



# **Algorithmique et Programmation avancées - MIA0501V**

**MISE EN ŒUVRE AVEC PYTHON**

**S. Ebersold**

# Algorithmique et Programmation avancées

## Mise en œuvre avec Python

Notions de base de la Programmation orientée Objets

- ▶ Types Abstraits de Données : Rappel
- ▶ Notions de classes, d'objets et de délégation
- ▶ Héritage et concepts liés

# Rappel : Notion de Type Abstrait de Données

- ▶ Qu'est ce qu'un TAD ?

# Principe

un Type abstrait de données

est :

Un descripteur d'un type de données

n'est pas :

Une implantation

# Principe

- ▶ un Type abstrait de données
- ▶ Fournit :
  - ▶ La description de la structure de la donnée
  - ▶ La description du comportement de la donnée

# Exemple : le type pile

- ▶ Comment représenter une pile ?
- ▶ Quels éléments composent une pile quelle qu'elle soit et indépendamment de son implantation ?
- ▶ Quelles opérations permettent de mettre en oeuvre le comportement attendu d'une pile quelle qu'elle soit ?
- ▶ Quelles contraintes a-t-on sur ces opérations ?

# Exemple : le type pile

► Point de départ :

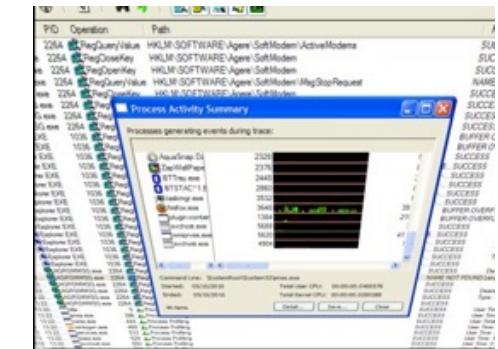


► Point commun entre ces 3 piles ?



► Qu'est-ce qui caractérise une pile ?

- ▶ Le dessus de la pile est seul accessible
- ▶ Tout nouvel élément est déposé sur le dessus de la pile
- ▶ .....



# Les piles : Définition informelle

- ▶ Une pile d'éléments est un objet dans lequel il est possible de déposer et retirer des éléments.
- ▶ Les dépôts et les retraits ont lieu selon l'ordre LIFO (Last In First Out).
- ▶ Il n'est pas possible de retirer un élément lorsqu'une pile est vide.
  
- ▶ Une seconde façon de voir la pile consiste à dire qu'il s'agit d'un objet sur lequel il est possible de déposer des éléments, d'observer l'élément qui a été le dernier posé et aussi de le supprimer.

# Exemple : le type pile

TAD : Pile

Données :

T : le type de l'information contenue dans la pile P

Opérations s'appliquant au type Pile :

Générateur

pile() : -> Pile

Opérations observatrices

hauteur?() : -> entier

vide?() : -> booléen

sommet?() : -> T

Opérations modificateurs

empiler(elt : T) : -> Pile

dépiler() : -> Pile

# Obtention d'un TAD

Rappel :

- ▶ Les TAD permettent de représenter des abstractions courantes
- ▶ Ils ne sont pas liés à une quelconque mise en œuvre dans un langage de programmation
- ▶ Ils se composent :
  - ▶ D'opérations (de base et évoluées)
  - ▶ De propriétés (comportement des opérations)

# Avantages de l'abstraction

- ▶ On peut et on doit utiliser un TAD sans se soucier des détails de son implantation
- ▶ On est libre d'implémenter un TAD «comme on veut» pour autant que l'on respecte ses spécifications abstraites
- ▶ On peut changer d'implémentation d'un TAD sans que cela change quoi que ce soit à son utilisation
- ▶ Cela augmente la simplicité du raisonnement lorsque l'on conçoit un algorithme utilisant un TAD
- ▶ Un TAD est à une structure de données concrète, ce qu'est un algorithme à un programme : un modèle

# Critères de qualité d'un TAD

- ▶ Facilité de compréhension et d'utilisation
- ▶ Séparation claire de la spécification et de la réalisation.
  - ▶ ⇒ Garantir l'indépendance du code qui utilise le TAD par rapport à sa mise en œuvre.
- ▶ Abstraction des données, protection
  - ▶ ⇒ Pour garantir la séparation spécification /réalisation.
- ▶ Efficacité
  - ▶ ⇒ Performance des algorithmes mis en œuvre
- ▶ Stabilité de la spécification
  - ▶ ⇒ Pour éliminer les risques de propagation des changements dans le code qui utilise le TAD.

# TAD : Avantages

- ▶ Briques de base pour une conception modulaire
- ▶ Construction d'applications complexes par assemblage de TAD plus simples
- ▶ Possibilité de tester et de valider le logiciel par étapes
- ▶ Approche de développement efficace et validée

# TAD : Inconvénients

- ▶ Coût en performances
- ▶ Les services sont documentés par leur coût
- ▶ L'interface d'un TAD doit être stable => effort de conception.

# Caractéristiques des types abstraits

- ▶ Syntaxe
- ▶ Sémantique
- ▶ Implantation

# Type abstrait = Syntaxe + Sémantique

- ▶ La syntaxe définit « la forme ».
    - ▶ Par exemple, le profil syntaxique d'une fonction
    - ▶ function PUISS(A : INTEGER ; B : REAL) : REAL
    - ▶ définit la forme d'utilisation de cette fonction.
  - ▶ La sémantique définit « la signification ».
    - ▶ Par exemple,
      - ▶ le commentaire associé à la fonction qui dit :
      - ▶ « la fonction PUISS calculera le nombre B élevé à la puissance A »
      - ▶ ou
      - ▶ la formule : «  $\forall A \in \text{INTEGER}, \forall B \in \text{REAL}, \text{PUISS}(A,B) = e^A * \ln(B)$  »

# Syntaxe

- ▶ Il y a différentes façons de définir un TAD mais toutes utilisent la notion de signature (syntaxe d'utilisation du type) qui permet de définir toutes les opérations applicables ainsi que leur syntaxe.
- ▶ La signature du TAD = les sortes (ensemble des types de données impliquées dans la description du TAD) + les opérations

# Syntaxe

- ▶ La signature d'un TAD est composée des :
  - ▶ nom des ensembles d'éléments (naturel, réel, chaînes de caractères, ...) utilisés lors de la définition du TAD.
    - ▶ Ces ensembles de valeurs sont appelés sortes
    - ▶ Ce sont les types dans les langages de programmation
  - ▶ noms et profils des opérations relatives au type abstrait (addition, soustraction pour les entiers, ajout d'élément pour les listes, ...).

# Syntaxe

- ▶ Ces opérations sont divisées en trois catégories :
  - ▶ Les constructeurs qui permettent d'obtenir un nouvel élément d'un TAD
  - ▶ Les opérations modificatrices qui permettent de modifier un élément existant
  - ▶ Les opérations observatrices qui permettent d'obtenir des informations sur un élément existant sans pouvoir le modifier
- ▶ Le profil d'une opération définit les sortes des arguments et des résultats des opérations.

# Syntaxe : Exemple : Vecteur

- ▶ sorte Vecteur utilise Naturel, Réel
  - ▶ constructeur
  - ▶ VecteurVide(Naturel, Naturel)
  - ▶ opération
  - ▶ ième?(Naturel) → Réel
  - ▶ premierIndice?() → Naturel
  - ▶ dernierIndice?() → Naturel
  - ▶ changer-ième(Naturel, Réel)
  - ▶ fin sorte
- 
- Sorte en cours de définition
- Profil

# Remarques

- ▶ Cette signature définit la syntaxe du TAD mais elle n'est pas suffisante pour définir complètement le TAD Vecteur :
- ▶ ⇒ Elle ne définit pas la sémantique de ce type.
  
- ▶ Les noms donnés aux sortes et opérations permettent de donner une idée au lecteur mais sont totalement basées sur l'intuition.

# Les constructeurs

- ▶ Un constructeur permet d'obtenir un nouvel élément de la sorte que l'on est en train de définir.
- ▶ Un constructeur est caractérisé par son nom et un ensemble éventuel de paramètres en entrée (leur sorte)
- ▶ Exemple :
  - ▶ constructeur
  - ▶ VecteurVide(Naturel, Naturel)
- ▶ Exemple d'utilisation :
  - ▶ VecteurVide(1,10)
  - ▶ est un élément correspondant à un vecteur de 10 réels dont les indices sont 1 à 10 et dont toutes les cases sont initialisées à 0

# Les opérations observatrices

- ▶ Une opération observatrice permet d'obtenir une information sur un élément existant.
- ▶ Une opération observatrice est caractérisée par son nom (terminé par "?"), un ensemble éventuel de paramètres en entrée (leur sorte) et la sorte du résultat attendu.
- ▶ Exemple :
  - ▶ operations
  - ▶ ième?(Naturel) → Réel
  - ▶ premierIndice?() → Naturel
- ▶ Une opération observatrice s'applique toujours à un élément de la sorte à laquelle elle appartient.
- ▶ Son résultat appartient à la sorte du résultat précisé dans le profil.

# Les opérations observatrices

- ▶ L'application d'une opération observatrice ne peut en aucun cas modifier un élément
- ▶ Remarque : L'application d'une opération est matérialisée par le symbole ". "
- ▶ Exemple d'utilisation :
  - ▶ VecteurVide(1,10).premierIndice?()
  - ▶ VecteurVide(5,20).ième?(6)

# Les opérations modificatrices

- ▶ Une opération modificatrice permet de modifier un élément existant
- ▶ Une opération modificatrice est caractérisée par son nom et un ensemble éventuel de paramètres en entrée
- ▶ Exemple:
  - ▶ opérations
  - ▶              changer-ième(Naturel, Réel)
- ▶ Une opération modificatrice s'applique toujours à un élément de la sorte à laquelle elle appartient
- ▶ Son résultat est l'élément qu'elle vient de modifier

# Les opérations modificatrices

- ▶ L'application d'une opération modificatrice est matérialisée par le symbole "•"
- ▶ L'application d'une opération modificatrice modifie un élément
- ▶ Exemple d'utilisation :
  - ▶ VecteurVide(1,10). changer-ième(5, 3.141592)
  - ▶ VecteurVide(1,10).changer-ième(5, 3.141592).changer-ième(4, 37.2)

# Sortes prédéfinies

- ▶ On distingue dans la définition d'un TAD :
  - ▶ Les sortes prédéfinies qui sont les sortes utilisées pour la définition d'un TAD (Naturel et Réel dans le cas du Vecteur de nombres réels).
  - ▶ Les sortes définies qui sont les nouvelles sortes apportées par le TAD (Vecteur dans le même exemple).
- ▶ Remarque importante : Il est interdit d'ajouter des opérations internes à une sorte prédéfinie.

# Caractéristiques des types abstraits

- ▶ Syntaxe
- ▶ Sémantique
- ▶ Implantation

# Sémantique

- ▶ Outre la syntaxe, il faut définir la sémantique du type abstrait
- ▶ La sémantique va décrire les propriétés du TAD

# Sémantique des TAD

- ▶ Les opérations ne peuvent être employées que lorsque c'est possible.
- ▶ Il faut donc préciser leurs « conditions d'exécution »
  - ▶ **Notion de pré-condition**
- ▶ Les propriétés sont exprimées en combinant les opérations : la spécification doit être cohérente
  - ▶ **Notion d'assertion sémantique**

# Description de la sémantique d'un TAD

- ▶ La sémantique est généralement définie par un ensemble d'axiomes.
- ▶ La sémantique d'un TAD doit permettre de comprendre le comportement des opérations.
- ▶ La définition des propriétés permet de définir ce que souhaite l'utilisateur du TAD et donne des informations sur le comportement attendu
- ▶ La sémantique ne définit pas la structure sous-jacente qui permettra l'implantation du type abstrait

# Exemples d'axiomes

- ▶ Deux axiomes pour le TAD vecteur :
- ▶ Pour tout  $v \in \text{Vecteur}$ ,  $e \in \text{Réel}$ ,  $i, j \in \text{Naturel}$ ,
  - ▶ Pour  $i=j$ ,  $v.\text{changer-ième}(j, e).\text{ième?}(i) = e$
  - ▶ Pour  $i \neq j$ ,  $v.\text{changer-ième}(j, e).\text{ième?}(i) = v.\text{ième?}(i)$
- ▶ Ces axiomes caractérisent la sémantique de l'opération changer-ième du TAD vecteur

# Difficultés du choix des axiomes

- ▶ dire la vérité, rien que la vérité (c'est le contrat)
- ▶ pas de contradiction (consistance)
- ▶ suffisamment d'axiomes (complétude)
- ▶ pour les observateurs: déduire une valeur et une seule pour tous les arguments vérifiant les conditions d'exécution (pré-conditions)
- ▶ préciser le moins possible les opérations internes, pour laisser la liberté à l'implanteur

# Les préconditions

- ▶ Pour quelles valeurs de k ces deux axiomes sont-ils utilisables ?
  - ▶ VecteurVide(i,j).ième?(k)
  - ▶ v.ième?(k)
- ▶ Problème :
  - ▶ ième est une opération partielle qui n'a pas de sens hors de l'intervalle [i,j]
- ▶ => Besoin de définir une précondition pour ième? :
  - ▶  $k \in [i,j]$

# Les préconditions

- ▶ Une précondition est une expression de la forme :
- ▶ « il n'est possible d'appliquer l'opération `ième?(val)` au vecteur `v` que si :
  - ▶ `val ∈ [v.premierIndice?(), v.dernierIndice?()]` »
- ▶ Il n'est alors autorisé d'utiliser une opération ayant une pré-condition que si celle-ci est vérifiée
- ▶ L'appelant doit donc faire le test et vérifier que la condition est remplie avant d'appeler l'opération

# Les préconditions

- ▶ Conditions supposées vraies au moment de l'appel de l'opération
- ▶ Conditions qui décrivent l'état initial avant l'exécution de l'opération
- ▶ Conditions qui peuvent porter sur :
  - ▶ les valeurs des paramètres,
  - ▶ l'état de l'élément courant ou
  - ▶ toute information pouvant avoir une incidence sur l'exécution de l'opération

# Les postconditions

- ▶ Une post-condition exprime un résultat, un effet attendu de l'exécution d'une opération
- ▶ C'est une expression de la forme :
- ▶ « Si on retire le sommet d'une pile, on a diminué sa taille de 1 »
- ▶ Une opération ayant une post-condition fausse en sortie est une opération défectueuse (un des axiomes du TAD est alors violé).

# Les postconditions

- ▶ Conditions vérifiées à la fin de l'exécution de l'opération si l'opération a été appelée alors que ses préconditions étaient satisfaites.
- ▶ Préconditions + Postconditions constituent un contrat.
- ▶ Ces conditions peuvent porter sur l'état de l'élément courant, la valeur renvoyée ou toute information modifiée par l'opération.

# Complétude des axiomes pour des TAD simples

- ▶ Pour chaque opération observatrice OP, il faut écrire les axiomes définissant totalement OP appliquée à :
  - ▶ tous les constructeurs du TAD
  - ▶ toutes les opérations modificatrices du TAD en passant comme paramètres à OP et aux autres opérations toutes les valeurs possibles des types attendus

# Exercices TAD

- ▶ Feuille de TD sur les TAD

# Caractéristique des types abstraits

- ▶ Syntaxe
  - ▶ Sémantique
- 
- Implantation : Programmation Orientée Objets

# Ecrire un TAD en Langage de programmation

- ▶ Nous allons implanter les TAD en Python
- ▶ Pour chaque TAD sa représentation interne est décrite par une **classe**
- ▶ Comme un TAD, une classe fournit un ensemble de « **méthodes** » créatrices, modificatrices ou observatrices
- ▶ Une classe va donc permettre de générer une famille d'« **objets** », variables dont le type est décrit par le TAD
- ▶ Une classe est construite sur un TAD

# Exemple de la classe Date : Le TAD Date

**sorte** Date **utilise** Booléen

constructeurs

    date (Naturel, Chaine de caractères, Naturel)

opérations

    Antérieure?(Date) → Booléen

    Postérieure?(Date) → Booléen

    Demain?() → Date

    Hier?() → Date

    Affecter(Naturel, Chaine de caractères, Naturel)

Fin sorte.

# Structure d'une classe

- ▶ Une classe comporte une partie structurelle : description des données du TAD qu'elle implante
  - ▶ Une classe va être décrite par des ATTRIBUTS
- ▶ Une classe comporte une partie comportementale : description des opérations du TAD qu'elle implante
  - ▶ Une classe va être décrite par des METHODES

# Structure d'une classe : Attributs

- ▶ Exemple : La classe date qui correspond au TAD date, est constituée
- ▶ **class** date:
  - """"Classe définissant une date caractérisée par :
  - son jour
  - son mois
  - son année
  - .... """
- ▶ Pour créer les attributs de la classe, il faut passer par son constructeur.

# Construction d'un objet – Constructeur

- ▶ En PYTHON, un objet est créé par appel d'un **constructeur** de la classe.
  - ▶ En PYTHON, le constructeur s'appelle `_init_`
  - ▶ Il prend en paramètre au moins la variable `self`
  - ▶ Cette méthode est développée dans le corps de la classe.
- 
- ▶ Exemple : pour la classe date qui correspond au TAD date:

**class** Date :

```
def __init__(self): # Notre méthode constructeur
    """Constructeur de notre classe. Chaque attribut va être instancié avec une valeur par défaut...
        self.jour = 01
        self.mois = « Janvier»
        self.annee = 01
```

# Construction d'un objet – Constructeur

```
def __init__(self, j, m, a):  
    """Constructeur de notre classe avec des valeurs initiales passées en paramètres""""  
    self.jour = j  
    self.mois = m  
    self.annee = a
```

# Construction d'un objet - Instanciation

- ▶ La déclaration de la variable *aujourd'hui* de type Date, aura pour effet d'appeler le constructeur *\_init\_* de la classe Date pour créer l'objet *aujourd'hui*.
- ▶ Cet objet est appelé « **instance de la classe** ».  
`Aujourd'hui = Date();`
- ▶ Déclarer une variable = déclarer son type = créer une instance de ce type.
  - ▶ Ecrire `Aujourd'hui = Date()` revient à appeler la méthode *\_init\_* sur la variable *aujourd'hui*.
- ▶ Les attributs de *aujourd'hui* auront pour valeur :
  - ▶ Pour jour : 01
  - ▶ Pour mois : « janvier »
  - ▶ Pour année : 01

# Méthodes représentant les opérations du TAD

- ▶ Les définitions des méthodes s'appliquent à l'intérieur de la classe
- ▶ Opérations observatrices (anterieure?),
  - ▶ consultent les attributs de la classe
  - ▶ sont des fonctions (sous-programmes qui retournent un résultat)
- ▶ Opérations modificatrices (Affecter)
  - ▶ modifient les valeurs des attributs de la classe (Affecter).
  - ▶ sont des procédures (sous-programmes qui s'appliquent à leurs paramètres)

# Méthodes représentant les opérations du TAD : observatrices

```
def anterieure?(self, d):
    if d.annee < self.annee:
        return True
    elif d.annee == self.annee:
        if d.mois < self.mois:
            return True
        elif d.mois == self.mois:
            if d.jour < self.jour:
                return True
    return False
```

- ▶ Les méthodes observatrices permettent aussi d'obtenir la valeur des attributs

```
def annee?(self):
    return self.annee
```

# Méthodes représentant les opérations du TAD : modificatrices

```
def Affecter(self):  
    j = input("Entrez un jour : ")  
    self.jour = int(j)  
    self.mois = input("Entrez un mois : ")  
    a = input("Entrez une année : ")  
    self.annee = int(a)
```

- ▶ Les méthodes modificatrices permettent aussi d'assigner une valeur aux attributs

## Les méthodes – exercices

- ▶ Exercice : Ecrire demain?, afficher et postérieure?

# Utilisation d' une classe

## ► Accès en lecture aux attributs d'une classe :

### ► L'opérateur « . »

► ***id\_variable. id\_champ*** permet d'accéder au champ *id\_champ* de la variable *id\_variable* :

aujourd'hui Date(16, « mars », 2018);

print « demain nous serons le », demain.jour, « », demain.mois, « », demain.annee

## ► Accès en écriture aux attributs d'une classe :

► En dehors de la définition de classe, **les attributs ne sont pas accessibles en modification et seules les méthodes de la classe peuvent être appliquées à la variable typée par la classe => Il faut appeler une méthode modificatrice**

# Utilisation d' une classe

## ► Accès aux opérations d'une classe :

- ▶ Utilisation du sélecteur « . » pour accéder aux opérations et aux attributs d'un objet.
- ▶ En dehors de la définition de classe, les attributs ne sont pas accessibles et seules les méthodes de la classe peuvent être appliquées à l'objet.

unJour Date(17, « mars », 2018);

unJour.afficher() ;

## ► Exercice : afficher la date du lendemain de unJour

# Utilisation d' une classe

## ► Accès aux opérations d'une classe :

- ▶ Utilisation du sélecteur « . » pour accéder aux opérations et aux attributs d'un objet.
- ▶ En dehors de la définition de classe, les attributs ne sont pas accessibles et seules les méthodes de la classe peuvent être appliquées à l'objet.

```
unJour Date(17, « mars », 2018);
```

```
unJour.afficher();
```

## ► Exercice : afficher la date du lendemain de unJour

```
"""affichage de la date du lendemain de unJour : """
```

```
demain = unJour.lendemain()
```

```
print(« le lendemain du », unJour.afficher(), « sera le », demain.afficher())
```

# Programmation Orientée Objets

- ▶ Qu'est-ce que ça signifie ?

# Les principes de l'approche par objets

- ▶ Principe fondamental :

- ▶ Programme => Organisme
- ▶ Fonctions assurées ? => Objets manipulés ?
- ▶ Ce que fait le système ? => Qui fait quoi dans le système ?  
(B. Meyer 1990)

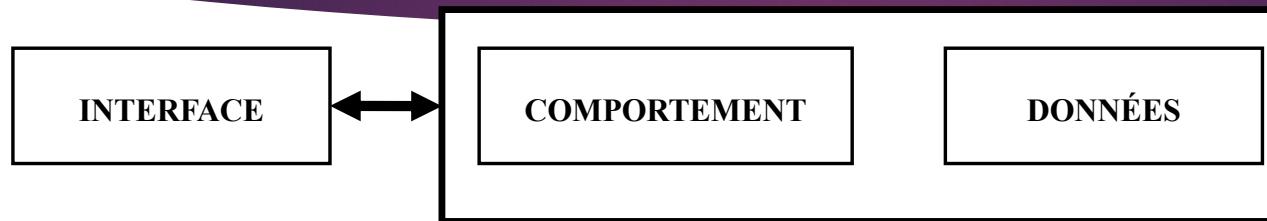
# Encapsulation

**Comportement**

**Données**

- ▶ Données : Description de la structure et de l'état :
  - ▶ Ensemble d'attributs (Variables)
- ▶ Comportement : Ensemble de méthodes (opérations, routines) :
  - ▶ Accès aux données
  - ▶ Envoi de messages

# Encapsulation



- ▶ Seules les méthodes constituant le comportement sont habilitées à modifier les données
- ▶ Activation d'une méthode :
  - ▶ Uniquement via un sélecteur de l'interface
  - ▶ Une primitive d'une classe peut être :
    - ▶ publique (accessible à tout appelant)
    - ▶ Protégée (accessible localement)
    - ▶ privée (accessible uniquement en interne)

# Le constructeur de base : La Classe

- ▶ **Concept fondamental de la technologie à objets**
- ▶ Entité conceptuelle décrivant un ensemble d'objets et chargée de les générer
- ▶ Construite sur la base d'un Type Abstrait de Données (doté éventuellement d'une implantation partielle)
- ▶ décrit les propriétés communes à un ensemble d'objets (instances de classe) :
  - ▶ Structure (Attributs) et comportement (Méthodes).
- ▶ Exemple : Classe PERSONNE :
  - ▶ Attributs : Nom, Prénom, Adresse, Date\_Naissance
  - ▶ Méthode : Vieillir, Déménager, Age?

# La classe comme Type Abstrait De Données

- ▶ Le TAD Décrit la spécification de la classe
- ▶ Description : en quatre parties :
  - ▶ Types : les types en cours de spécification (Liste, Pile, Personne,...)
  - ▶ Fonctions : nom et signature des opérations applicables sur les types
  - ▶ Axiomes : définition implicite des fonctions
  - ▶ Préconditions : précision du domaine des fonctions partielles

# La classe comme Module

- ▶ Point de vue du programmeur : Une classe est composée de :
  - ▶ Attributs : Stockage d'informations (conservant l'état du représentant)
  - ▶ Méthodes : Unités de calcul (fonctions ou procédures)
- ▶ Point de vue de l'utilisateur : Une classe est un ensemble de :
  - ▶ Requêtes : Informations qui peuvent être demandées à la classe
  - ▶ Commandes : Services (opérations) réalisés par la classe
- ▶ Un attribut n'est accessible qu'en lecture
- ▶ Visibilité : Définition de l'interface (Encapsulation et masquage d'information)
  - ▶ Définir un droit d'accès aux informations d'une classe : public, privé, protégé.

# La classe comme Type

- ▶ Une classe définit un type de données.
- ▶ Elle permet de définir des instances, les objets
- ▶ Elle comprend :
  - ▶ Un ensemble de ressources (Attributs)
  - ▶ Un ensemble d'opérations d'accès et de modification de ces ressources
  - ▶ Des propriétés qui la caractérisent
- ▶ Les opérations fournies par la classe en tant que module sont les opérations applicables sur les instances de cette classe vue comme un type
- ▶ Remarque : Dans un langage à objets, il est souhaitable que toute classe soit un type et que tout type soit une classe.

# Les Objets : Construction/Destruction

## ► **Instanciation :**

**Opération de création dynamique d'un objet, exemplaire conforme à la classe qui le décrit**

- Un objet est une instance, à l'exécution, d'une classe
- Règle fondamentale : Chaque objet est instance d'une classe
  - Ex.: Nov17 Date(17, «novembre», 2018);
    - création d'une instance Nov17 de la classe Date
- Chaque objet a son propre état (valeurs des attributs contenus dans la classe).
- Tous les objets instances d'une même classe ont le même comportement
  - (le comportement associé au type correspondant)

# Les Objets : Construction/Destruction

- ▶ Création d'un objet (appel au constructeur)
  - ▶ Réservation de la zone mémoire nécessaire au stockage de la structure de données (compilateur)
  - ▶ Initialisation de cette mémoire (valeurs fournies dans le constructeur `_init` défini par le développeur)
- ▶ La déclaration de la variable fait appel au constructeur `_init` qui crée l'objet
  - ▶ La déclaration de la variable `Nov17` de type `Date`, aura pour effet d'appeler le constructeur `_init_` de la classe `Date` pour créer l'objet `Nov17`.

# Structure des objets

- ▶ Attribut :
  - ▶ Atomique : désigne une valeur simple (title, titre,...)
  - ▶ Relationnel : établit une relation entre deux objets (auteur)
  - ▶ Composition, Agrégation : identifie les composants d'un objet (author)
- ▶ Identité d'un objet : Chaque objet a une identité unique : son adresse en mémoire.
- ▶ Référence : lien vers un objet.

# Structure d'un programme

- ▶ Statiquement : Un système est un ensemble de classes reliées par des relations.
- ▶ Dynamiquement : Un système est un ensemble d'objets communicant.
- ▶ L'exécution du programme principal correspond à l'activation d'une des méthodes de création de l'instance d'une classe dite classe racine du système : l'objet racine.
- ▶ En Python, c'est l'exécution du programme principal qui joue ce rôle

# Communication entre objets : envoi de messages

- ▶ Toute primitive (requête ou commande) est appliquée à un objet :
  - ▶ Ex. mon\_collier.afficher()
- ▶ En Python, une primitive est appliquée à l'objet courant (`self`) :
  - ▶ Ex. : `self.contenu.pop(0)`
- ▶ Seule une primitive de la classe (en tant que type) de l'objet peut lui être appliquée

# Exemple (2)

```
class Lifo():
    # On utilise une liste pour implanter la file : []
    #contenu
    #maxfile

    def __init__(self,maxfile=None):
        self.contenu=[]
        self.maxfile=maxfile

    def filevide(self) :
        if len(self.contenu)==0 : # teste si la taille de la pile file = 0
            return True # retourne True si la taille de la pile file = 0
        else :
            return False # retourne False si la taille de la file > 0

    def enfiler(self,e) :
        if len(self.contenu)==self.maxfile:
            print("File pleine: Impossible d'ajouter l'élément %s" %e)
        else:
            self.contenu.append(e) # ajouter l'élément e en queue de file

    def defiler(self) :
        if not self.filevide() :
            self.contenu.pop(0) # enlever l'élément en tête si la file n'est pas vide
        else :
            print("Pile vide")

    def taille(self) :
        return len(self.contenu) # retourne la taille de la pile

    def tete(self) :
        return self.contenu[0] # retourne le premier élément de contenu

    def afficher(self):
        if not self.filevide():
            for e in self.contenu:
                print( e)
        else :
            print("Pile vide")

    def purger (self):
        del self.contenu[:]
```

# Exercice

Ecrire un programme qui ajoute une perle verte au milieu d'un collier de 10 perles bleus

# Exemple de programme (1)

```
from lifo import Lifo
    #on crée la file contenant les perles bleues (ici des 'B')
mon_collier = Lifo()

for i in range (0,10):
    mon_collier.enfiler('B')
# on affiche le collier avec ses 10 'B'
mon_collier.afficher()

# On va enlever les 5 premières pour pouvoir mettre la 'V'
# on les met de côté dans une file temporaire
ma_file_temp=Lifo()

for i in range (0,5):
    ma_file_temp.enfiler(mon_collier.tete())
    mon_collier.defiler()

#On enfile la 'V'
mon_collier.enfiler('V')
#on remet dans le collier les perles B mises de côté dans la file temporaire
for i in range (0,5):
    mon_collier.enfiler(ma_file_temp.tete())
    ma_file_temp.defiler()

# on affiche le collier résultat
mon_collier.afficher()
```

# Les relations fondamentales

- ▶ L'Utilisation (relation Client/serveur) induisant la communication entre objets.
  - ▶ A un impact sur la dynamique du système
- ▶ L'Héritage (relation conceptuelle) induisant la hiérarchie des classes
  - ▶ A un impact sur le modèle statique
  - ▶ Permet la mise en oeuvre de concepts avancés (polymorphisme,...)
  - ▶ Implante la réutilisation et la classification,

# Relations d'utilisation

- ▶ Définition
- ▶ Une classe A utilise une classe B (A est cliente de B), lorsqu'elle définit une entité de type B sous l'une des formes suivantes :
  - ▶ Un attribut (variable de classe)
  - ▶ La valeur de retour d'une fonction
  - ▶ Un paramètre d'opération
  - ▶ Une variable locale

# Les différentes relations d'utilisation

- ▶ Association :
  - ▶ Une classe A est associée à une classe B si A utilise B et si les instances de B sont indépendantes des instances de A :
  - ▶ Une instance particulière peut appartenir à plusieurs associations et par conséquent être partagée.
  - ▶ L'association est bidirectionnelle.
- ▶ Ex. : Un avion est associé à une tour de contrôle qui gère plusieurs avions.

# Les différentes relations d'utilisation

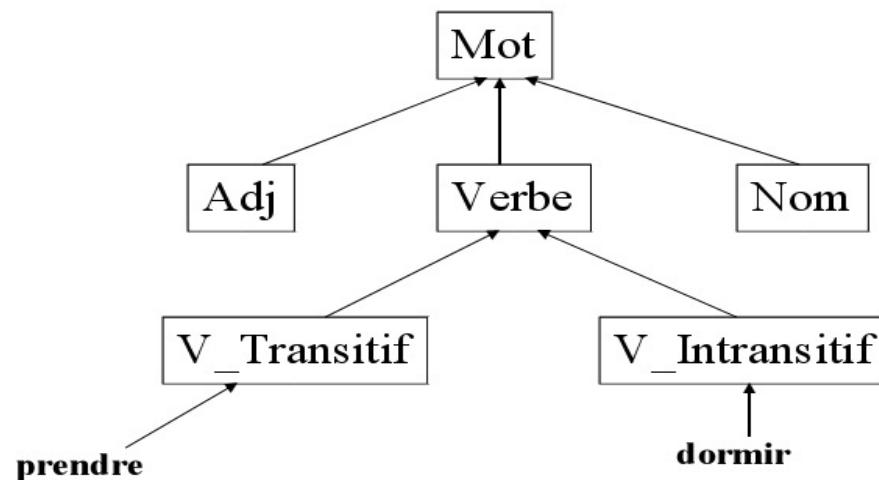
- ▶ Composition :
  - ▶ Une classe A est composée d'une classe B si les instances de B sont liées aux instances de A (même durée de vie)
- ▶ Ex. : Une voiture est composée de 4 roues.

# Les différentes relations d'utilisation : Exercices

- ▶ Feuilles de TDs 2 et 3
- ▶ Feuille de TD 4 jusqu'à la question 3 inclue

# L'héritage

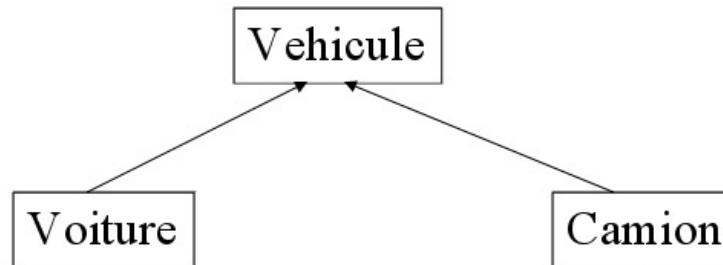
- On peut souvent classer les objets dans des arborescences de type de plus en plus génériques



# Exemple : généralisation de la classe Voiture

- La classe **Voiture** représente toutes sortes de voitures possibles
- On pourrait définir un camion comme une voiture très longue, très haute, etc.
- Mais un camion a des spécificités vis-à-vis des voitures : remorque, cargaison, boîte noire, etc.
- On pourrait créer une classe **Camion** qui ressemble à la classe **Voiture**
- Mais on ne veut pas réécrire tout ce qu'elles ont en commun

# Exemple : généralisation de la classe Voiture : Solution



- La classe **Vehicule** contient tout ce qu'il y a de commun à **Camion** et **Voiture**
- **Camion** ne contient que ce qu'il y a de spécifique aux camions

# Objectif principal de l'héritage

- On souhaite ne décrire qu'une seule fois le même traitement lorsqu'il s'applique à plusieurs classes
- Evite de recopier (notamment les modifications)
- On crée une classe plus générique à laquelle s'applique le traitement
- **Toutes les classes plus spécifiques, héritant de cette classe, héritent de ce traitement, elles peuvent l'exécuter**
- Le traitement n'est décrit qu'au niveau de la classe mère
- Les classes filles contiennent d'autres traitements plus spécifiques

# Relation d'Héritage

- ▶ Définition : Copie virtuelle des caractéristiques de classes existantes dans la définition d'une nouvelle classe.
- ▶ But :
  - ▶ Partage de connaissances
  - ▶ Définition de relations de sous-typage entre plusieurs classes
- ▶ Principe : Inclusion de modèles conceptuels
- ▶ Terminologie :
  - ▶ héritier, classe dérivée, sous-classe, descendant
  - ▶ parent, classe de base, sur-classe, ancêtre, super classe,...

# Héritage : Mise en œuvre en Python

- ▶ Class Voiture (Vehicule)
- ▶ Ve=vehicule() # Creation d'un objet de type vehicule
- ▶ Vo=voiture() # Creation d'un objet de type voiture ayant les caractéristiques de Vehicule

# Héritage : Mise en œuvre en Python

```
[class Personne:  
    """Classe représentant une personne"""  
    def __init__(self, nom):  
        """Constructeur de notre classe"""  
        self.nom = nom  
        self.prenom = "X"  
  
    def afficher(self):  
        print(self.prenom, self.nom)  
  
[class Etudiant(Personne):  
    """Classe définissant un étudiant.  
    Elle hérite de la classe Personne"""  
  
    def __init__(self, nom, noetu):  
        """Un étudiant se définit par son nom et son numéro"""  
        self.nom = nom  
        self.noetu = noetu  
    def afficher(self):  
        print(self.nom, self.prenom, self.noetu)
```

# Usage de l'héritage

- ▶ Une classe donnée hérite des attributs et des méthodes de sa classe mère
- ▶ On n'a donc pas besoin de les y écrire à nouveau, excepté le constructeur
- ▶ On s'assurera que tous les attributs ont bien été initialisés dans le constructeur de la classe fille (on pourra appeler le constructeur de la classe parente)
- ▶ On peut cependant redéfinir une méthode de la classe mère dans la classe fille, pour adapter le comportement correspondant à la classe descendante

# Héritage et redéfinition

- ▶ Il est possible de rajouter lors de la définition du constructeur de la classe fille, des paramètres au constructeur de la classe mère.
- ▶ Il est conseillé de redéfinir le constructeur (qui pourra appeler celui de la classe parente) (T 88)
- ▶ On peut redéfinir une méthode pour adapter son comportement à la classe descendante (T89)
- ▶ Si une méthode appelée sur la classe descendante n'y est pas (re)définie, c'est celle de la classe parente qui s'exécute (T90)

# Héritage – redefinition : Mise en œuvre en Python

```
class Personne:  
    """Classe représentant une personne"""  
    def __init__(self, nom):  
        """Constructeur de notre classe"""  
        self.nom = nom  
        self.prenom = "X"  
    def afficher(self):  
        print(self.prenom, self.nom)  
  
class Etudiant(Personne):  
    """Classe définissant un étudiant.  
    Elle hérite de la classe Personne"""  
  
    def __init__(self, nom, noetu):  
        """Un étudiant se définit par son nom et son numéro"""  
        Personne.__init__(self, nom)  
        self.noetu = noetu  
    def afficher(self):  
        print(self.nom, self.prenom, self.noetu)
```

# Héritage – redefinition : Mise en œuvre en Python

```
class Personne:  
    """Classe représentant une personne"""  
    def __init__(self, nom):  
        """Constructeur de notre classe"""  
        self.nom = nom  
        self.prenom = "X"  
    def afficher(self):  
        print(self.prenom, self.nom)  
  
class Etudiant(Personne):  
    """Classe définissant un étudiant.  
    Elle hérite de la classe Personne"""  
  
    def __init__(self, nom, noetu):  
        """Un étudiant se définit par son nom et son numéro"""  
        self.nom=nom  
        self.noetu = noetu  
    def afficher(self):  
        Personne.afficher()  
        print(self.noetu)
```

# Héritage – redefinition : Mise en œuvre en Python

```
class Personne:  
    """Classe représentant une personne"""  
    def __init__(self, nom):  
        """Constructeur de notre classe"""  
        self.nom = nom  
        self.prenom = "X"  
    def afficher(self):  
        print(self.prenom, self.nom)  
  
class Etudiant(Personne):  
    """Classe définissant un étudiant.  
    Elle hérite de la classe Personne"""  
  
    def __init__(self, nom, noetu):  
        """Un étudiant se définit par son nom et son numéro"""  
        self.nom=nom  
        self.noetu = noetu
```

## Usage de l'héritage (suite)

- Un objet de type **Voiture** peut utiliser toutes les méthodes de la classe **Vehicule**
- Il doit disposer d'une valeur pour tous les attributs de la classe **Vehicule**
- A tout moment, une méthode qui utilise un objet de type **Vehicule** peut manipuler un objet de type **Voiture** en guise de **Vehicule**
- Cette dernière propriété est le **polymorphisme**.

# Héritage et Polymorphisme : Mise en œuvre en Python

- ▶ Class Etudiant(Personne)
  - ▶ Max =Etudiant() // Creation d'un objet de type Etudiant
  - ▶ P= Personne() // Creation d'un objet de type Personne
  - ▶ P=Max // Affectation de Etudiant à Personne
  - ▶ ~~Max = P~~ // FAUX : On ne peut affecter qu'une instance de classe fille à une instance de classe mère
- ▶ Le polymorphisme permet à une instance de la classe Personne de devenir dynamiquement un Etudiant
- ▶ Utile pour gérer des collections génériques (ex. Figures géométriques)

# Héritage vs. Utilisation

- ▶ Sémantique associée généralement
  - ▶ héritage : est\_un
  - ▶ utilisation : a, est\_composé\_de, est\_associé\_à, est\_un ... \_et\_ ....
- ▶ Usage de l'héritage :
  - ▶ propager les propriétés,
  - ▶ utiliser le polymorphisme,
  - ▶ réutiliser
- ▶ Usage de l' utilisation
  - ▶ changer de type dynamique (ex. : équipe de football)
- ▶ Remarque : être c'est avoir un peu : on peut remplacer l'héritage par de la délégation (distorsions conceptuelles, lourdeur de programmation). Pas l'inverse !

# Héritage et appel de méthode

- ▶ La méthode invoquée sur un objet est par défaut celle de sa classe d'instanciation
- ▶ Si elle n'est pas définie, mais obtenue par héritage, c'est celle de la classe parente qui s'exécute
- ▶ Attention cependant à ce qu'elle existe bien dans la classe parente

# Types statiques vs. Types dynamiques

- ▶ Type statique : type de déclaration (ex. : Mot)
- ▶ Type dynamique : type réel à l'exécution (ex. : Verbe, nom, ...)
- ▶ Statiquement : à la compilation : contrôle que la méthode appelée est déclarée dans la classe de définition de l'objet sur lequel elle est invoquée
- ▶ Dynamiquement : à l'exécution : la méthode qui s'applique est celle qui est présente dans la classe définissant le type dynamique de l'objet sur lequel elle est invoquée (ou dans ses classes parentes, si elle n'existe pas)

# Types statiques vs. Types dynamiques

## ► Exemple :

```
Martin = Personne("Martin")
Moi = Etudiant("Max", 123456)
Martin.afficher()
Moi.afficher()
Martin = Moi
Martin.afficher()
```

- Martin a pour type statique Personne et pour type dynamique Etudiant (le type de Moi)
- On va donc chercher à lui appliquer les méthodes d'Etudiant

# Types statiques vs. Types dynamiques

## ► Exemple :

```
Martin = Personne("Martin")
Moi = Etudiant("Max", 123456)
Martin.afficher()
Moi.afficher()
Martin = Moi
Martin.afficher()
```

The diagram illustrates the state of variables after the execution of the provided code. It consists of three columns of text. The first column contains the variable names and their initial values or types. The second column shows arrows pointing from each variable to its corresponding value in the third column. The third column displays the final state of the variables after the assignment `Martin = Moi` is executed.

Martin = Personne("Martin")	→	<b>Martin</b>
Moi = Etudiant("Max", 123456)	→	<b>Max</b> 123456
Martin.afficher()	→	
Moi.afficher()	→	
Martin = Moi	→	
Martin.afficher()	→	<b>Max</b> 123456

# Classes abstraites

- ▶ Définitions :
  - ▶ Une classe est abstraite si elle n'est pas complètement définie (elle contient des primitives retardées).
  - ▶ Une primitive est retardée ou abstraite si elle n'a pas de code.
- ▶ Principe : Décrire une structure de données abstraite, indépendamment de l'implantation
- ▶ Intérêt : Factoriser des propriétés communes à un ensemble de classes (à lier au polymorphisme et à la liaison dynamique), laisser le choix d'implantation aux descendants, forcer les descendantes à implanter un comportement, ...

# Classes abstraites

- ▶ Conséquences :
  - ▶ Une classe abstraite ne peut pas être instanciée
  - ▶ Une classe descendante d'une classe abstraite doit définir toutes les primitives retardées pour pouvoir être instanciée (sans quoi elle reste abstraite)
- ▶ Exemples : notions de Figures (classer des figures par surface croissante), d'analyseur syntaxique (mettre des mots au pluriel), ...

# Abstraction : Mise en œuvre en Python

- ▶ Les classes abstraites ne font pas partie du cœur même de Python
- ▶ Elles sont disponibles via un module de la bibliothèque standard, `abc` (*Abstract Base Classes*).
  - ▶ Ce module contient :
    - ▶ la classe `ABC` pour définir respectivement une classe abstraite et
    - ▶ le décorateur `abstractmethod`, pour définir une méthode abstraite de cette classe.
- ▶ Une classe abstraite doit donc hériter d'`ABC`,
  - ▶ et utiliser le décorateur `abstractmethod` pour définir ses méthodes abstraites.

# Abstraction : Mise en œuvre en Python

```
import abc

class Polygone(abc.ABC):
    """Classe polygone abstraite"""
    @abc.abstractmethod
    def perimetre(self):
        pass
    @abc.abstractmethod
    def afficher(self):
        pass

class carre (Polygone):
    """Classe définissant un carre comme un polygone particulier"""

    def __init__(self):
        """Un étudiant se définit par son nom et son numéro"""
        self.nbcotes=4
        self.longueur = int(input("longueur d'un cote ? "))
    def afficher(self):
        print ("le périmètre est : ")
        print(self.perimetre())
    def perimetre(self):
        return self.longueur*self.nbcotes

C = carre()
C.afficher()
```

# Abstraction : Mise en œuvre en Python

```
import abc

class Polygone(abc.ABC):
    """Classe polygone abstraite"""
    @abc.abstractmethod
    def perimetre(self):
        pass
    @abc.abstractmethod
    def afficher(self):
        pass

class carre (Polygone):
    """Classe définissant un carre comme un polygone particulier"""

    def __init__(self):
        """Un étudiant se définit par son nom et son numéro"""
        self.nbcotes=4
        self.longueur = input("longueur d'un cote ? ")
    def afficher(self):
        print(self.perimetre())
    def perimetre(self):
        return self.longueur*self.nbcotes

P = Polygone()
P.perimetre()
P.afficher()
C = carre()
C.afficher()
```

P = Polygone()  
P.perimetre()  
P.afficher()  
TypeError: Can't instantiate abstract class Polygone with abstract methods afficher, perimetre

# Conclusion

- ▶ Qu'est-ce qu'un objet?
- ▶ Qu'est-ce qu'un système à objets?
- ▶ Qu'est-ce que la technologie objet ?
- ▶ Quelles relations lient les objets?
- ▶ Règles du développement par objets

# Qu'est-ce qu'un objet?

- ▶ Entité dynamique
- ▶ Instance d'une classe
  - ▶ => Typé : on ne peut lui appliquer que les méthodes décrites dans le type
  - ▶ => Modulaire : on ne peut lui appliquer que les méthodes décrites dans l'interface de la classe
- ▶ Communique avec d'autres objets (relation Client/Serveur) pour implémenter le système

# Qu'est-ce qu'un système dans un L.O.O?

- ▶ Une organisation constituée d'objets qui communiquent pour réaliser les fonctionnalités du système

# Qu'est-ce que la technologie objet ?

- ▶ Un principe d'architecture :  
    MODULE = TYPE
- ▶ Un principe d'exécution :  
    LA COMMUNICATION
- ▶ Une discipline épistémologique :  
    L'ABSTRACTION
- ▶ Une règle de classification :  
    L'HERITAGE
- ▶ Une exigence de validité :  
    LA CONCEPTION CONTRACTUELLE
- ▶ Une obligation d'ingénieur :  
    LA REUTILISABILITE et L'EXTENSIBILITE

# Quelles relations lient les objets?

- ▶ Une relation client/serveur fondée sur la communication via l'envoi de messages paramétrés.
- ▶ Cette relation est instance de la relation d'utilisation définie entre les classes d'instanciation des objets, classes constituant l'architecture physique et statique du système

# Règles du développement par objets

- ▶ Tout objet est instance d'une classe
- ▶ Le Système est constitué d'un ensemble de classes
- ▶ Le lancement de l'exécution du système est la création de l'objet racine
- ▶ Une méthode est activée par envoi de message à un objet
- ▶ Les attributs d'un objet ne sont accessibles que par les méthodes définies localement ou par héritage sur cette classe
- ▶ Par rapport à l'approche fonctionnelle, il faut opérer les changements suivants :
  - ▶      Programme                  Organisme
  - ▶      Fonctions assurées       Objets manipulés
  - ▶      Ce que fait le système       A qui il le fait ->

QUI FAIT QUOI ?

# Conclusion

- ▶ Notions intrinsèques à maîtriser :
  - ▶ Principes généraux de fonctionnement
  - ▶ Classes et objets
  - ▶ Relations inter-classes, relations inter-objets
  - ▶ Abstraction
  - ▶ Polymorphisme
  - ▶ Programmation contractuelle
- ▶ Quels problèmes la technologie à objets permet-elle de résoudre?
- ▶ Quelles sont ses limites?

# Exercices

- ▶ Feuille de TD 4 Question 4
- ▶ Feuille de TD 5