ; des yeux qui reçoivent la lumière ; une première partie du cerveau, le cortex visuel, qui transforme cette lumière en images ; et enfin une deuxième partie du cerveau, le cortex cérébral, qui transforme cette image en pensée. La communication se fait ici à sens unique, de l'écran vers la pensée.
Dans un deuxième temps, vous allez répondre aux questions posées plus loin. Il y aura interaction dans les deux sens, en rajoutant un autre objet, le clavier.
On est donc en présence d'objets ayant à la fois des caractéritiques, et des actions, qui leurs sont propres. Le clavier a comme caractéristiques ses touches, la manière dont il est connecté à l'ordinateur. Et ses actions peuvent être de communication vers l'ordinateur : envoyer le code d'une touche ; ou bien en provenance de l'ordinateur : configuration en azerty ou qwerty. Remarquez que l'objet clavier est ici défini de manière générale, on ne précise pas sa marque, son agencement de touches etc. De la même manière que l'on dit "j'ai pris la voiture" et non pas "j'ai pris la 2CV orange de 1964".

Les parties "exploration" de ce notebook ne demandent pas de programmer, juste de lire et de comprendre !

Exploration : retour sur des usages masqués de la classe de 1ère :

Comme M.Jourdain faisait de la prose sans le savoir, vous avez déjà utilisé l'an passé la programmation objet.

Regardons par exemple comment est programmée la fonction randint de la bibliothèque random :

In [ ]:
```
import random
import inspect

print(inspect.getsource(random.randint))
```

Excepté la présence d'un *self* surprenant (randint n'ayant besoin que de 2 paramètres a et b), rien de nouveau, on voit la définition d'une fonction très courte...

Inspectons maintenant tout le code de la bibliothèque random, on constate la présence de mots-clés comme class, une fonction **init** et si vous cherchez bien on y retrouve la fonction randint... mais inclue dans la définition de la classe

In [ ]:
```
print(inspect.getsource(random))
```

La fonction **init** présente en Python dans toute classe est appélée **constructeur** et la fonction randint est ce qu'on appelle une **méthode** de la classe. Ce vocabulaire est précisé dans la partie suivante, avec un exemple de création d'une nouvelle classe (car c'est tout l'intérêt de la programmation orientée objet, le programmeur définissant lui même ses propres classes avec ses méthodes et ses attributs)

Objectif belote ?

Nous allons utiliser ce principe d'"objets" pour jouer à la bataille, ou à d'autres jeux de cartes, comme la belote. La bataille est un jeu, qui utilise un paquet de cartes, constitué de... cartes. Le premier objet que l'on va créer est donc une carte "générique", comme une "voiture" générique. La **classe** Carte a :

- des **attributs/composants** (les caractérisques):
  - sa couleur (coeur, pique, carreau, trèfle)
  - sa hauteur (as ou 1, 2, ... jusqu'à roi)
  - sa valeur, qui n'est pas forcément la même que sa hauteur
  - *Remarque* : dans la majorité des langages de programation, les attributs sont censés être **privés**, c'est-à-dire qu'un objet extérieur qui veut y accéder ne peut pas les modifier directement. Il est obligé de passer par les méthodes de la classe. C'est le principe d'**encapsulation** des données. les attributs sont protégés d'une modification directe par un objet extérieur. En Python en terminale, on n'utilisera que des attributs publics : c'est-à-dire qu'ils seront directement modifiables, au risque de l'utilisateur qui pourra en faire n'importe quoi.
- des **méthodes** (les actions):
  - création/construction de la carte avec couleur, hauteur et valeur. Cette méthode particulière est le **constructeur**
  - Dans la majorité des langages (mais pas en Python donc) :
    - commmuniquer ses attributs (méthodes **get**). Ces méthodes sont des **accesseurs**
    - changer sa valeur (méthode **set**). Cette méthode est un **mutateur**
    - *Remarque* : on ne change pas la couleur ni la hauteur (sauf si on triche)
    - *Remarque* : les méthodes get et set sont publiques. Ce sont celles qui seront utilisées par les objets extérieurs pour interagir avec la carte.

Les méthodes permettent d'utiliser les objets à travers l'envoi de **messages**, le message étant l'utilisation de la méthode avec les paramètres associés dans la **signature** de la méthode.
Quand on crée une carte, on crée une **instance** de la classe carte. Pour cela, on utilise une méthode spéciale, le **constructeur** de la classe. Le constructeur crée l'objet en mémoire, et renvoie la référence sur l'objet (son adresse). En Python cette méthode est `__init__` . Dans certains mmorpg (cherchez meuporg sur youtube), ce vocabulaire d'instance est utilisé pour désigner une zone créée individuellement pour chaque groupe de joueurs. Les différents groupes explorent la même zone mais ne s'y croisent pas : ils ne sont pas dans la même instance. Une instance d'une calsse partage avec les autres instances les mêmes méthodes et la même structure, mais pas les mêmes valeurs des attributs.

Remarques :

- Une classe est une nouvelle structure de données, que l'on a construit. Cette structure a un comportement défini par ses méthodes. Une classe peut être vue comme un nouveau type.
- La programmation objet permet de découper plus facilement le travail à l'intérieur d'une équipe, chaque collaborateur pouvant programmer une classe indépendamment des autres. Un programmeur utilisant une classe créee par un de ses collègues n'a pas besoin de savoir "comment" ça marche, juste "ce qu'il peut faire" avec les objets de cette classe.

Exploration : première ébauche

Cette première version donne une classe dans laquelle les attributs sont publics. Comme on va le voir ci-après, les attributs publis sont accessibles à tout le monde. Et comme on l'a dit ci-dessus, on souhaite des attributs privés et non publics ; il est néanmoins intéressant de voir comment cela fonctionne

In [ ]:
```
class Carte:
    """
    Carte d'un jeu
    Attributs :
        couleur : chaine de caractères, normalement dans coeur, piq
ue, carreau, trèfle
        hauteur : entier ou chaine de caractères, dans 1, 2, ..., 1
0, valet, dame, roi
        valeur : entier
    Méthodes :
        constructeur __init__
    """
    def __init__(self,couleur,hauteur,valeur = 0):        #par défa
ut la valeur est à 0, on peut passer une
                                                          #valeur p
récise en paramètre ou pas
        self.couleur = couleur
        self.hauteur = hauteur
        self.valeur = valeur
```

Les instructions suivantes permettent de :

- lire les spécifications de la classe
- créer une carte
- récupérer un attribut
- modifier un attribut

Remarque : le script suivant montre comment un individu louche peut modifier un attribut public, pour en faire n'importe quoi. En fin de ce TD, un complément de code facultatif vous montre comment on procède en général (c'est-à-dire dans tous les langages objet... sauf Pyhton XD) pour éviter ces problèmes.

In [ ]:
```
print(Carte.__doc__)    # lecture des spécifications
roi_carreau = Carte('carreau','roi',13)    # création d'une carte
print(roi_carreau.hauteur, roi_carreau.couleur)     # lecture de la
hauteur et de la couleur de l'objet roiCarreau
print()
roi_carreau.couleur = 'fenetre'   # modification de la hauteur par
un individu mal intentionné
print(roi_carreau.hauteur, roi_carreau.couleur)    # et voilà le ré
sultat !
```

Complément/exploration : objets et référence mémoire

Une variable à laquelle on affecte un objet ne comprend pas l'objet, mais l'adresse mémoire de l'objet. On peut donc avoir plusieurs variables qui référencent le même objet, et ainsi modifier l'objet à l'aide de ces différentes variables. C'est une source d'erreurs potentielles.
Exemple :

In [ ]:
```
roi_carreau = Carte('carreau','roi',13)
roi_carreau_bis = Carte('carreau','roi',13)
roi_carreau_ter = roi_carreau
print(type(roi_carreau))
print(roi_carreau)
print(roi_carreau_bis)
print(roi_carreau_ter)
print("Classe Carte à l'adresse :",hex(id(Carte)))
print("roi_carreau à l'adresse :",hex(id(roi_carreau)))
print("roi_carreauBis à l'adresse :",hex(id(roi_carreau_bis)))
print("roi_carreauTer à l'adresse :",hex(id(roi_carreau_ter)))
print()
print("Attributs de roi_carreau : ", roi_carreau.hauteur, roi_carre
au.couleur, roi_carreau.valeur)
print("Attributs de roi_carreau_bis : ", roi_carreau_bis.hauteur, r
oi_carreau_bis.couleur,
      roi_carreau_bis.valeur)
print("Attributs de roi_carreau_ter : ", roi_carreau_ter.hauteur, r
oi_carreau_ter.couleur,
      roi_carreau_ter.valeur)

# Expliquer ce qui se passe lors des instructions suivantes, quel e
st le problème rencontré ? Justifier.
roi_carreau.hauteur = 'prince'
roi_carreau_bis.hauteur = 'empereur'

print()
print("Attributs de roi_carreau : ", roi_carreau.hauteur, roi_carre
au.couleur, roi_carreau.valeur)
print("Attributs de roi_carreau_bis : ", roi_carreau_bis.hauteur, r
oi_carreau_bis.couleur,
      roi_carreau_bis.valeur)
print("Attributs de roi_carreau_ter : ", roi_carreau_ter.hauteur, r
oi_carreau_ter.couleur,
      roi_carreau_ter.valeur)
```

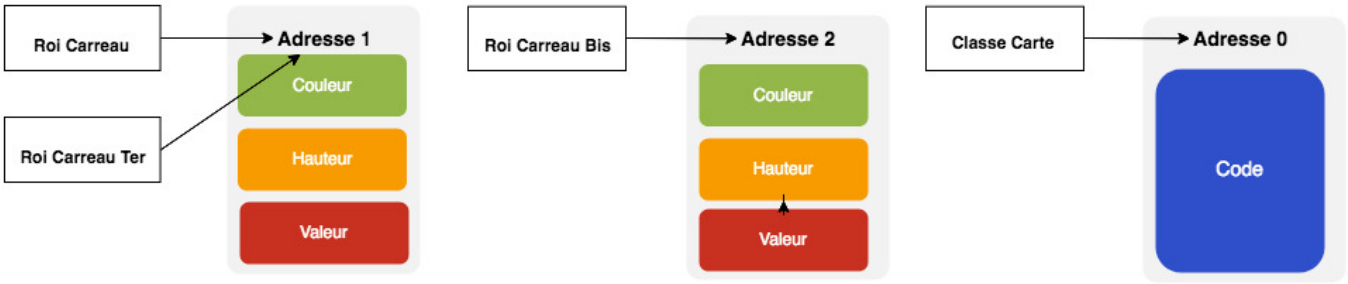
Origine du problème rencontré ci-dessus :

A compléter

Exploration : résumé graphique

En mémoire

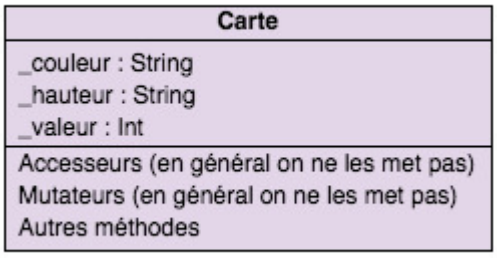
On a en mémoire la situation suivante, pour l'exemple précédent :



Remarque : le schéma précédent est outrageusement simplifié au point d'en être presque faux, mais ce qui compte c'est de comprendre le fonctionnement.

Représentation graphique d'une classe

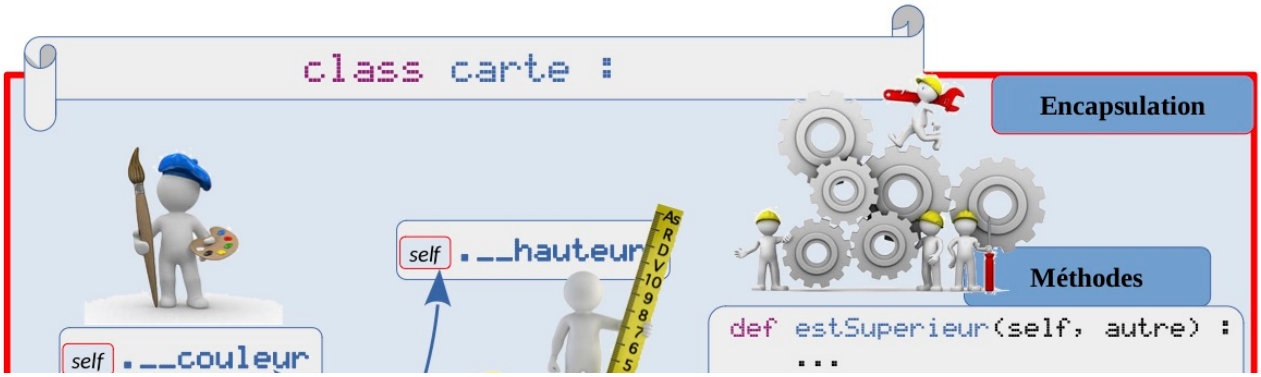
En général représente une classe sous cette forme :

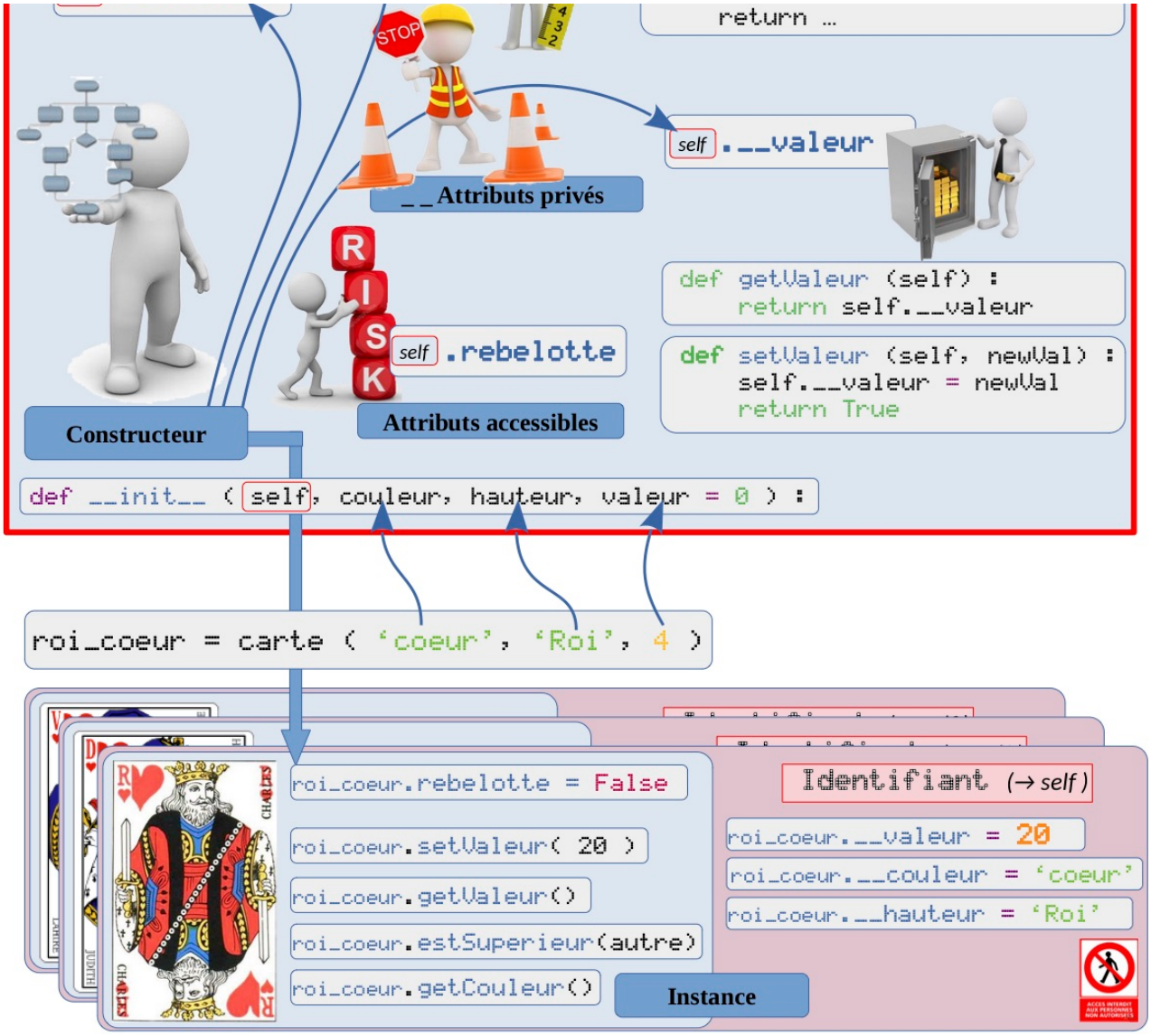


Ce qui se passe lorsque l'on crée et accède à un objet

Le schéma ci-dessous montre :

- l'encapsulation à l'intérieur de la classe ;
- la création d'un objet ;
- la modification des attributs.





On retrouve la situation des listes et autres types mutables.

La "bataille"

Avant de créer un jeu de belote, on va commencer par un jeu plus simple : la bataille. Pour jouer à la bataille, ou à tout autre jeu de cartes, il est nécessaire de pouvoir comparer des cartes entre elles. On crée deux méthodes, l'une pour "valeur strictement supérieure", l'autre pour "valeur égale".

Créer la méthode est_egale sur le même principe que "est_supérieure"..

On va également spécifier correctement notre classe. Dans la suite du notebook, pour des raisons de compacité/lisibilité, on ne spécifiera pas de manière aussi détaillée. Mais n'oubliez pas que les spécifications sont présentes à des fins de compréhension rapide, doivent être présentes, et complètes.

In [ ]:
```
class Carte:
    """Carte d'un paquet de cartes, pour jouer à différents jeux. O
n reste dans les paquets 32/52/54 cartes ou tarot

    Attributs :
        - couleur : chaine de caractères, en général coeur/carreau/
pique/trèfle, mais peut
            aussi être plus exotique batons/coupes/deniers/épées. O
n peut aussi avoir "atout" ou "joker"
        - hauteur : chaine de caractères, en général de "as" à "ro
i",
            variantes "un" à "vingt et un" popur les atouts, "aucun
e" pour les jokers
        - valeur : entier (en général), dépend du jeu.
            Dans un langage fortement typé c'est un flottant (valeu
rs 0.5 au tarot)

    Méthodes :
        init()
        estSuperieure(autre)  : renvoie un booléen vrai si l'objet
Carte est de valeur supérieure à celle
            d'un autre objet Carte
        estEgale(autre) : renvoie un booléen vrai si l'objet Carte
est de valeur égale à celle
            d'un autre objet Carte
    """

    def __init__(self,couleur,hauteur,valeur = 0):
        """
        Constructeur de la classe Carte
        @param:
            - couleur : chaine de caractères, en général coeur/carr
eau/pique/trèfle, mais peut
            aussi être plus exotique batons/coupes/deniers/épées. O
n peut aussi avoir "atout" ou "joker"
            - hauteur : chaine de caractères, en général de "as" à
"roi",
            variantes "un" à "vingt et un" popur les atouts, "aucun
e" pour les jokers
            - valeur : entier (en général), dépend du jeu.
            Dans un langage fortement typé c'est un flottant (valeu
rs 0.5 au tarot)
        Résultat :
            ne retourne rien, crée une nouvelle Carte
        """
        self.couleur = couleur
        self.hauteur = hauteur
        self.valeur = valeur

    # méthodes
    def est_superieure(self,autre):
        """
        Compare les valeurs de deux objets Carte
        @param : autre, objet de classe Carte
        @result : booléen Vrai si la valeur de la Carte self est su
```

```
périeure à la valeur de la Carte autre
        """
        return self.valeur > autre.valeur


    def est_egale(self,autre):
        # A compléter
        return

    #def __repr__(self):
        #return f'{self.hauteur} de {self.couleur}, valeur {self.va
leur}'
        #ou
        #return str(self.hauteur) + " de " + str(self.couleur) + ",
de valeur " + str(self.valeur)

roi_carreau = Carte('carreau','roi',13)
sept_pique =  Carte('pique','sept',7)
print(roi_carreau.est_superieure(sept_pique))
print(sept_pique.est_superieure(roi_carreau))
print(sept_pique.est_egale(sept_pique))
print(roi_carreau)
```

Dans la cellule précédente, quel est l'effet de l'instruction `print(roiCarreau)` ?

Décommentez les lignes de la méthode `__repr__` , et exécutez à nouveau la cellule. Quel est le changement ? Que fait la méthode `__repr__` ?

Objet composé d'objets

Dans un paquet de cartes, il y a plusieurs cartes. On va créer donc la classe Paquet.

On remarque que les listes des valeurs et des hauteurs possibles sont définies avant le constructeur. Ces variables sont partagées par toutes les instances de la classe : il n'y a qu'une seule copie de ces variables, créee lors du chargement de la classe. Ce sont des variables de classe, et il est interdit de les modifier.

In [ ]:
```
class PaquetCartes:
    """
    Paquet de cartes
    Attributs:
        - nom : nom du paquet, de préférence correspondant au nom d
u jeu pour
            lequel il va être utilisé
        - paquet : liste des cartes
        - nb_cartes : entier, soit 32 soit 52, le nombre de cartes
dans le paquet.
    """
    hauteurs = ["as","deux","trois","quatre","cinq","six","sept","h
uit","neuf","dix",
                "valet","dame","roi"]
    couleurs = ["coeur","pique","carreau","trèfle"]

    def __init__(self,nom,nbCartes = 32):
        """Constructeur du paquet de cartes"""
        self.nom = nom
        self.nb_cartes = nb_cartes
        self.paquet = []
        if nb_cartes == 32:
            for i in range(6,len(hauteurs)):
                for j in range(len(self.couleurs)):
                    self.paquet.append(Carte(couleurs[j],self.haute
urs[i],i + 1))
            for j in range(len(self.couleurs)):
                self.paquet.append(Carte(self.couleurs[j],self.haut
eurs[0],14))
        else:
            for i in range(1,len(self.hauteurs)):
                for j in range(len(self.couleurs)):
                    self.paquet.append(Carte(self.couleurs[j],self.
hauteurs[i],i+1))
            for j in range(len(self.couleurs)):
                self.paquet.append(Carte(self.couleurs[j],self.haut
eurs[0],14))

    def __repr__(self):
        chaine = ""
        for carte in self.paquet:
            chaine = chaine + carte.__repr__() + "\n"
        return chaine

paquet_bataille = PaquetCartes('bataille', 52)
print(len(paquet_bataille.paquet))
print(paquet_bataille)
print()
```

Testez le code précédent avec 32 cartes, et corrigez les erreurs.
Remarque : le paramètre `nbCartes` est renseigné par défaut à 32, il n'est pas focément nécessaire de le préciser lors de l'appel du constructeur. S'il est absent, il y aura 32 cartes dans le paquet, sinon il y aura 52 cartes.

Afficher les cartes

Quand on joue aux cartes, c'est assez important de savoir ce que l'on a en main, ou au moins de savoir quelle est la valeur de la carte jouée ! Affichons donc les 10 premières cartes avec le code suivant.

In [ ]:
```
#print(paquet_bataille)
for i in range(10):
    print(paquet_bataille.paquet[i])
```

Suivant que vous avez ou non défini la méthode `__repr__` dans la classe `Carte` , que constatez-vous ?

Si nécessaire, modifiez le le code des classes `Carte` et `PaquetCartes` , pour afficher les attributs de chaque carte et du paquet (utilisez `__repr__(self)`). Il peut être nécessaire de relancer les cellules avec les classes `Carte` et `PaquetCartes` . *Vous pouvez aussi recopier les deux classes dans la cellule suivante, pour ne pas avoir à relancer plusieurs cellules à chaque fois.*
Vérifiez également que la valeur des as est correcte

In [ ]:
```
class Carte:

class PaquetCartes:


paquetBataille = PaquetCartes('bataille', 52)
```

Avec les objets composés d'objets, les méthodes "enchainées"

On peut écrire des instructions appliquant une méthode sur le résultat d'une autre méthode :

In [ ]:
```
paquet_bataille.paquet[-3].valeur
```

Pour pouvoir jouer, il nous manque encore quelques méthodes dans notre classe PaquetCartes. Créer :

- la méthode `melange(self)` qui mélange le paquet comme son nom l'indique. Cette méthode renvoie `self` , elle modifie l'attribut `paquet` . On utilisera la méthode `shuffle(tableau_à_mélanger)` de la bibliothèque `random`
- la méthode `distribution_2_joueurs` beaucoup plus simple, où le paquet est divisé en deux (`donne` contient alors 2 listes). Pour l'instant l'objectif est de jouer à la bataille et rien de plus complexe.
- si vous le souhaitez, créezla méthode `distribution(nb_joueurs,nb_a_distribuer = 5)` qui renvoie une `donne` , une liste de listes de Cartes. Il y a autant de listes que de joueurs. On donne le nombre de cartes indiquées à chaque joueur (on peut supposer que ce nombre est compatible avec la taille du paquet, cela serait à préciser dans les spécifications). S'il reste des cartes après distribution, une dernière liste sera ajoutée avec celles-ci (la pioche).

Remarque : l'affichage des donnes de chaque joueur relève plutôt de la classe gérant le jeu. En effet dans certains jeux les cartes sont inconnues du joueur.

Vous pouvez modifier votre code dans la cellule précédente, ou bien recopier et mdoifier dans la cellule suivante.

In [ ]:
```
from random import shuffle #import random as rd -> rd.shuffle

class PaquetCartes:
    """
    Paquet de cartes
    Attributs:
        - nom : nom du paquet, de préférence correspondant au nom d
u jeu pour
            lequel il va être utilisé
        - paquet : liste des cartes
        - nb_cartes : entier, soit 32 soit 52, le nombre de cartes
dans le paquet.
    Méthode
        - constructeur
        - melange : mélange l'attribut paquet
        - distribution_2_joueurs : distribue tout le paquet de cart
es en deux mains. Renvoie donne, la liste des deux
            mains, qui sont des listes d'objets Carte de l'attribut
paquet
        - distribution(nb_joueurs, nb_a_distribuer = 8) : distribue
un certain nombre de cartes à un certain
            nombre de joueurs. L'utilisateur vérifie la cohérence d
u nombre de cartes dans le paquet avec
            le nombre de joueurs et le nombre de cartes à distribue
r.
            Renvoie donne, la liste des mains, qui sont des listes
d'objets Carte de l'attribut paquet
    """
    hauteurs = ["as", "deux", "trois", "quatre", "cinq", "six",
                "sept", "huit", "neuf", "dix", "valet", "dame", "ro
i"]
    couleurs = ["coeur","pique","carreau","trèfle"]

    def __init__(self,nom,nbCartes = 32):
        """Constructeur du paquet de cartes"""
        self.nom = nom
        self.nb_cartes = nb_cartes
        self.paquet = []
        if nb_cartes == 32:
            for i in range(6, len(self.hauteurs)):
                for j in range(len(self.couleurs)):
                    self.paquet.append(Carte(self.couleurs[j], sel
f.hauteurs[i],i + 1))
            for j in range(len(self.couleurs)):
                self.paquet.append(Carte(self.couleurs[j],self.haut
eurs[0],14))
        else:
            for i in range(1,len(self.hauteurs)):
                for j in range(len(self.couleurs)):
                    self.paquet.append(Carte(self.couleurs[j],self.
hauteurs[i],i+1))
            for j in range(len(self.couleurs)):
                self.paquet.append(Carte(self.couleurs[j],self.haut
eurs[0],14))
```

```
    def melange(self) :
        # à compléter
        pass  # instruction ne faisant rien, permet d'éviter une er
reur de syntaxe


    def distribution_2_joueurs(self):
        # la plus simple est de ne pas distribuer les cartes une pa
r une, à la place :
        # On donne la 1ère moitié du paquet au 1er joueur
        # On donne la 2ème moitié du paquet au 2ème joueur
        donne = []
        return donne

    def distribution(self, nbJoueurs, nbADistribuer = 8) :
        # dans cette méthode on distribue carte par carte
        donne = [[] for i in range(nbJoueurs)]
        return donne


    def distribution_bis(self, nb_joueurs, nb_a_distribuer = 8):
        # dans cette méthode on fait des slices, comme pour la dist
ribution
        # à deux joueurs
        # Si nbADistribuer = 0 alors on distribue un maximum de car
tes à
        # chaque joueur
        donne = []
        return donne


    def __repr__(self):
        chaine = ""
        for carte in self.paquet:
            chaine = chaine + carte.__repr__() + "\n"
        return chaine

paquet_bataille = PaquetCartes('bataille', 32)
print(len(paquet_bataille.paquet))
paquet_bataille.melange()
print(paquet_bataille)
print()
"""
donne = paquet_bataille.distribution_2_joueurs()
for main in donne :
    print(main)
    print(len(main))
    print()
"""
donne = paquet_bataille.distribution(4, 5)
for main in donne :
    print(len(main))
    print(main)
    print()
```

La classe "jeu de la bataille"

A vous de jouer, puisque vous allez créer la classe JeuBataille, dont les spécifications sont données ci-dessous. L'ordinateur joue contre lui-même.
Rappel des règles:

- deux joueurs se partagent le paquet
- la donne de chaque joueur est posée face cachée devant lui
- chaque joueur tire en même temps une carte. Le "en même temps" est en fait un tirage successif, d'abord le joueur numéro 1 puis le 2. Ceci pour des raisons d'ordre dans lequel on va ranger les cartes par la suite
- En cas d'inégalité des cartes, le joueur ayant la carte la plus forte l'emporte. Il met les deux cartes sous son paquet, face retournée.
- En cas d'égalité, on itère le processus. Lorsqu'un joueur retourne une carte plus forte que celle de son adversaire, il remporte tout le tas, qu'il retourne et place en dessous de son paquet
- Un joueur a perud lorsqu'il n'a plus de cartes à retourner.
- Il est théoriquement possible d'avoir un match nul

Une notion importante fait son apparition dans ce jeu. Il s'agit de la **file** de cartes (et non pile comme on aurait tendance à le dire dans le langage courant). Une file est une structure linéaire de données dans laquelle on ajoute des éléments d'un côté, et on les supprime de l'autre : c'est la file d'attente à la boulangerie. Ou : **FIFO** (first in first out), également **queue** en anglais. Les opérations sur les files ont des noms spécifiques :

- `fileVide()` : crée une file vide
- `tete(file)` : renvoie l'élément en tête de la file, la file étant non vide
- `enfiler(element, file)` : insère `element` en fin de file
- `defiler(file)` : supprime l'élément en tête de la file. On peut le renvoyer éventuellement
- `estFileVide(file)` : teste si la file est vide
- On peut éventuellement fixer une taille maximale à une file

Cette structure mérite amplement sa classe. Créez-là en suivant les spécifications ci-dessous, vous l'utiliserez dans "JeuBataille". Quelques tests sont proposés, vous pouvez en rajouter (essayez d'utiliser des `assert`).

In [ ]:
```
class File:
    """
    Gère les files FIFO
    Attributs :
        - file : liste d'éléments à priori du même type

        Dans un deuxième temps uniquement, rajouter les attributs s
uivants
        - nb_elements : taille de la file
        - premier : premier élément de la file
        - dernier : dernier élément de la file
    Méthodes :
        Les méthodes seront codées en deux temps. Premier temps : u
niquement avec l'attribut "file". Deuxième temps,
        pour ceux qui vont un peu plus vite : rajouter la gestion d
es autres attributs.
        - __init__(liste = []) : constructeur, renvoie une file vid
e si liste n'est pas renseigné.
            Sinon renvoie une file constituée des éléments de la li
ste
        - tete : renvoie l'élément en tête de la file, la file étan
t non vide
        - enfiler(element) : insère element en fin de file
        - defiler : supprime l'élément en tête de la file. On peut
le renvoyer éventuellement.
            Si la file est vide, renvoie None
        - estFileVide : teste si la file est vide
    """
    def __init__(self,liste = None):
        """Constructeur de la file
        Remarque : écrire self._file = liste copie l'adresse de lis
te dans self._file. Ceci peut poser
        des problèmes. En effet, si par la suite on modifie liste,
alors on modifiera aussi self._file"""
        self._file = []
        if liste == None:

        else:


    def estFileVide(self):
        pass

    def enfiler(self,element):
        pass

    def defiler(self):
        pass

    def __repr__(self):
        if self.estFileVide() :
            return 'File vide'
        else:
            return str(self.file)
```

```
    # une petite méthode d'instrumentation ça peut parfois aider
:-)
    def print_file(self):
        print("Contenu de la file : ")
        for element in self._file:
            if isinstance(element,Carte):   # on teste si "element"
est une instance de la classe Carte...
                print(element)        # ...dans le cadre particulie
r de ce notebook uniquement
            else:
                print(element, "n'est pas une carte")

# Dans les tests ci-après, modifier le code pour n'afficher que ce
que vous avez codé
ma_file = File([11,22,33,44,55])
ma_file.print_file()
ma_file.enfiler(66)
print("premier élément:", ma_file.premier, "dernier élément : ", ma
_file.dernier,
      "nombre d'éléments : ", ma_file.nb_elements)
long_ma_file = ma_file.nb_elements
for i in range(long_ma_file):
    print("on défile l'élément : ",ma_file.defiler())
print("on défile l'élément : ",ma_file.defiler())
print(ma_file)
print()

ma_file2 = File(donne[0])
ma_file2.printFile()
```

In [ ]:
```
class JeuBataille:
    """
    Jeu de bataille
    Attributs:
        - nom_joueur_1 : chaine
        - nom_joueur_2 : chaine
        - paquetBataille : liste des cartes
        - cartes_j1 : pile des cartes du joueur 1
        - cartes_j2 : pile des cartes du joueur 2
        - defausse : pile des cartes de la défausse
        - nb_tours : entier, nombre de tours de jeu
        - nb_batailles : entier, nombre de cas d'égalité lors des t
irages simultanés

        Remarque : certains de ces attributs auraient peut-être plu
tôt leur place en tant que variable
        dans la méthode jouer, et vice-versa (?).
        Dans la version proposée ici, le jeu étant automatique, c'e
st même certain. Mais si on veut
        plus visualiser/intervenir lors du jeu, il vaut mieux avoir
les attributs ci-dessus.

    Méthodes :
        - __init__
        - jouer : jeu de l'ordinateur contre lui-même. Il est conse
illé:
            de mettre très peu de cartes (8 au total max)
            et de préciser un nombre maximal de tours de jeu
            Renvoie :
                match_nul : booléen au nom explicite
                gagnant : chaîne de caractères
                self.nb_batailles : le nombre de batailles qu'il y
a eu pendant le jeu
                self.nb_tours : le nombre de tours de jeu
    """

    def __init__(self, nom_joueur_1 = 'ordi1', nom_joueur_2 = 'ordi
2'):
        """Constructeur du jeu"""
        self.nom_joueur_1 = nom_joueur1
        self.nom_joueur_2 = nom_joueur2
        self.paquet_bataille = PaquetCartes('bataille')
        donne = self.paquet_bataille.melange().distribution(2)
        #  on ne donne que quelques cartes de la donne aux joueurs,
ci-dessous on en donne 6.
        #  Pour cela, soit on détaille sur quelques lignes lignes
        donne_temp = donne[0]
        donne_temp = donne_temp[:6]
        self.cartes_j1 = File(donne_temp)
        #  Soit on ne met qu'une seule instruction
        self.cartes_j2 = File(donne[1][:6])
        self.defausse = File()
        self.nb_tours = 0
        self.nb_batailles = 0
```

```
    def jouer(self):
        # jouez avec très peu de cartes (2 à 10). Fixez un maximum
de nombre de tours de jeu
        match_nul = True
        gagnat = None
        return (match_nul, gagnant, self.nb_batailles, self.nb_tour
s)


baston = JeuBataille()

baston = JeuBataille()
print("Cartes j1")
baston.cartes_j1.printFile()
print()
print("Cartes j2")
baston.cartes_j2.printFile()
print()
print("Cartes défausse")
baston.defausse.printFile()
print()

(mat, gagnant, nb_batailles,tours) = baston.jouer()
if mat:
    print("match nul. En récompense, calculer la probabilité de cet
évènement.")
    print("Cartes j1 en fin de jeu")
    baston.cartes_j1.print_file()
    print()
    print("Cartes j2 en fin de jeu")
    baston.cartes_j2.print_file()
    print()
    print("Cartes défausse en fin de jeu")
    baston.defausse.print_file()
    print()
else:
    if gagnant == None:
        print("trop de tours de jeu. Il y a eu ",nb_batailles," bat
ailles")
        print("Cartes j1 en fin de jeu")
        baston.cartes_j1.printFile()
        print()
        print("Cartes j2 en fin de jeu")
        baston.cartes_j2.printFile()
        print()
        print("Cartes défausse en fin de jeu")
        baston.defausse.printFile()
        print()
    else:
        print(gagnant," a gagné en ",tours," tours de jeu, et ",nb_
batailles," batailles.")
```

Complément facultatif : deuxième ébauche de la classe carte

Rendons les attributs privés. Les attributs privés ne sont plus accessible de l'extérieur de la classe. Il suffit de les écrire avec `_` devant. Une remarque importante : en Python, c'est juste une convention, qui signale que l'accès (respectivement la modification) à/de cet attribut doit se faire par des getters/accesseurs (respectivement setters/mutateurs). D'une part, on peut quand même accéder à l'attribut malgré la présence du `_` , d'autre part, d'autres méthodes existent en Python pour éviter les manipulations délictueuses (on ne les verra pas).

In [ ]:
```
class Carte:
    """
    Carte d'un jeu
    """
    def __init__(self,couleur,hauteur,valeur = 0):
        self._couleur = couleur
        self._hauteur = hauteur
        self._valeur = valeur
```

Puis récupérons la hauteur :

In [ ]:
```
roiCarreau = Carte('carreau','roi',13)    # création d'une carte
print(roiCarreau._hauteur)     # lecture de l'objet roiCarreau
```

Complément facultatif : troisième ébauche, getters et setters

L'avantage des attributs privés, c'est qu'ils ne sont modifiables que par l'utilisation de méthodes publiques. Ces méthodes, puisqu'elles sont publiques, sont accessibles par n'importe quel objet. Mais leur définition étant interne à la classe, elles sont cohérentes avec celle-ci. Lorsqu'un attribut est signalé comme privé avec le tiret bas `_` , on evite donc d'y accéder comme précédememnt (`roiCarreau._hauteur`). On utilise à la place des accesseurs (getters en anglais) et des mutateurs (setters en anglais), comme ci-dessous :

In [ ]:
```
class Carte:
    """
    Carte d'un jeu
    """
    def __init__(self,couleur,hauteur,valeur = 0):
        self._couleur = couleur
        self._hauteur = hauteur
        self._valeur = valeur

    # méthodes getters/setters
    def getCouleur(self):
        return self._couleur
    def getHauteur(self):
        return self._hauteur
    def getValeur(self):
        return self._valeur

    def setCouleur(self,nouvCouleur):
        self._couleur = nouvCouleur
    def setHauteur(self,nouvHauteur):
        self._hauteur = nouvHauteur
    def setValeur(self,nouvValeur):
        if type(nouvValeur) == int :
            self._valeur = nouvValeur
        else:
            raise TypeError('la valeur doit être un entier')


roiCarreau = Carte('carreau','roi',13)
print(roiCarreau.getCouleur())      # l'appel d'une méthode doit vo
us rappeler "liste.sort()"
roiCarreau.setCouleur('fenetre')
print(roiCarreau.getCouleur())
roiCarreau.setValeur(-1.5)         # faire différents tests
print(roiCarreau.getValeur())
```

Remarque : si vous faites du Python plus avancé dans le supérieur, vous verrez qu'il y a des méthodes plus pythonesques que les getters et setters ; ce sont les propriétés et les décorateurs. Nous n'en ferons pas usage en terminale (l'objectif n'étant pas l'apprentissage de Python mais de la programmation). Vous pouvez aussi consulter le cours pour d'autres méthodes

La classe Carte "propre"

Reprendre la classe précédente. Le nettoyer et le compléter en tenant compte des remarques suivantes/précédentes :

- on ne change pas la couleur ni la hauteur (sauf si on triche)
- le but est de jouer à la belote ; la valeur d'une carte est comprise entre 0 et 20 points.

In [ ]:
```
```

Complément de complément : et la belote dans tout ça ?

On va programmer les règles du jeu, afin de pouvoir jouer entre humains. Les règles simplifiées :

- jeu de 32 cartes, 2 équipes de 2 joueurs. Les joueurs Nord et Sud jouent ensemble, de même qu'Est et Ouest. On tourne dans le sens des aiguilles d'une montre : si Sud est le 1er joueur, alors l'ordre est Sud-Ouest-Nord-Est
- La donne est de 5 cartes par joueur. La première carte de la défausse est dévoilée. On utilisera la méthode `.donne()`, sans suivre les règles de distribution usuelles, pour ceux qui les connaissent.
- La carte retournée donne la couleur de l'atout.
- Ordre/points des cartes :
  - non atout : As (11 points) dix (10 points) Roi (4 points) Dame (3 points) Valet (2 points) Neuf (0 points) Huit (0 points) Sept (0 points)
  - atout : Valet (20 points) Neuf (14 points) As (11 points) dix (10 points) Roi (4 points) Dame (3 points) Huit (0 points) Sept (0 points)
- Le premier joueur à parler (choisi au hasard lors de la première partie) "prend" la carte retournée ou non. S'il ne la prend pas, le joueur suivant peut choisir de la prendre, etc. jusqu'au dernier. Si personne ne prend, la partie est annulée.
- Une fois l'atout pris, on finit de distribuer les cartes. Celui qui a pris reçoit deux cartes supplémentaires, les autres joueurs trois (chacun a donc huit cartes et il n'y a plus de défausse).
- Le premier joueur a avoir parlé commence. Lors des tours suivants, c'est celui qui a emporté le pli qui commence.
- l'ordinateur doit vérifier qu'il n'y a pas de triche :
  - Lorsqu'une couleur est jouée, les autres joueurs doivent suivre s'ils le peuvent. S'ils ne peuvent pas, ils doivent couper s'ils ont de l'atout. S'ils n'ont pas d'atout, ils "pissent" en se défaussant d'une carte au choix.
  - Si un premier joueur a déjà coupé, et qu'un deuxième doit et peut couper alors il doit monter en ordre de la carte si possible.
  - on peut rajouter la règle suivante: si dans une équipe un des partenaires est maitre à l'atout, son coéquipier n'est pas obligé d'en mettre.
  - Si la pli a été coupé, l'atout le plus fort l'emporte. Sinon c'est la carte la plus forte qui est maître.
- Le programme doit également compter les points. L'équipe ayant fait le dernier pli gagne 10 points ("dix de der"). La manche est gagnante si 82 points ou plus ont été faits.

Dans un deuxième temps, on peut implémenter une ou plusieurs des règles suivantes:

- Si la partie est annulée, on mélange et distribue à nouveau les cartes. C'est le joueur à gauche du joueur précédent qui commence.
- Une partie se comporte de plusieurs manches, jusqu'à ce qu'une des équipes marque 501 points ou plus
- Il y a deux tours de table pour prendre ou non. Lors du deuxième tour, les joueurs peuvent choisir n'importe quelle couleur d'atout (on peut même rajouter "sans atout" et "tout atout").

In [ ]:
```
class JeuBelote():
```

Pour aller encore plus loin : polymorphisme et héritage

Cette partie n'est pas au programme de Terminale NSI, c'est un approfondissement à faire éventuellement chez soi
Les jeux de cartes sont très nombreux. Ils n'utilisent pas tous les mêmes cartes, et la manière de distribuer les cartes est différente. Sans aller jusqu'aux cartes de type Pokémon ou Magic, intéressons-nous aux cartes du tarot. Pour ceux qui ne connaissent pas le tarot, il y a une figure de plus, le Cavalier, entre la Dame et le Valet. Et aussi 22 atouts : 21 numérotés de 1 à 21, et un atout spécial appelé l'Excuse (que beaucoup connaissent très bien pour tous les devoirs rendus en retard).
Créer une deuxième classe PaquetTarot est lourd, il est plus intéressant d'avoir une super-classe Paquet et deux sous-classes PaquetClassique et PaquetTarot (voire plus : il y a aussi des jeux de 54 cartes avec joker).

Avant de donner un exemple d'héritage, remarquons que la valeur des cartes peut avoir deux sens:

- `valeur` peut représenter l'ordre des cartes
- `valeur` peut représenter le nombre des points lors du décompte final
  Modifier la classe Carte en tenant compte de ces deux possibilités. On rajoutera l'attribut `points` .

In [ ]:
```
class Carte:
```

Classes abstraites

Puisque l'on va utiliser des types de paquets très différents, notre classe intiale PaquetCartes n'a plus d'intérêt que comme une abstraction de ces paquets particuliers (belote, bridge, tarot, pokemon, etc...). Le paquet de cartes ne sera jamais instancié par la super-classe PaquetCartes, mais par une des sous-classes PaquetCartesClassique, PaquetCartesTarot, etc. Dans ce cas particulier, la classe PaquetCartes est une classe abstraite.
Remarque : une super-classe n'est pas forcément abstraite. Si on a une super-classe Voiture et une sous-classe VoitureCourse, où l'on aura en plus quelques attributs et/ou méthodes, on peut instancier les deux.

In [ ]:
```
import random as rd  #version prof
import abc

class PaquetCartes(metaclass = abc.ABCMeta):
    """
    Paquet de cartes
    Attributs:
        - nom : nom du paquet, de préférence correspondant au nom d
u jeu pour
            lequel il va être utilisé
        - paquet : liste des cartes

    Méthodes :
        - __init__
        - getPaquet
        - melange() mélange le paquet de cartes
        - distribution(nbJoueurs,nbADistribuer = 0) renvoie donne,
une liste de listes de cartes.

    """

    def __init__(self,nom,nbCartes):
        """Constructeur du paquet de cartes"""
        self.nom = nom
        self.nbCartes = nbCartes
        self.paquet = []

    def melange(self):   #version prof
        rd.shuffle(self.paquet)
        return(self)

    @abc.abstractmethod       # comme précisé, la méthode est abstr
aite, ce qui rend la classe abstraite
    def distribution(self):
        """distribue les cartes aux joueurs"""
```

Essayons d'instancier la classe:

In [ ]:
```
mon_paquet = PaquetCartes("jeu",10)
```

Les sous-classes

En exemple ci-dessous la sous-classe PaquetBelote (constructeur à compléter)

- La déclaration `class PaquetBelote(PaquetCartes)` précise que PaquetBelote est une sous-classe de PaquetCartes.
- on **redéfinit** la méthode `distribution` .
- Examinez le code pour la méthode `distribution(self)` : comment est faite la distribution ?

Remarque :Le vocabulaire officiel est super-classe et sous-classe, mais on utilise souvent classe mère et classe fille. Ce qui permet de réaliser des héritages plus complexes, avec des classes grand-mère, arrière-grand-mère. En Python, on peut aussi avoir une classe fille qui hérite d'une classe mère et d'une classe père. C'est également possible en C++ mais pas en Java, PHP ni C# par exemple, sauf dans le cas particulier des interfaces que l'on verra ultérieurement.

In [ ]:
```
class PaquetBelote(PaquetCartes):
    """
    Paquet de cartes pour la belote
    """
    _hauteurs = ["sept","huit","neuf","valet","dame","roi","dix","as"]
    _couleurs = ["coeur","pique","carreau","trèfle"]
    _points = [0,0,0,2,3,4,10,11]    # on met les points par défaut pour non atout

    def __init__(self):
        """Constructeur du paquet de cartes"""
        PaquetCartes.__init__(self,"belote",32)
        for i in range(len(self._hauteurs)):
            for j in range(len(self._couleurs)):
                #self._paquet.append(Carte(self._couleurs[j],self._hauteurs[i],???,self._points[i])) # version élève
                self.paquet.append(Carte(self._couleurs[j],self._hauteurs[i],i,self._points[i]))

    def distribution(self):    # on redéfinit la méthode distribution
        aDistribuer = File(self.paquet)
        donne = [[],[],[],[],[]]
        for i in range(4):
            donne[i].append(aDistribuer.defiler())
            donne[i].append(aDistribuer.defiler())
        for i in range(4):
            donne[i].append(aDistribuer.defiler())
            donne[i].append(aDistribuer.defiler())
            donne[i].append(aDistribuer.defiler())
        for i in range(aDistribuer.getNb_elements()):
            donne[4].append(aDistribuer.defiler())
        return donne

paquet_belote = PaquetBelote()
paquet_belote.melange()
donne = paquet_belote.distribution()

for main in donne:
    print(len(main))

for i in range(5):
    print()
    for carte in donne[i]:
        print(carte.getTout())
```

Créez la sous-classe PaquetTarot. Au tarot:

- il y a trois, quatre ou cinq joueurs.
- Les cartes sont distribuées trois par trois. A trois joueurs, il reste 6 cartes, sinon il en reste 3.
- Ces cartes restantes sont données à la fin d'un tour de distribution (sauf le dernier). Pour simplifier, on choisira une des méthodes suivantes :
  - soit par trois
  - soit une par une
- Le roi, les atouts 1, 21 et l'excuse rapportent 4,5 points. La dame 3,5, le cavalier 2,5, le valet 1,5 et les autres cartent rapportent 0,5.
- L'ordre des cartes est, de la moins forte à la plus forte:
  - les couleurs de l'as au roi
  - les atouts du 1 au 21
  - l'excuse est imprenable (c'est plus compliqué que ça en fait), et ne peut pas prendre de pli non plus.

In [ ]:

<hr style="color:black; height:1px />

frederic.mandon @ ac-montpellier.fr, Lycée Jean Jaurès - Saint Clément de Rivière - France (2015-2019)