

R308

Consolidation de la programmation

Manuel Munier

UPPA STEE - IUT des Pays de l'Adour - Département RT
manuel.munier@univ-pau.fr
<https://munier.perso.univ-pau.fr/teaching/butrt-r308/>

2023-2024



Plan du cours

- 1 Introduction
- 2 Programmation Orientée Objet
- 3 Structures Dynamiques

Plan du cours

- 1 Introduction
 - Présentation du R308
 - Déroulement
- 2 Programmation Orientée Objet
- 3 Structures Dynamiques

Ce que dit le PPN

- Objectifs du module
 - ▷ Principes fondamentaux de la programmation orientée objet
 - Classes/Objets/attributs/méthodes/constructeurs
 - Notion d'héritage, agrégation
 - Format et description de données (affichage, expr. textuelle pour un json/yaml/xml, date avec timezone, retour sur l'encodage)
 - ▷ Sérialisation des objets (texte versus binaire)
 - ▷ Gestion des erreurs/exceptions
- Pré-requis
 - ▷ R107 : Fondamentaux de la programmation
 - ▷ R207 : Sources de données
 - ▷ R208 : Analyse et traitement de données structurées

Ce que l'on va (essayer) de faire. . .

- Contenu de ce module
 - ✓ programmation orientée objet
 - ✓ sérialisation des objets
 - ✓ introduction aux structures de données avancées (listes, files, piles, arbres, etc.)
- **Ce que nous ne ferons pas !!!**
 - ✗ algorithmique : boucles, tests, tableaux à 1 dimension, dictionnaires, fonctions, . . .
 - ✗ programmation Python "de base", c'est-à-dire "hors objets"
 - ✗ ligne de commande : gestion fichiers & répertoires, . . .

⇒ `if (lacunes>0) then do_révisions([R107,R207,R208]);`

Pourquoi s'intéresser à la POO ?

- La paradigme OO est devenu incontournable
 - meilleure structuration des données et des traitements
 - programmation par (assemblage de) composants
 - notions de services, d'interfaces exposées, etc.
- Avec de nombreux avantages d'un point de vue GL...
 - séparation des concepts, modularité
 - meilleure gestion du cycle de vie des composants (dév, debug)
 - facilité de réutilisation
 - déploiement, maîtrise des interfaces, etc.

Plan du cours

- 1 Introduction
 - Présentation du R308
 - Déroulement

Déroulement

- Très peu de temps (2 cours d'1h30 ☹), donc :
 - pas le temps de faire de la théorie \Rightarrow apprentissage par la pratique, sur des exemples
 - on va se concentrer sur les fondamentaux :
 - **POO** \leadsto objets, classes, attributs, méthodes,...
 - **POO** \leadsto héritage
 - sérialisation des objets (ex : JSON)
 - **structures des données avancées** \leadsto listes, arbres,... + algorithmique associée (et en particulier la notion de récursivité)

Plan du cours

- 1 Introduction
- 2 Programmation Orientée Objet
 - Introduction
 - Objets en Python
 - Constructeurs
 - Encapsulation
 - Héritage
- 3 Structures Dynamiques

Introduction : motivations

- Il est parfois nécessaire de **regrouper** plusieurs données dans une seule variable
 - 3 entiers jour, mois, année pour représenter une date
 - nom, prénom, date_naissance, ... pour une personne
 - ref, libellé, prix, catégorie, ... pour un article
 - point (lui même composé de X et Y), longueur, largeur pour un rectangle
- ⇒ On définit pour cela des structures de données
- ~> tuple en Python
 - ~> record en Pascal
 - ~> struct en langage C

Introduction : motivations

Pb n°1 Si les données sont mieux structurées, cela reste néanmoins très "statique"

- les données d'un côté
- les traitements (fonctions) en "vrac" dans le programme

Pb n°2 Du coup, les fonctions doivent connaître la structure (interne) de ces données pour pouvoir les manipuler

- `lendemain(date)`
- `ajouter(date,nb_jours)`
- `surfaceRect(rectangle), surfaceCercle(cercle),...`
- `moyenne(tableau,nb_cases_remplies)`
- `ajouter(tableau,nb_cases_remplies,valeur)`

Introduction : concept d'objet

Définition d'un objet

L'idée consiste à **encapsuler** au sein d'une même entité à la fois

- les données (appelées **attributs**)
- les traitements (appelés **méthodes**)

- Les méthodes ont accès à tous les attributs de l'objet sur lequel elles s'exécutent (en plus de leurs paramètres)
- Exemples :
 - `date.lendemain()`
 - `date.ajouter(nb_jours)`
 - `rectangle.surface()`, `cercle.surface()`,...
 - `tableau.moyenne()`, `tableau.ajouter(valeur)`

Introduction : concept d'objet

Abstraction de données

Seules les méthodes d'un objet ont accès aux attributs de cet objet.

- Les attributs sont "masqués"
 - ⇒ ils ne sont pas accessibles de l'extérieur
 - ⇒ l'objet ne peut être manipulé qu'au travers de ses méthodes

⇒ Modularité

- les objets peuvent être testés individuellement
- un changement sur les attributs n'impacte pas l'extérieur (à condition que les signatures des méthodes restent identiques)
- les algorithmes sont plus "clairs" 😊

Objets en Python

- En POO il est donc nécessaire de définir la structure des objets que l'on va utiliser
 - ▷ **attributs** ~> quelles sont les informations "portées" par chaque objet
 - ▷ **méthodes** ~> "fonctions" attachées à chaque objet
 - ▷ **constructeurs** ~> fonctions spéciales permettant de créer un objet
- ⇒ En POO, une **classe** est un type d'objet
- **Problème** : Contrairement à Java, C++ ou Pascal, le langage Python est **non typé** (comme JavaScript ou PHP d'ailleurs)

Objets en Python

- La définition de classes en Python n'est qu'une "surcouche syntaxique" 😞
 - ▷ on peut parler de programmation par prototypage

⇒ Inconvénients

- ▷ pas "POO only" : on peut mixer objets et fonctions classiques (même pb en C++)
 - ⇒ peut perturber les débutants en OO
- ▷ quiconque peut ajouter des attributs et des méthodes à des objets, sans pour autant "mettre à jour" la classe \leadsto pas très rigoureux...

⇒ Avantages

- ▷ peut apporter plus de souplesse... quand on sait exactement ce que l'on fait !

Objets en Python

- Par contre, autant prendre tout de suite de "bonnes habitudes" en POO...
 - ▷ pour programmer "proprement" en OO (cf. GL)
 - ▷ pour bien comprendre l'intérêt des approches OO
 - programmation par assemblage de composants
 - notion de services \leadsto SOA
 - dissociation interface et implémentation des services
 - ▷ pouvoir faire de l'OO dans d'autres langages, parfois plus stricts...

Objets en Python : exemple n°1 (Point)

- Nous voulons définir des objets "points" composés de 2 coordonnées x et y
- v1 Version "basique" : comme un simple tuple... pas vraiment POO!!!
- rien dans la classe \leadsto pas d'encapsulation
 - les attributs sont ajoutés par le main \leadsto pas d'abstraction

classe Point

```

class Point:
    "Définition d'un point"

def main():
    pt = Point()
    pt.x = 10.5
    pt.y = 5.4
    print("pt=(%2.1f,%2.1f)" % (pt.x,pt.y))

# Programme principal
main()
  
```

affichage

```
pt=(10.5,5.4)
```

Objets en Python : exemple n°1 (Point)

- Remarque : chaque objet a ses propres attributs

classe Point

```
class Point:
    "Définition d'un point"

def main():
    pt = Point()
    pt.x = 10.5
    pt.y = 5.4
    print("pt=(%2.1f,%2.1f)" % (pt.x,pt.y))

    pt2 = Point()
    pt2.x = -3
    pt2.y = 12
    print("pt=(%2.1f,%2.1f)" % (pt.x,pt.y))
    print("pt2=(%2.1f,%2.1f)" % (pt2.x,pt2.y))

# Programme principal
main()
```

affichage

```
pt=(10.5,5.4)
pt=(10.5,5.4)
pt2=(-3.0,12.0)
```

Objets en Python : exemple n°1 (Point)

- Remarque : distinction entre variable et objet \rightsquigarrow notion de référence
 - ici, les 2 variables pt2 et pt sont des références vers le même objet en mémoire!

classe Point

```

class Point:
    "Définition d'un point"

def main():
    pt = Point()
    pt.x = 10.5
    pt.y = 5.4
    print("pt=(%2.1f,%2.1f)" % (pt.x,pt.y))

    pt2 = pt
    pt2.x = -3
    pt2.y = 12
    print("pt=(%2.1f,%2.1f)" % (pt.x,pt.y))
    print("pt2=(%2.1f,%2.1f)" % (pt2.x,pt2.y))

# Programme principal
main()
  
```

affichage

```

pt=(10.5,5.4)
pt=(-3.0,12.0)
pt2=(-3.0,12.0)
  
```

Objets en Python : instantiation

- En POO il faut impérativement distinguer 2 choses :
 - l'**objet** lui même
 - ~> *une zone de la mémoire contenant les attributs, les méthodes, sa classe (~type de l'objet), etc...*
 - une **référence** sur cet objet
 - ~> *"l'adresse" de cette zone mémoire (~pointeur)*
 - ~> *plusieurs références (variables différentes) peuvent désigner le même objet (même "adresse" comme valeur)*

Instantiation

Une **instance** est un objet créé à partir d'une classe par un mécanisme appelé **instanciation**.

Objets en Python : exemple n°1 (Point)

- Une **méthode** est une "fonction" s'exécutant sur l'état d'un objet

v2 Ajout d'une méthode

- le 1^{er} paramètre doit être **self** (réf. à l'objet lui-même)

classe Point

```
class Point:
    "Définition d'un point"
    def deplacer(self, dx, dy):
        self.x = self.x+dx
        self.y = self.y+dy

def main():
    pt = Point()
    pt.x = 10.5
    pt.y = 5.4
    print("pt=(%2.1f,%2.1f)" % (pt.x,pt.y))

    pt.deplacer(3,2)
    print("pt=(%2.1f,%2.1f)" % (pt.x,pt.y))

# Programme principal
main()
```

affichage

```
pt=(10.5,5.4)
pt=(13.5,7.4)
```

Constructeurs

- Modularité & tests (GL) \leadsto une méthode prend un objet "stable" et le rend dans un état "stable"

Pb Quel est l'état¹ de l'objet lors de son instantiation ?

Constructeur

Un **constructeur** est une méthode particulière qui est invoquée automatiquement lors de l'instanciation d'un objet. Son objectif est d'initialiser tous les attributs de l'objet.

- En Python, un constructeur porte le nom `__init__()`

1. valeurs de ses attributs

Constructeurs : exemple n°1 (Point)

v3 Notion de constructeur pour initialiser "proprement" l'objet

- on constate que le constructeur a bien créé les 2 attributs x et y

classe Point

```
class Point:
    "Définition d'un point"
    def __init__(self):
        self.x = 0
        self.y = 0

    def deplacer(self, dx, dy):
        self.x = self.x+dx
        self.y = self.y+dy

def main():
    pt = Point()
    print("pt=(%2.1f,%2.1f)" % (pt.x,pt.y))

    pt.deplacer(3,2)
    print("pt=(%2.1f,%2.1f)" % (pt.x,pt.y))

# Programme principal
main()
```

affichage

```
pt=(0.0,0.0)
pt=(3.0,2.0)
```

Constructeurs : exemple n°1 (Point)

v3.1 Utilisation de constructeurs paramétrés

- le constructeur a bien initialisé les 2 attributs x et y avec les 2 paramètres

classe Point

```
class Point:
    "Définition d'un point"
    def __init__(self,a,b):
        self.x = a
        self.y = b

    def deplacer(self, dx, dy):
        self.x = self.x+dx
        self.y = self.y+dy

def main():
    pt = Point(10.5,5.4)
    print("pt=(%2.1f,%2.1f)" % (pt.x,pt.y))

    pt.deplacer(3,2)
    print("pt=(%2.1f,%2.1f)" % (pt.x,pt.y))

# Programme principal
main()
```

affichage

```
pt=(10.5,5.4)
pt=(13.5,7.4)
```

Constructeurs : exemple n°1 (Point)

NB Contrairement à la quasi majorité des langages OO, Python ne supporte malheureusement pas la **surcharge de méthodes**

- en Python, la 2^{ème} définition du constructeur va remplacer la 1^{ère} 😞

classe Point ~> erreur

```
class Point:
    "Définition d'un point"
    def __init__(self):
        self.x = 0
        self.y = 0

    def __init__(self, a, b):
        self.x = a
        self.y = b

def main():
    pt1 = Point() # -> erreur car ce constructeur n'existe plus !
    print("pt=(%2.1f,%2.1f)" % (pt1.x,pt1.y))
    pt2 = Point(10.5,5.4)
    print("pt=(%2.1f,%2.1f)" % (pt2.x,pt2.y))

# Programme principal
main()
```

classe Point

```
class Point:
    "Définition d'un point"
    def __init__(self, a=0, b=0):
        self.x = a
        self.y = b

def main():
    pt1 = Point() # -> ok; utilise les valeurs par défaut
    print("pt=(%2.1f,%2.1f)" % (pt1.x,pt1.y))
    pt2 = Point(10.5,5.4)
    print("pt=(%2.1f,%2.1f)" % (pt2.x,pt2.y))

# Programme principal
main()
```

Encapsulation : exemple n°1 (Point)

v4 Encapsulation \equiv attributs protégés

- le nom des attributs est préfixé par `__` \Rightarrow attributs privés \Rightarrow **méthodes d'accès**

classe Point

```
class Point:
    "Définition d'un point"
    def __init__(self,a,b):
        self.__x = a
        self.__y = b

def main():
    pt = Point(10.5,5.4)
    print("pt=(%2.1f,%2.1f)" % (pt.__x,pt.__y))

# Programme principal
main()
```

affichage \rightsquigarrow erreur d'accès

```
Traceback (most recent call last):
  File "test221.py", line 15, in <module>
    main()
  File "test221.py", line 11, in main
    print("pt=(%2.1f,%2.1f)" % (pt.__x,pt.__y))
AttributeError: 'Point' object has no attribute '__x'
```

classe Point

```
class Point:
    "Définition d'un point"
    def __init__(self,a,b):
        self.__x = a
        self.__y = b

    def afficher(self):
        print("pt=(%2.1f,%2.1f)" % (self.__x,self.__y))

def main():
    pt = Point(10.5,5.4)
    pt.afficher()

# Programme principal
main()
```

affichage

```
pt=(10.5,5.4)
```

Encapsulation

- Si les attributs sont privés, seul l'objet lui-même y a accès
- ⇒ Tout membre "extérieur" (autre objet, fonction, etc.) devra donc obligatoirement passer par les méthodes d'accès proposées par cet objet
 - ▷ **constructeurs** pour initialiser l'objet
 - ▷ **accesseurs** pour lire les attributs (ex : des méthodes `get_XXX()`)
 - ▷ **mutateurs** pour modifier les attributs (ex : des méthodes `set_XXX()`)
- Au passage, ces méthodes d'accès peuvent également...
 - filtrer, mettre en forme les informations
 - contrôler les valeurs "injectées", déclencher des calculs, etc.
 - mettre en place une politique d'accès : *read only*, *write only*, etc.

Remarques

- En POO, toute la "mécanique" est cachée dans les objets
 - organisation des attributs
 - algorithmes des méthodes
 - Les objets "extérieurs" doivent utiliser les "services" proposés par les objets
 - constructeurs pour instancier des objets
 - méthodes (publiques) pour consulter / modifier les objets
- ⇒ "Propreté" d'un point de vue GL
- traitements locaux
 - interfaces bien identifiées
 - impact d'un changement minimisé
- Ex modification des spécifications, ajout de fonctionnalités, correction de bug,...*

Héritage

- L'héritage est un mécanisme qui nous permet de créer une nouvelle classe (**classe fille** ou **sous-classe**) qui est basée sur une classe existante (**classe mère** ou **super-classe**), en ajoutant de nouveaux attributs et méthodes en plus de la classe existante.
- Ce faisant, la classe fille **hérite** des attributs et des méthodes de la classe mère

syntaxe

```
class mere:
    # corps de la classe mère

class enfant(mere):
    # corps de la classe enfant
```

Héritage : exemple n°2 (Employe)

classes Personne et Employe

```
class Personne():
    # Constructeur
    def __init__(self, nom, prenom):
        self.nom = nom
        self.prenom = prenom

    def afficher(self):
        print("Nom      : ",self.nom)
        print("Prénom  : ",self.prenom)

class Employe(Personne):
    # Constructeur
    def __init__(self, nom, prenom, job):
        # appel du constructeur de la classe mère (Personne)
        Personne.__init__(self, nom, prenom)

        # ajout d'un attribut
        self.job = job

    def afficher(self):
        Personne.afficher(self)
        print("Job      : ",self.job)
```

programme

```
def main():
    # création d'une variable d'instance
    p=Personne("Doe", "John")
    # appel d'une fonction de la classe Personne via son instance
    p.afficher()

    # création d'une instance de la sous-classe
    e=Employe("Bond","James","agent secret")
    e.afficher()

# Programme principal
main()
```

affichage

```
Nom      : Doe
Prénom   : John

Nom      : Bond
Prénom   : James
Job      : agent secret
```

Héritage : exemple n°2 (Employe)

classes Personne et Employe

```
class Personne():
    # Constructeur
    def __init__(self, nom, prenom):
        self.nom = nom
        self.prenom = prenom

    def afficher(self):
        print("Nom      : ",self.nom)
        print("Prénom  : ",self.prenom)

class Employe(Personne):
    # Constructeur
    def __init__(self, nom, prenom, job):
        # appel du constructeur de la classe mère (Personne)
        # utilisation du mot clé super()
        super().__init__(nom, prenom)

        # ajout d'un attribut
        self.job = job

    def afficher(self):
        super().afficher()
        print("Job      : ",self.job)
```

programme

```
def main():
    # création d'une variable d'instance
    p=Personne("Doe", "John")
    # appel d'une fonction de la classe Personne via son instance
    p.afficher()

    # création d'une instance de la sous-classe
    e=Employe("Bond","James","agent secret")
    e.afficher()

# Programme principal
main()
```

affichage

```
Nom      : Doe
Prénom   : John

Nom      : Bond
Prénom   : James
Job      : agent secret
```

Héritage

- L'héritage est un concept **très important** en POO !!!
- Via l'héritage nous pouvons définir de nouvelles classe et...
 - ▷ ajouter de nouveaux attributs
 - ▷ ajouter de nouvelles méthodes
 - ▷ "masquer" des attributs de la classe mère
 - ▷ redéfinir des méthodes de la classe mère (**surcharge**)
 - Ex méthode `afficher()` de `Personne` redéfinie dans la sous-classe `Employe`
⇒ **liaison dynamique** ≡ choix de la "bonne" méthode lors de l'exécution

Synthèse

- En POO, un programme est un ensemble de petites entités autonomes (les **objets**) qui interagissent et communiquent par messages (les **appels de méthodes**)
 - L'accent est mis sur l'autonomie de ces entités et sur les traitements locaux
- ⇒ Programmer avec des objets nécessite un changement d'état d'esprit de la part du programmeur

Synthèse

- 3 critères de qualité en Génie Logiciel
 - **fiabilité** : un composant fonctionne dans tous les cas de figure
 - **extensibilité** : on peut ajouter de nouvelles fonctionnalités sans modifier l'existant
 - **réutilisabilité** : les composants peuvent être réutilisés (en partie ou en totalité) pour construire de nouvelles applications

⇒ 1 mot : **modularité**

Synthèse

- Cette modularité a un coût
 - à partir du cahier des charges, il faut identifier les objets
- En amont de la phase de programmation ont ainsi été développées des méthodes d'analyse et/ou de conception orientées objet
 - Ex UML (*the Unified Modeling Language*)
- **Ne concerne pas le module R308 😊**
 - ↪ les sujets préciseront les objets/méthodes à programmer
 - ↪ UML sera abordé "plus tard"...

Plan du cours

- 1 Introduction
- 2 Programmation Orientée Objet
- 3 Structures Dynamiques
 - Introduction
 - Concept de SDD
 - Listes chaînées
 - Arbres binaires

Introduction : motivations

- En programmation, on ne connaît pas toujours à l'avance le nombre de variables (de cases mémoire) dont on aura besoin, ni forcément leur taille, etc.
 - ⇒ **allocation dynamique** (de mémoire) \leadsto "création à la volée de nouvelles variables"
- On est parfois limité par la taille du *pool* des variables
 - ⇒ l'allocation dynamique se fait dans le "**tas**"
 - ⇒ seules les références "racines" se trouvent dans le *pool* des variables
- Les structures de données "de bases" telles que les tuples, les tableaux, les listes, etc. manquent de souplesse
 - ⇒ les structures de données dynamiques permettent de "**déconnecter**" et de "**reconnecter**" facilement des cellules (aka zones mémoire) entre elles

Introduction : motivations

- Remarques
 - Si dans les langages traditionnels (ex : C, Java) les tableaux sont restés très statiques, les nouveaux langages tels que Python ou JavaScript proposent des tableaux "plus souples"
 - augmentation automatique de la taille
 - fonctionnalités avancées : taille, insertion, suppression, tri, somme, moyenne, etc.
 - ⇒ mais en réalité **ce ne sont plus des "tableaux"** au sens algorithmique ; ils sont gérés comme des objets 😊
 - Beaucoup de langages proposent maintenant des bibliothèques de classes pour faciliter la vie des programmeurs
 - `Hashtable` en Java
 - listes et dictionnaires en Python ou en JavaScript
 - ⋮

Concept de SDD : présentation

- À la base, les SDD (**S**tructures de **D**onnées **D**ynamiques) sont constituées de **cellules**
- Conceptuellement, chaque cellule contient
 - ▷ une **valeur**
 - ▷ des **liens** vers d'autres cellules
- En pratique
 - ▷ la valeur d'une cellule peut bien sûr être une donnée composée (ex : tuple)
 - ▷ les liens sont des adresses mémoire (ex : C) ou des références objet en POO (ex : Java, Python)

Exemples de SDD

- **Listes chaînées** \leadsto chaque cellule a un seul suivant ; pas de cycle
- **Listes doublement chaînées** \leadsto chaque cellule a 2 liens : un suivant et un précédent ; pas de cycle
- **Listes circulaires** \leadsto listes simplement ou doublement chaînées avec possibilité de cycle
- **Arbres binaires** \leadsto chaque cellule a 2 suivants : un fils gauche et un fils droit ; pas de cycle
- **Arbres n-aires** \leadsto chaque cellule a plusieurs (une liste de) fils ; pas de cycle
- **Graphes** \leadsto chaque cellule peut avoir plusieurs suivants ; cycles possibles

Listes chaînées : classe Node (base)

classe Node

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

myNode = Node(10)
print("The data in the node is:", myNode.data)
print("The next attribute in the node is:", myNode.next)
```

affichage

```
The data in the node is: 10
The next attribute in the node is: None
```

Listes chaînées : classe LinkedList

classes Node et LinkedList

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

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

myLinkedList = LinkedList()
myNode1 = Node(10)
myNode2 = Node(20)
myNode3 = Node(30)
myNode4 = Node(40)
myLinkedList.head = myNode1
myNode1.next = myNode2
myNode2.next = myNode3
myNode3.next = myNode4

print("The elements in the linked list are:")
print(myLinkedList.head.data, end=" ")
print(myLinkedList.head.next.data, end=" ")
print(myLinkedList.head.next.next.data, end=" ")
print(myLinkedList.head.next.next.next.data)
```

affichage

```
The linked list is:
10 20 30 40
```

Listes chaînées : parcours d'une liste chaînée

méthode printList() (itérative)

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

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

    def printList(self):
        current = self.head
        while current is not None:
            print(current.data, end=" ")
            current = current.next

myLinkedList = LinkedList()
myNode1 = Node(10)
myNode2 = Node(20)
myNode3 = Node(30)
myNode4 = Node(40)
myLinkedList.head = myNode1
myNode1.next = myNode2
myNode2.next = myNode3
myNode3.next = myNode4

print("The elements in the linked list are:")
myLinkedList.printList()
```

affichage

The elements in the linked list are:
10 20 30 40

Listes chaînées : parcours récursif d'une liste chaînée

méthode printNode() (récursive)

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

    def printNode(self):
        print(self.data, end= " ")
        if self.next is not None:
            self.next.printNode()

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

    def printListRec(self):
        if self.head is not None:
            self.head.printNode()
        print()

myLinkedList = LinkedList()
myNode1 = Node(10)
myNode2 = Node(20)
myNode3 = Node(30)
myNode4 = Node(40)
myLinkedList.head = myNode1
myNode1.next = myNode2
myNode2.next = myNode3
myNode3.next = myNode4

print("The elements in the linked list are:")
myLinkedList.printListRec()
```

affichage

The elements in the linked list are:
10 20 30 40

Arbres binaires : classe Node (base)

classe Node

```
class Node:  
    def __init__(self, value):  
        self.left = None  
        self.right = None  
        self.data = value  
  
root = Node(10)  
  
root.left = Node(34)  
root.right = Node(89)  
root.left.left = Node(45)  
root.left.right = Node(50)
```

résultat

