1) introduction

Jusqu'à présent nous avons vu un seul paradigme de programmation (un paradigme est une manière de voir les choses, une représentation du monde) : la programmation impérative. La programmation impérative repose sur des notions qui vous sont familières :

- la séquence d'instructions (les instructions d'un programme s'exécutent l'une après l'autre)
- l'affectation (on attribue une valeur à une variable, par exemple : a = 5)
- l'instruction conditionnelle (if / else)
- la boucle (while et for)

La programmation impérative est loin d'être le seul paradigme de programmation (même si c'est sans doute le plus courant). Nous allons étudier deux autres paradigmes : le paradigme objet et le paradigme fonctionnel.

2) le paradigme fonctionnel

Comme dit l'année dernière dans la partie du cours consacrée aux effets de bord, le paradigme fonctionnel cherche à éviter au maximum les effets de bord, dit autrement, en programmation fonctionnelle on va éviter de modifier les valeurs associées à des variables. Pour ce faire, on va chercher au maximum à utiliser les fonctions (d'où le nom de programmation fonctionnelle), mais ces fonctions ne devront pas modifier les variables : en programmation fonctionnelle, on s'efforce de coder des fonctions qui ne modifient pas l'état courant des variables. Les fonctions utilisées en programmation fonctionnelle sont parfois appelées "fonction pure" : le résultat renvoyé par une fonction pure doit uniquement dépendre des paramètres passés à la fonction et pas des valeurs externes à la fonction (elle ne doit pas non plus engendrer d'effet de bord):

Intéressons-nous au programme Python suivant :

```
i = 5
def fct():
  if i > 5:
    return True
  else :
    return False
fct()
```

La fonction ci-dessus n'est pas une fonction pure, car la valeur renvoyée par la fonction fct (True ou False) dépend d'une valeur extérieure à la fonction.

Alors que dans ce cas :

```
def fct(i):
  if i > 5:
    return True
  else :
    return False
fct(5)
```

La fonction ci-dessus est une fonction pure, car la valeur renvoyée par la fonction fct (True ou False) dépend uniquement du paramètre passé à la fonction.

Même si certains langages de programmation ont été conçus pour "imposer" au programmeur le paradigme fonctionnel (Lisp, Scheme, Haskell…), il est tout à fait possible d'utiliser le paradigme fonctionnel avec des langages de programmation plus "généralistes" (Python par exemple).

Nous allons maintenant travailler sur un exemple de programme Python utilisant le paradigme fonctionnel :

Considérons le programme suivant :

```
l = [4,7,3]
def ajout(i):
  l.append(i)
```

Le programme ci-dessus ne respecte pas le paradigme fonctionnel, car nous avons un effet de bord (la variable l est modifiée par la fonction ajout).

Alors que dans le cas ci-dessous :

```
def ajout(i,l):
  tab = l + [i]
  return tab
```

La fonction ajout ne modifie aucune variable, elle crée un nouveau tableau (tab) à partir du tableau l et du paramètre i (le signe + permet de créer un nouveau tableau, ce nouveau tableau est constitué des éléments contenus dans le tableau l auxquels on ajoute la valeur i), la fonction renvoie le tableau ainsi créé.

D'une façon plus générale, la méthode append de Python ne respecte pas le paradigme fonctionnel puisque append modifie une donnée existante. Le paradigme fonctionnel va amener le programmeur non pas à modifier une valeur existante, mais plutôt à créer une nouvelle grandeur à partir de la grandeur existante : une grandeur existante n'est jamais modifiée, donc aucun risque d'effet de bord.

3) le paradigme objet

La programmation orientée objet repose, comme son nom l'indique, sur le concept d'objet.

Un objet dans la vie de tous les jours, vous connaissez, mais en informatique, qu'est ce que c'est ? Une variable ? Une fonction ? Ni l'un ni l'autre, c'est un nouveau concept.

Imaginez un objet (de la vie de tous les jours) très complexe (par exemple un moteur de voiture) : il est évident qu'en regardant cet objet, on est frappé par sa complexité (pour un non spécialiste). Imaginez que l'on enferme cet objet dans une caisse et que l'utilisateur de l'objet n'ait pas besoin d'en connaître son principe de fonctionnement interne pour pouvoir l'utiliser. L'utilisateur a, à sa disposition, des boutons, des manettes et des écrans de contrôle pour faire fonctionner l'objet, ce qui rend son utilisation relativement simple. La mise au point de l'objet (par des ingénieurs) a été très complexe, en revanche son utilisation est relativement simple. Programmer de manière orientée objet, c'est un peu reprendre cette idée : utiliser des objets sans se soucier de leur complexité interne. Pour utiliser ces objets, nous n'avons pas à notre disposition des boutons, des manettes ou encore des écrans de contrôle, mais des attributs et des méthodes (nous aurons l'occasion de revenir longuement sur ces 2 concepts). Un des nombreux avantages de la programmation orientée objet (POO), est qu'il existe des milliers d'objets (on parle plutôt de classes, mais là aussi nous reviendrons sur ce terme de classe est peu plus loin) prêts à être utilisés (vous en avez déjà utilisé de nombreuses fois sans le savoir). On peut réaliser des programmes extrêmement complexes uniquement en utilisant des classes préexistantes.

Les idées sous-tendant le paradigme objet datent des années 60. Mais il faudra attendre le début des années 70 et la mise au point du langage Smalltalk pour que le paradigme objet gagne en popularité chez les informaticiens. Aujourd'hui de nombreux langages permettent d'utiliser le paradigme objet : C++, Java,...

Pour nous initier à la programmation orientée objet nous allons utiliser un langage que vous connaissez bien : Python. Python permet d'utiliser le paradigme impératif (comme nous l'avons fait jusqu'à présent), mais il permet aussi d'utiliser le paradigme objet. Il est même possible, comme nous le verrons plus loin, d'utiliser les 2 paradigmes dans un même programme.

La création d'une classe en python commence toujours par le mot class. Ensuite toutes les instructions de la classe seront indentées :

```
class LeNomDeMaClasse:
  #instructions de la classe
#La définition de la classe est terminée.
```

La classe est une espèce de moule (nous reviendrons plus tard sur cette analogie qui a ses limites), à partir de ce moule nous allons créer des objets (plus exactement nous parlerons d'instances). Par exemple, nous pouvons créer une classe voiture, puis créer différentes instances de cette classe (Peugeot407, Renault Espace,...). Pour créer une de ces instances, la procédure est relativement simple :

```
peugeot407 = Voiture()
```

Cette ligne veut tout simplement dire : "crée un objet (une instance) de la classe Voiture que l'on nommera peugeot407."

Ensuite, rien ne nous empêche de créer une deuxième instance de la classe Voiture :

```
renaultEspace = Voiture()
```

Nous rencontrons ici la limite de notre analogie avec le moule. En effet 2 objets fabriqués avec le même moule seront (définitivement) identiques, alors qu'ici nos 2 instances pourront évoluer différemment.

Pour développer toutes ces notions (et d'autres), nous allons écrire un premier programme :

Nous allons commencer par écrire une classe Personnage (qui sera dans un premier temps une coquille vide) et, à partir de cette classe créer 2 instances : bilbo et gollum :

```
class Personnage:
  pass
gollum = Personnage()
bilbo = Personnage()
```

Pour l'instant, notre classe ne sert à rien et nos instances d'objet ne peuvent rien faire. Comme il n'est pas possible de créer une classe totalement vide, nous avons utilisé l'instruction pass qui ne fait rien. Ensuite nous avons créé 2 instances de la classe Personnage : gollum et bilbo.

Comme expliqué précédemment, une instance de classe possède des attributs et des méthodes. Commençons par les attributs :

Un attribut possède une valeur (un peu comme une variable). Nous allons associer un attribut vie à notre classe Personnage (chaque instance aura un attribut vie, quand la valeur de vie deviendra nulle, le personnage sera mort !)

Ces attributs s'utilisent comme des variables, l'attribut vie pour bilbo sera noté :

```
bilbo.vie
```

de la même façon l'attribut vie de l'instance gollum sera noté :

```
gollum.vie
```

Considérons maintenant le programme suivant :

```
class Personnage:
  pass
gollum=Personnage()
gollum.vie=20
bilbo=Personnage()
bilbo.vie=20
```

Comme pour une variable il est possible d'utiliser la console Python pour afficher la valeur référencée par un attribut. Il suffit de taper dans la console gollum.vie ou bilbo.vie (sans bien sûr avoir oublié d'exécuter le programme au préalable.). Si nous tapons dans la console *gollum.vie* nous aurons 20 comme réponse, même chose si nous tapons *bilbo.vie*

Cette façon de faire n'est pas très "propre" et n'est pas une bonne pratique

En effet, nous ne respectons pas un principe de base de la POO : l'encapsulation

Il ne faut pas oublier que notre classe doit être "enfermée dans une caisse" pour que l'utilisateur puisse l'utiliser facilement sans se préoccuper de ce qui se passe à l'intérieur, or, ici, ce n'est pas vraiment le cas.

En effet, les attributs (gollum.vie et bilbo.vie), font partie de la classe et devraient donc être enfermés dans la "caisse" !

Pour résoudre ce problème, nous allons définir les attributs, dans la classe, à l'aide d'une méthode (une méthode est une fonction définie dans une classe) d'initialisation des attributs.

Cette méthode est définie dans le code source par la ligne :

```
def __init__ (self)
```

La méthode *init* est automatiquement exécutée au moment de la création d'une instance. Le mot self est obligatoirement le premier argument d'une méthode (nous reviendrons ci-dessous sur ce mot self)

Nous retrouvons ce mot self lors de la définition des attributs. La définition des attributs sera de la forme :

```
self.vie=20
```

Le mot self représente l'instance. Quand vous définissez une instance de classe (bilbo ou gollum) le nom de votre instance va remplacer le mot self.

Dans le code source, nous allons avoir :

```
class Personnage:
  def __init__ (self):
    self.vie=20
```

Ensuite lors de la création de l'instance gollum, python va automatiquement remplacer self par gollum et ainsi créer un attribut gollum.vie qui aura pour valeur de départ la valeur donnée à self.vie dans la méthode *init*

Il se passera exactement la même chose au moment de la création de l'instance bilbo, on aura automatiquement la création de l'attribut bilbo.vie.

Si nous saisissons le programme suivant :

```
class Personnage:
  def __init__(self):
    self.vie=20
gollum=Personnage()
bilbo=Personnage()
```

et que nous tapons dans la console Python *gollum.vie*, nous obtiendrons bien 20.

Au moment de la création de l'instance gollum, on passe comme argument le nombre de vies (gollum=Personnage (20)). Ce nombre de vies est attribué au premier argument de la méthode **init** , la variable nbreDeVie (nbreDeVie n'est pas tout à fait le premier argument de

la méthode **init** puisque devant il y a self, mais bon, self étant obligatoire, nous pouvons dire que nbreDeVie est le premier argument non obligatoire).

N.B. Je parle bien de variable pour nbreDeVie (car ce n'est pas un attribut de la classe personnage puisqu'elle ne commence pas par self).

Nous pouvons passer plusieurs arguments à la méthode **init** (comme pour n'importe quelle fonction).

Nous allons créer 2 nouvelles méthodes :

- Une méthode qui enlèvera un point de vie au personnage blessé
- Une méthode qui renverra le nombre de vies restantes

Intéressons-nous à ce programme :

```
class Personnage:
  def __init__(self, nbreDeVie):
    self.vie=nbreDeVie
  def donneEtat (self):
    return self.vie
  def perdVie (self):
    self.vie=self.vie-1
gollum = Personnage(20)
bilbo = Personnage(15)
```

si dans la console Python vous tapez successivement :

- *gollum.donneEtat()* vous allez obtenir 20
- *bilbo.donneEtat()* vous allez obtenir 15
- gollum.perdVie()
- gollum.donneEtat() vous allez obtenir 19
- bilbo.perdVie()
- bilbo.donneEtat() vous allez obtenir 14

Vous avez sans doute remarqué que lors de "l'utilisation" des instances biblo et gollum, nous avons uniquement utilisé des méthodes et nous n'avons plus directement utilisé des attributs (plus de "gollum.vie"). Il est important de savoir qu'en dehors de la classe l'utilisation des attributs est une mauvaise pratique en programmation orientée objet : les attributs doivent rester "à l'intérieur" de la classe, l'utilisateur de la classe ne doit pas les

utiliser directement. Il peut les manipuler, mais uniquement par l'intermédiaire d'une méthode (la méthode self.perdVie() permet de manipuler l'attribut self.vie)

Pour l'instant nous avons utilisé les méthodes uniquement en tapant des instructions dans la console, il est évidemment possible d'utiliser ces méthodes directement dans votre programme :

```
class Personnage:
  def __init__(self, nbreDeVie):
    self.vie=nbreDeVie
  def donneEtat (self):
    return self.vie
  def perdVie (self):
    self.vie=self.vie-1

bilbo = Personnage(15)
bilbo.perdVie()
point=bilbo.donneEtat()
```

Après l'exécution du programme ci-dessus, la variable *point* aura pour valeur 14

Selon le type d'attaque subit, le personnage peut perdre plus ou moins de points de vie. Pour tenir compte de cet élément, il est possible d'ajouter un paramètre à la méthode perdVie :

```
class Personnage:
  def __init__(self, nbreDeVie):
    self.vie=nbreDeVie
  def donneEtat (self):
    return self.vie
  def perdVie (self,nbPoint):
    self.vie=self.vie-nbPoint
bilbo = Personnage(15)
bilbo.perdVie(2)
point=bilbo.donneEtat()
```

Après l'exécution du programme ci-dessus, la variable *point* aura pour valeur 13

Il est possible d'ajouter une part d'aléatoire dans la méthode perdVie :

```
import random
class Personnage:
  def __init__(self, nbreDeVie):
    self.vie=nbreDeVie
  def donneEtat (self):
    return self.vie
```

```
  def perdVie (self):
    if random.random()>0.5:
      nbPoint = 1
    else :
      nbPoint = 2
    self.vie=self.vie-nbPoint
bilbo = Personnage(15)
bilbo.perdVie()
point=bilbo.donneEtat()
```

N.B : `random.random()` renvoie une valeur aléatoire comprise entre 0 et 1

Comme vous l'avez remarqué, il est possible d'utiliser une instruction conditionnelle (if / else) dans une méthode. Il est donc possible d'utiliser dans le même programme le paradigme objet et le paradigme impératif.

Il est maintenant possible d'organiser un combat virtuel entre nos 2 personnages grâce à la classe *Personnage* que nous venons de créer :

```
import random

class Personnage:
  def __init__(self, nbreDeVie):
    self.vie=nbreDeVie
  def donneEtat (self):
    return self.vie
  def perdVie (self):
    if random.random()>0.5:
      nbPoint = 1
    else :
      nbPoint = 2
    self.vie=self.vie-nbPoint
def game():
  bilbo = Personnage(20)
  gollum = Personnage(20)
  while bilbo.donneEtat()>0 and gollum.donneEtat()>0 :
    bilbo.perdVie()
    gollum.perdVie()
  if bilbo.donneEtat()<=0 and gollum.donneEtat()>0:
    msg = f"Gollum est vainqueur, il lui reste encore {gollum.donneEtat()} points
  elif gollum.donneEtat()<=0 and bilbo.donneEtat()>0:
    msg = f"Bilbo est vainqueur, il lui reste encore {bilbo.donneEtat()} points a
  else :
    msg = "Les deux combattants sont morts en même temps"
  return msg
```

Nous avons encore ici la démonstration qu'il est possible d'utiliser le paradigme objet et le paradigme impératif dans un même programme.

4) conclusion

Il est important de bien comprendre qu'un programmeur doit maitriser plusieurs paradigmes de programmation (impératif, objet ou encore fonctionnelle). En effet, il sera plus facile d'utiliser le paradigme objet dans certains cas alors que dans d'autres situations, l'utilisation du paradigme fonctionnel sera préférable. Être capable de choisir le "bon" paradigme en fonction des situations fait partie du bagage de tout bon programmeur.

Il est aussi important de bien comprendre que la frontière entre ces différents paradigmes est parfois floue, par exemple on utilise très souvent de l'impératif en programmation orientée objet.

activité 14.1

Soit la classe *Personnage* suivante :

```
class Personnage:
    def __init__(self, nbreDeVie):
        self.vie=nbreDeVie
    def donneEtat (self):
        return self.vie
    def perdVie (self,nbPoint):
        self.vie=self.vie-nbPoint
```

Ajoutez une méthode *soigne* qui permettra d'augmenter l'attribut *vie* d'une valeur *nbr* (*nbr* sera un paramètre de la méthode *soigne*).

Testez cette méthode en saisissant dans la console Python les instructions suivantes (les unes après les autres) :

- toto = Personnage(15)
- toto.donneEtat()
- toto.perdVie(2)
- toto.soigne(3)
- toto.donneEtat()

activité 14.2

Écrivez une classe *Voiture* qui aura un attribut *vitesse* et 3 méthodes :

- une méthode *accelere* qui permettra d'incrémenter l'attribut vitesse d'une unité
- une méthode *freine* qui permettra de diminuer la vitesse
- une méthode *getVitesse* qui renverra la valeur de la vitesse

activité 14.3

À partir de la classe créée à l'activité 14.2, écrivez un programme qui permettra d'atteindre la vitesse de 3 km/h, d'afficher cette vitesse dans la console puis de freiner jusqu'à l'arrêt complet du véhicule.

activité 14.4

Le but de cette longue activité est d'implémenter en Python les algorithmes sur les arbres binaires précédemment étudiés. Il sera donc sans doute nécessaire de reprendre ce qui a été vu sur la structure de données "arbre" et sur "les algorithmes sur les arbres binaires".

Comme nous l'avons déjà dit, Python ne propose pas de structure de données permettant d'implémenter directement les arbres binaires. Il va donc être nécessaire de créer cette structure. Pour programmer ce type de structure, nous allons utiliser le paradigme objet.

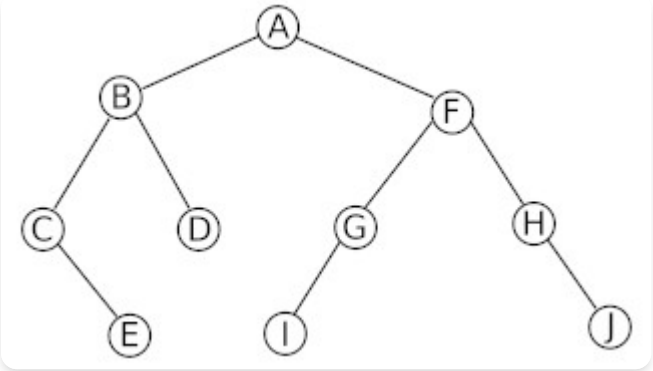
Vous trouverez ci-dessous la classe *ArbreBinaire* qui va nous permettre d'implémenter des arbres binaires.

```
class ArbreBinaire:
    def __init__(self, valeur):
        self.valeur = valeur
        self.enfant_gauche = None
        self.enfant_droit = None
    def insert_gauche(self, valeur):
        if self.enfant_gauche == None:
            self.enfant_gauche = ArbreBinaire(valeur)
        else:
            new_node = ArbreBinaire(valeur)
            new_node.enfant_gauche = self.enfant_gauche
            self.enfant_gauche = new_node
    def insert_droit(self, valeur):
        if self.enfant_droit == None:
            self.enfant_droit = ArbreBinaire(valeur)
        else:
            new_node = ArbreBinaire(valeur)
            new_node.enfant_droit = self.enfant_droit
            self.enfant_droit = new_node
    def get_valeur(self):
        return self.valeur
    def get_gauche(self):
        return self.enfant_gauche
    def get_droit(self):
        return self.enfant_droit
```

1-
Étudiez attentivement la classe *ArbreBinaire* (méthodes et attributs). Vous pouvez, par exemple, vous interroger sur l'utilité de toutes les méthodes de cette classe.

Voici un exemple d'utilisation de cette classe pour construire un arbre binaire :

Soit l'arbre binaire suivant (arbre 1) :



Voici le programme qui va permettre de construire cet arbre à l'aide de la classe *ArbreBinaire* :

```
class ArbreBinaire:
    def __init__(self, valeur):
        self.valeur = valeur
        self.enfant_gauche = None
        self.enfant_droit = None
    def insert_gauche(self, valeur):
        if self.enfant_gauche == None:
            self.enfant_gauche = ArbreBinaire(valeur)
        else:
            new_node = ArbreBinaire(valeur)
            new_node.enfant_gauche = self.enfant_gauche
            self.enfant_gauche = new_node
    def insert_droit(self, valeur):
        if self.enfant_droit == None:
            self.enfant_droit = ArbreBinaire(valeur)
        else:
            new_node = ArbreBinaire(valeur)
            new_node.enfant_droit = self.enfant_droit
            self.enfant_droit = new_node
    def get_valeur(self):
        return self.valeur
    def get_gauche(self):
        return self.enfant_gauche
    def get_droit(self):
        return self.enfant_droit

########fin de la classe########

######début de la construction de l'arbre binaire############

racine = ArbreBinaire('A')
racine.insert_gauche('B')
racine.insert_droit('F')

b_node = racine.get_gauche()
b_node.insert_gauche('C')
b_node.insert_droit('D')
```

```
f_node = racine.get_droit()
f_node.insert_gauche('G')
f_node.insert_droit('H')

c_node = b_node.get_gauche()
c_node.insert_droit('E')

g_node = f_node.get_gauche()
g_node.insert_gauche('I')

h_node = f_node.get_droit()
h_node.insert_droit('J')

######fin de la construction de l'arbre binaire###########
```

2-
Étudiez attentivement le programme ci-dessus afin de comprendre le principe de “construction d’un arbre binaire”

Il est possible d’afficher un arbre binaire dans la console Python, pour cela, nous allons écrire une fonction “affiche”. Cette fonction renvoie une série de tuples de la forme (valeur,arbre_gauche, arbre_droite), comme “arbre_gauche” et “arbre_droite” seront eux-mêmes affichés sous forme de tuples, on aura donc un affichage qui ressemblera à : (valeur, (valeur_gauche,arbre_gauche_gauche,arbre_gauche_droite), (valeur_droite,arbre_droite_gauche,arbre_droite_droite)), mais comme “arbre_gauche_gauche” sera lui-même représenté par un tuple… Nous allons donc avoir des tuples qui contiendront des tuples qui eux-mêmes contiendront des tuples…

Pour l’arbre binaire défini ci-dessus, on aura :

```
('A', ('B', ('C', None, ('E', None, None)), ('D', None, None)), ('F', ('G', ('I',
```

Voici le programme augmenté de la fonction *affiche* :

```
class ArbreBinaire:
    def __init__(self, valeur):
        self.valeur = valeur
        self.enfant_gauche = None
        self.enfant_droit = None
    def insert_gauche(self, valeur):
        if self.enfant_gauche == None:
            self.enfant_gauche = ArbreBinaire(valeur)
        else:
            new_node = ArbreBinaire(valeur)
            new_node.enfant_gauche = self.enfant_gauche
            self.enfant_gauche = new_node
```

```
    def insert_droit(self, valeur):
        if self.enfant_droit == None:
            self.enfant_droit = ArbreBinaire(valeur)
        else:
            new_node = ArbreBinaire(valeur)
            new_node.enfant_droit = self.enfant_droit
            self.enfant_droit = new_node
    def get_valeur(self):
        return self.valeur
    def get_gauche(self):
        return self.enfant_gauche
    def get_droit(self):
        return self.enfant_droit

#######fin de la classe########

######début de la construction de l'arbre binaire###########

racine = ArbreBinaire('A')
racine.insert_gauche('B')
racine.insert_droit('F')

b_node = racine.get_gauche()
b_node.insert_gauche('C')
b_node.insert_droit('D')

f_node = racine.get_droit()
f_node.insert_gauche('G')
f_node.insert_droit('H')

c_node = b_node.get_gauche()
c_node.insert_droit('E')

g_node = f_node.get_gauche()
g_node.insert_gauche('I')

h_node = f_node.get_droit()
h_node.insert_droit('J')

######fin de la construction de l'arbre binaire###########

def affiche(T):
    if T != None:
        return (T.get_valeur(),affiche(T.get_gauche()),affiche(T.get_droit()))
```

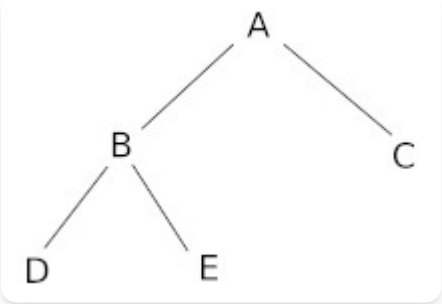
3-

Vérifiez que “affiche(racine)” renvoie bien :

```
('A', ('B', ('C', None, ('E', None, None)), ('D', None, None)), ('F', ('G', ('I',
```

N.B : la fonction *affiche* n'a pas une importance fondamentale, elle sert uniquement à vérifier que les arbres programmés sont bien corrects.

4-
Programmez à l'aide de la classe *ArbreBinaire*, l'arbre binaire suivant (arbre 2) :



Vérifiez votre programme à l'aide de la fonction "affiche"

Vous allez maintenant pouvoir commencer à travailler sur l'implémentation des algorithmes sur les arbres binaires :

5-
Programmez la fonction *hauteur* qui prend un arbre binaire T en paramètre et renvoie la hauteur de T

Testez votre fonction en utilisant l'arbre vu plus haut (schéma "arbre 1").

6-
Programmez la fonction *taille* qui prend un arbre binaire T en paramètre et renvoie la taille de T
Testez votre fonction en utilisant l'arbre vu plus haut (schéma "arbre 1").

7-
Programmez la fonction *parcours_infixe* qui prend un arbre binaire T en paramètre et qui permet d'obtenir le parcours infixe de l'arbre T

Testez votre fonction en utilisant l'arbre vu plus haut (schéma "arbre 1").

8-
Programmez la fonction *parcours_prefixe* qui prend un arbre binaire T en paramètre et qui permet d'obtenir le parcours préfixe de l'arbre T

Testez votre fonction en utilisant l'arbre vu plus haut (schéma "arbre 1").

9-
Programmez la fonction *parcours_suffixe* qui prend un arbre binaire T en paramètre et qui permet d'obtenir le parcours suffixe de l'arbre T

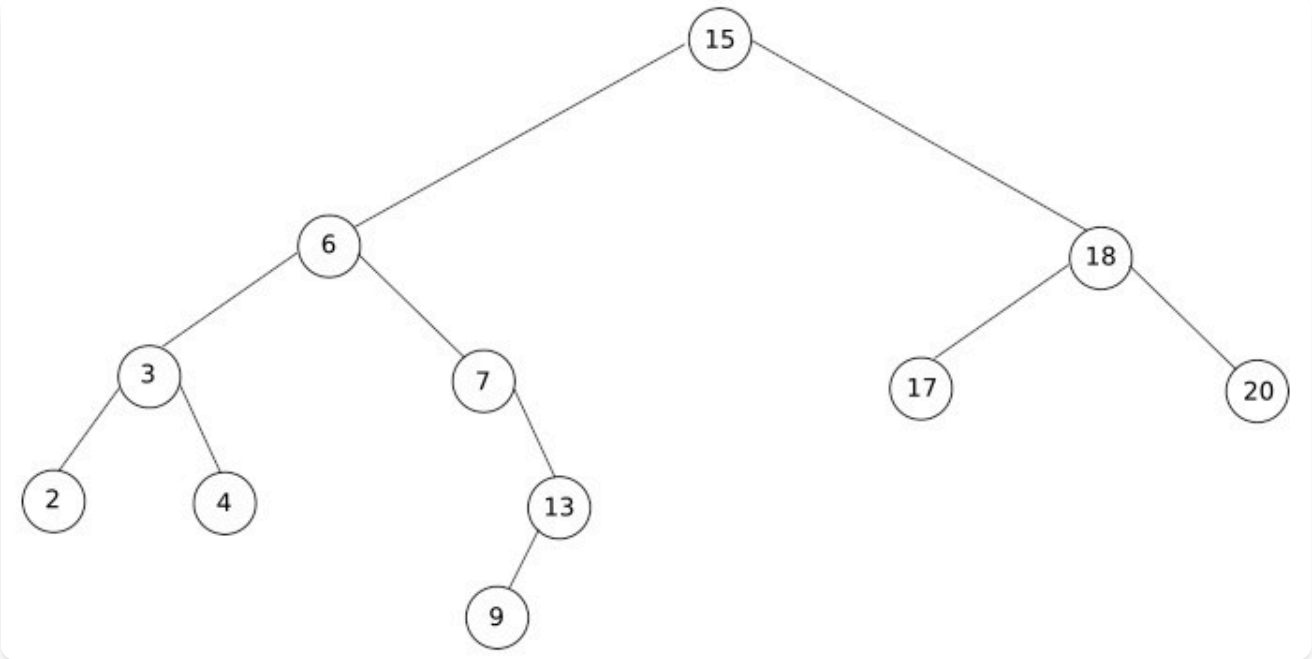
Testez votre fonction en utilisant l'arbre vu plus haut (schéma "arbre 1").

10-
Programmez la fonction *parcours_largeur* qui prend un arbre binaire T en paramètre et qui permet d'obtenir le parcours en largeur de l'arbre T

Testez votre fonction en utilisant l'arbre vu plus haut (schéma "arbre 1").

Nous allons maintenant travailler sur les arbres binaires de recherche.

11-
Programmez, à l'aide de la classe *ArbreBinaire*, l'arbre binaire de recherche ci-dessous (arbre 3) :



Vérifiez votre réponse à l'aide de la fonction *affichage*

12-
Afin de vérifier que l'arbre binaire "Arbre 3" est bien un arbre binaire de recherche, utilisez la fonction *parcours_infixe* programmée dans le "projet 3.7".

13-
Programmez la fonction *arbre_recherche* qui prend un arbre binaire T et un entier k en paramètres et qui renvoie True si k appartient à T et False dans le cas contraire
Testez votre fonction en utilisant l'arbre vu plus haut (schéma "arbre 3") avec k = 13 et k = 16.

14-
Programmez la fonction *arbre_recherche_ite* (version itérative de la fonction *arbre_recherche*) qui prend un arbre binaire T et un entier k en paramètres et qui renvoie True si k appartient à T et False dans le cas contraire

Testez votre fonction en utilisant l'arbre vu plus haut (schéma "arbre 3") avec k = 13 et k = 16.

15-
Programmez la fonction "arbre_insertion" qui prend T (un arbre binaire) et y (un objet de type *ArbreBinaire*) en paramètres et qui insert y dans T

Testez votre fonction en utilisant l'arbre vu plus haut (schéma "arbre 3") avec y.valeur = 16.

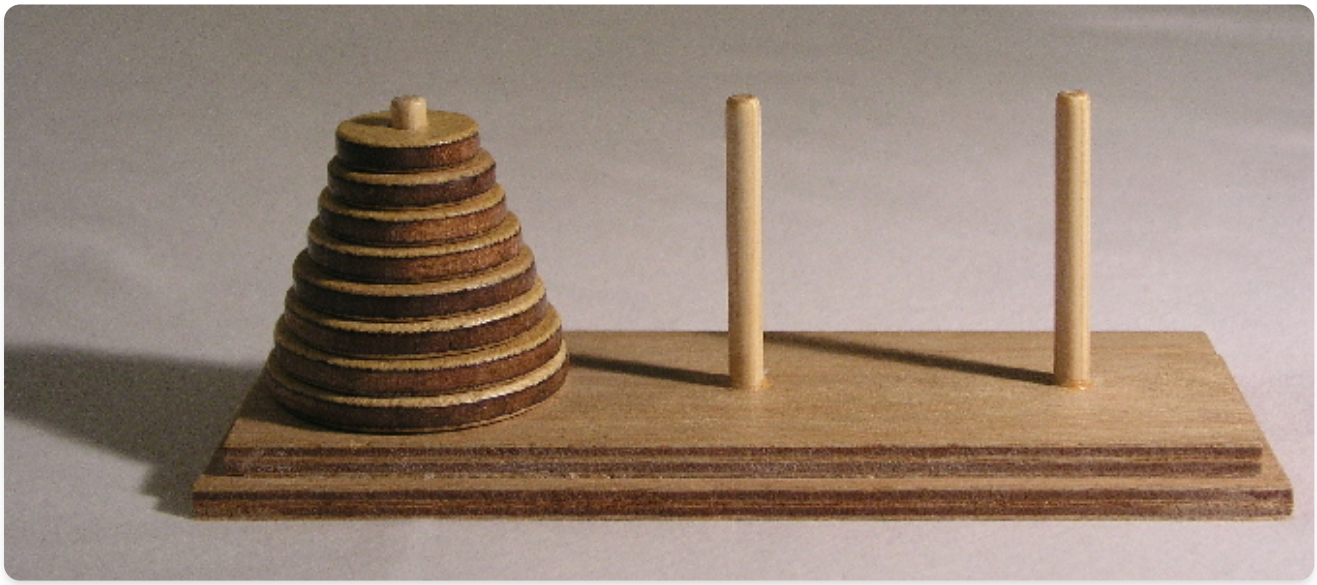
activité 14.5*

Cette activité porte sur le problème des Tours de Hanoi. Voici un extrait de l'article Wikipédia consacré aux Tours de Hanoi (https://fr.wikipedia.org/wiki/Tours_de_Hano%C3%AF) :

Les tours de Hanoï (originellement, la tour d'Hanoïa) sont un jeu de réflexion imaginé par le mathématicien français Édouard Lucas, et consistant à déplacer des disques de diamètres différents d'une tour de « départ » à une tour d'« arrivée » en passant par une tour « intermédiaire », et ceci en un minimum de coups, tout en respectant les règles suivantes :

- on ne peut déplacer plus d'un disque à la fois ;
- on ne peut placer un disque que sur un autre disque plus grand que lui ou sur un emplacement vide.

On suppose que cette dernière règle est également respectée dans la configuration de départ.



Dans la suite de cette activité nous allons utiliser la classe Tour donnée ci-dessous :

```
class Tour:
    def __init__(self, nom, n = 0):
        self.s=[]
        self.nom = nom
```

```
        for i in range (n, 0, -1):
            self.s.append(i)
    def empile(self, d):
        assert len(self.s) == 0 or self.s[-1] > d, 'Mouvement interdit'
        self.s.append(d)
        print(f"Ajout du disque {d} sur la tour ", self.nom)

    def depile(self):
        assert len(self.s) != 0, 'Impossible, la tour est vide'
        d = self.s.pop()
        print(f"Retire le disque {d} de la tour ",self.nom)
        return d

    def affiche(self):
        if len(self.s)==0:
            print(f"la tour {self.nom} est vide")
        else :
            print (f"Tour {self.nom}")
            for d in reversed(self.s):
                print("| ",d," |")
```

1-
Après avoir étudié attentivement la classe Tour, écrivez les instructions Python permettant de créer 3 instances de la classe Tour (une instance permettant de modéliser une tour du jeu). Une tour sera créée avec 2 disques (cette tour sera nommée “A”), les 2 autres tours seront au départ vide (tour “B” et tour “C”). On notera que chaque disque est identifié par un entier (cet entier représente le diamètre du disque correspondant).

2-
Écrivez une fonction *mouvement*, cette fonction prendra 2 paramètres *t1* et *t2*, tous les deux de type Tour (instance de Tour). Le but de cette fonction est de permettre le passage d’un disque de la tour *t1* vers la tour *t2*.

3-
Écrivez la suite d’instructions permettant de résoudre le jeu avec 2 disques (la tour “A” sera la tour de “départ”, la tour “B” sera la tour “intermédiaire” et la tour “C” sera la tour “arrivée”). Vous vérifierez votre réponse en affichant le contenu de la tour “C” après l’exécution de votre programme.

4-
Écrivez une fonction récursive *resoudre* permettant de résoudre le jeu “Tours de Hanoi” dans tous les cas (avec n disques). Cette fonction prendra 4 paramètres : un entier représentant le nombre de disques et trois instances de la classe Tour (représentant la tour “départ”, la tour “arrivée” et la tour “intermédiaire”).

Deux conseils :

- réfléchissez bien au cas de base de votre fonction récursive
- inspirez-vous de ce que vous avez fait dans la question 3

activité 14.6*

Vous avez déjà eu l'occasion de travailler sur les listes dans le chapitre 5. Dans ce chapitre, nous avons utilisé les tuples pour réaliser l'implémentation de cette structure de données. Le but de cette activité est de réaliser une autre implémentation des listes, non plus cette fois en utilisant des tuples mais des listes chaînées (revoir aussi le chapitre 5 pour les listes chaînées).

Rappels sur les listes chaînées :

Dans une liste chaînée, à chaque élément de la liste on associe 2 cases mémoire : la première case contient l'élément et la deuxième contient l'adresse mémoire de l'élément suivant.

Pour dans un premier temps implémenter les listes chaînées en Python, nous allons utiliser 2 classes :

- la classe Node
- la classe LinkedList

```
class Node:
    def __init__(self,value):
        self.value = value
        self.next = None
    def __str__(self):
        return str(self.value)
```

La classe Node permet d'implémenter un élément de la liste chaînée (les 2 cases : celle qui contient la valeur et celle qui pointe vers l'élément suivant). Cette classe Node possède deux attributs :

- l'attribut self.value qui correspond à la case qui contient la valeur (première case)
- l'attribut self.next qui correspond à la case qui "pointe" vers la valeur suivante (deuxième case)

Ne vous souciez pas de la méthode *_str_* qui permet juste d'afficher la valeur d'un élément.

```
class LinkedList:
    def __init__(self):
        self.head=None
```

```
        self.tail=None
    def __iter__(self):
        cur_node = self.head
        while cur_node:
            yield cur_node
            cur_node = cur_node.next
    def copy(self):
        l = LinkedList()
        node = self.head
        while node is not None:
            new_node = Node(node.value)
            if l.head is None:
                l.head = new_node
                l.tail = new_node
            else:
                l.tail.next = new_node
                l.tail = l.tail.next
            node = node.next
        return l
```

La classe LinkedList permet d'implémenter une liste chaînée. Cette classe LinkedList possède deux attributs :

- l'attribut self.head correspond au premier élément de la liste chaînée.
- l'attribut self.tail correspond au dernier élément de la liste chaînée.

Ne vous souciez pas de la méthode *_iter_* qui permet de parcourir une liste à l'aide d'une boucle "for" (vous n'aurez pas à l'utiliser directement) et de la méthode "copy" qui permet de réaliser la copie d'une liste.

Le but de cette activité est d'écrire les fonctions qui ont déjà été écrites dans le chapitre 5, mais en utilisant les classes Nodes et LinkedList à la place des tuples :

- fonction "newList" : permet d'obtenir une liste vide (la fonction ne prend aucun paramètre et renvoie une liste vide)
- fonction "showList" : permet d'afficher une liste (la fonction prend en paramètre une liste et renvoie une chaîne de caractères)
- fonction "isEmpty" : permet de tester si une liste est vide (la fonction prend en paramètre une liste et renvoie True si la liste est vide et False dans le cas contraire)
- fonction "car" : permet d'obtenir le dernier élément ajouté à la liste (la fonction prend en paramètre une liste et renvoie un entier)
- fonction "cdr" : permet d'obtenir une liste contenant tous les éléments d'une liste à l'exception du dernier élément ajouté (la fonction prend en paramètre une liste et renvoie une liste)

- fonction “cons” : permet de construire une liste à partir d’un élément et d’un autre liste (la fonction prend en paramètres une valeur et une liste et renvoie une liste)

On donne ci-dessous les fonctions à compléter. Les fonctions “newList”, “isEmpty” et “showList” sont fournies.

À noter les lignes “l1 = l.copy()” dans la fonction cons et la fonction cdr. Ces 2 lignes permettent de créer une copie de la liste qui a été passée en paramètre afin d’éviter de modifier cette même liste. Un simple “l1 = l” pour créer la copie ne suffirait pas (une modification de l1 entraînerait une modification de l), il est donc nécessaire d’utiliser la méthode “copy” de la classe LinkedList.

```
 def newList():
    return LinkedList()
def showList(l):
    li = [str(x) for x in l]
    return " - ".join(li)
def isEmpty(l):
    return l.head is None
def cons(v,l):
    l1 = l.copy()
    ....
def car(l):
    ....
def cdr(l):
    l1 = l.copy()
    ....
```

Après avoir complété les fonctions ci-dessus, exécutez le programme ci-dessous :

```
l = newList()
l1 = cons(15,cons(12,cons(2,l)))
v = car(l1)
l2 = cdr(l1)
l3 = cons(4,cons(5, l2))
```

Ensuite, tapez successivement dans la console :

- v
- showList(l1)
- showList(l2)
- showList(l3)

Voici les résultats que vous devriez obtenir :

```
>>> v
15
>>> showList(l1)
'2 - 12 - 15'
>>> showList(l2)
'2 - 12'
>>> showList(l3)
'2 - 12 - 5 - 4'
```

activité 14.7*

Vous avez déjà eu l'occasion de travailler sur les piles dans le chapitre 5. Dans ce chapitre, nous avons utilisé les tableaux (listes Python) pour réaliser l'implémentation de cette structure de données. Le but de cette activité est de réaliser une autre implémentation des piles, non plus cette fois en utilisant des tableaux, mais des listes chaînées (revoir aussi le chapitre 5 pour les listes chaînées).

Rappels sur les listes chaînées :

Dans une liste chaînée, à chaque élément de la liste on associe 2 cases mémoire : la première case contient l'élément et la deuxième contient l'adresse mémoire de l'élément suivant.

Pour dans un premier temps implémenter les listes chaînées en Python, nous allons utiliser 2 classes :

- la classe Node
- la classe LinkedList

```
class Node:
    def __init__(self,value):
        self.value = value
        self.next = None
    def __str__(self):
        return str(self.value)
```

La classe Node permet d'implémenter un élément de la liste chaînée (les 2 cases : celle qui contient la valeur et celle qui pointe vers l'élément suivant). Cette classe Node possède deux attributs :

- l'attribut self.value qui correspond à la case qui contient la valeur (première case)
- l'attribut self.next qui correspond à la case qui "pointe" vers la valeur suivante (deuxième case)

Ne vous souciez pas de la méthode *_str_* qui permet juste d'afficher la valeur d'un élément.

```
class LinkedList:
    def __init__(self):
        self.head=None
        self.tail=None
    def __iter__(self):
        cur_node = self.head
        while cur_node:
            yield cur_node
            cur_node = cur_node.next
```

La classe LinkedList permet d'implémenter une liste chaînée. Cette classe LinkedList possède deux attributs :

- l'attribut self.head correspond au premier élément de la liste chaînée.
- l'attribut self.tail correspond au dernier élément de la liste chaînée.

Ne vous souciez pas de la méthode *_iter_* qui permet de parcourir une liste à l'aide d'une boucle "for" (vous n'aurez pas à l'utiliser directement)

Le but de cette activité est d'écrire une classe "Stack" ("Pile" en français). Cette classe vous permettra d'implémenter la structure de données pile.

La classe "Stack" utilisera les classes "LinkedList" et "Node".

```
class Stack:
    def __init__(self):
        self.ll = LinkedList()
    def isEmpty(self):
        return self.ll.head is None
    def show(self):
        print("")
        if self.isEmpty():
            print("La pile est vide")
        else :
            for n in self.ll:
                print ("|",n.value,"|")
            print("-----")
    def push(self,v):
        ......
    def pop(self):
        ......
```

Comme vous pouvez le constater ci-dessus, les méthodes "*_init_*", "isEmpty" (renvoie True si la pile est vide et False dans le cas contraire), "show" (permet d'afficher la pile) sont données.

Il vous reste donc à implémenter la méthode “push” (permet de placer un élément v au sommet de la pile) et la méthode “pop” (permet de “dépiler” la pile, cette méthode renvoie la valeur qui vient d’être “dépilée”).

Pour vérifier la correction des méthodes “push” et “pop” que vous aurez écrites, vous pourrez exécuter le programme suivant :

```
p = Stack()
p.push(5)
p.push(8)
p.show()
v=p.pop()
print("valeur renvoyée par pop : ",v)
p.show()
v = p.pop()
print("valeur renvoyée par pop : ",v)
p.show()
```

Vous devriez alors obtenir le résultat suivant :

```
| 8 |
| 5 |
-----
valeur renvoyée par pop :  8

| 5 |
-----
valeur renvoyée par pop :  5

La pile est vide
```

activité 14.8*

Vous avez déjà eu l’occasion de travailler sur les files dans le chapitre 5. Dans ce chapitre, nous avons utilisé les tableaux (listes Python) pour réaliser l’implémentation de cette structure de données. Le but de cette activité est de réaliser une autre implémentation des files, non plus cette fois en utilisant des tableaux, mais des listes chaînées (revoir aussi le chapitre 5 pour les listes chaînées).

Rappels sur les listes chaînées :

Dans une liste chaînée, à chaque élément de la liste on associe 2 cases mémoire : la première case contient l’élément et la deuxième contient l’adresse mémoire de l’élément suivant.

Pour dans un premier temps implémenter les listes chaînées en Python, nous allons utiliser 2 classes :

- la classe Node
- la classe LinkedList

```
class Node:
    def __init__(self,value):
        self.value = value
        self.next = None
    def __str__(self):
        return str(self.value)
```

La classe Node permet d'implémenter un élément de la liste chaînée (les 2 cases : celle qui contient la valeur et celle qui pointe vers l'élément suivant). Cette classe Node possède deux attributs :

- l'attribut self.value qui correspond à la case qui contient la valeur (première case)
- l'attribut self.next qui correspond à la case qui "pointe" vers la valeur suivante (deuxième case)

Ne vous souciez pas de la méthode *_str_* qui permet juste d'afficher la valeur d'un élément.

```
class LinkedList:
    def __init__(self):
        self.head=None
        self.tail=None
    def __iter__(self):
        cur_node = self.head
        while cur_node:
            yield cur_node
            cur_node = cur_node.next
```

La classe LinkedList permet d'implémenter une liste chaînée. Cette classe LinkedList possède deux attributs :

- l'attribut self.head correspond au premier élément de la liste chaînée.
- l'attribut self.tail correspond au dernier élément de la liste chaînée.

Ne vous souciez pas de la méthode *_iter_* qui permet de parcourir une liste à l'aide d'une boucle "for" (vous n'aurez pas à l'utiliser directement)

Le but de cette activité est d'écrire une classe "Queue" ("File" en français). Cette classe vous permettra d'implémenter la structure de données file.

La classe "Queue" utilisera les classes "LinkedList" et "Node".

```
class Queue:
    def __init__(self):
        self.ll = LinkedList()
    def isEmpty(self):
        return self.ll.head is None
    def show(self):
        if self.isEmpty() :
            print("la file est vide")
        else :
            l = [str(x) for x in self.ll]
            print(" - ".join(l))
    def enQueue(self,v):
        ......
    def deQueue(self):
        ......
```

Comme vous pouvez le constater ci-dessus, les méthodes "*_init_*", "isEmpty" (renvoie True si la file est vide et False dans le cas contraire), "show" (permet d'afficher la file) sont données. Il vous reste donc à implémenter la méthode "enQueue" (permet de placer un élément v dans la file) et la méthode "deQueue" (permet de "défiler" la file, cette méthode renvoie la valeur qui vient d'être "défilée").

Pour vérifier la correction des méthodes "enQueue" et "deQueue" que vous aurez écrites, vous pourrez exécuter le programme suivant :

```
q = Queue()
q.enQueue(5)
q.enQueue(8)
q.show()
v = q.deQueue()
print("valeur renvoyée par deQueue : ",v)
q.show()
v = q.deQueue()
print("valeur renvoyée par deQueue : ",v)
q.show()
```

Vous devriez alors obtenir le résultat suivant :

```
5 - 8
valeur renvoyée par deQueue :  5
8
```

```
valeur renvoyée par deQueue :  8
la file est vide
```

exercices du bac

- Sujet 3 2021 Exercice 1
- Sujet 4 2021 Exercice 2
- Sujet 5 2021 Exercice 3
- Sujet 6 2021 Exercice 1
- Sujet 10 2021 Exercice 4
- Sujet 2 2022 Exercice 5
- Sujet 3 2022 Exercice 1
- Sujet 5 2022 Exercice 1
- Sujet 6 2022 Exercice 2
- Sujet 7 2022 Exercice 4
- Sujet 8 2022 Exercice 4
- Sujet 10 2022 Exercice 2
- Sujet 12 2022 Exercice 3
- Sujet 14 2022 Exercice 2

Ce qu'il faut savoir

paradigme fonctionnel

- la programmation fonctionnelle est un paradigme de programmation comme la programmation impérative ou la programmation orientée objet.
- le paradigme fonctionnel repose sur l'utilisation de fonction
- le paradigme fonctionnel cherche à éviter au maximum les effets de bord => en programmation fonctionnelle, on s'efforce de coder des fonctions qui ne modifient pas l'état courant des variables globales.
- Les fonctions utilisées en programmation fonctionnelle sont parfois appelées "fonction pure" : le résultat renvoyé par une fonction pure doit uniquement dépendre des paramètres passés à la fonction et pas des valeurs externes à la fonction

paradigme objet

La programmation orientée objet (poo) est un paradigme de programmation qui repose sur la notion de classe, la notion d'attribut et la notion de méthode (la poo repose aussi sur les notions d'héritage et de polymorphisme, mais ces 2 notions ne sont pas au programme de NSI). En poo on travaille sur des objets (des instances plus exactement), chaque instance est créée à partir d'un "moule" : la classe. Les attributs représentent l'état d'un objet alors que les méthodes représentent le comportement d'un objet.

Ce qu'il faut savoir faire

paradigme fonctionnel

être capable d'écrire un programme simple en Python qui respecte le paradigme fonctionnel

paradigme objet

- vous devez être capable d'analyser et comprendre un programme Python simple qui utilise le paradigme objet
- vous devez être capable d'écrire un programme Python simple qui utilise le paradigme objet