Classe Vecteur : Implémentation

```
def __init__(self):
  self._x = 0
  self._y = 0

def base(self, v):
  self._x = 0
  self._y = 1
  v._x = 1
  v._y = 0

def egaux(self, v):
  eq = (self._x == v._x and self._y ==
     v._y)
  return eq

def nul(self):
  n = (self._x == 0 and self._y == 0)
  return n
```

```
def colinéaires(self, v):
  co = (not nul(self) and not nul(v) and
  (self._x * v._y - v._x * self._y ) == 0)
  return co

def addition(self, v):
  self._x = self._x + v._x
  self._y = self._y + v._y

def multiplication(self, k):
  self._x = k * self._x
  self._y = k* self._y
```

Classes Génériques

- Certaines classes peuvent être déclinées de différentes manières
  - Par ex. les structures appelées à stocker plusieurs valeurs d'un même type comme les piles, files...
  - Implémentation ne dépend pas du type de valeur stockée
- **Idée** : on souhaiterait donc pouvoir écrire ces classes une seule fois, et les réutiliser quel que soit le type de valeur qu'on souhaite y stocker.

Classes Génériques avec Python

- En Python et dans les langages faiblement typés, on se contentera en général de ne pas contraindre le type des éléments manipulés par la classe
  - Utilisation du type le plus abstrait (**object** en Python).
- On utilisera ensuite les pré/post-conditions pour imposer le type des éléments d'un objet donné.
  - Par exemple, une fonction prenant en paramètre d'entrée une pile pourra imposer dans ses pré-conditions que cette pile ne contienne que des entiers
  - Mais le respect de cette pré-condition restera de la responsabilité du programmeur!

M2103 – Structures de Données

Cours 2
Allocation Dynamique – Listes Chaînées

Plan du Cours

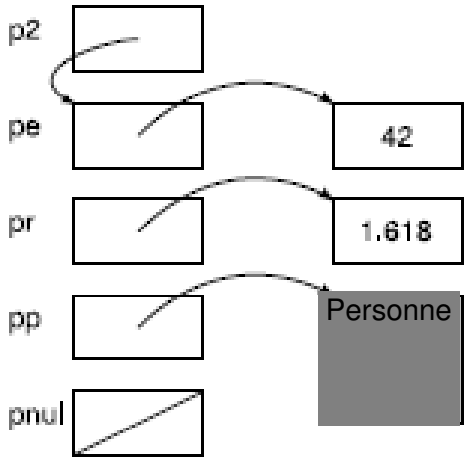
- Allocation Dynamique - Introduction
- Rappel – Type Pointeur
- Type Abstrait de Données Liste
  - Implémentation : Classe TabListe
  - Problèmes liés à l'utilisation d'un tableau
- Listes Chaînées (Simples)
  - Introduction
  - Ajout
  - Suppression
  - Maillon Tête Factice
- Accès
- Listes Chaînées Doubles
  - Insertion
  - Suppression
- Listes Chaînées Circulaires

Allocation Dynamique

- On connaît les tableaux
  - Structure de données statique
    - Chaque tableau occupe en mémoire la taille maximum envisagée
- Que faire si on ne souhaite pas “perdre de place”?
  - Structure de données dynamique
    - A chaque instant, la place occupée par les données dépend uniquement de la taille de celles-ci (idéalement)
- Mécanisme d'allocation dynamique de la mémoire

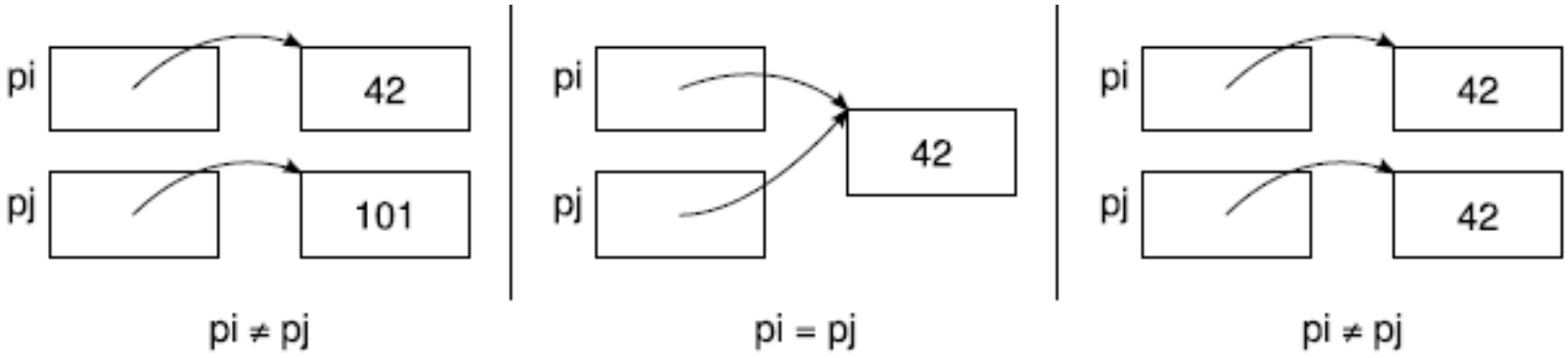
Rappel - Manipulation de variables allouées dynamiquement : le type pointeur

- Variable qui contient l'adresse en memoire d'une autre variable
  - Ne peut pointer que vers des variables d'un seul type
  - Pour déclarer une variable de type pointeur, on donne le type des variables pointées, suivi d'une étoite
    - int* pe //pointeur vers entier
    - float* pr //pointeur vers réel
    - Personne* pp //pointeur vers Personne
    - int** p2 //pointeur vers pointeur
- Un pointeur peut egalement ne pointer vers aucune variable : on l'appelle alors pointeur nul



Opérations sur variables de type pointeur : Comparaison

- Deux pointeurs sont égaux s'ils contiennent la même adresse mémoire
  - i.e. ils pointent vers la même variable
- On notera que deux pointeurs peuvent pointer vers deux variables distinctes mais ayant la même valeur
  - Les pointeurs sont bien différents, puisqu'ils contiennent des adresses mémoire différentes.
  - La valeur des variables pointées n'a aucune influence sur la comparaison des pointeurs



Opérations sur variables de type pointeur : Déréférencement

- Opération permettant, à partir d'un pointeur, d'accéder à la variable pointée (*)
  - Si ***pe*** est un pointeur sur entier, ****pe*** représente l'entier pointé par ***pe***
- Peut être placée à gauche d'une affectation afin de changer la valeur de la variable pointée
  - Example : *pe = 42
  - Le fait d'affecter *pe n'impacte pas le pointeur : il pointe toujours vers la même variable, ce n'est que la valeur de cette variable qui change
  - Modifcation toutefois « visible » depuis d'autres pointeurs pointant vers la même variable
- Attention !
  - Toujours s'assurer lorsqu'on déréférence un pointeur qu'il n'est pas nul
  - Déréférencer un pointeur non initialisé est plus dangereux…

Type Abstrait de Données `Liste`

- Une Liste est une collection d'éléments, chacun avec une position distincte, i.e. les éléments ont un ordre d'apparition

- Opérations
  - Création d'une liste
  - Ajout d'un élément
  - Retourner la position d'un élément donné de la liste
  - Modification d'un élément à une position valide donnée avec récupération de l'ancien élément
  - Suppression d'un élément à une position valide donnée avec récupération de cet élément

Classe TabListe : Interface

```
def __init__(self):
 """
 :sortie self:
 :post-cond: tableau déclaré, initialisation nb éléments
 """

def ajout(self,x):
 """
 :entrée-sortie self:
 :entrée x: object
 :pré-cond: le tableau n'est pas plein
 :post-cond: ajout de x à TabListe
 """

def retourner_pos(self,x):
 """
 :entrée self:
 :entrée x: object
 :sortie i: int
 :pré-cond: l'élément x se trouve dans la liste
 :post-cond: i est sa position
 """
```

```
def set(self,id,nouvVal):
 """
 :entrée-sortie self:
 :entrée id: int
 :entrée nouvVal: object
 :sortie vieilleVal: object
 :pré-cond: l'élément à modifier est à la position id
 :post-cond: retourne l'ancien élément à la position id
 """

def suppr(self,id):
 """
 :entrée self:
 :entrée id: int
 :sortie supprVal: object
 :pré-cond: l'élément à supprimer est à la position id
 :post-cond: retourne l'élément supprimé supprVal
 """
```

Classe TabListe : Implémentation

```
MAX = 100

def __init__ (self) :
  self._tab_liste = empty(MAX, object)
  self._nb_elem = 0

def ajout (self,x) :
  self._tab_liste[self._nb_elem] = x
  self._nb_elem = self._nb_elem+1

def retourner_pos (self,x) :
  for i in range (0, self._nb_elem):
    if self._tab_liste[i] == x:
      return i
```

```
def set (self,id,nouvVal) :
  vieilleVal = self._tab_liste[id]
  self._tab_liste[id] = nouvVal
  return vieilleVal

def suppr (self,id) :
  supprVal = self._tab_liste[id]
  self._nb_elem = self._nb_elem-1
  for i in range (id, self._nb_elem) :
    self._tab_liste[i]=self._tab_liste[i+1]
  return supprVal
```

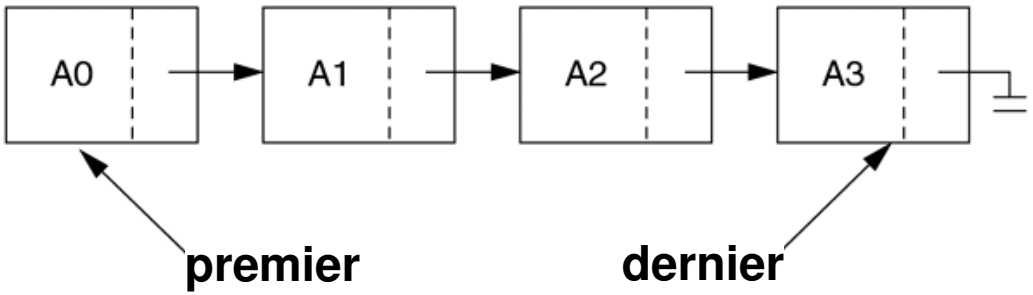
Implémentation des opérations de Liste avec un tableau

- Problèmes liés à l'utilisation d'un tableau :
  - Suppression (et éventuellement ajout) à l'origine de décalages de données
    - Moitié des éléments en moyenne
  - La taille du tableau arrive à son maximum
    - Nécessaire de déclarer un nouveau tableau (par exemple de taille double)
    - Peut être à l'origine d'un gâchis d'espace mémoire

Listes Chaînées : Introduction (1/2)

```
Tmaillon = Struct (
        valeur = object,        // un élément
        suivant = SAME )        // lien vers le maillon suivant
```

- Une collection A représentée par une ***liste chaînée*** peut éviter ces problèmes en stockant les éléments sans avoir besoin de mémoire contiguë et en maintenant des liens entre les éléments par ordre de position
  - Un lien externe référençant le ***premier*** élément de la liste doit être mis en oeuvre pour accéder à la liste
  - Si un nouvel élément doit être inséré, un lien externe référençant le ***dernier*** élément est modifié



Listes Chaînées : Introduction (2/2)

- Ajout d'un nouveau dernier élément x :

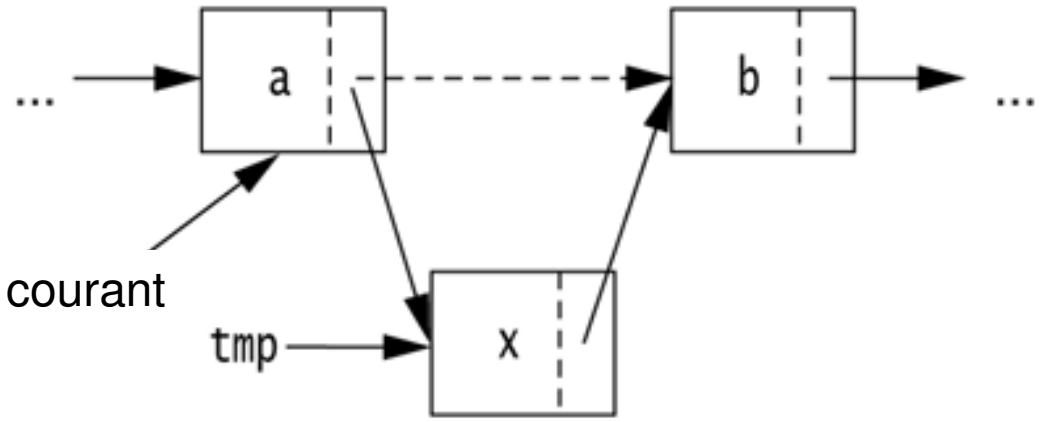
```
NouveauMaillon = Tmaillon(valeur=x, suivant=None)   // Création d'un maillon
dernier.suivant = NouveauMaillon                    // Attache le maillon à la liste
dernier = dernier.suivant                           // Ajuste le dernier maillon
```

- Pour accéder aux éléments dans la liste, nous utilisons une référence sur le maillon considéré au lieu d'un indice

Ajout

- Nous considérons l'insertion d'un élément x **après** le maillon référencé par le lien *courant*

```
tmp = Tmaillon (valeur=x, suivant=courant.suivant)   // Création d'un maillon
                                                     // Placement de x dans le maillon
                                                     // Le suivant de courant devient le suivant de tmp
courant.suivant = tmp                                // tmp est placé après le maillon courant
```

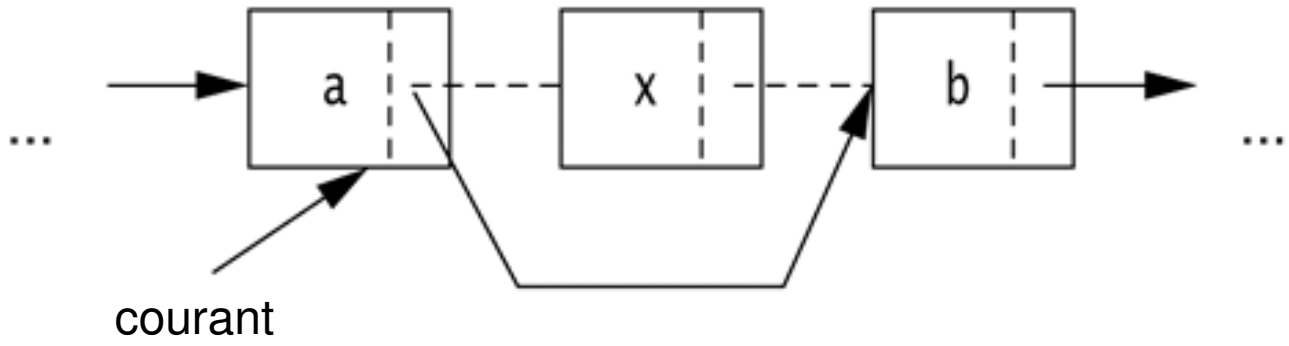


Suppression

- Suppression de l'élément situé après le maillon référencé par le lien ***courant***

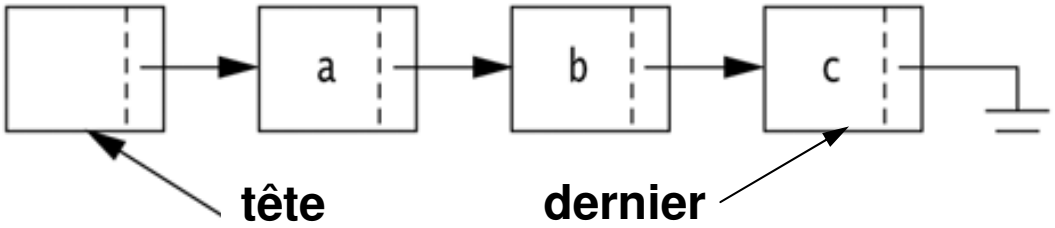
courant.suivant = courant.suivant.suivant

// le maillon situé après le maillon courant est contourné



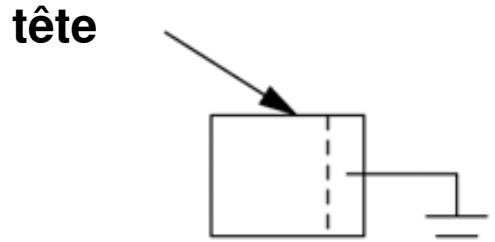
Pour aller plus loin : Maillon Tête Factice

- Insertion/Suppression d'un nouveau premier élément nécessitent de considérer des cas particuliers en l'état
- Au lieu d'implémenter ces cas particuliers
  - Introduction d'un maillon ***tête*** factice sans donnée en tant que premier maillon
    - Tous les maillons avec des données ont donc un maillon précédent



- Une liste chaînée vide n'a donc que le maillon vide initialisé

tete.suivant = None



Exercices

Ecrire les méthodes d'initialisation et d'ajout de la classe LCListe (implémentation d'une Liste à l'aide d'une liste chaînée)

- en considérant une liste chaînée simple
- en considérant une liste chaînée avec tête factice

Considérer une liste chaînée implémentée avec tête factice. Décrire des algorithmes en temps constant afin de :

- Insérer un élément x avant la position référencée par **`courant`**
- Supprimer l'élément stocké dans le maillon référencé par **`courant`**, celui-ci n'étant pas le dernier maillon de la liste