T NSI **Initiation à la programmation orientée objet**

A. Historique et concepts de base

La **programmation orientée objet** (POO) est un paradigme de programmation qui consiste à considérer un programme comme un ensemble d'objets en interaction.
Les **objets** peuvent représenter un concept du monde réel (comme un meuble, une personne, ou un livre par exemple…).

Histoire : La programmation orientée objet a fait ses débuts dans les années 1960 avec les réalisations dans le langage Lisp. Cependant, elle a été formellement définie avec les langages Simula (vers 1970, créé par les norvégiens Ole-Johan Dahl et Kristen Nygaard, prix Turing 2001 et médaille John von Neumann en 2002), puis SmallTalk (Allan Kay au Palo Alto Research Center de Xerox). Puis elle s'est développée dans les langages anciens comme le Fortran, le Cobol et est même incontournable dans les langages plus récents comme Java.

Les objets possèdent des caractéristiques qui lui sont propres : les **attributs**.
Par exemple, pour une personne, sa date de naissance, sa taille, son poids…
Sur ces objets, on peut réaliser des actions, qui vont correspondre à des fonctions propres à ces objets : ces fonctions s'appellent des **méthodes**.
Par exemple, pour une personne, la méthode *grandir(h)* permettra d'augmenter sa taille de h cm.
Pour définir cette personne, avec ses attributs ainsi que les méthodes qui lui sont associées, on peut définir une **classe**. Une classe est une sorte de "moule" permettant de créer des objets ayant tous les mêmes noms d'attributs et les mêmes méthodes pour agir sur eux.

Les objets appartenant à une même classe sont appelés **instances** de cette classe. Ainsi, les instances d'une même classe ne diffèrent les unes des autres que par la valeur de leurs attributs.
Par exemple, si on a créé une classe Personne possédant les attributs "année_de_naissance", "taille" (en cm) et "poids" (en kg), on peut définir un individu nommé Paul et un autre nommé Pierre par :
Paul = Personne(2000, 182, 85) et Pierre = Personne(2005, 165, 68).
Paul et Pierre sont deux instances de la classe Personne.

Un concept important dans la POO est **l'encapsulation** : la fonctionnalité interne de l'objet et les variables qu'il utilise pour effectuer son travail, sont en quelque sorte enfermées dans l'objet. Les autres objets et le monde extérieur ne peuvent y accéder qu'à travers des méthodes bien définies : l'**interface** de l'objet.
Pour ce faire, une donnée peut être déclarée en accès :
▪**Public :** les autres objets peuvent accéder à la valeur de cette donnée et/ou la modifier.
▪**Privé :** les autres objets n'ont pas le droit d'accéder directement à la valeur de cette donnée, ni de la modifier. En revanche, ils peuvent le faire indirectement par des méthodes de l'objet concerné (si celles-ci existent en accès public) (interface).
Une méthode peut aussi être publique ou privée.

Par exemple, pour modifier la taille de Paul et le faire grandir de 2 cm, on utilisera la méthode grandir(h) : Paul.grandir(2). On pourra ainsi s'assurer, par exemple, que h est positif.

B. Exemple en Python

On reprend l'exemple précédent. On définit une classe Personne à l'aide de trois caractéristiques (année de naissance, taille en cm, poids en kg) et on définit une fonction propre à la classe (grandir) qui admet un paramètre numérique (exprimé en cm).

```
class Personne:
    """Classe définissant une personne par :
        -son année de naissance
        -sa taille en cm
        -son poids en kg
"""
    def __init__(self,annee, taille, poids):
        """Constructeur de la classe personne
"""
        self.annee = annee
        self.taille = taille
        self.poids = poids

    def grandir(self,h):
        """Méthode permettant d'augmenter la taille de h cm.
"""
        self.taille += h


Paul = Personne(2000,182,85)
init_taille = Paul.taille
Paul.grandir(2)
new_taille = Paul.taille
print("Paul pèse",Paul.poids,"kg.")
print("Paul mesure à présent",new_taille,"cm après avoir
grandi de 2 cm en 1 an.")
print("Avant cela, il mesurait",init_taille,"cm.")
```

Ce programme affiche :

Paul pèse 85 kg.
Paul mesure à présent 184 cm après avoir grandi de 2 cm en 1 an.
Avant cela, il mesurait 182 cm.

Ici, les **attributs** de la **classe** Personne sont : année, taille, poids. Ils sont ici définis comme **Publics**. Pour les déclarer **Privés**, il faudrait les faire précéder d'un *double underscore* :
par exemple : self.__taille = taille

Ici, on accède donc aux attributs d'une classe simplement en tapant :
nom_objet . attribut

Documentation : le texte placé entre trois guillemets """ …""" permet de documenter les classes et méthodes et peut être affiché avec la méthode __doc__.
Exemple :

```
print(Paul.__init__.__doc__)
```

renvoie :
Constructeur de la classe personne

Constructeur d'une classe :

Le constructeur d'une classe est une fonction
▪ dont le nom est nécessairement __init__ :
▪ admettant au moins un paramètre nommé `self` placé obligatoirement en premier s'il y en a plusieurs.

Le paramètre `self` représente l'objet cible : c'est une variable qui contient une référence vers l'objet qui est en cours de création. Grâce à ce dernier, on va pouvoir accéder aux attributs et fonctionnalités de l'objet cible.

Instanciation et variables d'instance :

Comme vu précédemment, une classe est une sorte de moule qui sert à fabriquer des objets.
L'instanciation est l'opération qui consiste à créer un objet. L'objet ainsi créé est alors appelé une **instance** de la classe.
Exemple : si on reprend la classe Personne de l'exemple précédent, en définissant :
`Paul = Personne(2000,182,85)`
nous avons défini une instance de la classe Personne.
On peut ainsi définir plusieurs instances d'une même classe, chacune ayant des attributs qui lui sont propres.
Si on demande d'afficher la variable Paul, on voit apparaitre quelque chose de la forme :
<__main__.Personne object at 0x00000262F6CD8F28>
Cela signifie que Paul est une référence vers l'objet créé, c'est-à-dire une indication sur son emplacement en mémoire. Ici, par exemple, l'objet Personne se trouve en mémoire à la position 0x00000262F6CD8F28.

Un **attribut** de la classe est représenté par une variable appelée **variable d'instance**, accessible à l'aide du paramètre self. L'idée est que le constructeur initialise ces variables d'instances, qui seront stockées dans l'objet et en mémoire pour toute la durée de vie de l'objet.

Il est important de faire la différence entre les deux types de variables qui se trouvent dans le code de ce constructeur :
- la variable `self.taille` (par exemple) représente la variable d'instance, c'est-à-dire celle associée à l'objet, qui existe à partir de la création de l'objet jusqu'à sa destruction.
- la variable `taille` représente le paramètre reçu par le constructeur et n'existe que dans le corps de ce dernier.
Le paramètre `self` permet donc d'accéder aux variables d'instances, c'est-à-dire aux attributs de l'objet cible, depuis le constructeur.

Méthodes d'une classe :

Une **méthode** d'une classe est une fonction définie au sein de la classe et admettant au moins le paramètre `self` (obligatoirement en premier s'il y a plusieurs paramètres).

Le paramètre `self` représente l'objet cible sur lequel la méthode est appelée. Il permet notamment d'avoir accès aux variables d'instance de l'objet.

Dans l'exemple précédent, `grandir(self, h)` est une méthode admettant deux paramètres, dont le second est un nombre. Cette méthode redéfinit la variable d'instance `self.taille` en lui ajoutant la valeur passée en second paramètre.
Une méthode est appelée en spécifiant l'objet cible (ici `Paul`) suivi de la méthode en utilisant l'opérateur d'appel : le point. Cela donne ici : `Paul.grandir(2)`

La méthode `grandir(self, h)` est une méthode publique.
Pour définir une méthode privée, il faut la faire précéder d'un *double underscore* :
`def __ma_méthode(self,…) :`

Les méthodes dédiées :

Pour utiliser ou modifier les attributs, on utilisera de préférence des méthodes dédiées dont le rôle est de faire l'**interface** entre l'utilisateur de l'objet et la représentation interne de l'objet (ses attributs). Il existe deux familles :
▪Les **accesseurs** (ou **getters**) : pour obtenir la valeur d'un attribut.
▪Les **mutateurs** (ou **setters**) : pour modifier la valeur d'un attribut.

Dans notre cas, on souhaite accéder de manière publique à tous les accesseurs. En règle générale, on nomme ces accesseurs en commençant par `Get` suivi du nom de notre attribut.
Exemple : si nos **attributs** sont maintenant **privés**, on peut accéder à **taille** avec :

```
def GetTaille(self):
     return self.__taille


Paul = Personne(2000,182,85)
print(Paul.GetTaille() )
```
renvoie : `182`

Pour les mutateurs, on les nommera en commençant par `Set` suivi du nom de l'attribut à modifier.

Exercice :

On veut ajouter un nouvel attribut aux objets de la classe Personne : le "statut juridique", c'est-à-dire s'il est Majeur ou Mineur en 2020. On veut donc initialiser une variable d'instance self.__statut dans la méthode constructeur __init__.

Cependant, cet attribut ne peut pas être ajouté dans la liste des paramètres du constructeur, car l'utilisateur pourrait se tromper au moment de créer l'objet :
Par exemple : Jean = Personne (2015, 107, 20, "Majeur") est aberrant.
Le statut doit être **calculé** à partir de l'année de naissance.

1) Pour faire cela "proprement", créer une méthode privée def __SetStatut(self) qui affecte à la variable d'instance self.__statut le statut de l'objet, à savoir "Majeur" ou "Mineur" selon son année de naissance.

2) Initialiser dans __init__ la variable d'instance self.__statut simplement par un appel au mutateur : self.__SetStatut( ) .

3) Créer un accesseur GetStatut(self) qui renvoie le statut de l'objet.

Redéfinition d'opérations (surcharge d'opérateur) :

Intéressons-nous aux opérations que l'on peut faire sur des objets. Par exemple, si on souhaite définir un objet `vecteur` représentant un vecteur du plan, il serait utile de définir la **somme** de deux de ces vecteurs, qui est aussi un objet de classe vecteur, en utilisant le même symbole "+" que pour la somme de deux nombres. Python permet de définir une telle opération de la manière suivante :

```
class vecteur:
     def __init__(self, x, y) :
          self.x = x
          self.y = y

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

u,v = vecteur(-2,8) , vecteur(3,-5)
w = u + v
print( f"w( {w.x} , {w.y} )" )
```

retourne : w(1 , 3)

Cette capacité du langage Python s'appelle la **surcharge d'opérateur**.
On a étendu à la classe vecteur l'action de l'opérateur "+" défini par la méthode interne __add__.

TP Programmation orientée objet

Création et manipulation d'un jeu de cartes

A. Création d'une carte à jouer

On souhaite créer une classe Carte permettant de créer des cartes à jouer (correspondant à l'élément de base d'un jeu de 32 ou 52 cartes) "informatiquement".
Dans le monde réel, une carte à jouer est définie par :
Sa couleur : Carreau, Cœur, Pique ou Trèfle.
Sa valeur : de 2 à 10 (cartes numérotées) et de 11 à 13 pour les figures, et 14 pour l'as.
(La valeur du Valet est 11, celle de la Dame est 12 et celle du Roi est 13).
Sa figure : Aucune (pour les cartes numérotées et l'as), Valet, Dame ou Roi.

Les attributs de la classe Carte seront donc : valeur, couleur, figure.
La classe Carte doit aussi être munie de méthodes permettant d'interagir avec la carte pour attribuer une valeur ou récupérer la figure d'une carte par exemple.

Créer une classe Carte constituée des méthodes suivantes (à compléter). On enregistrera le fichier Python en **carte.py** et on l'utilisera comme un **module** dans un autre fichier Python.

```
class Carte:
    def __init__(self,val,coul):
            #Appeler le mutateur SetValeur pour initialiser la valeur de la carte.
            #La figure sera déterminée automatiquement à partir de la valeur.
            #Appeler le mutateur SetCouleur pour initialiser la couleur de la carte.

    def GetValeur(self):

    def GetCouleur(self):

    def GetFigure(self):

    def __SetFigure(self,val):
        """Méthode privée pour changer la figure en fonction
de la nouvelle valeur val.
"""
    def SetValeur(self, val):
        """Retourne True si la valeur de la carte a pu être
changée par val et False sinon (valeur non comprise dans
l'intervalle [2 ; 14]). Cette méthode modifie aussi la figure
associée à  val, par un appel à __SetFigure.
"""
    def SetCouleur(self,coul):
        """Retourne True si la couleur de la carte a été
changée par coul et False sinon (lorsque coul non comprise
dans le domaine des couleurs).
"""
```

Tester la classe Carte en tapant dans un fichier :

```
from carte import *

ma_carte = Carte( 11, "Trèfle" )
print(ma_carte.GetFigure( ) )
```

```
if ma_carte.SetValeur(13) :
        print( ma_carte.GetFigure( ) )
```

Dans la console, on doit obtenir :
Valet
Roi

B. Création d'un paquet de cartes à jouer

La classe Carte va nous servir à créer un jeu de cartes complet, de 32 ou 52 cartes.
On dit que l'objet "jeu de cartes" est un objet agrégé (constitué) de 32 ou 52 cartes (objets de type Carte). Comme c'est un objet, on va donc devoir créer une classe JeuDeCarte.
Compléter :

```
from carte import*
from random import*

class JeuDeCarte :
        """ Classe définissant un jeu de cartes caractérisé par :
-son nombre de cartes Nbcartes
-son paquet de cartes (une liste de cartes)"""
        def __CreerPaquet(self) :
                """Méthode privée pour créer le paquet de cartes classé
par valeurs et couleurs : il est donc non mélangé."""
                monpaquet = [ ]
                if self.__Nbcartes == 32 :
                        num_debut = 7
                else:
                        num_debut = 2
                for coul in …

                return monpaquet

        def __init__(self, nb) :
                """Constructeur de la classe JeuDeCarte. nb est le nombre de cartes
du jeu."""
                self.__Nbcartes = …..
                self.__Paquet = ……

        def GetNbcartes(self):
                """Retourne le nombre de cartes du jeu de cartes"""

        def GetPaquet(self):
                """Retourne le paquet de cartes"""

        def MelangerPaquet(self):
                """Mélange aléatoirement le paquet de cartes avec la méthode shuffle du
module random."""
```

Utiliser la classe JeuDeCarte pour créer un jeu de 32 cartes, afficher toutes ses cartes en console, le mélanger, l'afficher à nouveau.
On peut compléter cette classe assez sommaire par d'autres méthodes, comme distribuer une carte, afficher le jeu, ou sécuriser la création d'un jeu en contraignant la création à 32 ou 52 cartes etc.