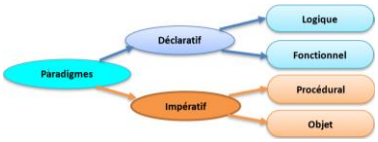
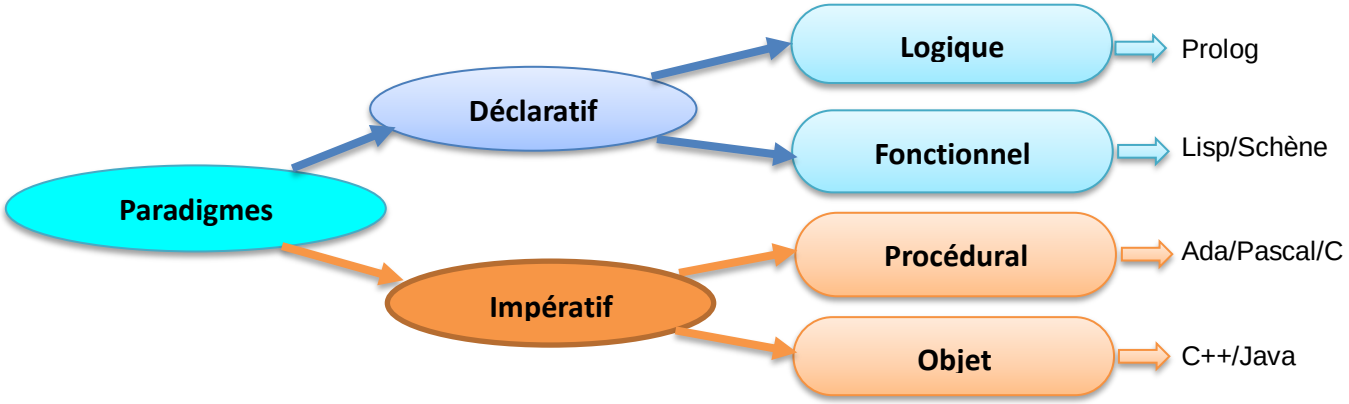
NSI Terminale S3 : Paradigmes de programmation



Un paradigme de programmation définit un style de programmation, une manière de coder.
Il existe beaucoup de paradigmes mais ils sont classés dans deux grandes familles :

- Les paradigmes déclaratifs
- Les paradigmes impératifs.

Ces paradigmes ont vu le jour suivant les langages :



Dans cette séquence, nous allons voir que nous pouvons programmer ces 4 paradigmes avec Python.

1. Programmation logique :

a) Définition :

La programmation **logique** est une forme de programmation qui définit les applications à l'aide d'un ensemble de faits élémentaires les concernant et de règles de logique leur associant des conséquences plus ou moins directes.
Ces faits et ces règles sont exploités par un démonstrateur de théorème ou moteur d'inférences, en réaction à une question ou requête.
Elle est particulièrement adaptée aux besoins de l'intelligence artificielle, dont elle est un des principaux outils.
Cette programmation est née à Marseille en 1972 avec pour but de laisser le compilateur faire ke gros du travail (PROgrammation LOGique : PROLOG)

b) Exemple :

Pour coder avec ce type de programmation, il faut importer la bibliothèque pyDatalog puis créer les faits et les règles.
Nous allons tester avec les faits suivants :

- Un ours est grand
- Un éléphant est grand
- Un chat est petit
- Un chien est petit
- Un ours est brun
- Un éléphant est gris
- Un chat est noir
- Un chien est blanc

Les règles :

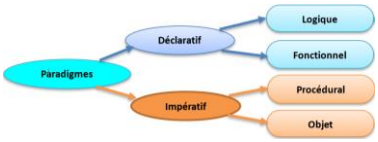
- Un animal est sombre s'il est noir
- Un animal est sombre s'il est brun
- Un animal est clair s'il est gris
- Un animal est clair s'il est blanc

Un exemple de code de ceci est le suivant :
Il faut d'abord lui donner tous les termes qui vont être utilisé : ours, elephant, chat, chien, petit, grand, brun, gris, noir, blanc, sombre, clair, couleur, taille, X

```
from pyDatalog import pyDatalog
pyDatalog.create_terms('ours,elephant,chat,chien,petit,grand,brun,gris,noir,blanc,sombre,clair,couleur,taille,X')
```

Puis tous les faits :

```
+taille("ours","grand")
+taille("elephant","grand")
+taille("chat","petit")
+taille("chien","petit")
+couleur("ours","brun")
+couleur("elephant","gris")
+couleur("chat","noir")
+couleur("chien","blanc")
```



Puis toutes les règles :

```
sombre(X) <= (couleur(X,"noir"))
sombre(X) <= (couleur(X,"brun"))
clair(X) <= (couleur(X,"gris"))
clair(X) <= (couleur(X,"blanc"))
```

Puis on peut poser les questions, exemple pour afficher tous les animaux clairs :

```
print(clair(X))
```

```
X
--------
chien
elephant
**************
X
----
ours
```

Pour afficher les animaux sombre et grand :

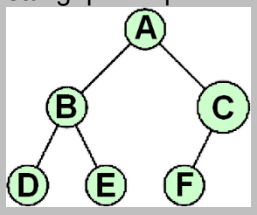
```
print(sombre(X) &(taille(X,'grand')))
```

I) Coder sous python l'exemple ci-dessus et sauvegarder sous animaux.py puis tester.
Rajouter les questions qui permettent d'afficher :
- Tous les animaux petits.
- Tous les animaux grand et clair.

Sauver à nouveau sous animaux.py
Faire une capture du résultat et de la partie de programme ajoutée.

II) Coder sous python l'arbre généalogique d'après les faits et règles suivants :

- A est le père de B
- A est le père de C
- B est le père de D
- B est le père de E
- D est le père de G
- C est le père de F



Les règles :

- X et Y sont frères si Z est père de X et de Y et que X et Y différents,

Codé : frere(X,Y) <= ((pere(Z,X)) & (pere(Z,Y)) & ~(X==Y)

- X et Y sont cousins si (Z est le père de X) et (W le père de Y) et (Z et W sont frères)
- X est le petit-fils de Y si Y est le père de Z et du Z est le père de X

Sauvegarder-sous famille.py.
Rajouter les lignes de codes pour afficher les frères, les cousins et les petits enfants puis sauver et tester.
Faire une capture du résultat.

III) Ce mode de programmation permet aussi de programmer facilement de la récursivité.
Coder le factoriel :

```
from pyDatalog import pyDatalog
pyDatalog.create_terms("N,Factoriel")
Factoriel[N]=N*Factoriel[N-1]
Factoriel[1]=1
print(Factoriel[4]==N)
```

Sauver sous factoriel.py, tester et capturer le résultat.

2. Programmation fonctionnelle :

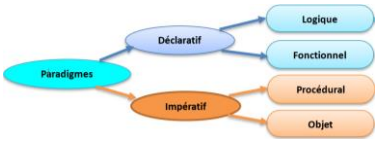
a) Définition :

La programmation **fonctionnelle** est apparue dans les années 1950 (Langage Lips). Elle consiste en l'évaluation de fonctions mathématiques : si on a les mêmes paramètres on a toujours le même résultat : ce sont des **fonctions pures**. Elles ne permettent pas d'affectations. Rien à voir avec les fonctions. Ces fonctions ne dépendent pas de l'état du programme. Un programme écrit en style fonctionnel se caractérise essentiellement par une chose : l'absence d'effets de bord.
Exemple :

Programmation non fonctionnelle	Programmation fonctionnelle

`compteur = 0`
`def compter():`
`  global compteur`
`  compteur += 1`
	`def compter(compteur):`
	`  return compteur+1`

compteur est global et peut être modifié en dehors de son état local (effet de bord)	Compteur est modifié localement puis on renvoie le résultat



b) Les fonctions pures :

i) La fonction lambda :

C'est une fonction anonyme qui permet de créer des fonctions sous forme condensé.

Exemple :

Pour faire une addition de deux nombres :

```
def addition(n,m):
    return (lambda n,m:n+m)(n,m)
print(addition(1,5))
```

```
*** Console de processus distant Réinitialisée ***
6
```

On donne les paramètres (n et m) puis le calcul à faire n+m et enfin les valeurs de n et m.

Cela peut être fait en une seule ligne de code :

```
print((lambda n,m:n+m)(1,5))
```

```
*** Console de processus distant Réinitialisée ***
6
```

IV) A l'aide de cette fonction lambda, afficher en une ligne de code :

- x*y avec x=3 et y=7 puis x=1 et y=3 (2 lignes de code)
- x/y avec x=6 et y=4 puis x=9 et y=3(2 lignes de code)
- 3*x+5*y+6*k avec x=2, y=4 et k=12 puis avec x=1, y=2 et k=3 (2 lignes de code)

Sauver sous fonctionnel.py, tester et capturer le résultat.

ii) La fonction map :

La fonction **map** prend en argument **une fonction** et **une collection de données**. Elle crée une nouvelle collection vide, applique la fonction à chaque élément de la collection d'origine et insère les valeurs de retour produites dans la nouvelle collection. Finalement, elle renvoie la nouvelle collection.

Exemple :

Un map simple qui prend en entrée une liste de mots et renvoie une liste contenant la longueur de chacun de ces noms grâce à la fonction len.

```
print(list(map(len,["Bonjour","Super","le","cours","sur","la","programmation","fonctionnelle"])))
```

```
*** Console de processus distant Réinitialisée ***
[7, 5, 2, 5, 3, 2, 13, 13]
```

V) A l'aide des fonctions map, lambda, et abs (valeur absolue) afficher en une ligne de code :

- une fonction qui permet de rajouter 2 à chaque éléments d'une liste : [10,8,4,1,3,6] puis [4,7,2,0,3,4,9] (2 lignes de codes, une par liste)
- une fonction qui affiche la valeur absolue de chaque éléments d'une liste : [-1,3,-5,-7,4] puis [9,-4,-7,-6,4,8]

Sauver sous fonctionnel.py, tester et capturer le résultat.

iii) La fonction filter :

La fonction **filter** prend en argument **une fonction** et **une collection de données**. Elle crée une nouvelle collection vide, applique le filtre à chaque élément de la collection d'origine et insère les valeurs de retour produites dans la nouvelle collection. Finalement, elle renvoie la nouvelle collection.

Exemple :

On a une liste dont on veut récupérer les éléments supérieurs à 10 :

```
print(list(filter(lambda n:n<10,[8,12,9,14,15,3,13])))
```

```
*** Console de processus distant Réinitialisée ***
[8, 9, 3]
```

iv) La fonction reduce:

La fonction **reduce** prend en entrée **une fonction** et **une collection d'éléments**. Elle renvoie une valeur créée en combinant les éléments de la collection.

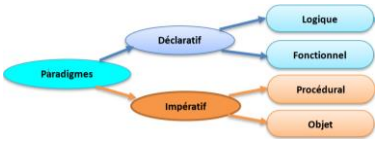
Pour utiliser reduce, il faut l'importer de la librairie functools

Exemple :

Faire l'addition des éléments d'une liste :

```
from functools import reduce
print(reduce(lambda x,y:x+y,[1,2,3,4,5]))
```

```
*** Console de processus di
15
```



On peut utiliser la même ligne de code pour concaténer des lettres d'une liste

```
print(reduce(lambda x,y:x+y,['T','O','T','O']))
```

```
*** Console de processus distant Réinitialisée ***
TOTO
```

VI) Que fait la ligne de code ci-dessous (argumentez) ?

```
print(reduce(lambda x,y:x*y,range(1,5)))
```

Coder cette ligne, sauver sous fonctionnel.py et capturer le résultat.
Tester cette même fonction pour les valeurs de 1 à 9, sauver et capturer le résultat.

c) Exemple sur une base de données :

Pour cet exemple, nous allons créer 4 élèves à l'aide d'un dictionnaire qui contient le nom, le prénom et les notes de NSI, philo, math et anglais.
Puis il suffit de créer une base composée de ces 4 élèves.
On cherche les élèves qui ont en-dessous de 10 (pbs)
On créer les lettres à envoyer aux parents.

VII) Pour cela coder le programme ci-dessous et sauver sous examens.py.
Tester et capturer le résultat.

```
#Création d'une base de données élèves
p1 = {"nom":"DUPONT","prenom":"Maxime","notes": [("NSI", 12), ("philo", 8), ("math", 15), ("anglais", 5)]}
p2 = {"nom":"PETIT","prenom":"Toto","notes": [("NSI", 10), ("philo", 2), ("math", 18), ("anglais", 16)]}
p3 = {"nom":"DURAND","prenom":"Titi","notes": [("NSI", 5), ("philo", 9), ("math", 10), ("anglais", 4)]}
p4 = {"nom":"DUBOIS","prenom":"David","notes": [("NSI", 20), ("philo", 19), ("math", 15), ("anglais", 14)]}
db = [p1,p2,p3,p4] #Création de la base de données
pbs=[(p["nom"],m) for p in db for m,n in p["notes"] if n<10] #on cherche les élèves qui ont des notes inférieur à 10
print(pbs) #affiche le nom et la matières des élèves qui ont en dessous de 10
print(["Cher monsieur %s, vous devez repasser : %s" %p for p in pbs]) #on créé toutes les lettres a envoyer
```

VIII) Dans le programme précédent rajouter trois élèves :

Nom	Prénom	NSI	Philo	Math	Anglais
MARTIN	Pierre	10	15	8	10
LEROY	Paul	18	12	14	16
MOREAU	Jacques	13	17	10	15

Rajouter l'affichage des lettres à envoyer aux élèves qui ont plus de 16/20 dans une matière pour les féliciter (Cher monsieur NOM PRENOM, félicitations, vous avez obtenu plus de 16/20 en : XXXXX)

```
'Cher monsieur PETIT Toto, félicitations, vous avez obtenu plus de 16/20 en math
```

et sauver sous examens.py.
Tester et capturer le résultat.

3. Programmation procédurale :

Un langage impératif donne des ordres, là ou un langage fonctionnel utilise des expressions. Il utilise :

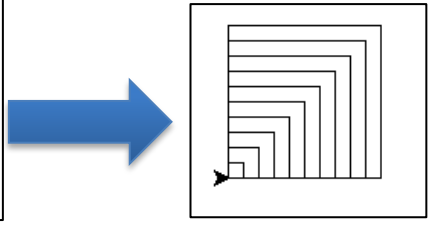
- L'affectation de variable
- Les boucles d'itérations
- La définition de fonctions…

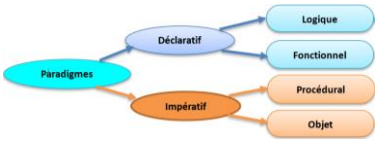
C'est le langage que vous utilisez depuis le début de l'année.
Exemple : utilisation de turtle

```
from turtle import *
avance=forward
gauche=left
droite=right
recule=backward

def rectangle (L,l): #Définition de la fonction rectangle
    for loop in range(2):
        avance(L)
        gauche(90)
        avance(l)
        gauche(90)

for i in range (10): #On trace 10 carrés de plus en plus grand
    rectangle(10*(i+1),10*(i+1))
```





4. Programmation orienté objet :

La programmation orientée objet a fait ses débuts dans les années 1960 avec des réalisation dans le langage Lips. Cependant elle a été formellement définie dans les années 1970 avec les langages Simula puis Smaltalk. En réalité, vous avez déjà programmé avec des objets en utilisant les méthode len(), append(),pop()….

a) Vocabulaire :

La conception objet possède un aspect intuitif fort dans la mesure où l'on s'efforce de calquer la **représentation informatique** sur **des entités physiques ou conceptuelles** apparaissant dans le monde à modéliser.
Un objet comprend **une partie figée** qui **représente son état et les liens** qui l'unissent à d'autres objets.
Un objet comprend **une partie dynamique** qui **décrit son comportement**, c'est-à-dire toutes les opérations qu'on peut lui appliquer ainsi que sa manière de réagir aux événements de l'environnement.
Sa **partie fixe** est constituée **d'un ensemble de champs ou attributs.**
Sa **partie dynamique** d'opérations appelées **méthodes**.
Certaines de ces méthodes constituent la partie visible de l'objet. C'est par elles que l'on s'adresse à lui. Elles constituent autant de services qu'on peut demander à l'objet de fournir.
Certains champs peuvent être **partiellement ou totalement inaccessibles** à d'autres objets : c'est **l'encapsulation.**
Un objet en accès **public** peut **être accessible et modifiable** par un autre objet.
Un objet en accès **privé ne** peut **être directement accessible** par un autre objet mais cela peut se faire indirectement par une méthode prévue dans la classe de cet objet**.**
Quand des objets possèdent **une structure et des comportements en commun**, on peut les regrouper sous forme de **classe**.
En résumé :

- **Une classe** c'est un plan de conception : ex: humain.
- **Un objet** c'est une instance de classe : ex: Pierre
- **Un attribut** c'est une variable de classe : ex: prenom, age…
- **Une propriété** c'est une manière de manipuler les attributs ex: lecture seule, accès non autorisé en dehors de la classe…
- **Une méthode** c'est une fonction d'une classe : ex: manger, parler, dormir…
- **L'Héritage** c'est une classe fille qui hérite d'une classe mère : ex : classe mère « animaux » et une classe fille : « panda », « chien », « chat » … donc la classe « chat » hérite de la classe « animaux ».

b) Codage d'une classe :

Le paramètre **self** représente l'instance courante de l'objet, on le retrouve dans toutes les méthodes de la classe.
On trouve des méthodes spéciales comme « **__init__** » pour initialiser l'objet. (Attention il y a 2 soulignés avant et après init)
Par convention, une classe **commence toujours par une majuscule** contrairement aux fonctions qui sont tout en minuscules.

Créons la classe mère : Alphabet_majuscule()

```
import string
class Alphabet_majuscules():#classe mère
    def __init__(self):
        self.lettres = string.ascii_uppercase

maj=Alphabet_majuscules() #on créé un objet maj
print(maj.lettres) #on affiche la méthode lettre de l'objet maj
```

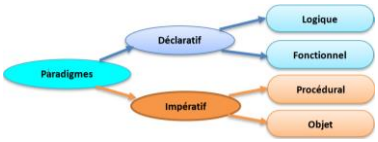
```
*** Console de processus dis
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Créons la classe fille : Alphabet_minuscule()

```
class Alphabet_minuscules(Alphabet_majuscules):#classe fille de Aphabet majuscules
    def __init__(self):
        Alphabet_majuscules.__init__(self) #on récupère les majuscules
        self.lettres = self.lettres.lower ()#on les mets en minucules

minuscules=Alphabet_minuscules()#on créé un objet minuscules
print(minuscules.lettres)#on affiche la méthode lettre de l'objet minuscule
```

```
abcdefghijklmnopqrstuvwxyz
```



Créons la classe petite-fille : Alphabet_tri() qui va permettre de trier les voyelles des consonnes.

```
class Alphabet_tri (Alphabet_minuscules):# classe fille de minuscules donc petite-fille de majuscules
    def __init__(self):
        Alphabet_minuscules.__init__(self) #on récupère les les minuscules
        self.voyelles = [] #on créé la méthode voyelles
        self.consonnes = [] #on créé la méthode consonnes
        for lettre in self.lettres :
            if lettre in "aeiouy":
                self.voyelles.append (lettre) #on ajoute les voyelles
            else :
                self.consonnes.append (lettre) #on ajoute les consonnes
    def listes_vers_chaines (self) :
        self.voyelles_chaine = "".join (self.voyelles)#on créé une chaine avec la liste de voyelles
        self.consonnes_chaine = "".join (self.consonnes)#on crée une chaine avec la liste des consonnes

test=Alphabet_tri() #on créé l'objet test
print(test.lettres) #on affiche la liste des lettres
print(test.voyelles) #on affiche la liste des voyelles
test.listes_vers_chaines() #on applique la méthode liste vers chaine
print(test.consonnes_chaine)# affiche les consonnes sous forme de chaine
```

```
*** Console de processus distant Réinitialisée ***
abcdefghijklmnopqrstuvwxyz
['a', 'e', 'i', 'o', 'u', 'y']
bcdfghjklmnpqrstvwxz
```

IX) Coder ces trois classes et les exemples puis sauver sous alphabet.py. Capturer le résultat.

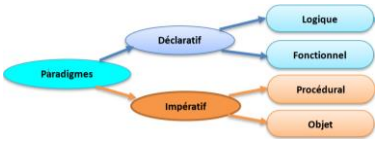
c) Exemple d'utilisation de classe : un jeu de cartes

X) Commençons par coder la classe Cartes composée de la documentation entre 3 """……""" et du constructeur puis sauver sous Cartes.py

```
class Cartes:
    """Cette classe définit une carte par sa valeur,sa couleur et sa figure"""
    #C'est le docstring : décrit la classe et renvoyé en tapant Cartes.__doc__
    def __init__(self,val,coul):#on définit le constructeur
        self.__valeur=val #l'attribut valeur est privé car __
        self.__couleur=coul #l'attribut couleur est privé car __
        if val==13:
            self.__figure="Roi"
        else:
            if val==12:
                self.__figure="Dame"
            else:
                if val==11:
                    self.__figure="Valet"
                else:
                    self.__figure="Aucune"#du 10 à l'as, il n'y a pas de figure
```

Ajouter les lignes de codes suivantes, sauver et capturer le résultat. Interpréter le résultat.

```
ma_carte=Cartes(13,"Coeur")
print(ma_carte)
print(ma_carte.__doc__)
```



Pour **obtenir** une valeur d'un attribut on va utiliser les **accesseurs** (getters). Nous allons créer des méthodes pour accéder de manière publique à ces accesseurs. En général, ces méthodes commencent par **Get**.

XI) Continuons par coder les 3 méthodes Get dans la même classe composée de la documentation entre 3 """……""" et des valeurs à retournées puis sauver sous Cartes.py

```
    def Get_valeur(self):
        """Retourne la valeur d'une carte"""
        return self.__valeur
    def Get_couleur(self):
        """Retourne la couleur d'une carte"""
        return self.__couleur
    def Get_figure(self):
        """Retourne la figure d'une carte"""
        return self.__figure
```

Tester en rajoutant trois lignes pour afficher la valeur, la couleur et la figure de ma_carte créée précédemment. Sauver puis capturer le résultat.

Dans la même logique, pour **modifier** une valeur d'un attribut on va utiliser les **mutateurs** (setters). Nous allons créer des méthodes pour rendre la modification publique de ces mutateurs. En général, ces méthodes commencent par **Set**.
On souhaite que la modification de la figure reste privée et donc on va créer une méthode Set_figure(val) avec 2 _ avant afin que cela soit le cas.

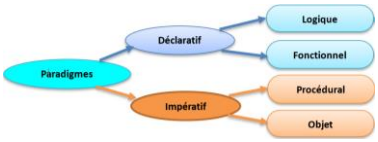
XII) Continuons par coder les 3 méthodes Set dans la même classe composée de la documentation entre 3 """……""" et des valeurs à modifier puis sauver sous Cartes.py

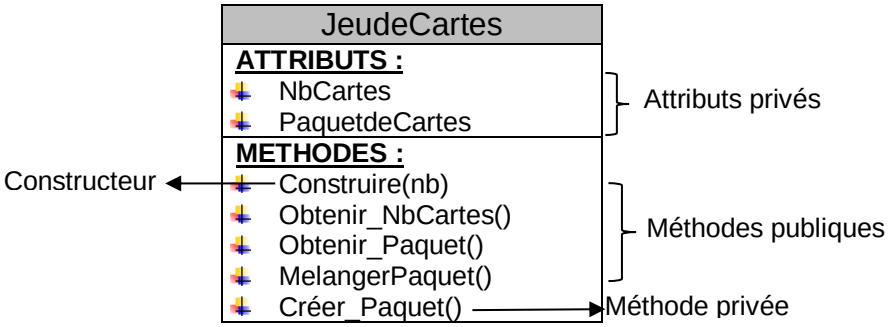
```
    def __Set_figure(self,val):
        """Méthode privée pour changer la figure d'une carte suivant sa valeur"""
        if val==13:
            self.__figure="Roi"
        else:
            if val==12:
                self.__figure="Dame"
            else:
                if val==11:
                    self.__figure="Valet"
                else:
                    self.__figure="Aucune"#du 10 à l'as, il n'y a pas de figure
```

```
    def Set_valeur(self,val):
        """Retourne Vrai si la valeur de la carte a été changée par val et
        faux sinon (valeur non comprise en 2 et 14 : l'as c'est 14)"""
        if 2<=val<=14:
            self.__valeur=val
            self.__Set_figure(val)
            return True
        else:
            return False
    def Set_couleur(self,coul):
        """Retourne Vrai si la couleur de la carte a été changée par coul et
        faux sinon (valeur non comprise dans la liste ["Coeur","Pique","Carreau","Trèfle"]"""
        liste_couleurs=["Coeur","Pique","Carreau","Trèfle"]
        if coul in liste_couleurs:
            self.__couleur=coul
            return True
        else:
            return False
```

Tester en rajoutant 5 lignes pour modifier la valeur, la couleur (on souhaite avoir un valet de pique) et afficher la valeur, la couleur et la figure de ma_carte créée précédemment. Sauver puis capturer le résultat.

Pour le moment, nous avons programmé une carte, il serait bien de créer une classe JeudeCartes qui va être une classe fille de la classe Cartes puis un jeu de cartes est composé de cartes.





Pour mélanger un paquet de cartes (32 ou 52) il faudra la méthode shuffle qui est dans la librairie random qu'il faut importer : `import random`

XIII) Après avoir importé la librairie tout en haut du code, créer la classe JeudeCartes sous la classe Cartes puis sauver sous Cartes.py

```
class JeudeCartes:
    """Classe définissant une jeu de cartes caractérisé par son nombre et son paquet de cartes"""
    def __CreerPaquet(self):
        """Méthode pour créer un jeu de cartes de 32 ou 52 cartes non mélangé"""
        monpaquet=[]
        if self.__NbCartes==32:
            num_debut=7
        else:
            num_debut=2
        for coul in ["Coeur","Pique","Carreau","Trèfle"]:
            for i in range (num_debut,15,1):#Le 15 est exclu
                nouvelleCarte=Cartes(i,coul)
                monpaquet.append(nouvelleCarte)
        return monpaquet
    def __init__(self,nb):
        """Constructeur de la classe JeudeCartes"""
        self.__NbCartes=nb
        self.__PaquetdeCartes=self.__CreerPaquet()
    def GetNbCartes(self):
        """Retourne de Nb de cartes du jeu 32 ou 52"""
        return self.__NbCartes
    def GetPaquet(self):
        """Retourne le paquet de cartes"""
        return self.__PaquetdeCartes
    def MelangerCartes(self):
        """Mélange le paquet de cartes"""
        random.shuffle(self.__PaquetdeCartes)
```

On remarque qu'il faut mettre __CreerPaquet en premier car on l'utilise pour le constructeur.
Créons un jeu de 32 cartes puis affichons le à l'aide des lignes de code suivantes :

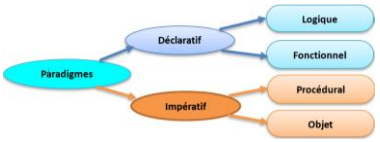
```
mon_jeu=JeudeCartes(32)
print(mon_jeu.GetNbCartes())
mon_paquet=mon_jeu.GetPaquet()
for i in range (len(mon_paquet)):
    print("Valeur : "+str(mon_paquet[i].Get_valeur())+" Couleur : "+mon_paquet[i].Get_couleur()+ " Figure : "+mon_paquet[i].Get_figure())
print("**********MELANGE DES CARTES**********")
```

Sauver et capturer le résultat.

XIV) Rajouter des lignes de code pour mélanger le paquet puis afficher le paquet mélangé puis sauver sous Cartes.py, tester et capturer le résultat.

XV) Rajouter dans la classe JeudeCarte la méthode DistribuerCarte qui enlève une carte du PaquetdeCartes (avec la méthode pop) et qui la renvoie à l'utilisateur puis sauver sous Cartes.py, tester en créant une carte distribuée et afficher celle et le mon_jeu afin de vérifier qu'elle n'y soit plus et capturer le résultat.

On pourrais rajouter un classe pour les joueurs, il faudrait aussi contraindre le jeu de départ à 52 ou 32 cartes…



5. Exercices POO :

a) Classe point :

On souhaite créer une classe qui permet de définir un point soit par ces coordonnées cartésiennes (x,y) ou par ces coordonnées polaires (rho, theta) avec rho : $\rho = \sqrt{x^2 + y^2}$ et théta :$\theta = atan2(y, x)$. Il faudra importer la librairie math et convertir théta en ° qui est donné en radians par la fonction atan2.

XVI) Donner les attributs et méthodes de la classe Point en complétant le tableau ci-dessous.

Point
ATTRIBUTS : . . . .
METHODES : Construire(abs,ord) . . . .

XVII) Créer la classe point comprenant deux méthode privée CalculRho et CalculThéta, le constructeur et quatre méthodes publiques GetX, GetY, GetRho et GetThéta. Tester avec 4 points :

- A=(2,3)
- B=(-5,6)
- C=(2,-5)
- D=(-1,-5), afficher pour chacun d'eux rho et théta. Sauver sous Point.py, tester et capturer le résultat.

b) Classe Pile :

Soit la classe Pile suivante :

Pile
ATTRIBUTS : Pile (privé)
METHODES : Construire() Empiler(x) Depiler() Est_vide() Obtenir_Pile()

La méthode « Est-vide() » renvoie vrai si la pile est vide et faux sinon.

XVIII) Créer la classe Pile décrite ci-dessus et sauver sous Pile_File.py puis tester en créant une pile vide : ma_pile puis empiler les valeurs 3,7,6,8 puis dépiler 2 fois et empiler 5. Afficher si la pile est vide ou non et afficher ma_pile. Capturer le résultat.

c) Classe File :

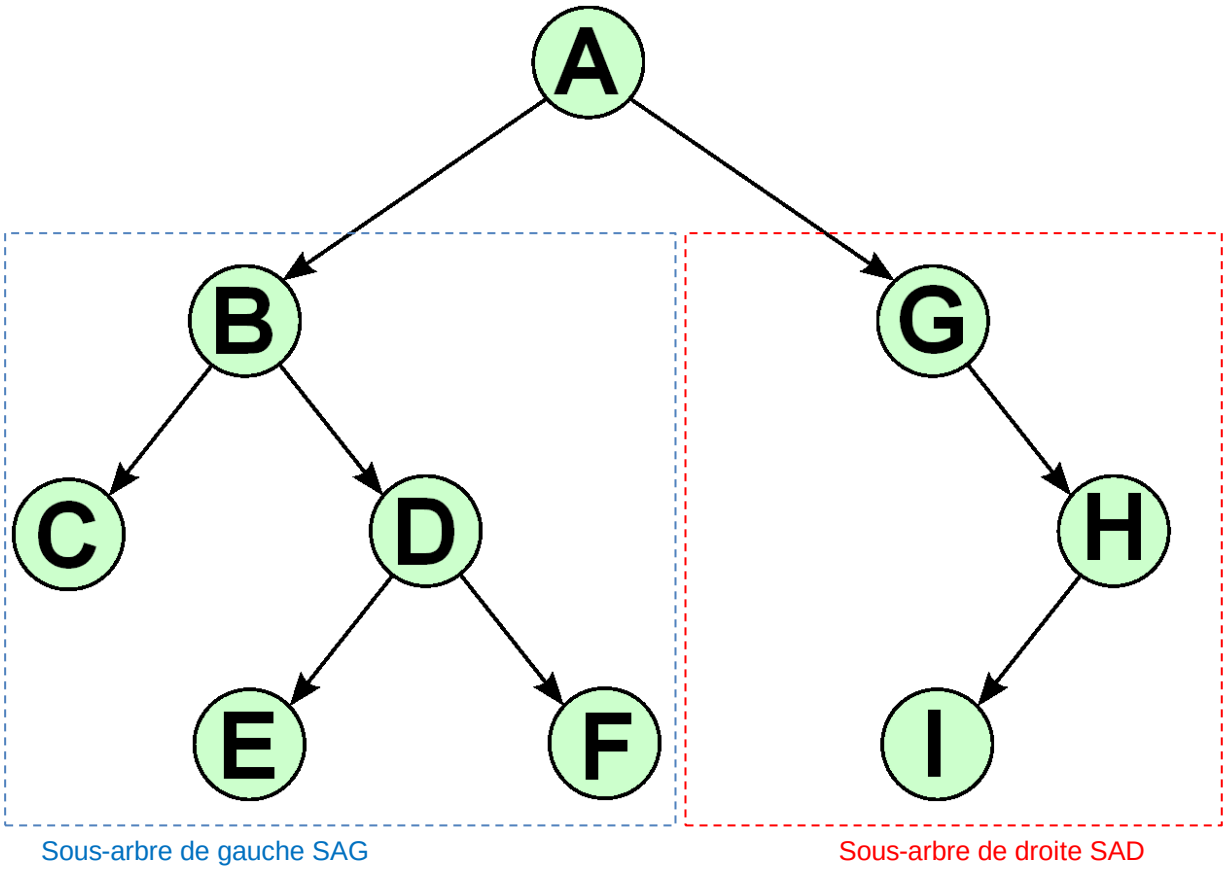
Soit la classe File suivante :

File
ATTRIBUTS : File (privé)
METHODES : Construire() Enfiler(x) Defiler() Est_vide() Obtenir_File()

La méthode « Est-vide() » renvoie vrai si la file est vide et faux sinon.

XIX) Créer la classe File décrite ci-dessus et sauver sous Pile_File.py puis tester en créant une file vide : ma_file puis enfiler les valeurs 3,7,6,8 puis défiler 2 fois et enfiler 5. Afficher si la file est vide ou non et afficher ma_file. Capturer le résultat.

d) Les arbres :



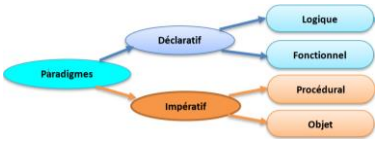
A chaque nœud de l'arbre binaire, on associe :
- Une clé (valeur associée au nœud)
- Un sous-arbre de gauche SAG
- Un sous-arbre de droite SAD

Nous allons créer les classes suivantes (la file a été créée en question c))

Nœud	Arbre	File
ATTRIBUTS : - Clé (public)	**ATTRIBUTS :** - Racine (public) - Sag (public) - Sad (public)	**ATTRIBUTS :** - File (privé)
METHODES : - Construire()	**METHODES :** - Construire() ……(seront préciser par la suite)	**METHODES :** - Construire() - Enfiler(x) - Defiler() - Est_vide() - Obtenir_File()

Utilisation :
```
a= Arbre('A')
a.Sag=Arbre('B')
a.Sag.Sag=Arbre('C')
a.Sag.Sad=Arbre('D')
a.Sag.Sad.Sag=Arbre('E')
a.Sag.Sad.Sad=Arbre('F')
a.Sad=Arbre('G')
a.Sad.Sad=Arbre('H')
a.Sad.Sad.Sag=Arbre('I')
```

XX) Coder les trois classes décrites ci-dessus (on copiera celle de la file de l'exercice précédent) puis créer l'arbre a ci-dessus et sauver sous arbre.py



Pour calculer la taille de l'arbre a :

si a est vide
  alors taille=0
  sinon taille(a)=1+taille(Sag)+taille(Sad)

XXI) Ajouter dans la classe arbre la méthode Taille(self,a) d'après l'algorithme ci-dessus et sauver sous arbre.py . Tester cette méthode en affichant la taille de l'arbre créé auparavant. Capturer le résultat.

Pour calculer la hauteur de l'arbre a :

si a est vide (Sad et Sag =None)
  alors la hauteur =1
  sinon hauteur(a)=max(taille(Sag),taille(Sad))-1

XXII) Ajouter dans la classe arbre la méthode Hauteur(self,a) d'après l'algorithme ci-dessus et sauver sous arbre.py . Tester cette méthode en affichant la hauteur de l'arbre créé auparavant. Capturer le résultat.

Parcours en largeur de l'arbre :

Créer une liste lst vide
Si a n'est pas vide :
    Alors Créer une file F vide et enfiler a
    Tant que F n'est pas vide :
        arbre_courant vaut l'arbre défilé
        Ajouter à la liste lst la clé de la racine de l'arbre_courant (arbre_courant.Racine.Cle)
        Si le sous-arbre de gauche de l'arbre_courant n'est pas vide :
            Alors enfiler celui-ci dans F.
        Si le sous-arbre de droite de l'arbre_courant n'est pas vide :
            Alors enfiler celui-ci dans F.
Retourner lst.

XXIII) Ajouter dans la classe arbre la méthode ParcoursLargeur(self) d'après l'algorithme ci-dessus et sauver sous arbre.py. Tester cette méthode en affichant le parcours en largeur de l'arbre créé auparavant. Capturer le résultat.

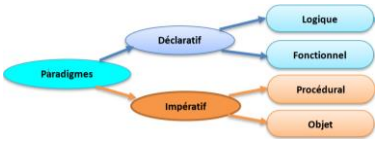
Parcours en profondeur Préfixe :

Dans le parcours préfixe, la racine est traitée avant les appels récursifs sur les sous-arbres de gauches et droite.
Cette méthode a besoin en paramètres d'une liste vide (qui contiendra le parcours préfixe) et d'un arbre.
Si arbre n'est pas vide :
    Ajouter la clé de la racine de arbre à la fin de la liste.
    Appeler la méthode Parcours_Prefixe avec la liste précédente et le sous-arbre de gauche.
    Appeler la méthode Parcours_Prefixe avec la liste précédente et le sous-arbre de droite.

XXIV) Ajouter dans la classe arbre la méthode ParcoursPrefixe(self,lst,arbre) d'après l'algorithme ci-dessus et sauver sous arbre.py. Tester cette méthode en affichant le parcours en préfixe de l'arbre créé auparavant. Pour cela créer une list_pref vide, appliquer la méthode puis faites un print. Capturer le résultat.

Parcours en profondeur infixe :

Dans le parcours infixe, la racine est traitée entre les appels récursifs sur les sous-arbres de gauches et droite.
Cette méthode a besoin en paramètres d'une liste vide (qui contiendra le parcours infixe) et d'un arbre.
Si arbre n'est pas vide :
    Appeler la méthode Parcours_Infixe avec la liste précédente et le sous-arbre de gauche.
    Ajouter la clé de la racine de arbre à la fin de la liste.
    Appeler la méthode Parcours_Infixe avec la liste précédente et le sous-arbre de droite.



XXV) Ajouter dans la classe arbre la méthode ParcoursInfixe(self,lst,arbre) d'après l'algorithme ci-dessus et sauver sous arbre.py. Tester cette méthode en affichant le parcours en infixe de l'arbre créé auparavant. Pour cela créer une list_in vide, appliquer la méthode puis faites un print. Capturer le résultat.

Parcours en profondeur postfixe :

Dans le parcours infixe, la racine est traitée après les appels récursifs sur les sous-arbres de gauches et droite.

Cette méthode a besoin en paramètres d'une liste vide (qui contiendra le parcours postfixe) et d'un arbre.

Si arbre n'est pas vide :

Appeler la méthode Parcours_Postfixe avec la liste précédente et le sous-arbre de gauche.
Appeler la méthode Parcours_Posfixe avec la liste précédente et le sous-arbre de droite.
Ajouter la clé de la racine de arbre à la fin de la liste.

XXVI) Ajouter dans la classe arbre la méthode ParcoursPostfixe(self,lst,arbre) d'après l'algorithme ci-dessus et sauver sous arbre.py. Tester cette méthode en affichant le parcours en postfixe de l'arbre créé auparavant. Pour cela créer une list_post vide, appliquer la méthode puis faites un print. Capturer le résultat.

```
La taille de l'arbre a est : 9
La hauteur de l'arbre a est : 4
Parcours en largeur :  ['A', 'B', 'G', 'C', 'D', 'H', 'E', 'F', 'I']
Parcours en préfixe :  ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I']
Parcours en infixe :  ['C', 'B', 'E', 'D', 'F', 'A', 'G', 'I', 'H']
Parcours en postfixe :  ['C', 'E', 'F', 'D', 'B', 'I', 'H', 'G', 'A']
```