Numérique et science informatique

Lycée Hoche

année scolaire 2024-2025

Contents

0.1 Classes

En première, on a déjà affaire aux notions de classes et de méthodes. En effet, le type «list Python» régulièrement manipulé permet d'illustrer ces notions :

```
>>> ma_liste = [12, 24, 28]
>>> type(ma_liste)
<class 'list'>
```

Ainsi, l'affichage du type d'une variable fait référence à une classe (ici list) et présente le mot-clé correspondant (class). De plus, l'usage du mot méthode (par exemple append) était régulièrement utilisé sans vraiment être expliqué .Par exemple:

```
>>> ma_liste.append(31)
>>> ma_liste
[12, 24, 28, 31]
```

Regardons sur un autre exemple la méthode randint de la bibliothèque random qu'on a déjà été amené à utiliser. Le code suivant :

```
import random
import inspect

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

permet d'accéder au code source de la méthode randint en affichant en console :

```
def randint(self, a, b):
'''Return random integer in range [a, b], including both end points. '''

return self.randrange(a, b+1)
```

On y retrouve la documentation de la méthode ainsi qu'un mot-clé self qui peut surprendre (la méthode randint attendant uniquement deux paramètres a et b). On peut également inspecter toute la bibliothèque random avec print(inspect.getsource(random)) afin d'exhiber les mots-clés class ainsi que __init__ pour évoquer le constructeur.

0.1.1 Présentation des mots clés

Comme observé en inspectant le module random, la création d'une classe se fait à l'aide du mot-clé **class**.Cependant, lors de la création de la classe, on ne connait pas à l'avance le nom des objets qui seront créés. D'où la nécessité de représenter le futur objet créé à l'aide d'un mot-clé self.

```
class Rectangle:
    def __init__(self, longueur, largeur):
        self.longueur = longueur
        self.largeur = largeur
```

Définition

- Le **constructeur** __init__ permet de définir des **attributs** au futur objet créé ainsi que de leur affecter les valeurs désirées.
- Pour créer une **instance** d'une classe (autre expression pour désigner un objet), on fait suivre le nom de la classe de la liste des arguments attendus par __init__

```
rectangle1 = Rectangle(6, 4)
```

Le code présent dans le constructeur est exécuté et le mot-clé **self** dans le constructeur représente l'instance de la classe en cours de création.

On peut ensuite vérifier que notre instance de classe possède bien les deux attributs

```
>>> rectangle1.longueur
6
>>> rectangle1.largeur
4
```

0.1.2 les méthodes de classe

Les méthodes sont définies à la suite, dans la création de la classe :

```
def calcule_aire(self):
     return self.longueur * self.largeur
```

Dans la définition d'une **méthode**, le premier argument, appelé self par convention, représente l'objet sur lequel la méthode est appliquée.

Voici un exemple à l'aide de la console, où l'on retrouve les usages des méthodes rencontrées en première sur les listes.

```
>>> rectangle1.calcule_aire()
24
```

À ce stade, un élément n'a pas été évoqué : qu'en est-il de la modification de la valeur d'un attribut d'une instance de la classe ?En Python, il est tout à fait possible de modifier directement un attribut, par exemple :

```
rectangle1.longueur = 10
```

Ceci dit, notamment pour des raisons de « sécurité » du code, cette pratique est déconseillée. La dernière partie de ce document revient plus en détail sur cette notion d'encapsulation des données.

Voici deux autres exemples de classe:

```
class Chien :
    def __init__(self, nom, age, race):
        self.nom = nom
        self.age = age
        self.race = race

class Personne :
    def __init__(self, prenom, nom, age, chien):
        self.prenom = prenom
        self.nom = nomself.age = age
        self.chien = chien

#Instanciation :création d'objets

chien1 = Chien('Medor',2, 'Berger Allemand')
chien2 = Chien("Brutus", 4, "Chihuahua")
personne1 = Personne('Alain', 'Térieur', 42, chien1)
```

0.1.3 Les méthodes spéciales

Les méthodes spéciales ne sont pas au programme de NSI. Cependant, si dans un premier temps nous avons complété la classe Chien par exemple ainsi :

```
def affiche_informations(self):
   return "Ce chien de la race " + self.race + " a pour nom " + \
    self.nom +   " et pour age " + str(self.age) + " an(s)."
```

Nous pouvons alors afficher les informations relatives à une instance de classe grâce à cette méthode :

```
>>> chien1.affiche_informations()
Ce chien de la race Berger Allemand a pour nom Medor et pour age 2 ans.
```

Il peut être légitime de vouloir obtenir cet affichage avec l'instruction print(chien1), instruction qui, par défaut, renvoie un affichage précisant notamment l'adresse mémoire dans laquelle est stockée la valeur de la variable :

```
>>> print(chien1)
__main__.Chien object at 0x10a98af60>
```

Une méthode est prédéfinie pour définir cela : __str __ et il suffit de la modifier pour obtenir l'affichage voulu. Ainsi, en renommant la méthode affiche _informations en __str__, on obtient exactement le résultat souhaité.

Indépendamment de cette considération d'affichage, il peut être intéressant de présenter dans certains cas des méthodes spéciales comme __eq__ ou __add__ (il ne semble pas pertinent de tester l'égalité ou d'ajouter deux chiens...).
Considérons par exemple une classe Point avec deux instances de cette classe :

```
class Point:
    def __init__(self, abscisse, ordonnee):
        self.abscisse = abscisse
        self.ordonnee = ordonnee
point1 = Point(3,4)
point2 = Point(3,4)
```

L'affichage suivant peut surprendre

```
>>> point1 == point2
False
```

En effet, par défaut, l'opérateur == compare les identifiants et donc ne renvoie ici vrai que si les 2 opérandes de part et d'autre du signe '==' désignent la même instance :

```
>>> point1
 __main__.Point at 0x10a931550
>>>> point2
 __main__.Point at 0x10a931668>
```

Cependant ceci ne correspond pas à la représentation que nous nous faisons de l'égalité de deux points. Pour éviter cet écueil, il suffit d'ajouter à la classe Point la méthode __eq__ qui est appelée en cas de test d'égalité

```
def __eq__(self, autrePoint):
    return self.abscisse == autrePoint.abscisse and
         self.ordonnee == autrePoint.ordonnee
```

Permettant d'obtenir le comportement attendu :

```
>>> point1 == point2
True
```

On peut retrouver un comportement identique avec la somme de deux vecteurs, qui est un nouveau vecteur :

```
class Vecteur:
    def __init__(self, abscisse, ordonnee):
        self.abscisse = abscisse
        self.ordonnee = ordonnee

    def __add__(self, autre_vecteur):
        return Vecteur(self.abscisse + \
         autre_vecteur.abscisse, self.ordonnee + \
         autre_vecteur.ordonnee)

    def __str__(self):

    return "Vecteur de coordonnees ("+str(self.abscisse)+" \
     +"str(self.ordonnee)+")"

v1 = Vecteur(1, 2)
v2 = Vecteur(-3, 4
```

```
>>> v3 = v1 + v2
>>> print(v3)
Vecteur de coordonnées (-2, 6)
```

On retrouve bien un comportement qui rappelle l'exemple donné ci-dessus sur les listes.Enfin, il peut être intéressant de présenter la fonction (bien nommée) isinstance, qui renvoie un booléen. Elle peut être présentée comme une généralisation de la fonction type utilisée sur les types de base Python.

```
>>> isinstance(v3, Vecteur)
True
>>> isinstance(v3, Point)
False
```

Elle peut notamment permettre un peu de programmation défensive, par exemple en précisant dans la méthode __add__ ci-dessus que le second paramètre doit bien être un vecteur :

```
def __add__(self, autre_vecteur):
    assert isinstance(autre_vecteur, Vecteur)
    return Vecteur(self.abscisse + autre_vecteur.abscisse, \
     self.ordonnee + autre_vecteur.ordonnee)
```

0.2 A propos de l'encapsulation

La notion d'encapsulation n'est pas vraiment au programme. Nous en présentons néanmoins le principe.

Il s'agit de ne rendre accessibles et modifiables les attributs d'instance uniquement à l'aide de méthodes. En effet, en préfixant un attribut par un double underscore, celui-ci ne semble plus directement accessible (et donc modifiable, on appelle cela un attribut **privé**) :

```
class Exemple:

    def __init__(self, attribut1):
        self.__attribut1 = attribut1
```

```
>>> objet1 = Exemple(«test»)
 objet1.__attribut1AttributeError: 'Exemple' object has no attribute '__attribut1'
```

Cependant, en inspectant la variable objet1, on se rend compte que Python lui a affecté un attribut _Exemple__attribut1, à la fois accessible et modifiable à l'extérieur de la classe :

```
>>> objet1._Exemple__attribut1 = "modification"
```

En ajoutant par exemple une méthode renvoie_attribut1 (que l'on a pu nommer ici get_attribut1 pour respecter les conventions de la programmation objet, mais celles-ci ne semblent pas nécessaires en Terminale) à notre classe :

```
def renvoie_attribut1(self):
    return self.__attribut1
```

On obtient alors en console :

```
>>> objet1.renvoie_attribut1()
"modification"
```

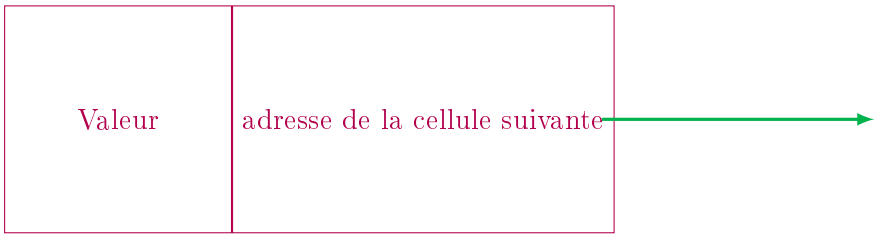
Ainsi, en Python, l'encapsulation ne reste que purement théorique, et un utilisateur d'une classe peut toujours accéder à n'importe quel attribut d'instance (et le modifier).

Le langage Python simplifie l'accès aux attributs (ainsi que leur modification en dehors de la classe), en donnant une souplesse lorsque les classes sont utilisées par exemple dans l'implémentation de structures de données.

Cependant, autant un accès simple aux attributs d'instance n'est pas très gênant, autant laisser l'utilisateur les modifier à sa guise peut être dangereux.

0.3 Implémentation des piles et des files utilisant la programmation orientée objet

Que cela soit pour les listes, les piles ou les files, ces structures sont définies par un ensemble d'éléments ordonnés et reliés entre eux. Pour faciliter les implantations de ces structures en utilisant la programmation orientée objet, nous allons introduire une classe Cellule qui permet de définir un objet contenant deux attributs :
• un attribut valeur définissant la valeur contenue dans cette "cellule" (nombre, texte, un autre objet de nature quelconque...) ;
• un attribut suivant définissant l'adresse d'un autre objet Cellule qui "suit" cet objet (qui est l'objet suivant dans l'ordre de la structure).




```
'''Implantation de la classe Cellule Elle sera utilisée pour les listes, les piles et

class Cellule :


    def __init__(self, valeur = None, suivant = None) :

    self.valeur = valeur
    self.suivant = suivant
```

Chaque valeur de la liste est encapsulé dans un objet Cellule (défini plus haut) qui contient une valeur ainsi que l'adresse de la cellule suivante

Une proposition très minimaliste d'implantation peut être :
On importe la classe "Cellule" que l'on a défini précédemment (et qui doit être dans le même dossier)

```
from Cellule import Cellule
class Liste :
    def __init__(self) :
        '''
        Crée une liste vide.
        L'attribut head est un objet Cellule qui définit la cellule         en tête d
        '''
        self.head = None

    def estVide(self) :
        '''
        Renvoie ``True`` si la liste est vide et ``False`` sinon.
        '''
        return self.head == None

    def insererEnTete(self, element) :
        '''
        Paramètres        ----------
        element : N importe quel type
        Description : L élément à ajouter en tête de la liste
           Ajoute un élément en tête de liste.
        '''
        nouvelle_cellule = Cellule(element, self.head)
        self.head = nouvelle_cellule

    def tete(self) :
        '''
        Renvoie la valeur de l élément en tête de liste.
        '''
        if not(self.estVide()) :
            return self.head.valeur

    def queue(self) :
        '''
         Renvoie la liste privée de son premier élément (queue de la liste)
        '''
        subList = None
        if not(self.estVide()) :
            subList = Liste()
            subList.head = self.head.suivant
        return subList
```

Celle-ci peut être complétée par des méthodes implantant de nouvelles possibilités Par exemple :

- renvoyer la longueur de la liste ;
- accéder au ième élément d'une liste ;
- ajouter un élément à la fin de la liste ;
- rechercher un élément dans une liste en renvoyant "Vrai" si l'élément est présent, "Faux" sinon.

0.4 Implémentation des piles utilisant la programmation objet

Chaque valeur contenue dans la pile est encapsulée dans un objet Cellule (défini plus haut) qui contient une valeur ainsi que l'adresse de la cellule suivante.Un attribut top est attaché à la pile. C'est en fait un objet de type *Cellule* correspondant au sommet.

À partir du sommet, on peut donc accéder à chaque valeur contenue dans la pile.L'empilement et le dépilement va donc consister à « mettre à jour » l'attribut top de cet objet.Une proposition d'implantation peut être :

```
from Cellule import Cellule

class Pile :


    def __init__(self) :
        ''' Crée une pile vide.
        L'attribut 'top' est un objet Cellule qui définit la cellule  constituant
        le'sommet' de la pile.
        '''
        self.top = None

    def estVide(self) :
        '''
        Renvoie ``True`` si la pile est vide et ``False`` sinon.
        '''
        return self.top == None

    def sommet(self) :

        '''
        Renvoie la valeur de l'élément au sommet de la pile.
        '''
        if not(self.estVide()) :
            return self.top.valeur
        else :
            return None

    def empiler(self, element) :
        '''
         Paramètres    element : est de n'importe quel type
        Description : L'élément à empiler sur la pile.
        Ajoute un élément au sommet de la pile.
        '''
        nouvelleCellule = Cellule(element, self.top)
        self.top = nouvelleCellule

    def depiler(self) :
        '''
        Dépile et renvoie l'élément situé au sommet de la pile.
        '''
        if not(self.estVide()) :
            valeur = self.top.valeur
            self.top = self.top.suivant
            return valeur
        else :
            return None
```

```
    def __len__(self) :
        '''
        Renvoie le nombre d'éléments de la pile.

        '''
        taille  = 0
        celluleCourante = self.top
        while(celluleCourante != None) :

            celluleCourante=celluleCourante.suivant
            taille += 1
        return taille
```

0.5 Implémentation des files utilisant la programmation objet

Chaque valeur contenue dans la file est encapsulée dans un objet Cellule (définit plus haut) qui contient une valeur ainsi que l'adresse de la cellule suivante.Deux attributs (de type Cellule) sont attachés à la file :

- un attribut head correspondant à la tête (sortie) de la file. C'est ici qu'on récupère les éléments qu'on retire de la file .
- un attribut end correspondant au bout (entrée) de la file. C'est à partir de cet emplacement qu'on ajoute des éléments à la file.

Ajouter un élément à la file consiste en une opération où l'on modifie l'attribut 'end' de la file en créant un nouvel objet Cellule et en affectant correctement les attributs de l'objet.
Retirer un élément consiste à modifier l'attribut head en récupérant les valeurs de la cellule correspondante et en réaffectant cet attribut à la cellule suivante.

Le seul cas particulier auquel il faut faire attention est l'ajout du premier élément :
il correspond à la fois à l'attribut head et end (c'est à la fois la tête et le bout de la file, donc même cellule).Une proposition d'implantation peut être :

```
'''On importe la classe'Cellul' qu'on a défini précédemment.
Doit être dans le même dossier
'''

from Cellule import Cellule
class File :
    def __init__(self) :

        '''
        Créé une file vide.L'attribut'head' est un objet Cellule
        qui définit la cellule constituant la tête (sortie) de la file.
        L'attribut 'en' est un objet Cellule qui définit la cellule
        constituant le bout (entrée) de la file.
        '''
        self.head = None
        self.end = None
```

```
def estVide(self) :
    '''
    Renvoie ``True`` si la file est vide et ``False`` sinon.
    '''
    return self.head == None


def tete(self) :

   '''
   Renvoie la valeur de l'élément en tête de la file (premier élément).
   '''
   if not(self.estVide()) :
     return self.head.valeur
   else :
     return None



def ajouter(self, element) :
   '''
   Paramètres  element : N'importe quel type
   Description : L'élément à ajouter au bout de la file.
   Ajoute un élément au bout de la file.
   '''
   dernierCellule = Cellule(element, None)
   if(self.estVide()) :
    #Cas particulier si la file ne contient rien.
    # La tête == Le bout de la file.

       self.head = dernierCellule
   else :
       self.end.suivant = dernierCellule
       self.end = dernierCellule

 def retirerTete(self) :
      '''
       Retire et renvoie l'élément situé à la tête de
       la file (premier élément).
      '''

      if not(self.estVide()) :
          valeur = self.head.valeur
          self.head = self.head.suivant
          return valeur
      else :
         return None
```

Toutes les opérations sur la file s'exécutent en temps constant : elles ne dépendent pas du nombre d'éléments stocké dans la file. Cette implémentation est donc optimale en terme de complexité algorithmique.