Chapitre 10

Blackjack (vingt et un)

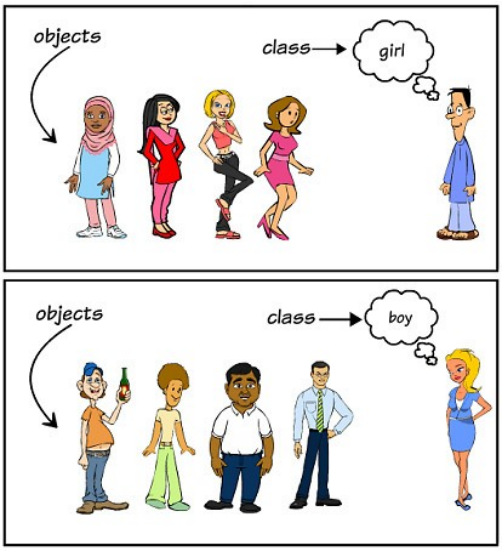
10.1. Nouveaux thèmes abordés dans ce chapitre

- Programmation orientée objet
- Sons

10.2. Programmation orientée objet (POO)

Un *paradigme* est une manière de programmer.

La **programmation orientée objet (POO)** est un paradigme de programmation informatique élaboré au début des années 1960. Il consiste en la définition et l'interaction de « briques logicielles » appelées *objets*. Un objet représente un concept, une entité du monde physique, comme une voiture, une personne, etc.



Python est un langage orienté objet, mais on peut très bien programmer en Python sans utiliser ce paradigme. D'ailleurs, c'est ce que l'on a fait jusqu'à présent ! L'intérêt d'une approche par objets est que chaque objet peut être développé séparément, ce qui est très pratique pour travailler sur un projet en équipe.

Nous présentons dans ce paragraphe les notions essentielles de la POO (classe, objet, attribut, méthode) et comment les programmer en Python Des notions plus évoluées telle que l'héritage et le polymorphisme seront présentées plus tard dans ce chapitre.

Classe

Une classe est un ensemble d'items qui ont une structure commune et un comportement commun, par exemple une date, un étudiant, un arbre, une carte à jouer, etc.

Par convention, les noms de classe commencent par une majuscule.

```
class Carte:
    "Définition d'une carte"
```

Objet / instance

Un objet est une instance d'une classe.

Un objet est unique et identifiable avec un rôle bien défini. Il est persistant[1] et change d'état.

```
ma_carte = Carte()
```

Attribut

Un attribut est une propriété observable des objets d'une classe. Dans notre exemple, une carte est définie par une valeur et une couleur :

```
ma_carte.valeur = « roi »
ma_carte.couleur = « pique »
```

Méthode

Une méthode est une opération qui peut changer la valeur des certains attributs d'un objet. Dans l'exemple ci-dessous, il y a deux méthodes : la méthode spéciale `__init__()`, appelée *constructeur*, permet d'initialiser la valeur d'une carte (par défaut l'as de carreau) ; la méthode `afficher()` permet d'écrire une carte, par exemple « as de carreau ».

```
class Carte(object):
    "Définition d'une carte"

    def __init__(self,val='as',coul='carreau'):
        self.valeur = val
        self.couleur = coul

    def afficher(self):
        print(self.valeur+" de "+self.couleur)

ma_carte=Carte('roi','pique')
ma_carte.afficher()
```

Le constructeur est une méthode de notre objet se chargeant de créer nos attributs. En vérité, c'est même la méthode qui sera appelée quand on voudra créer notre objet.

Quand une méthode est appelée, une référence vers l'objet principal est passée en tant que premier argument. Par convention le premier argument appelé par les méthodes est toujours `self`. Voilà pourquoi la fonction `__init__(self,val='as',coul='carreau')` a trois arguments, mais elle est appelée avec seulement deux arguments : `Carte('roi','pique')`.

10.3. Exemple de classe : un paquet de cartes

Voici un exemple plus complet de classes : un paquet de 52 cartes. Le but du programme est simple : on va mélanger 52 cartes et les faire défiler à l'écran. On aura besoin de 52 images des cartes, que l'on va mettre dans le répertoire « cartes » et, pour améliorer le résultat à l'écran, deux

1 **Persistant :** Un objet est créé à un certain moment, subit des changements d'états, et il est seulement détruit suite à une requête directe d'un utilisateur ou via un objet autorisé.

sons au format wave, que l'on mettra dans le répertoire « sons ».

paquet.py

```
# Affichage de cartes mélangées

from random import randrange
from tkinter import *
from winsound import PlaySound

couleur = ('pique','trefle','carreau','coeur')
valeur = ('as','2','3','4','5','6','7','8','9','10','valet','dame','roi')

class Carte(object):

    def __init__(self,val='as',coul='carreau'):
        self.valeur = val
        self.couleur = coul

    def dessin_carte(self):
        "Renvoi du nom du fichier image de la carte"
        # les cartes sont dans le répertoire "cartes"
        nom = "cartes/"+self.valeur+"_"+self.couleur+".gif"
        return PhotoImage(file=nom)


class Paquet_de_cartes(object):

    def __init__(self):
        "Construction de la pile des 52 cartes"
        self.cartes =[]
        for coul in range(4):
            for val in range(13):
                nouvelle_carte = Carte(valeur[val], couleur[coul])
                self.cartes.append(nouvelle_carte)

    def battre(self):
        "Mélanger les cartes"
        PlaySound("sons/distrib.wav", 2)
        t = len(self.cartes)
        for i in range(t):
            h1, h2 = randrange(t), randrange(t)
            self.cartes[h1], self.cartes[h2] = self.cartes[h2], self.cartes[h1]

    def tirer(self):
        "Tirer la première carte de la pile"
        PlaySound("sons/ramass.wav", 1)
        t = len(self.cartes)
        if t>0:
            carte = self.cartes[0]   # choisir la première carte du jeu
            del(self.cartes[0])      # et la supprimer du jeu
            return carte
        else:
            return None


###################### Programme de test ##########################

def jouer():
    global carte1, lab1
    c = jeu.tirer()
    if c != None:
        carte1 = c.dessin_carte()
        lab1.configure(image=carte1)

def reinit():
    global jeu
    jeu = Paquet_de_cartes()
    jeu.battre()
    jouer()

# fenetre graphique
fenetre = Tk()
fenetre.title("Cartes")
jeu = Paquet_de_cartes()
```

```
jeu.battre()
c = jeu.tirer()
carte1 = c.dessin_carte()
lab1 = Label(fenetre, image=carte1)
lab1.grid(row=0, column=0)
bouton1 = Button(fenetre,text='Tirer',command=jouer)
bouton1.grid(row=1, column=0, sticky=W)
bouton2 = Button(fenetre,text='Recommencer',command=reinit)
bouton2.grid(row=1, column=0, sticky=E)

# demarrage :
fenetre.mainloop()
```

10.4. Blackjack (règles simplifiées)

Les règles complètes sont sur Wikipédia :

http://ow.ly/jtKfh

La partie oppose tous les joueurs contre le croupier (pour simplifier, il n'y aura ici qu'un seul joueur). Le but est de faire plus de points que le croupier sans dépasser 21. Dès qu'un joueur fait plus que 21, on dit qu'il « saute » et il perd sa mise initiale. La valeur des cartes est établie comme suit :

- de 2 à 10 → valeur nominale de la carte
- une figure → 10 points
- un as → 1 ou 11 (au choix)

Un blackjack est composé d'un as et d'une « bûche » (carte ayant pour valeur 10, donc 10, valet, dame ou roi). Cependant, si le joueur atteint le point 21 en 3 cartes ou plus on compte le point 21 et non pas blackjack.

Nous allons ici suivre les règles européennes.

Au début de la partie le croupier distribue une carte face visible à chaque joueur et tire une carte face visible également pour lui. Il tire ensuite pour chacun des joueurs une seconde carte face visible et tire une seconde carte face cachée pour lui au blackjack américain. Au blackjack européen, le croupier tire sa seconde carte après le tour de jeu des joueurs.

Puis il demande au premier joueur de la table (joueur situé à sa gauche) l'option qu'il désire choisir. Si le joueur veut une carte, il doit l'annoncer en disant « Carte ! ». Le joueur peut demander autant de cartes qu'il le souhaite pour approcher 21 sans dépasser. Si après le tirage d'une carte, il a dépassé 21, il perd sa mise et le croupier passe au joueur suivant. S'il décide de s'arrêter, en disant « Je reste », le croupier passe également au joueur suivant.

Le croupier répète cette opération jusqu'à ce que tous les joueurs soient servis.

Ensuite, il joue pour lui selon une règle simple et codifiée « la banque tire à 16, reste à 17 ». Ainsi, le croupier tire des cartes jusqu'à atteindre un nombre compris entre 17 et 21 que l'on appelle un point. S'il fait plus de 21, tous les joueurs restants gagnent mais s'il fait son point, seuls gagnent ceux ayant un point supérieur au sien (sans avoir sauté). Dans cette situation, le joueur remporte l'équivalent de sa mise. En cas d'égalité le joueur garde sa mise mais n'empoche rien en plus. À noter que le blackjack (une bûche et un as en deux cartes) est plus fort que 21 fait en ayant tiré plus de deux cartes.

10.5. Code du programme

blackjack.py

```
# Blackjack
# règles : http://fr.wikipedia.org/wiki/Blackjack_(jeu)

from random import randrange
from tkinter import *
from winsound import PlaySound

couleur = ('pique','trefle','carreau','coeur')
valeur = ('as','2','3','4','5','6','7','8','9','10','valet','dame','roi')

def calculer_main(cartes):
    # Les cartes sont des numéros allant de 1 (as) à 13 (roi)
    val = 0
    for carte in cartes:
        if 1 < carte <= 10:
            val += carte
        elif carte > 10:     # buche
            val += 10
    # valeur de l'as ou des as
```

Ce programme n'est pas tout à fait terminé...

En effet, vous constaterez que ce programme « bugue » quand on clique à des moments inopportuns.

Que pourrait-on faire pour éviter ces problèmes ?

```
    nb_as = cartes.count(1) # Attention s'il y a plusieurs as dans la main
    while nb_as>1:
        val += 1    # un seul as peut valoir 11 pts. Les autres valent 1.
        nb_as -= 1
    if nb_as == 1 and val + 11 <= 21:
        return val + 11
    elif 1 in cartes:
        return val + 1
    else:
        return val


class Carte(object):

    def __init__(self,val='as',coul='carreau'):
        self.valeur = val
        self.couleur = coul

    def dessin_carte(self):
        "Renvoi du nom du fichier image de la carte"
        # les cartes sont dans le répertoire "cartes"
        nom = "cartes/"+self.valeur+"_"+self.couleur+".gif"
        return PhotoImage(file=nom)


class Joueur(object):
    """Main du joueur"""

    def __init__(self):
        "Construction de la liste des 52 cartes"
        self.main = []

    def ajouter(self, c):
        "Ajoute la carte c"
        # 1:as, 2...10, 11:valet, 12:dame, 13:roi
        self.main.append(valeur.index(c.valeur)+1)

    def total(self):
        "Calcule le total des points de la main"
        return calculer_main(self.main)

    def nb_cartes(self):
        return len(self.main)

    def reinit(self):
        self.main = []


class Paquet_de_cartes(object):
    """Paquet de cartes"""

    def __init__(self):
        "Construction de la liste des 52 cartes"
        self.cartes = []
        for coul in range(4):
            for val in range(13):
                nouvelle_carte = Carte(valeur[val], couleur[coul])
                self.cartes.append(nouvelle_carte)

    def battre(self):
        "Mélanger les cartes"
        PlaySound("sons/distrib.wav",2)
        t = len(self.cartes)
        for i in range(t):
            h1, h2 = randrange(t), randrange(t)
            self.cartes[h1], self.cartes[h2] = self.cartes[h2], self.cartes[h1]

    def tirer(self):
        "Tirer la première carte de la pile"
        PlaySound("sons/ramass.wav",1)
        t = len(self.cartes)
        if t>0:
            carte = self.cartes[0]   # choisir la première carte du jeu
```

```
            del(self.cartes[0])      # et la supprimer du jeu
            return carte
        else:
            return None


def reinit():
    global mes_cartes, ses_cartes, mon_score, son_score, moi, coupier
    global jeu, can

    can.delete(ALL)
    can.create_text(60,30,text="Croupier",fill='black',font='Arial 18')
    can.create_text(60,470,text="Joueur",fill='black',font='Arial 18')
    can.update()
    croupier.reinit()
    ses_cartes = []
    jeu = Paquet_de_cartes()
    jeu.battre()     # mélange des cartes
    sleep(1)         # pause d'une seconde
    c = jeu.tirer()
    ses_cartes.append(c.dessin_carte())
    can.create_image(100, 150, image=ses_cartes[0])
    croupier.ajouter(c)
    son_score = can.create_text(150,30,text=str(croupier.total()),fill='black',font='Arial 18')
    can.update()
    sleep(1)

    moi.reinit()     # joueur humain
    mes_cartes = []
    c = jeu.tirer()
    mes_cartes.append(c.dessin_carte())
    can.create_image(100, 350, image=mes_cartes[0])
    can.update()
    sleep(1)
    moi.ajouter(c)
    c = jeu.tirer()
    mes_cartes.append(c.dessin_carte())
    can.create_image(150, 350, image=mes_cartes[1])
    can.update()
    moi.ajouter(c)
    mon_score = can.create_text(150,470,text=str(moi.total()),fill='black',font='Arial 18')
    can.update()
    if moi.total()==21 :
        can.create_text(250,470,text="Blackjack",fill='red',font='Arial 18')
        can.update()
        sleep(3)
        reinit()


def hit(): # le joueur tire une carte
    global mes_cartes, moi, mon_score, son_score
    c = jeu.tirer()
    mes_cartes.append(c.dessin_carte())
    moi.ajouter(c)
    n = moi.nb_cartes()
    can.create_image(150+50*(n-2), 350, image=mes_cartes[n-1])
    can.delete(mon_score)
    mon_score = can.create_text(150,470,text=str(moi.total()),fill='black',font='Arial 18')
    can.update()
    if moi.total()>21:
        can.create_text(250,30,text="Gagné",fill='red',font='Arial 18')
        can.update()
        sleep(3)
        reinit()


def stay(): # le joueur s'arrête et le croupier joue
    global ses_cartes, croupier, moi, son_score, mon_score
    c = jeu.tirer()
    ses_cartes.append(c.dessin_carte())
    croupier.ajouter(c)
    n = croupier.nb_cartes()
    can.create_image(100+50*(n-1), 150, image=ses_cartes[n-1])
    can.delete(son_score)
```

```
    son_score = can.create_text(150,30,text=str(croupier.total()),fill='black',font='Arial 18')
    can.update()
    if croupier.total()>21:
        can.create_text(250,470,text="Gagné",fill='red',font='Arial 18')
        can.update()
        sleep(3)
        reinit()
    elif croupier.total()==21 and n==2:
        can.create_text(250,30,text="Blackjack",fill='red',font='Arial 18')
        can.update()
        sleep(3)
        reinit()
    elif croupier.total()<17:
        sleep(2)
        stay()
    else:
        if croupier.total()>moi.total():
            can.create_text(250,30,text="Gagné",fill='red',font='Arial 18')
        elif croupier.total()<moi.total():
            can.create_text(250,470,text="Gagné",fill='red',font='Arial 18')
        can.update()
        sleep(3)
        reinit()


# fenetre graphique
moi = Joueur()
mes_cartes = []
croupier = Joueur()
ses_cartes = []
jeu = Paquet_de_cartes()
fenetre = Tk()
fenetre.title("Blackjack")
can = Canvas(fenetre, width=600, height=500, bg ='white')
can.pack(side=TOP, padx=5, pady=5)
b2 = Button(fenetre, text='Quitter', width=15, command=fenetre.quit)
b2.pack(side=RIGHT)
b1 = Button(fenetre, text='Carte !', width=15, command=hit)
b1.pack(side=LEFT)
b1 = Button(fenetre, text='Je reste', width=15, command=stay)
b1.pack(side=LEFT)
reinit()

# demarrage :
fenetre.mainloop()
fenetre.destroy()
```

Il y a dans ce programme trois classes. Par rapport à notre premier programme, on a rajouté la classe `joueur`, qui est utilisée pour gérer les cartes qu'a en main le joueur.

```
class Joueur(object):
    """Main du joueur"""

    def __init__(self):
        "Construction de la liste des 52 cartes"
        self.main = []

    def ajouter(self, c):
        "Ajoute la carte c"
        # 1:as, 2...10, 11:valet, 12:dame, 13:roi
        self.main.append(valeur.index(c.valeur)+1)

    def total(self):
        "Calcule le total des points de la main"
        return calculer_main(self.main)

    def nb_cartes(self):
        return len(self.main)

    def reinit(self):
        self.main = []
```

Exercice 10.1

Écrivez une classe `Cercle()`. Les attributs seront son centre et son rayon. Les méthodes seront `perimetre` qui renverra le périmètre du cercle, `aire` qui donnera son aire et `contient` qui indiquera si oui un non un point est dans le cercle.

Par défaut, ce sera un cercle centré à l'origine de rayon 1.

Exemple d'utilisation de cette classe :

```
c = Cercle((0,0),5)
p = (2,5)
print(c.perimetre())
print(c.aire())
print(c.contient(p))

31.41592653589793
78.53981633974483
False
```

Exercice 10.2

Définissez une classe `Piece()` qui permette d'instancier des pièces de monnaie truquées. On donnera en paramètre la probabilité d'obtenir pile (un nombre réel entre 0 et 1). La valeur par défaut est 0.5.

Une méthode `lancer()` simulera le jet de la pièce et renverra le résultat (pile ou face).

Exemple d'utilisation de cette classe :

```
p1 = Piece(0.1)
print(p1.lancer())
p2 = Piece()
print(p2.lancer())

face
face
```

Exercice 10.3

Définissez une classe `De()` qui permette d'instancier des objets simulant des dés. Un dé pourra pourra avoir un nombre de faces quelconque, avec un nombre quelconque écrit dessus. Une méthode `lancer()` simulera le jet du dé et renverra le résultat d'une face au hasard.

Par défaut, on définira un dé standard : 6 faces numérotées de 1 à 6.

Exercice 10.4

Modifiez le programme blackjack pour implémenter les règles complètes, que vous trouverez sur Wikipédia : http://fr.wikipedia.org/wiki/Blackjack_(jeu)

Contentez-vous dans un premier temps de programmer un seul joueur contre le croupier.

10.6. POO: notions avancées

10.6.1. Encapsulation

L'encapsulation est un mécanisme consistant à rassembler les données et les méthodes au sein d'une structure en cachant l'implémentation de l'objet, c'est-à-dire en empêchant l'accès aux données

par un autre moyen que les services proposés. L'encapsulation permet donc de garantir l'intégrité des données contenues dans l'objet.

Exemple

Dans le programme de blackjack que nous avons implémenté ci-dessus, on ne manipule jamais les cartes directement : on passe par les méthodes `battre` et `tirer` pour respectivement mélanger le jeu et tirer la première carte de la pile. L'usage de ces méthode garantit le bon fonctionnement de ces deux actions.

10.6.2. Héritage

L'héritage est un principe propre à la programmation orientée objet, permettant de créer une nouvelle classe à partir d'une classe existante. Le nom d' « héritage » vient du fait que la classe dérivée contient les attributs et les méthodes de sa superclasse. L'intérêt majeur de l'héritage est de pouvoir définir de nouveaux attributs et de nouvelles méthodes pour la classe dérivée, qui viennent s'ajouter à ceux et celles héritées.

Par ce moyen, on crée une hiérarchie de classes de plus en plus spécialisées. Cela a comme avantage majeur de ne pas avoir à repartir de zéro lorsque l'on veut spécialiser une classe existante. De cette manière il est possible d'acheter dans le commerce des librairies de classes, qui constituent une base, pouvant être spécialisées à loisir.

```
class Personnage(object):
    def __init__(self,nom,force,vie):
        self._nom = nom
        self._force = force
        self._vie = vie

    def getNom(self) : return self._nom
    def getForce(self) : return self._force
    def getVie(self) : return self._vie

class Guerrier(Personnage):
    def __init__(self,nom,force,vie):
        Personnage.__init__(self,nom,force,vie) # héritage

class Mage(Personnage):
    def __init__(self,nom,force,vie,mana):
        Personnage.__init__(self,nom,force,vie) # héritage
        self._mana = mana

    def getMana(self) : return self._mana
```

Héritage multiple

Certains langages orientés objet, tels que le C++ ou Python, permettent de faire de l'héritage multiple, c'est-à-dire regrouper au sein d'une seule et même classe les attributs et méthodes de plusieurs classes.

Par exemple, on peut imaginer une classe « centaure », qui hériterait à la fois des caractéristiques des classes « homme » et « cheval ».

```
class Personnage(object):
    def __init__(self,nom,force,vie):
        self._nom = nom
        self._force = force
        self._vie = vie

    def getNom(self) : return self._nom
    def getForce(self) : return self._force
    def getVie(self) : return self._vie


class Homme(Personnage):
    def __init__(self,nom,force,vie,intelligence):
        Personnage.__init__(self,nom,force,vie) # héritage
```

```
        self._intelligence = intelligence

    def getIntelligence(self) : return self._intelligence


class Cheval(Personnage):
    def __init__(self,nom,force,vie,vitesse):
        Personnage.__init__(self,nom,force,vie) # héritage
        self._vitesse = vitesse

    def getVitesse(self) : return self._vitesse


class Centaure(homme,cheval):   # héritage multiple
    def __init__(self,nom,force,vie,intelligence,vitesse):
        Homme.__init__(self,nom,force,vie,intelligence)
        self._vitesse = vitesse


chiron = Centaure("Chiron",10,6,5,15)
print(chiron.getVitesse())
```

Dans le design objet de tous les jours, l'héritage multiple est peu utilisé, car il introduit des dépendances généralement plus rigides que souhaitées.

10.6.3. Polymorphisme

Le mot *polymorphisme* vient du grec et signifie « qui peut prendre plusieurs formes ». Cette caractéristique est un des concepts essentiels de la programmation orientée objet. Alors que l'héritage concerne les classes (et leur hiérarchie), le polymorphisme est relatif aux méthodes des objets.

On distingue généralement trois types de polymorphisme :

Ces trois types de polymorphisme existent en Python.

- le polymorphisme *ad hoc* (également appelé surcharge),
- le polymorphisme paramétrique (également appelé généricité),
- le polymorphisme d'héritage (également appelé redéfinition).

Le polymorphisme *ad hoc*

Le polymorphisme *ad hoc* permet de définir des **opérateurs** dont l'utilisation sera différente selon le type des paramètres qui leur sont passés. Il est donc possible par exemple de surcharger l'opérateur « + » et de lui faire réaliser des actions différentes selon qu'il s'agit d'une opération entre deux entiers (addition : 3+4 = 7) ou entre deux chaînes de caractères (concaténation : 'parle'+'ment' = 'parlement').

Le polymorphisme paramétrique

Le polymorphisme paramétrique représente la possibilité de définir plusieurs **fonctions** de même nom mais possédant des paramètres différents (en nombre et/ou en type). Le polymorphisme paramétrique rend ainsi possible le choix automatique de la bonne méthode à adopter en fonction du type de donnée passée en paramètre.

Ainsi, on peut par exemple définir plusieurs méthodes homonymes `add(a,b)` effectuant une « somme » de valeurs `a` et `b`.

Par exemple, en Python 3 :

```
def add(a,b):
    return a + b

print(add(12, 5))                    # somme de deux entiers
print(add('parle', 'ment'))          # concaténation de deux strings
print(add(3.8, 9))                   # somme d'un réel et d'un entier
print(add([4, 9], [3.2, 6, 'a']))    # concaténation de deux listes
```

produira comme résultat :

```
17
parlement
12.8
[4, 9, 3.2, 6, 'a']
```

Le polymorphisme d'héritage

La possibilité de redéfinir une méthode dans des classes héritant d'une classe de base s'appelle la spécialisation. Il est alors possible d'appeler la méthode d'un objet sans se soucier de son type intrinsèque : il s'agit du polymorphisme d'héritage**.** Ceci permet de faire abstraction des détails des classes spécialisées d'une famille d'objets, en les masquant par une interface commune (qui est la classe de base).

Reprenons notre exemple des personnages virtuels et ajoutons une méthode « `attaquer` ».

```
class Personnage(object):
    def __init__(self,nom,force,vie):
        self._nom = nom
        self._force = force
        self._vie = vie

    def getNom(self) : return self._nom
    def getForce(self) : return self._force
    def getVie(self) : return self._vie


class Guerrier(Personnage):
    def __init__(self,nom,force,vie):
        Personnage.__init__(self,nom,force,vie) # héritage
    def attaquer(self,qui):
        print(self._nom,"pourfend",qui.getNom(),"de son épée")


class Mage(Personnage):
    def __init__(self,nom,force,vie,mana):
        Personnage.__init__(self,nom,force,vie) # héritage
        self._mana = mana

    def getMana(self) : return self._mana
    def attaquer(self,qui):
        print(self._nom,"ensorcelle",qui.getNom())


merlin = Mage("Merlin",3,10,1)
duroc = Guerrier("Duroc",9,10)
merlin.attaquer(duroc)
duroc.attaquer(merlin)
```

Le résultat affiché sera :

```
Merlin ensorcelle Duroc
Duroc pourfend Merlin de son épée
```

Exercice 10.5

Créez une classe « Nain » où un personnage de ce type porte une attaque avec sa hache.

Créez un personnage de cette classe et nommez-le « Gimli ».

Exercice 10.6

Imaginez d'autres races et sous-races (par exemple les elfes, les trolls, les trolls-cyclopes, etc.) et implémentez un système pour simuler des combats entre ces personnages.

Les attributs vie, force, mana (et éventuellement d'autres que vous pourrez

ajouter) seront modifiés selon les blessures infligées, dont la gravité sera déterminée aléatoirement. À vous de définir les règles ! Un personnage aura perdu le combat quand ses points de vie seront à zéro.

Organisez ensuite un tournoi où chaque personnage se battra en duel avec chacun des autres. Il y aura 10 personnages différents, pas forcément tous de race différente. Le tournoi sera donc composé de 45 combats.

Quelques idées de personnages fantastiques...

Sorcier

Harpie

Sirène

Troll-cyclope

10.7. Ce que vous avez appris dans ce chapitre

- Vous avez vu les notions fondamentales de la programmation orientée objet (POO) : classe, objet, attribut, méthode, encapsulation, héritage, polymorphisme.
- Python est fait pour ce paradigme, qui est particulièrement intéressant quand on développe un projet en groupe.
- Vous avez pour la première fois incorporé des sons à vos programmes.