



RÉSUMÉ THÉORIQUE – FILIÈRE DÉVELOPPEMENT DIGITAL
M103 - Programmer en Orienté Objet



100 heures



SOMMAIRE



01 - APPRÉHENDER LE PARADIGME DE LA POO

- Introduire la POO
- Définir une classe
- Créer un objet
- Connaître l'encapsulation
- Manipuler les méthodes

02 - CONNAÎTRE LES PRINCIPAUX PILIERS DE LA POO

- Définir l'héritage
- Définir le polymorphisme
- Caractériser l'abstraction
- Manipuler les interfaces

03 - CODER DES SOLUTIONS ORIENTÉES OBJET

- Coder une solution orientée objet
- Manipuler les données
- Utiliser les expressions régulières
- Administrer les exceptions

04 - MANIPULER LES MODULES ET LES BIBLIOTHÈQUES

- Manipuler les modules
- Manipuler les bibliothèques

MODALITÉS PÉDAGOGIQUES



1

LE GUIDE DE SOUTIEN

Il contient le résumé théorique et le manuel des travaux pratiques.



2

LA VERSION PDF

Une version PDF est mise en ligne sur l'espace apprenant et formateur de la plateforme WebForce Life.



3

DES CONTENUS TÉLÉCHARGEABLES

Les fiches de résumés ou des exercices sont téléchargeables sur WebForce Life



4

LA VERSION PDF

Une version PDF est mise en ligne sur l'espace apprenant et formateur de la plateforme WebForce Life.



5

DES RESSOURCES EN LIGNES

Les ressources sont consultables en synchrone et en asynchrone pour s'adapter au rythme de l'apprentissage



PARTIE 1

Appréhender le paradigme de la POO

Dans ce module, vous allez :

- Comprendre le principe de la POO
- Maîtriser la définition d'une classe
- Maîtriser le concept d'objet
- Connaitre le principe de l'encapsulation
- Savoir manipuler les méthodes



 05 heures



CHAPITRE 1

Introduire la POO

Ce que vous allez apprendre dans ce chapitre :

- Acquérir une compréhension de la méthode de POO
- Connaitre le principe de la POO
- Citer ses avantages par rapport aux autres paradigmes de programmation



01 heure



CHAPITRE 1

Introduire la POO

1. **Introduction à la programmation Orientée Objet**
2. Brève historique de l'évolution des langages de programmation Orientée Objet
3. Connaissance des avantages de la POO par rapport aux autres paradigmes

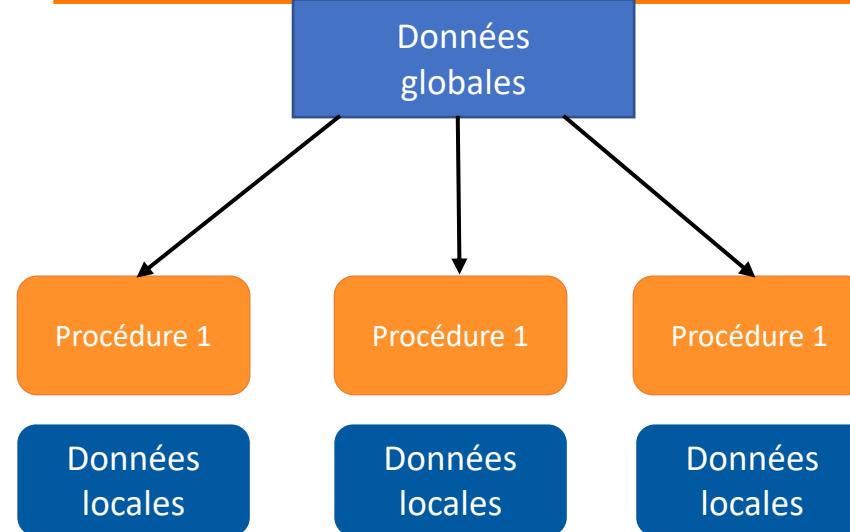
01 - Introduire la POO

Introduction à la programmation Orientée Objet

Programmation procédurale

- Dans **la programmation procédurale**, le programme est divisé en des blocs d'instructions, appelés **procédures** ou **fonctions**.
- Ensuite, pour résoudre chaque problématique, une ou plusieurs procédures/fonctions sont utilisées
- Dans **la programmation procédurale**, les données et le traitement de ces données sont **séparées**

Programmes = Procédures/Fonctions + Données



Programmation procédurale

Dans la programmation procédurale :

- Le programme crée des données
- Les procédures et les fonctions manipulent ces données-là.
- **Programmer dans ce cas revenait à :**
- définir un certain nombre de variables de types différents (entier, chaîne de caractères, structures, tableaux, etc.)
- écrire des procédures ou fonctions pour les manipuler

Inconvénients :

- Difficulté de réutilisation du code
- Difficulté de la maintenance de grandes applications

CalculSomme100PremierNombre
Var

i,S: entier

Début /*début de la procédure*/

S:=0

Pour i:=1 à 100 Faire

S:=S+1

FinPour

Ecrire(" La somme des 100 premiers
nombres est ",S);

Fin /*Fin de la procédure*/

Début /*début algorithme*/

Somme

Fin /*fin algorithme*/



01 - Introduire la POO

Introduction à la programmation Orientée Objet



Programmation orientée objet

- Séparation (données, procédures) est elle utile?
- Pourquoi privilégier les procédures sur les données, plutôt que les fonctions ?
- Pourquoi ne pas considérer que les programmes devraient ne manipuler que des objets qui renferment les opérations (procédures) qu'on a défini (dans ces objets) ?

- Les **langages orientés objets** proposent des réponses à ces interrogations.
- Ils se basent sur un seul type d'entités : Ce sont **les objets**
- Ainsi, un **objet** est une entité qui regroupe des propriétés **statiques** et d'autres **dynamiques**.
- Un programme est constitué d'un ensemble d'objets chacun disposant d'une partie procédures (traitement) et d'une partie données.

01 - Introduire la POO

Introduction à la programmation Orientée Objet



Programmation procédurale

Que doit faire le système
dans sa globalité ?



Programmation Orientée Objet

Quelles entités mon
programme va manipuler ?

De quels modules doit être
composé mon programme ?

01 - Introduire la POO

Introduction à la programmation Orientée Objet



Programmation procédurale



Programmation Orientée Objet

Quelles procédures dois-je
implémenter ?
Créditer un compte ?
Débiter un compte ?

Programme de Gestion
des comptes bancaires

Comment se présente (structure)
un compte bancaire ?



CHAPITRE 1

Introduire la POO

1. Introduction à la programmation Orientée Objet
2. **Brève historique de l'évolution des langages de programmation Orientée Objet**
3. Connaissance des avantages de la POO par rapport aux autres paradigmes

01 - Introduire la POO

Brève historique de l'évolution des POO

Les années 70

Les années 80

Les années 90

De nos jours

- Les concepts de la POO ont vu le jour vers la fin de années 1960.
- Les premiers langages de programmation véritablement orientés objet ont été .
 - **Simula** (1966) : a été le premier langage à regrouper données et procédures.
 - **Simula I** (1972) : a formalisé les notions d'objet et de classe.
 - **Smalltalk** (1972) : a généralisé de la notion d'objet.

01 - Introduire la POO

Brève historique de l'évolution des POO



Les années 70

Les années 80

Les années 90

De nos jours

- Un peu plus d'une décennie plus tard (80'), de nombreux langages basés sur la POO ont été publiés, tel que :
 - Eiffel créé par Bertrand Meyer
 - C++ créé par Bjarne Stroustrup
 - Objective C, qu'on utilise notamment dans le développement iOS.

01 - Introduire la POO

Brève historique de l'évolution des POO

Les années 70

Les années 80

Les années 90

De nos jours

Le **23 mai 1995** a marqué la naissance du célèbre langage **Java** par la société Sun Microsystems. Il a été racheté, en 2009, par le géant Oracle.

01 - Introduire la POO

Brève historique de l'évolution des POO

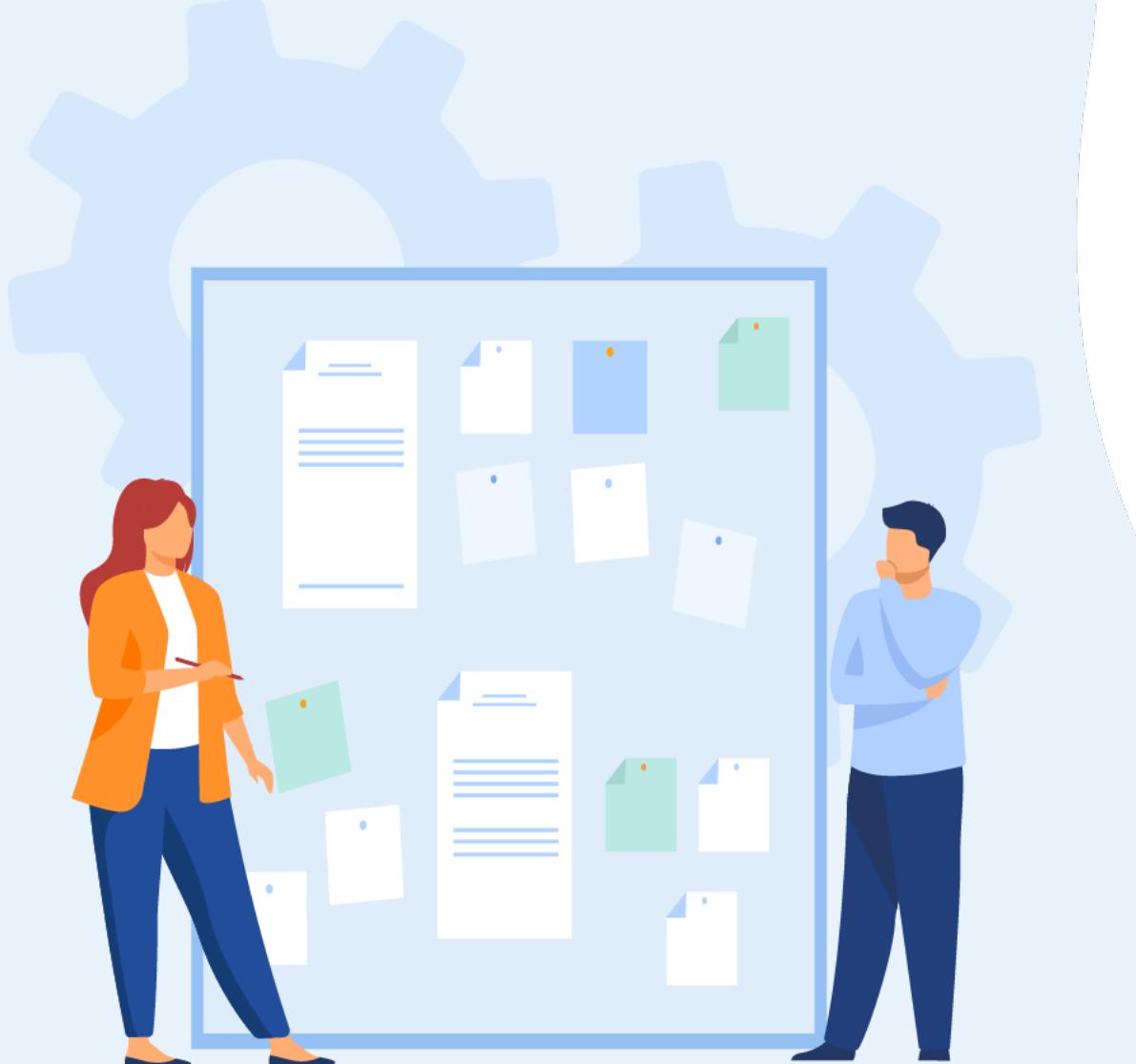
Les années 70

Les années 80

Les années 90

De nos jours

- Aujourd'hui, nous avons le choix entre une grande panoplie de langages basés sur les principes de la POO tels que :
 - **PHP (à partir de la version 5),**
 - **C#,**
 - **Ruby,**
 - **Python, etc.**



CHAPITRE 1

Introduire la POO

1. Introduction à la programmation Orientée Objet
2. Brève historique de l'évolution des langages de programmation Orientée Objet
3. **Connaissance des avantages de la POO par rapport aux autres paradigmes**

01 - Introduire la POO

Connaissance des avantages de la POO par rapport aux autres paradigmes



Objectif et avantages de la POO

Objectif : Faciliter la conception, l'exploitation et la maintenabilité de gros programmes.

Avantages :

- **Modularité** : Chaque classe représente un module compact renfermant des données et un ensemble de comportement. Ceci donne lieu à des applications moins complexes.
- **Abstraction** :

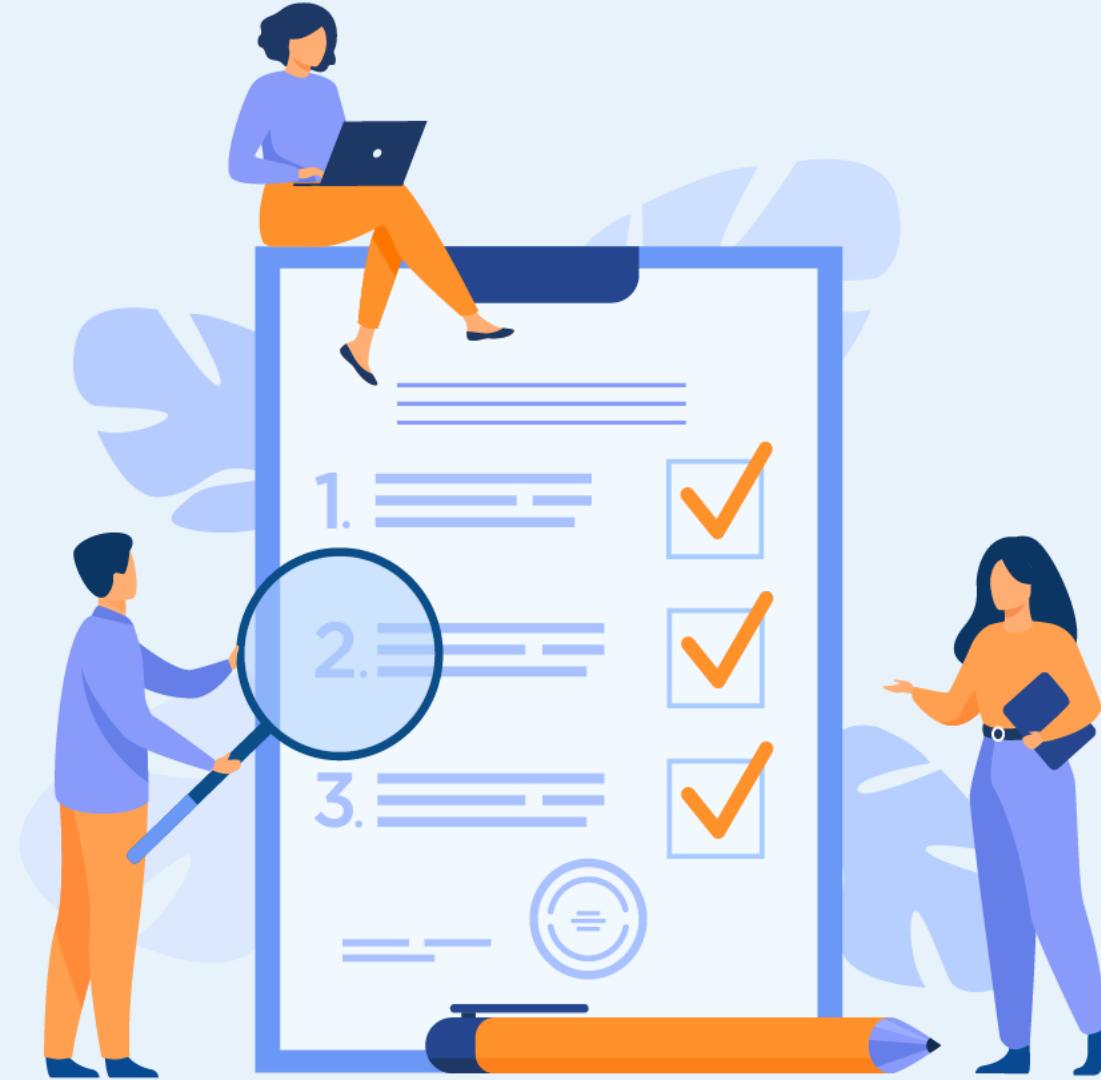
- Les objets sont des entités qui sont semblables à deux qu'on rencontre dans le monde réel (un compte, un étudiant, une entreprise, etc.).
- Les détails qui ne sont pas utiles à l'utilisateur lui sont masqués. Ce volet est garanti par la POO..

Exemple : Pour affecter une note à un étudiant, il suffit de saisir la note et de sélectionner l'étudiant concerné.

→ Comment une note donnée a été affectée à un étudiant donné est masqué pour l'utilisateur.

- **Réutilisabilité** :

- La POO facilite la réutilisation de composants logiciels. Par exemple, une classe peut être réutilisé dans plusieurs programmes, sans besoin de la définir à chaque fois.
- La définition d'une relation d'héritage entre les classes est motivée essentiellement par le besoin d'éviter la duplication de code
- → **Maintenabilité simplifiée et productivité boosté.**



CHAPITRE 2

Définir une classe

Ce que vous allez apprendre dans ce chapitre :

- Comprendre le concept de classe
- Savoir modéliser une classe
- Savoir définir les composantes d'une classe



CHAPITRE 2

Définir une classe

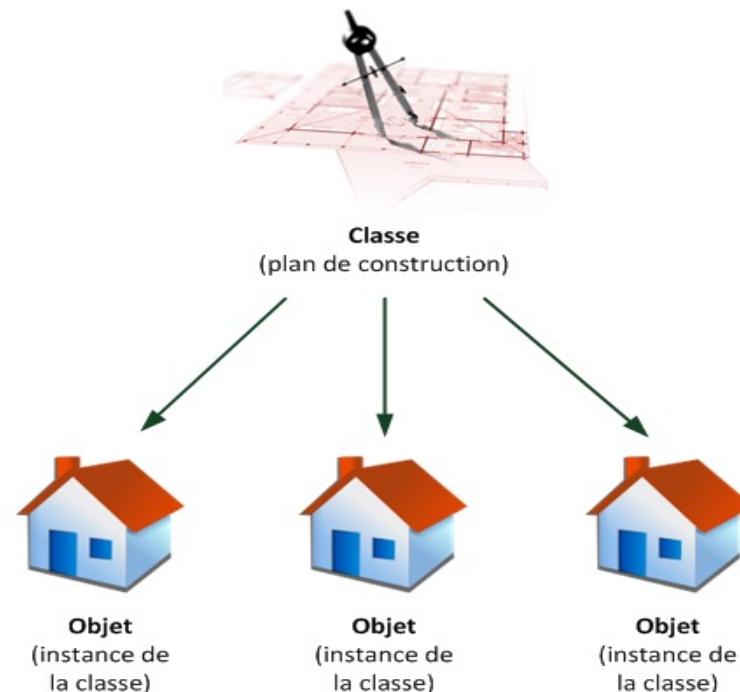
1. **Définition d'une classe**
2. Modélisation d'une classe
3. Composantes d'une classe

02 - Définir une classe

Définition d'une classe

Notion de classe

- Une classe peut être assimilée à la notion de type que l'on voit dans les langages de programmation procédurales.
- Une classe est considéré comme un **modèle** à partir duquel vont être créés un ensemble d'objets. Ces objets ont **des données ou des propriétés communes** et les mêmes **comportements**.





CHAPITRE 2

Définir une classe

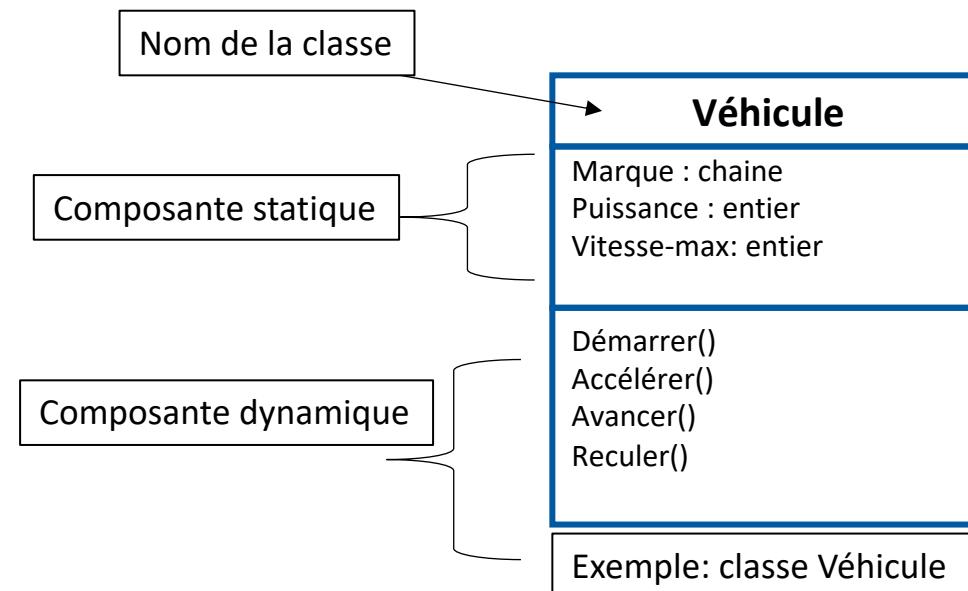
1. Définition d'une classe
2. **Modélisation d'une classe**
3. Composantes d'une classe

02 - Définir une classe

Modélisation d'une classe

Modélisation d'une classe

- Une classe est caractérisée par :
 - **Un nom**
 - **Une composante statique** : des champs (ou **attributs**). Ils caractérisent l'état des objets pendant l'exécution du programme
 - **Une composante dynamique** : des **méthodes** représentant le comportement des objets de cette classe. Elles manipulent les champs des objets et caractérisent les actions pouvant être effectuées par les objets





CHAPITRE 2

Définir une classe

1. Définition d'une classe
2. Modélisation d'une classe
3. Composantes d'une classe

02 - Définir une classe

Composantes d'une classe

Attribut

- Un **attribut** appelé également **champ** ou **donnée membre** correspond à une propriété de la classe
- **Un attribut est défini par:**
 - un nom,
 - un type de données
 - une valeur initiale (éventuellement)
- **Un attribut peut être de type :**
 - simple: entier, réel, chaîne de caractères, caractère, etc
 - Objet de type classe: Etudiant, Voiture, etc

Etudiant
Nom : chaîne
CIN : entier
anneeNaiss: chaîne
note1, note2 : réel

Etudiant
Nom : chaîne
CIN : entier
anneeNaiss : chaîne
note1, note2 : réel

Attributs de type simple

Attribut de type objet

Voiture
Marque : chaîne
Couleur : chaîne
MT:Moteur

Moteur
Marque : chaîne
Puissance : chaîne

02 - Définir une classe

Composantes d'une classe

Accès aux attributs

- Pour accéder à un attribut d'un objet on indique le nom de la référence de l'objet suivi par le nom de l'attribut dans l'objet de la manière suivante :

nomObjet. nomAttribut

- nomObjet** : nom de la référence à l'objet
- nomAttribut** = nom de l'attribut

Visibilité des attributs

- La visibilité des attributs définit les droits d'accès aux données d'une classe :
- Publique (+)** :
 - Toute classe peut accéder aux données d'une classe définie avec le niveau de visibilité publique.
- Protégée (#)** :
 - L'accès aux données est réservé aux méthodes des classes héritières
(A voir ultérieurement dans la partie Héritage)
- Privée (-)** :
 - L'accès aux données est limité aux méthodes de la classe elle-même

Nom_de_Classe

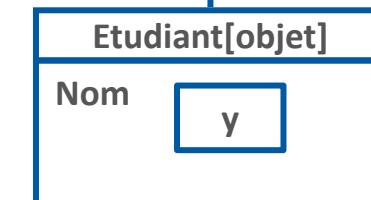
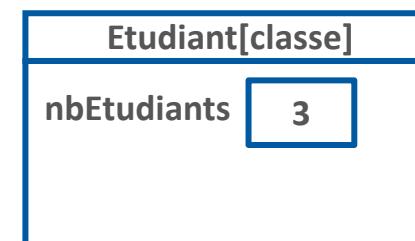
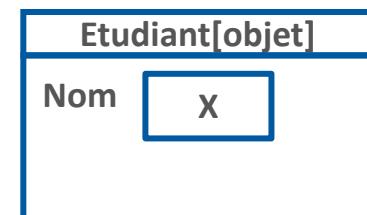
#attribut1 : type
- attribut2 : type
+ attribut3: type

+méthode1()
-méthode2()

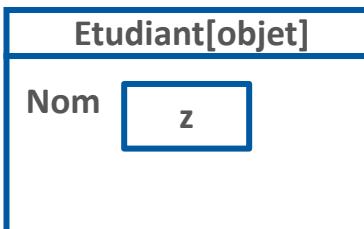
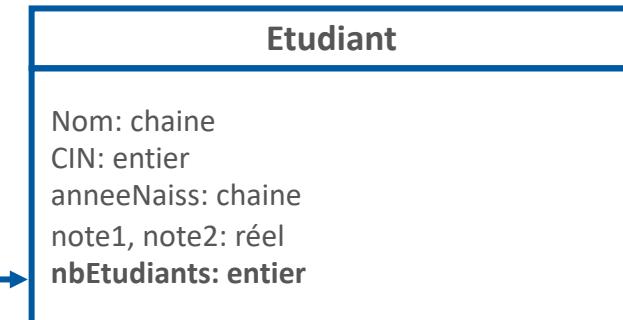
02 - Définir une classe

Composantes d'une classe

- Il peut s'avérer nécessaire de définir un attribut dont la valeur est **partagée** par tous les objets d'une classe. **On parle d'attribut de classe**
- Ces données sont, de plus, **stockées une seule fois**, pour toutes les instances d'une classe
- Accès :**
 - Depuis, une méthode de la classe comme pour tout autre attribut
 - Via une instance de la classe
 - À l'aide du nom de la classe



Attribut de classe



02 - Définir une classe

Composantes d'une classe

Constructeur

- Un **constructeur** est une méthode particulière invoquée implicitement lors de la création d'un objet
- Un constructeur **permet d'initialiser** les données des objets (les attributs) de la classe dont elle dépend
- Le constructeur ne doit pas avoir un type de retour
- Une classe peut posséder plusieurs constructeurs, mais un objet donné n'aura pu être produit que par un seul constructeur

Types de Constructeurs

- **Constructeur par défaut**
 - Un constructeur sans aucun paramètre est appelé un constructeur par défaut
 - Si nous ne créons pas de constructeur, la classe appellera automatiquement le constructeur par défaut lorsqu'un objet est créé
- **Constructeur paramétré:**
 - Un constructeur avec au moins un paramètre s'appelle un constructeur paramétré
- **Constructeur de copie:**
 - Le constructeur qui crée un objet en copiant les données d'un autre objet s'appelle un constructeur de copie

02 - Définir une classe

Composantes d'une classe

Destructeur

- Le destructeur est une méthode particulière qui permet **la destruction d'un objet non référencé**.
- Le destructeur ne doit pas avoir un type de retour

Un destructeur permet de :

- **Gérer les erreurs**
- **Libérer** les ressources utilisées de manière certaine
- **Assurer la fermeture** de certaines parties du code.





CHAPITRE 3

Créer un objet

Ce que vous allez apprendre dans ce chapitre :

- Comprendre le concept d'objet
- Maitriser les concept d'instanciation d'une classe et la destruction d'un objet



CHAPITRE 3

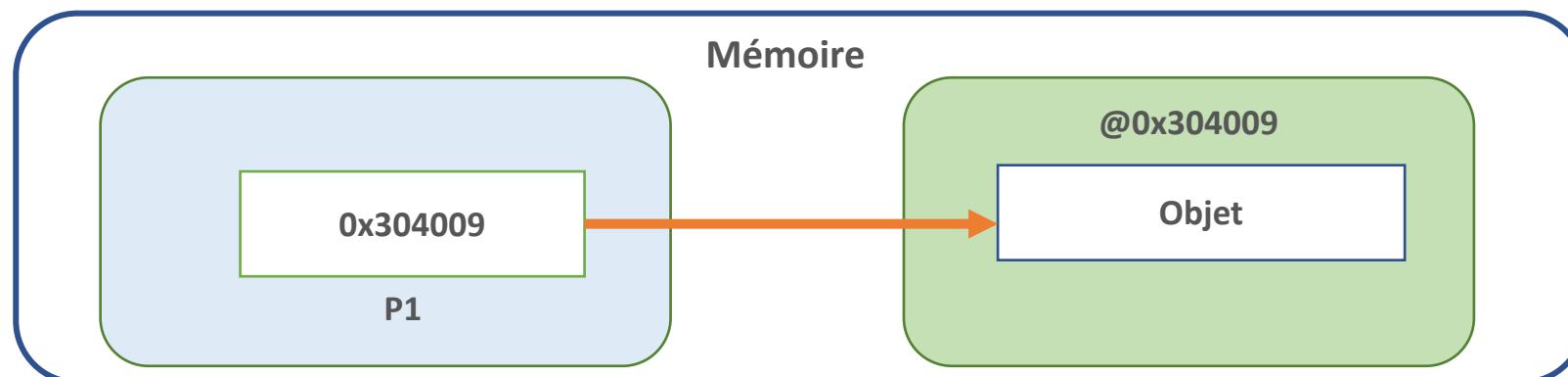
Créer un objet

1. **Définition d'un objet**
2. Instantiation
3. Destruction explicite d'un objet

Notion d'objet

OBJET= Référent + Etat + Comportement

- Chaque objet doit avoir un nom (référence) « qui lui est propre » pour l'identifier
- La référence permet d'accéder à l'objet, mais n'est pas l'objet lui-même. Elle contient l'adresse de l'emplacement mémoire dans lequel est stocké l'objet.

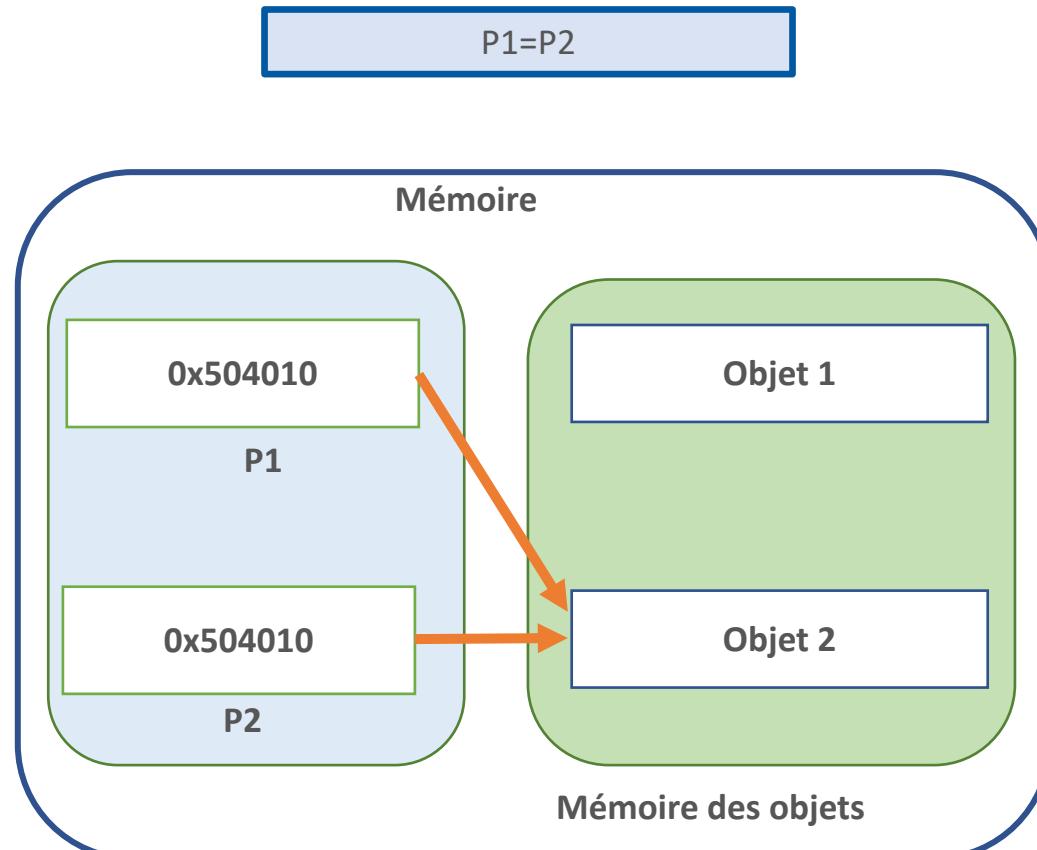
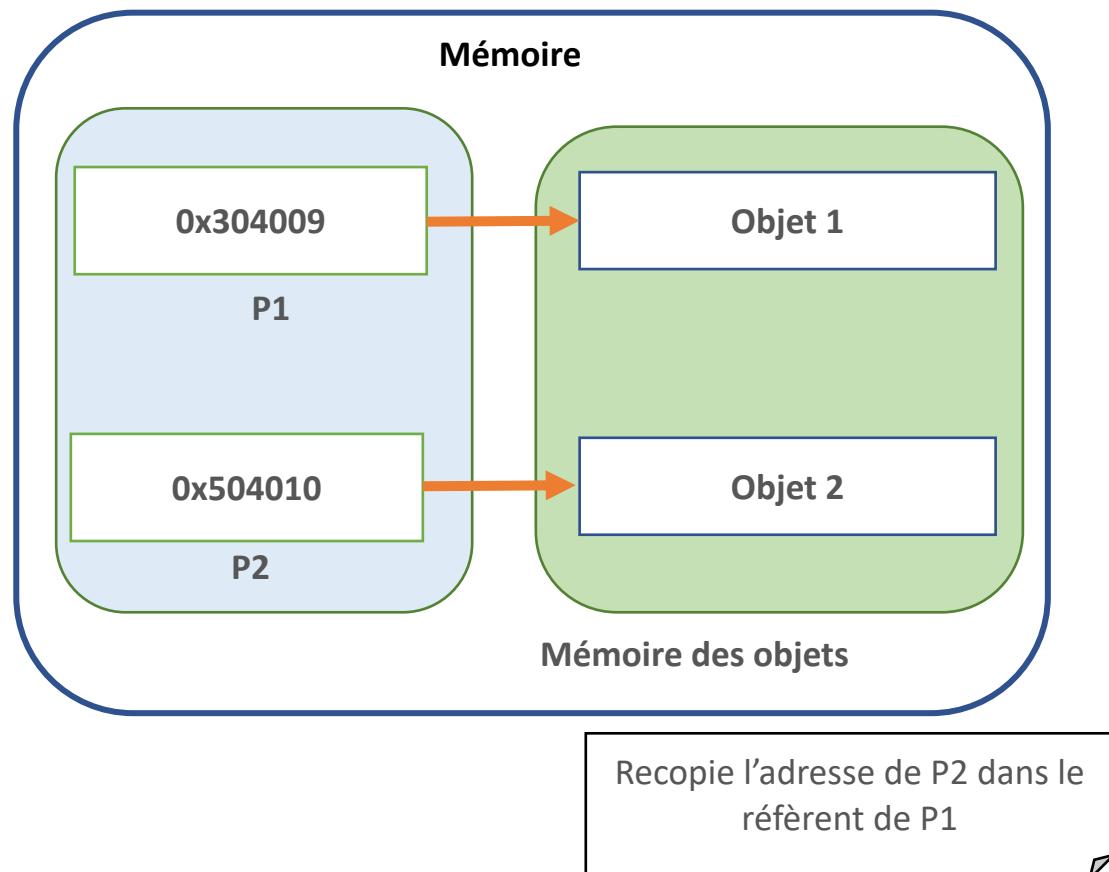


03 - Créer un objet

Définition d'un objet

Notion d'objet

- Plusieurs référents peuvent référer un même objet → **Adressage indirect**



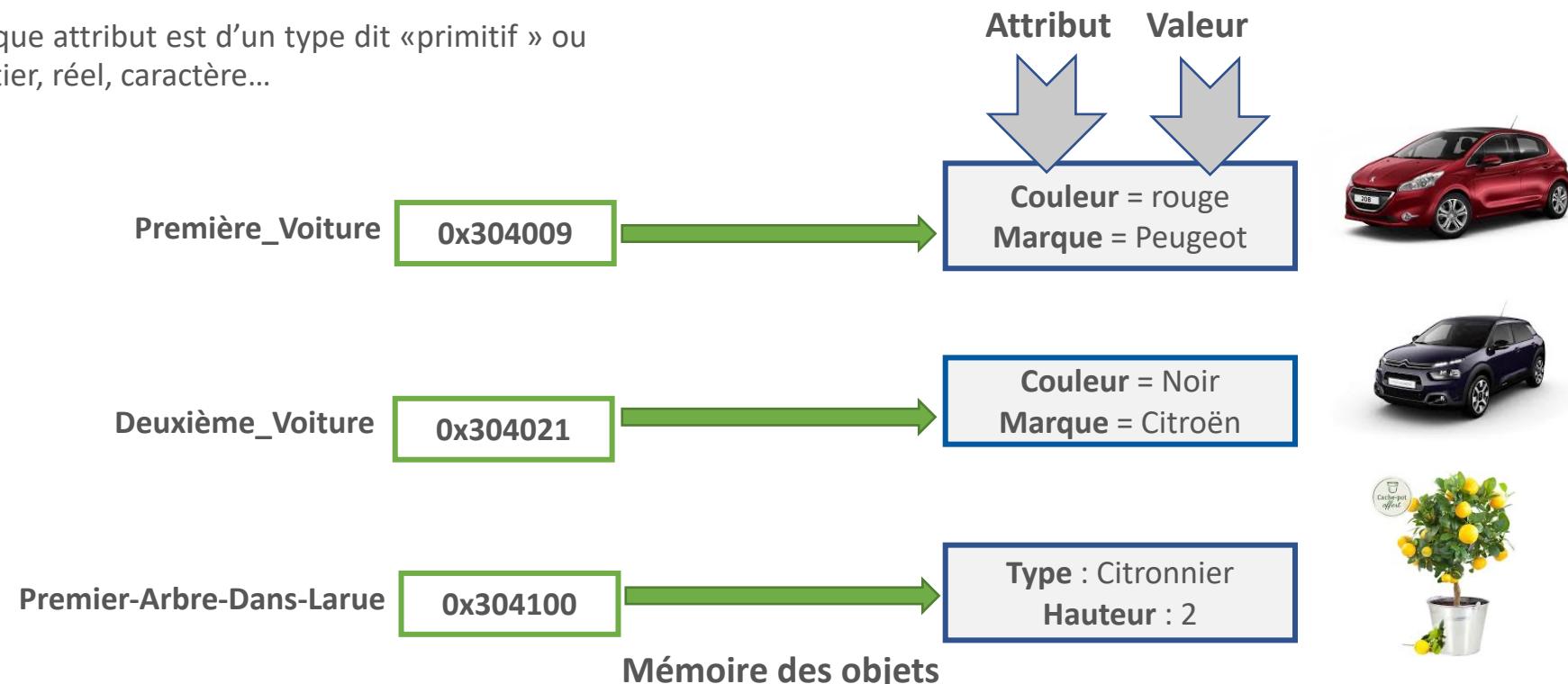
03 - Créer un objet

Définition d'un objet

Notion d'objet

- Un objet est capable de sauvegarder un état c'est-à-dire un ensemble d'information dans les **attributs**
- Les **attributs** sont l'ensemble des informations permettant de représenter l'état de l'objet
- Exemple d'objets où chaque attribut est d'un type dit «primitif » ou « prédéfini », comme entier, réel, caractère...

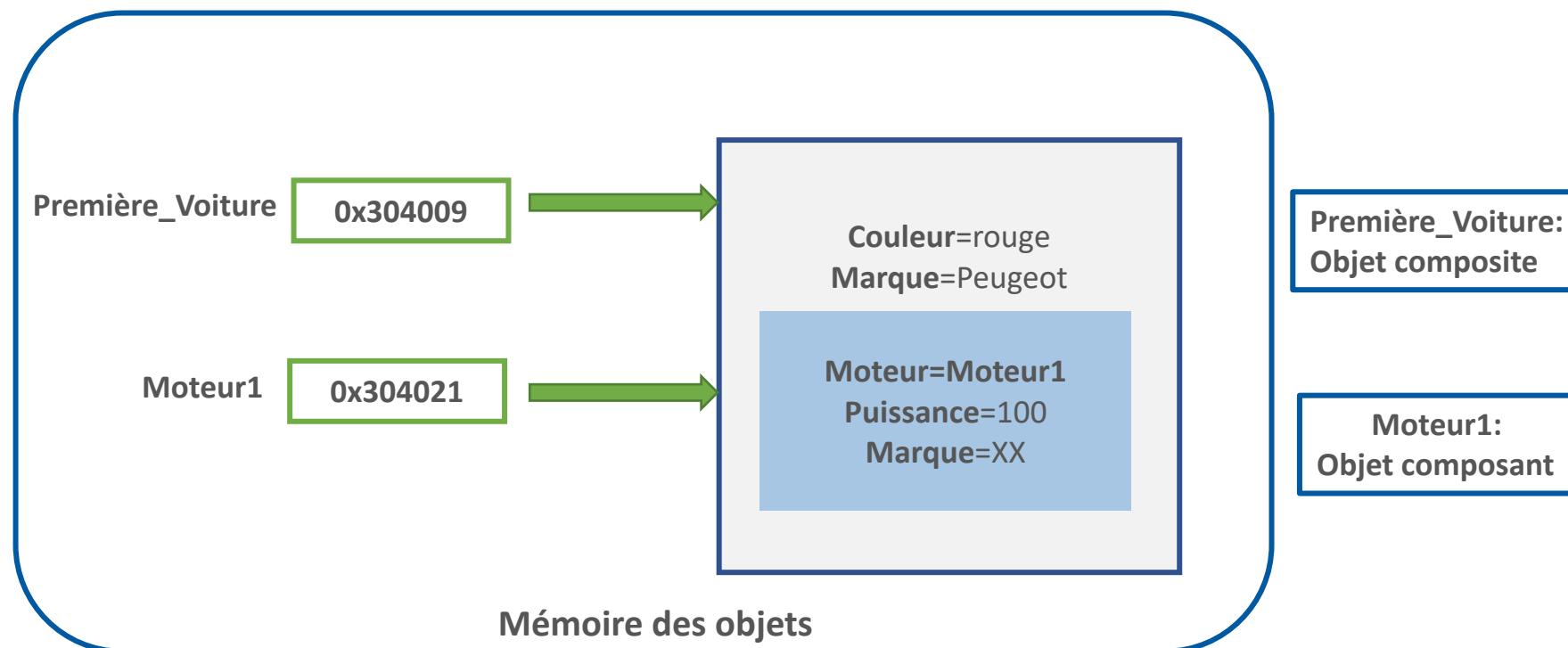
OBJET= Référent + Etat + Comportement



03 - Créer un objet

Définition d'un objet

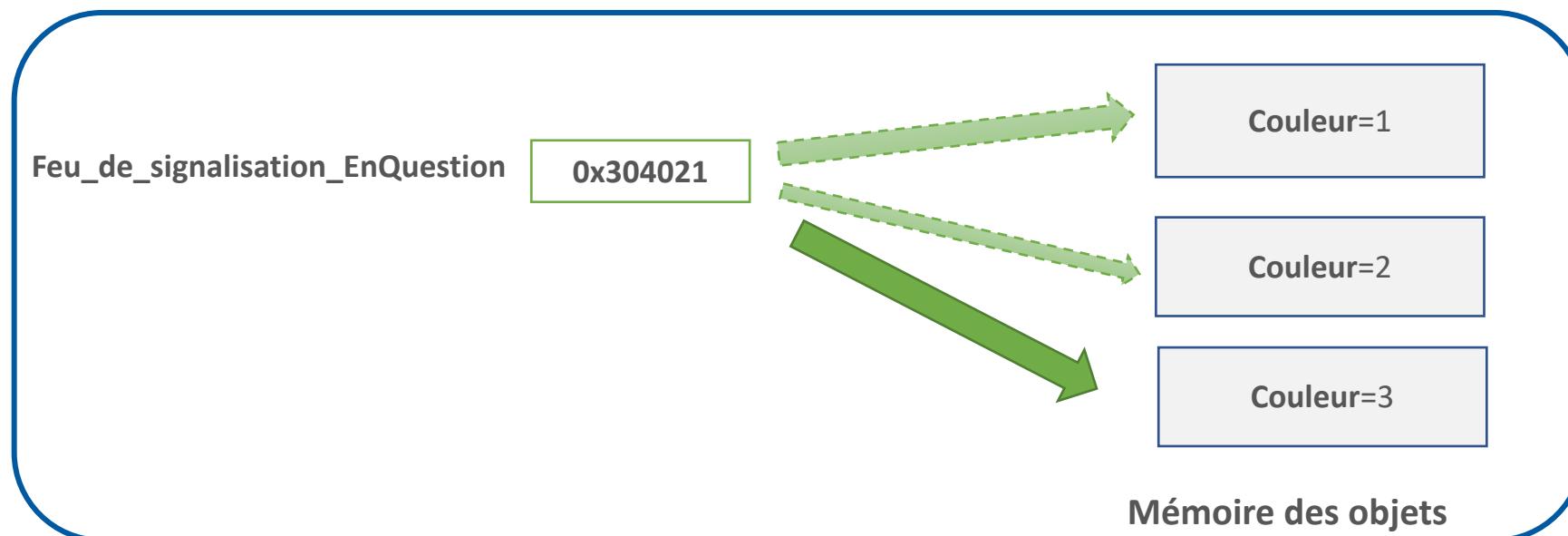
- Un objet stocké en mémoire peut être placé à l'intérieur de l'espace mémoire réservé à un autre. Il s'agit d'un **Objet Composite**



03 - Créer un objet

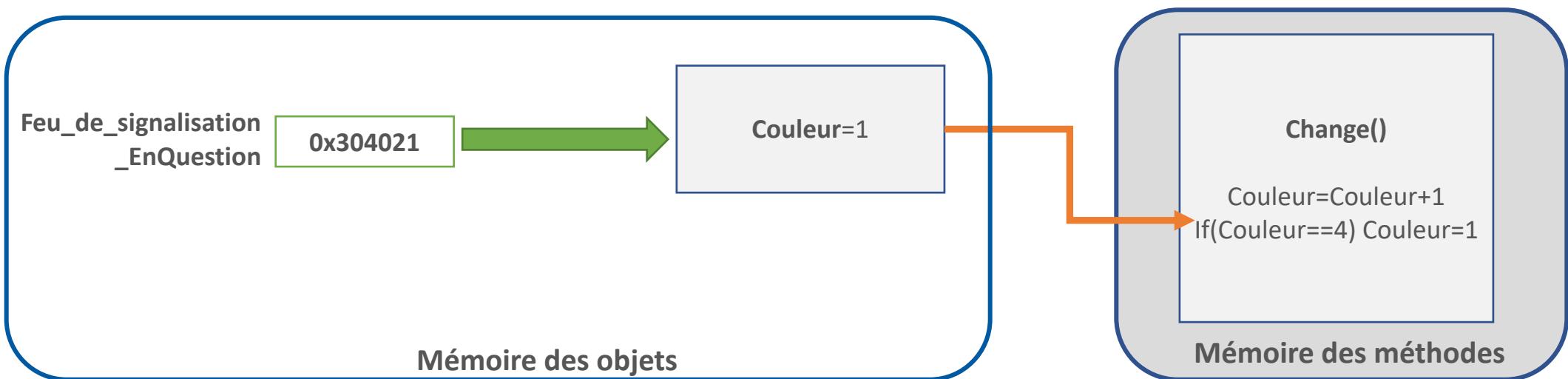
Définition d'un objet

- Les objets changent d'état
- Le cycle de vie d'un objet se limite à une succession de changements d'états
- Soit l'objet Feu_de_signalisation_EnQuestion avec sa couleur prenant trois valeurs



OBJET= Référent + Etat + Comportement

Mais quelle est la cause des changements d'états ?
– LES METHODES





CHAPITRE 3

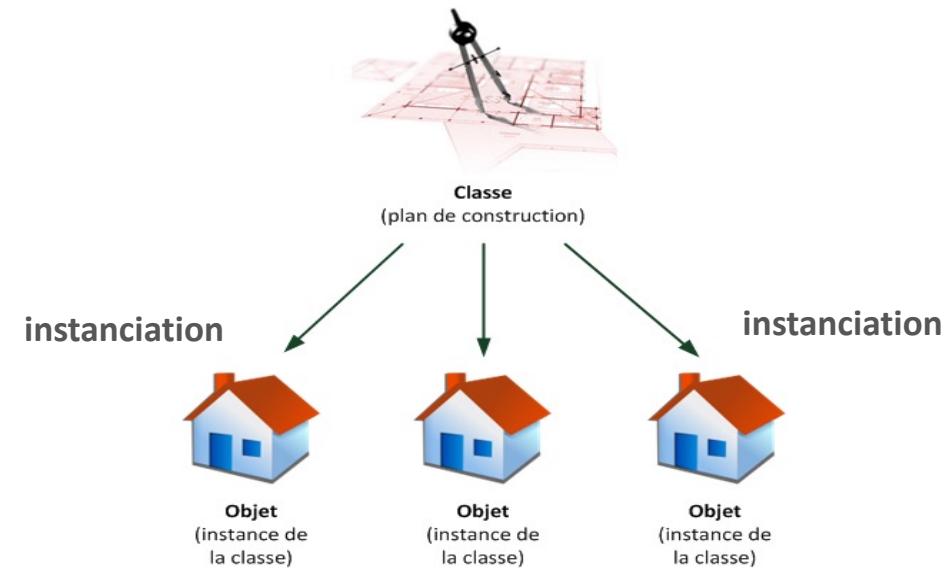
Créer un objet

1. Définition d'un objet
2. **Instanciation**
3. Destruction explicite d'un objet

03 - Crée un objet

Instanciation

- L'instanciation d'un objet est constituée de deux phases :
 - **Une phase de ressort de la classe** : créer l'objet en mémoire
 - **Une phase du ressort de l'objet** : initialiser les attributs
- L'instanciation explicite d'un objet se fait via un **constructeur**. Il est appelé automatiquement lors de la création de l'objet dans la mémoire.





CHAPITRE 3

Créer un objet

1. Définition d'un objet
2. Instanciation
3. **Destruction explicite d'un objet**

03 - Créer un objet

Destruction explicite d'un objet

- **Rappel :** un destructeur est une méthode particulière qui permet **la destruction d'un objet non référencé**
 - La destruction explicite d'un objet s'effectue en faisant appel au destructeur de la classe

CHAPITRE 4

Connaitre l'encapsulation



Ce que vous allez apprendre dans ce chapitre :

- Comprendre le principe d'encapsulation
- Connaitre les modificateurs et les accesseurs d'une classe : Syntaxe et utilité



CHAPITRE 4

Connaitre l'encapsulation

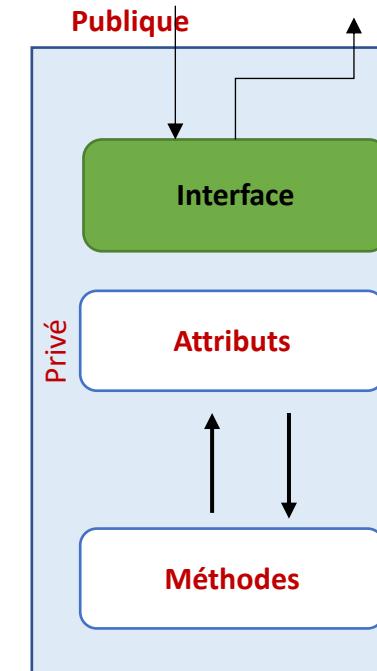
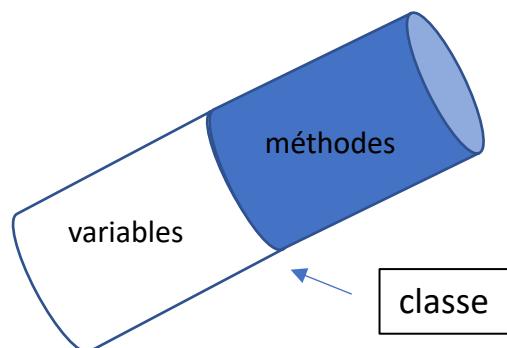
1. Principe de l'encapsulation
2. Modificateurs et accesseurs (getters, setters, ...)

04 - Connaitre l'encapsulation

Principe de l'encapsulation

Principe de l'encapsulation

- L'encapsulation est le fait de réunir à l'intérieur d'une même entité (objet) le code (méthodes) + données (attributs)
- L'encapsulation consiste à protéger l'information contenue dans un objet
- **Il est donc possible de masquer les informations d'un objet aux autres objets.**
- L'encapsulation consiste donc à masquer les détails d'implémentation d'un objet, en définissant une **interface**
- L'interface est la **vue externe d'un objet**, elle définit les services **accessibles** (Attributs et Méthodes) aux utilisateurs de l'objet
 - **interface = liste des signatures des méthodes accessibles**
 - **Interface = Carte de visite de l'objet**



04 - Connaitre l'encapsulation

Principe de l'encapsulation

- Les objets ne restreignent leur accès qu'aux méthodes de leur classe

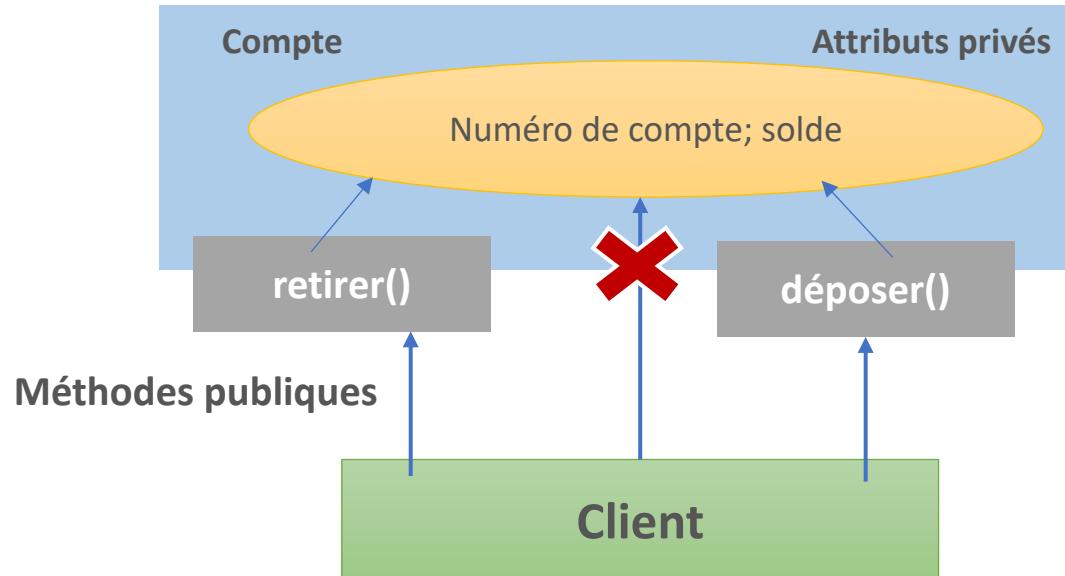
→ Ils protègent leurs attributs

- Cela permet d'avoir un contrôle sur tous les accès

Exemple :

- Les attributs de la classe Compte sont privés
- Ainsi le solde d'un compte n'est pas accessible par un client qu'à travers les méthodes retirer() et déposer() qui sont publiques

→ Un client ne peut modifier son solde qu'en effectuant une opération de dépôt ou de retrait





CHAPITRE 4

Connaitre l'encapsulation

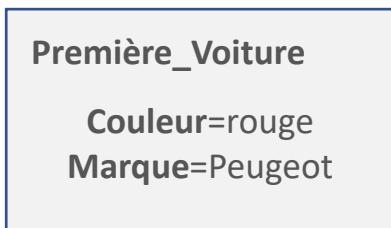
1. Principe de l'encapsulation
2. **Modificateurs et accesseurs (getters, setters, ...)**

04 - Connaitre l'encapsulation

Modificateurs et accesseurs (getters, setters, ...)

Modificateurs et accesseurs

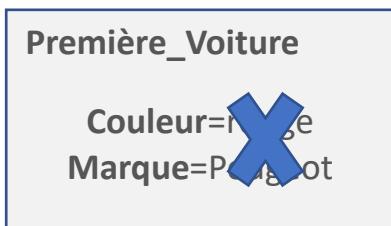
- Pour interagir avec les attributs d'un objet de l'extérieur, il suffit de créer des méthodes publiques dans la classe appelée **Accesseurs et Modificateurs**
- Accesseurs (getters):** Méthodes qui retournent la valeur d'un attribut d'un objet (un attribut est généralement privé)
- La notation utilisée est **getXXX** avec XXX l'attribut retourné
- Modificateurs (setters):** Méthodes qui modifient la valeur d'un attribut d'un objet
- La notation utilisée est **setXXX** avec XXX l'attribut modifié



getCouleur()

Quelle est la couleur de Première_Voiture?

Rouge



setCouleur(Bleu)

Couleur = Bleu



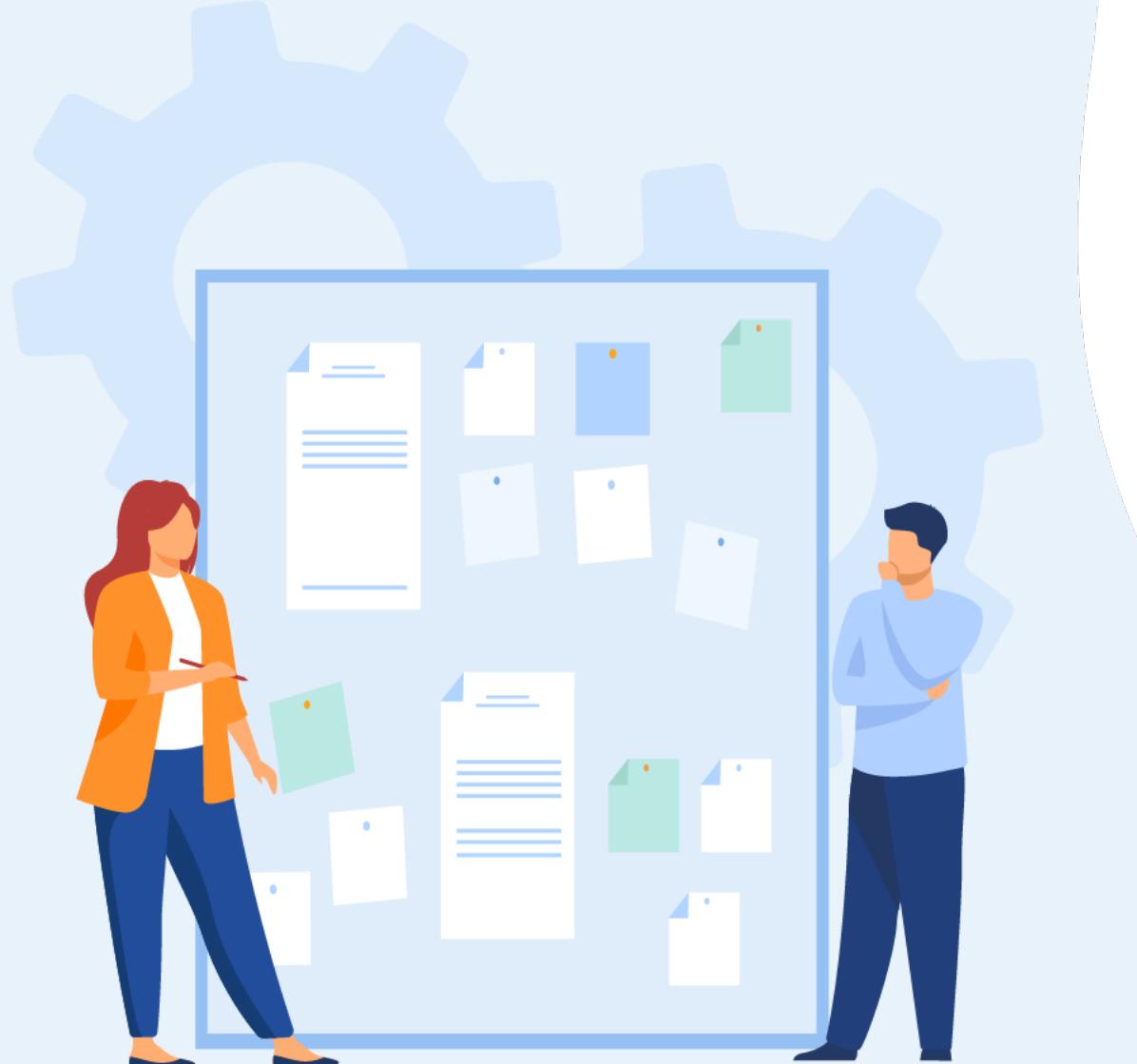
CHAPITRE 5

Manipuler les méthodes



Ce que vous allez apprendre dans ce chapitre :

- Manipuler les méthodes : définition et appel
- Différencier entre méthode d'instance et méthode de classe



CHAPITRE 5

Manipuler les méthodes

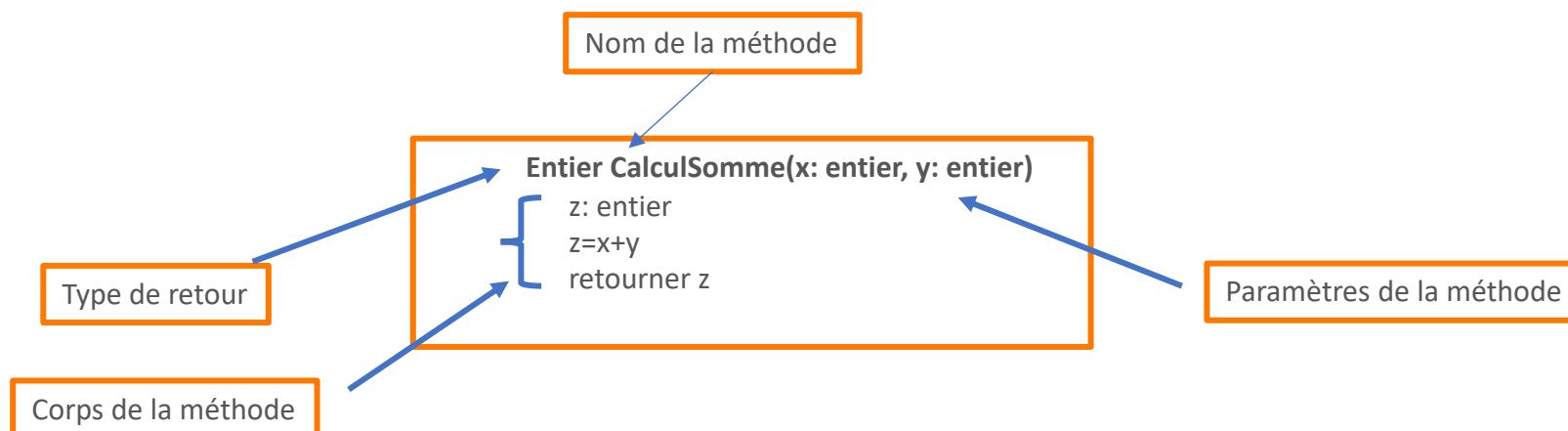
1. **Définition d'une méthode**
2. Visibilité d'une méthode
3. Paramètres d'une méthode
4. Appel d'une méthode
5. Méthodes de classe

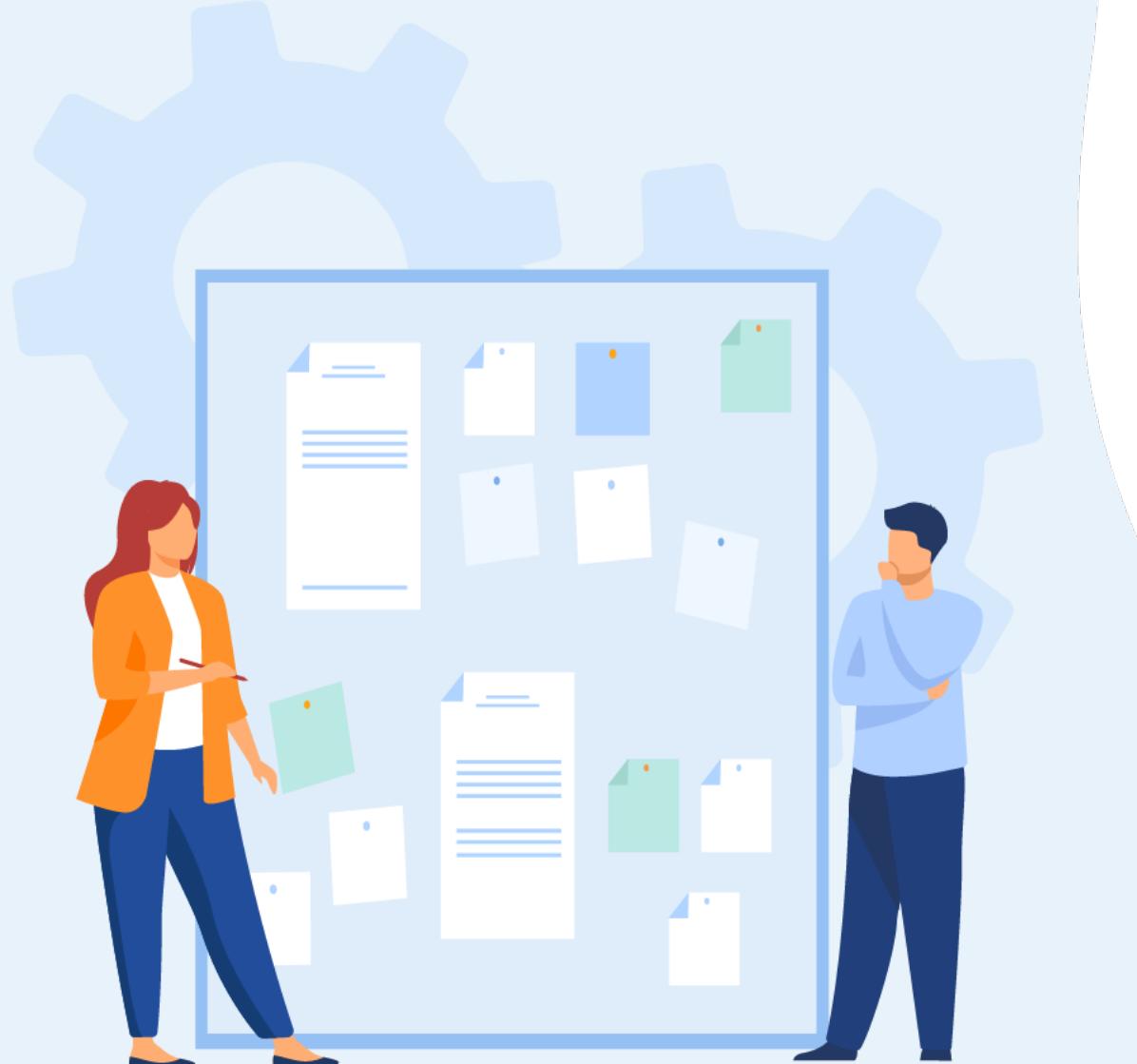
05 - Manipuler les méthodes

Définition d'une méthode

- La définition d'une méthode ressemble à celle d'une procédure ou d'une fonction. Elle se compose d'une entête et un bloc.
- L'**entête** précise :
 - Un nom
 - Un type de retour
 - Une liste (éventuellement vide) de **paramètres** typés en entrée
 - Le **bloc** est constitué d'une suite d'instructions (un bloc d'instructions) qui constituent le corps de la méthode

Exemple de syntaxe :





CHAPITRE 5

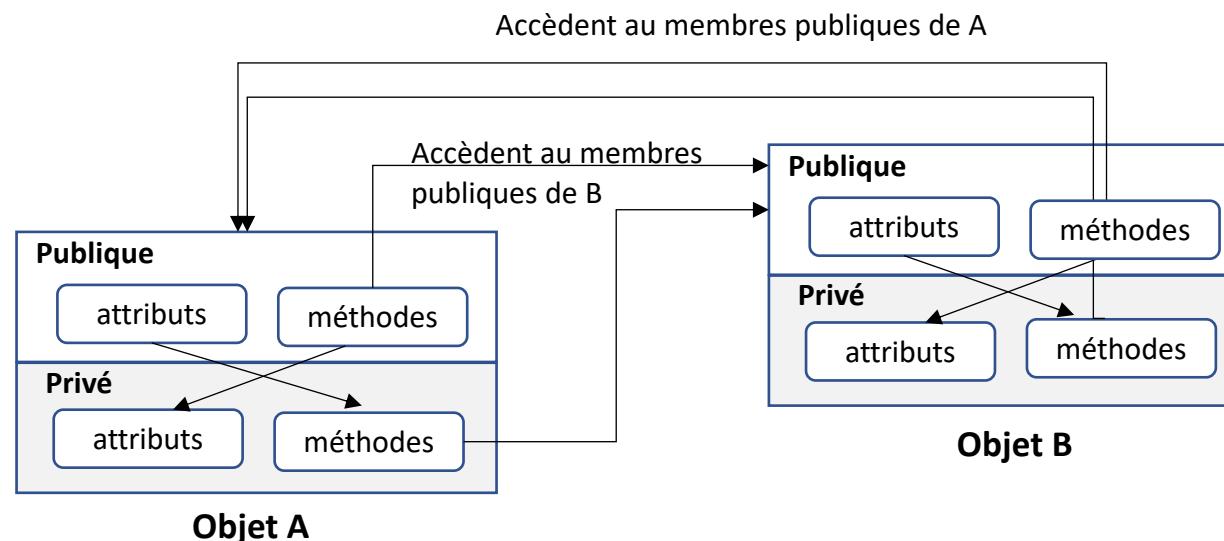
Manipuler les méthodes

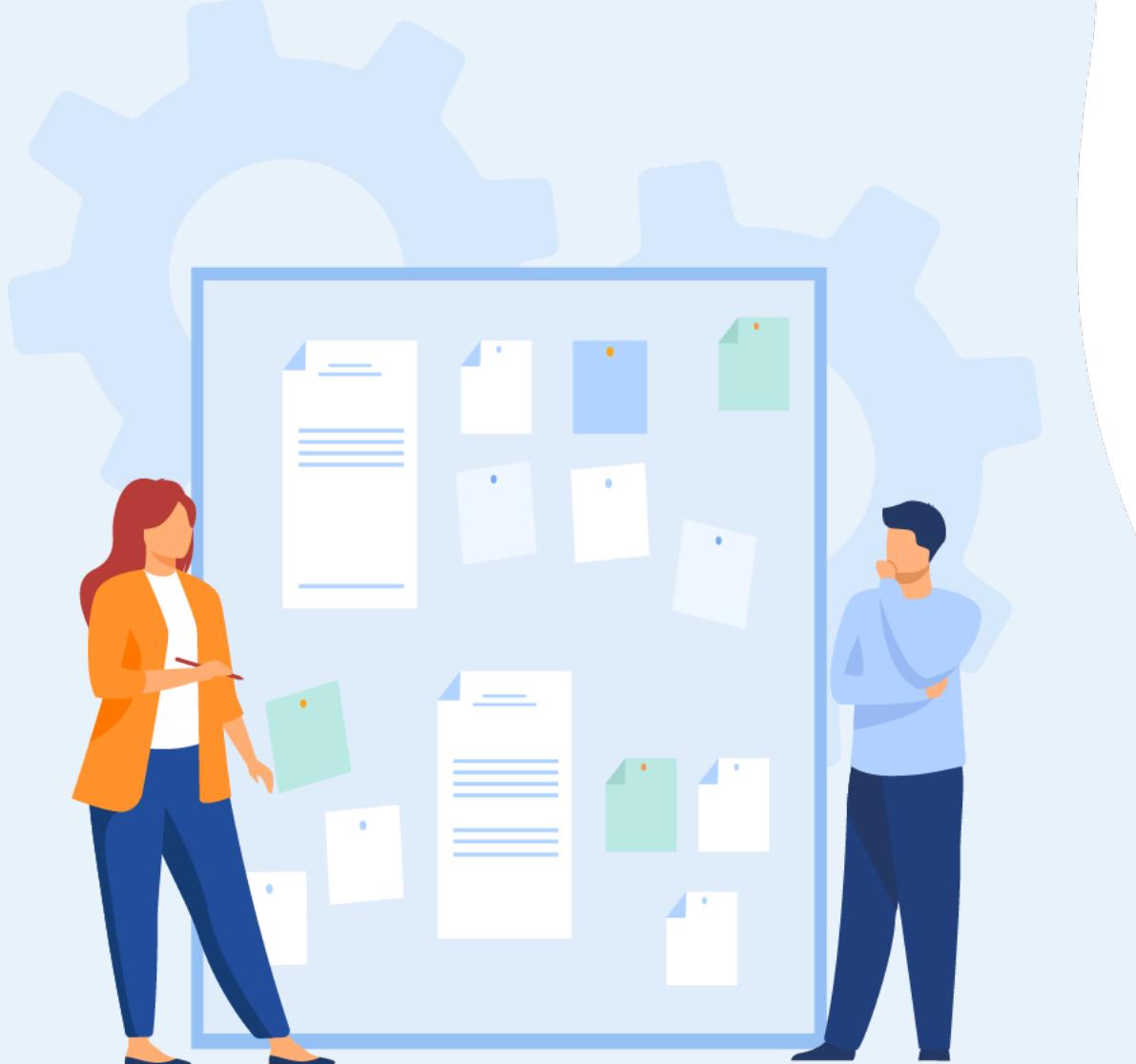
1. Définition d'une méthode
2. **Visibilité d'une méthode**
3. Paramètres d'une méthode
4. Appel d'une méthode
5. Méthodes de classe

05 - Manipuler les méthodes

Visibilité d'une méthode

- La visibilité d'une méthode définit les droits d'accès autorisant l'appel aux méthodes de la classe
- **Publique (+) :**
 - L'appel de la méthode est autorisé aux méthodes de toutes les classes
- **Protégée (#) :**
 - L'appel de la méthode est réservé aux méthodes des classes héritières
(A voir ultérieurement dans la partie Héritage)
- **Privée(-) :**
 - L'appel de la méthode est limité aux méthodes de la classe elle-même





CHAPITRE 5

Manipuler les méthodes

1. Définition d'une méthode
2. Visibilité d'une méthode
- 3. Paramètres d'une méthode**
4. Appel d'une méthode
5. Méthodes de classe

05 - Manipuler les méthodes

Paramètres d'une méthode

- Il est possible de passer des arguments (appelés aussi **paramètres**) à une **méthode**, c'est-à-dire lui fournir le nom d'une variable afin que la **méthode** puisse effectuer des opérations sur ces arguments ou bien grâce à ces arguments.

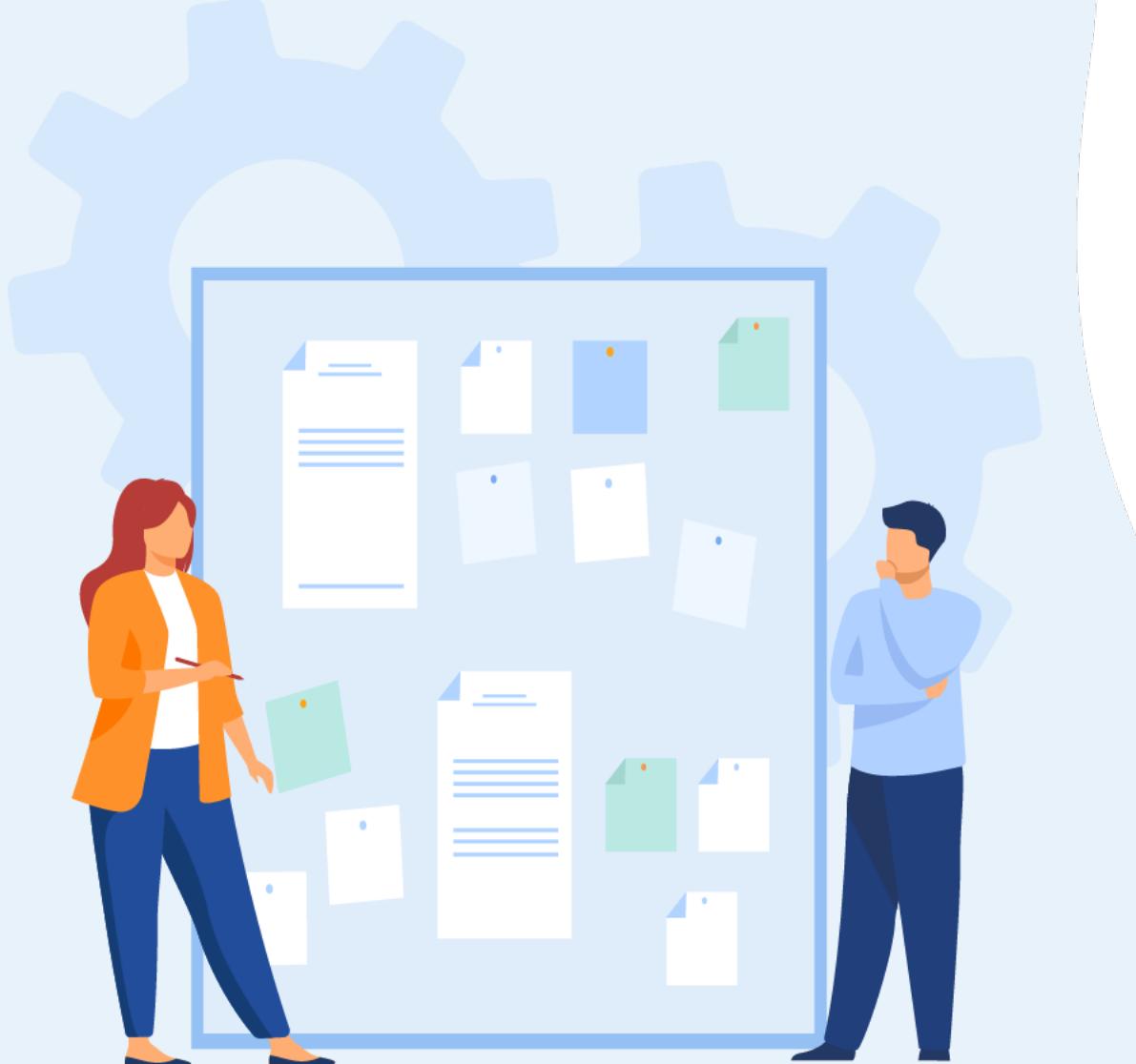
Entier CalculSomme(x: entier, y: entier)

z: entier

$z=x+y$

retourner z

Paramètres de la méthode



CHAPITRE 5

Manipuler les méthodes

1. Définition d'une méthode
2. Visibilité d'une méthode
3. Paramètres d'une méthode
- 4. Appel d'une méthode**
5. Méthodes de classe

05 - Manipuler les méthodes

Appel d'une méthode

- Les méthodes ne peuvent être définies comme des composantes d'une classe.
- Une méthode est appelée pour un objet. Cette méthode est appelée **méthode d'instance**.
- L'appel d'une méthode pour un objet se réalise en nommant l'objet suivi d'un point suivi du nom de la méthode et sa liste d'arguments.

nomObjet.nomMéthode(arg1, arg2,..)

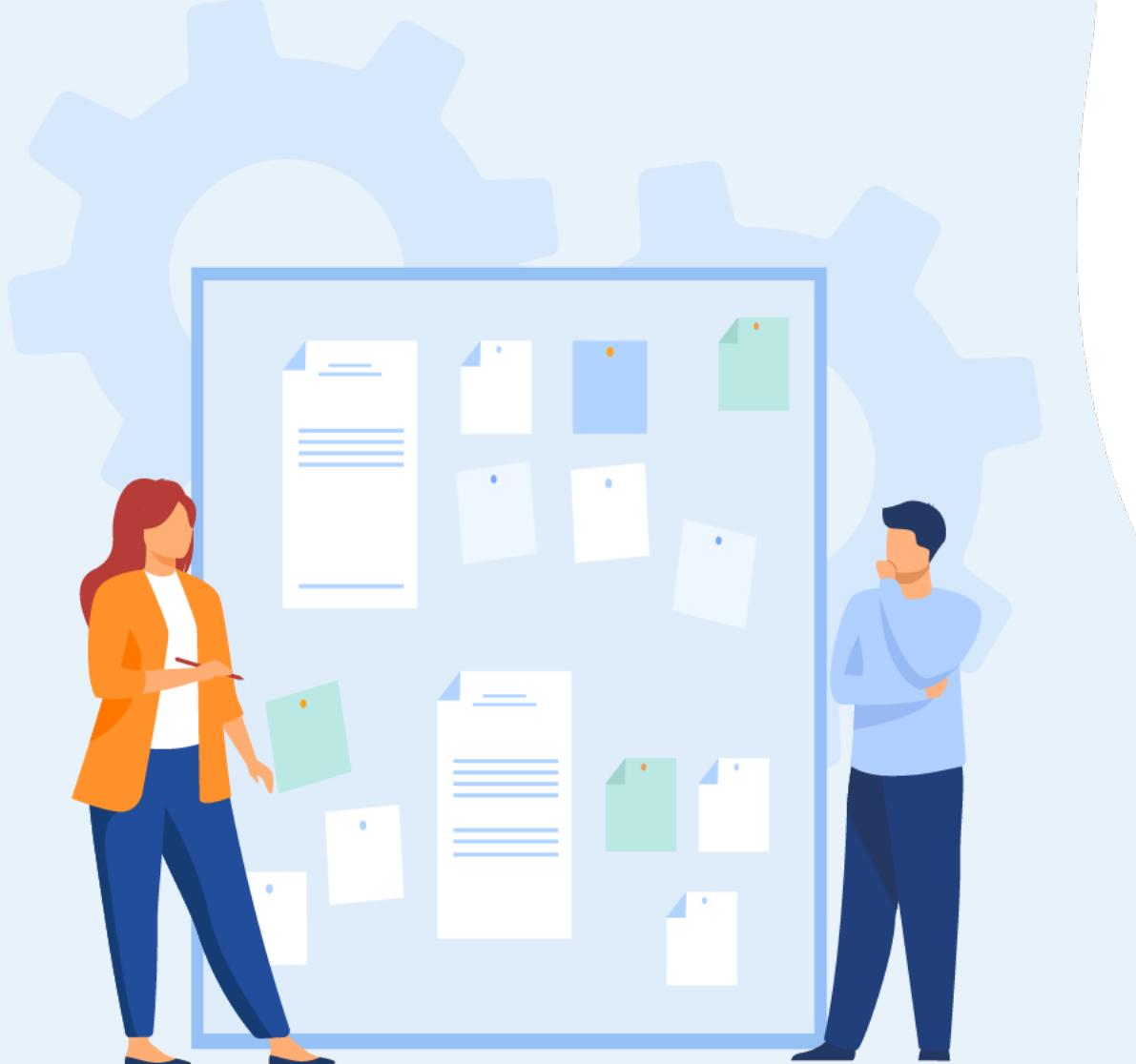
Où :

- **nomObjet** : nom de la référence à l'objet
- **nomMéthode** : nom de la méthode

Exemple : la classe *Véhicule*

<i>Véhicule</i>
<i>Marque :string</i>
<i>Puissance :integer</i>
<i>Vitesse-max :integer</i>
<i>Vitesse-courante :int</i>
<i>Démarrer()</i>
<i>Accélérer()</i>
<i>Avancer()</i>
<i>Reculer()</i>

v est objet de type Véhicule
v.Démarrer()



CHAPITRE 5

Manipuler les méthodes

1. Définition d'une méthode
2. Visibilité d'une méthode
3. Paramètres d'une méthode
4. Appel d'une méthode
5. **Méthodes de classe**

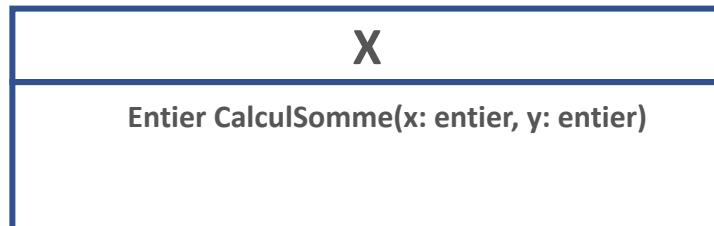
05 - Manipuler les méthodes

Méthodes de classe

- Méthode de classe est une méthode déclarée dont l'exécution ne dépend pas des objets de la classe
- Étant donné qu'elle est liée à la classe et non à ses instances, on préfixe le nom de la méthode par le nom de la classe

nomClasse.nomMethode(arg1, arg2,..)

Exemple :



X.CalculSomme (2,5)

- Puisque les méthodes de classe appartiennent à la classe, elles ne peuvent en aucun cas accéder aux attributs d'instances qui appartiennent aux instances de la classe
- Les méthodes d'instances accèdent aux attributs d'instance et méthodes d'instance
- Les méthodes d'instances accèdent aux attributs de classe et méthodes de classe
- Les méthodes de classe accèdent aux attributs de classe et méthodes de classe
- Les méthodes de classe n'accèdent pas aux attributs d'instance et méthodes d'instance



PARTIE 2

Connaître les principaux piliers de la POO

Dans ce module, vous allez :

- Maitriser le principe d'héritage en POO
- Maitriser le concept du polymorphisme en POO
- Maitriser le principe de l'abstraction en POO et son utilité
- Manipuler les interfaces (définition et implémentation)



15 heures



CHAPITRE 1

Définir l'héritage

Ce que vous allez apprendre dans ce chapitre :

- Comprendre le principe de l'héritage et distinguer ses types
- Comprendre le principe de surcharge de constructeur
- Maîtriser le chainage des constructeurs
- Maîtriser la visibilité des membres d'une classe dérivée



06 heures



CHAPITRE 1

Définir l'héritage

1. **Principe de l'héritage**
2. Types d'héritage
3. Surcharge des constructeurs et chaînage
4. Visibilité des attributs et des méthodes de la classe fille

01 - Définir l'héritage

Principe de l'héritage



Principe de l'héritage

- L'héritage est une notion qui définit une relation de **spécialisation** ou de **généralisation** entre différentes classes (Exp : une relation d'héritage entre les classes Employé et Directeur, et la classe Salarié).
- Lorsqu'une classe A hérite d'une classe B :
 - A s'octroie toutes les propriétés de B, en plus de ses propres propriétés.
 - A est alors considéré comme une spécialisation de B.
 - Dans le même temps, B peut être vu comme une généralisation de A.
 - A est appelée classe fille de B, ou classe dérivée de B.
 - B est dite classe mère ou super-classe de A.
- **Toute instance (objet) de A peut être considéré comme un objet de B.**
- **Une instance de B ne peut être toujours considéré comme un objet de A.**

01 - Définir l'héritage

Principe de l'héritage



Principe de l'héritage

Exemple :

- Considérons la définition des 3 classes Personne, Etudiant et Employé suivantes :

Personne
Nom: chaîne CIN: entier anneeNaiss: entier
age()

Etudiant
Nom: chaîne CIN: entier anneeNaiss: entier note1, note2: réel
age() moyenne()

Employé
Nom: chaîne CIN: entier anneeNaiss entier prixHeure: réel nbreHeure: réel
age() Salaire()

Problème :

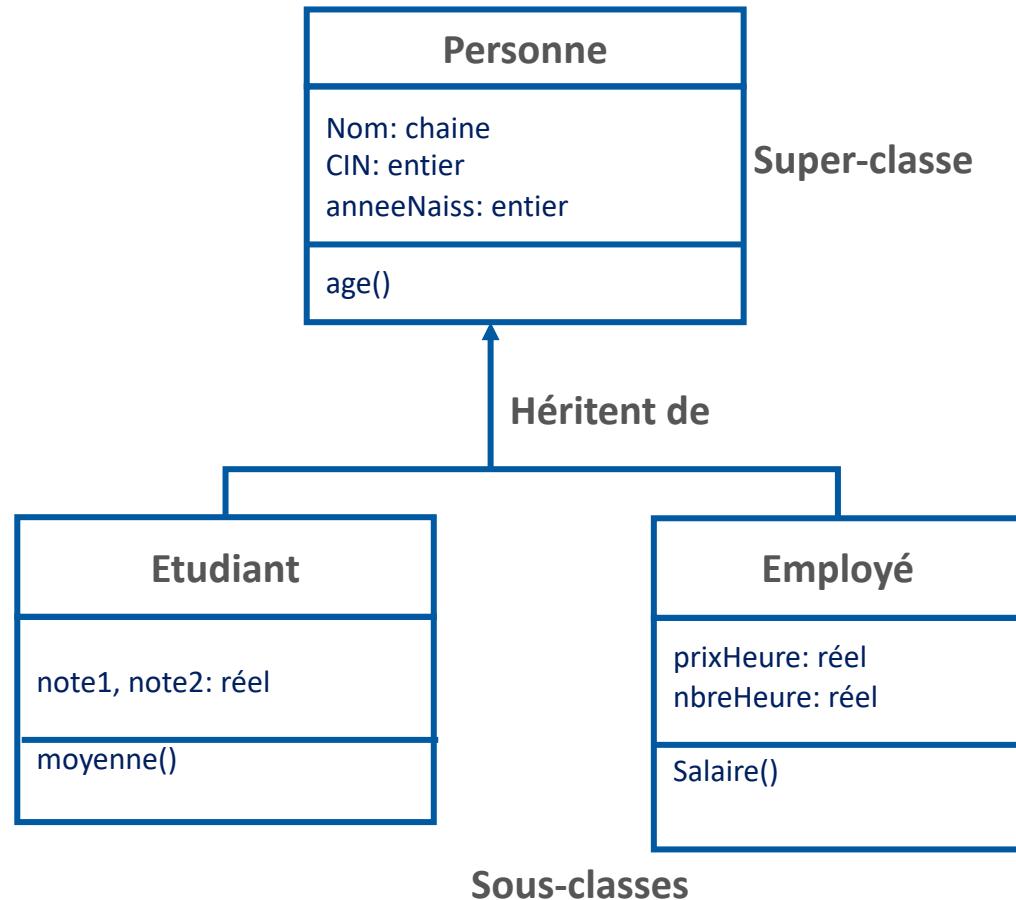
- Duplication du code.
- Toute modification d'un attribut ou d'une méthode doit être répétée dans chaque classe où ils ont été déclarés. (Exp : si on souhaite modifier le type de l'attribut **anneeNaiss**, il faut le faire dans chacune des 3 classes).

01 - Définir l'héritage

Principe de l'héritage

Solution :

- Placer dans la **classe mère** les propriétés en commun à toutes les autres classes.
- On ne garde dans les classes filles que les attributs ou méthodes qui leur sont **spécifiques**.
- Les classes dérivées **héritent** automatiquement des attributs (et des méthodes) qui n'ont pas besoin d'être réécrits.



01 - Définir l'héritage

Principe de l'héritage



Intérêt de l'héritage

- L'héritage permet de factoriser le code en **regroupant les caractéristiques en commun** à plusieurs classes au sein d'une seule (la classe mère).
- **La création de nouvelles classes** s'en trouve facilitée grâce à une hiérarchie bien définie des classes.



CHAPITRE 1

Définir l'héritage

1. Principe de l'héritage
2. **Types d'héritage**
3. Surcharge des constructeurs et chaînage
4. Visibilité des attributs et des méthodes de la classe fille

01 - Définir l'héritage

Types d'héritage

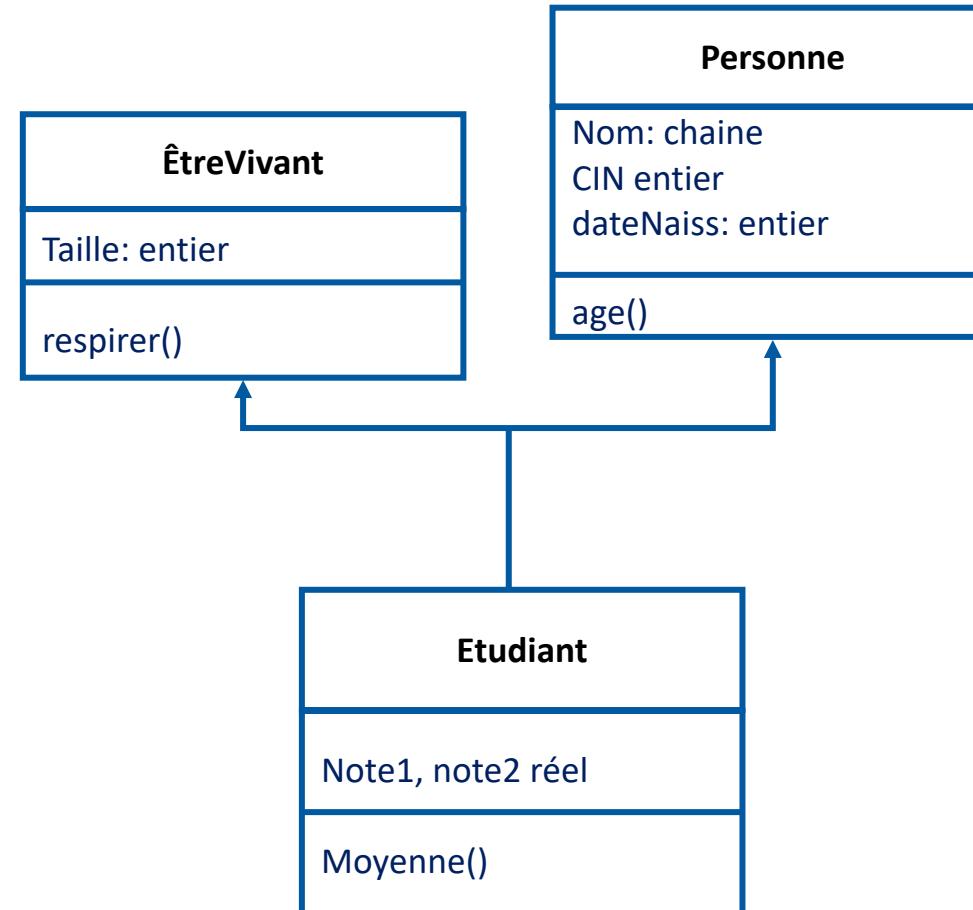
Héritage multiple

- Une classe peut hériter de plusieurs classes

Exemple :

- Un Etudiant est à la fois une Personne et un EtreVivant
- La classe Etudiant hérite des attributs et des méthodes des
- deux classes

→ Il deviennent ses propres attributs et méthodes



01 - Définir l'héritage

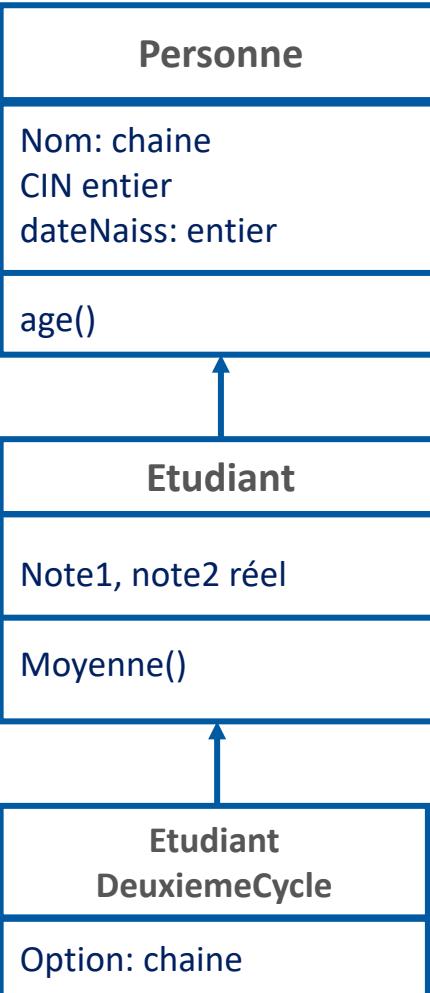
Types d'héritage

Héritage en cascade

- Une classe sous-classe peut être elle-même une super-classe.

Exemple :

- Etudiant hérite de Personne
 - EtudiantDeuxièmeCycle hérite de Etudiant
- EtudiantDeuxièmeCycle hérite de Personne





CHAPITRE 1

Définir l'héritage

1. Principe de l'héritage
2. Types d'héritage
3. **Surcharge des constructeurs et chaînage**
4. Visibilité des attributs et des méthodes de la classe fille

01 - Définir l'héritage

Surcharge des constructeurs et chaînage



Surcharge du constructeur

- Les constructeurs portant le **même nom** mais **une signature différente** sont appelés constructeurs surchargés
- La signature d'un constructeur comprend les types des paramètres et le nombre des paramètres
- Dans une classe, les constructeurs peuvent avoir différents nombres d'arguments, différentes séquences d'arguments ou différents types d'arguments
- Nous pouvons avoir plusieurs constructeurs mais lors de la création d'un objet, le compilateur choisit la méthode qui doit être appelée en fonction du nombre et du type des arguments

Surcharge du constructeur
CréerPersonne

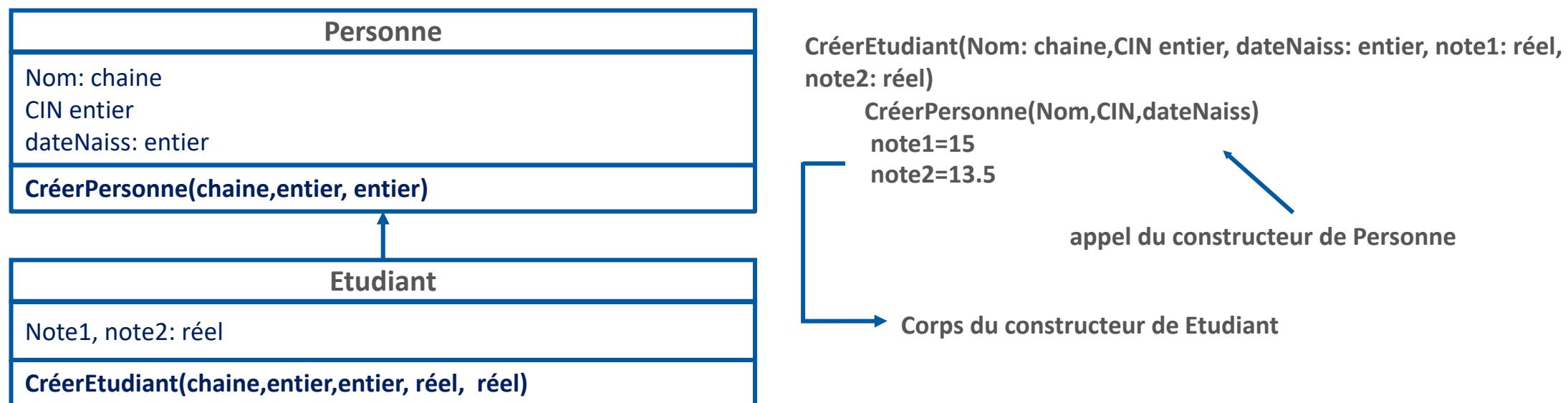
Personne
Nom: chaine CIN: entier dateNaiss: entier
age() CréerPersonne(chaine, entier, entier) CréerPersonne(chaine, entier) CréerPersonne(entier, chaine, entier)

01 - Définir l'héritage

Surcharge des constructeurs et chaînage

Chainage des constructeurs

- Le chaînage de constructeurs est une technique **d'appel d'un constructeur de la classe mère à partir d'un constructeur de la classe fille**
- Le chaînage des constructeurs permet **d'éviter la duplication du code** d'initialisation des attributs qui sont hérités par la classe fille





CHAPITRE 1

Définir l'héritage

1. Principe de l'héritage
2. Types d'héritage
3. Surcharge des constructeurs et chaînage
4. **Visibilité des attributs et des méthodes de la classe fille**

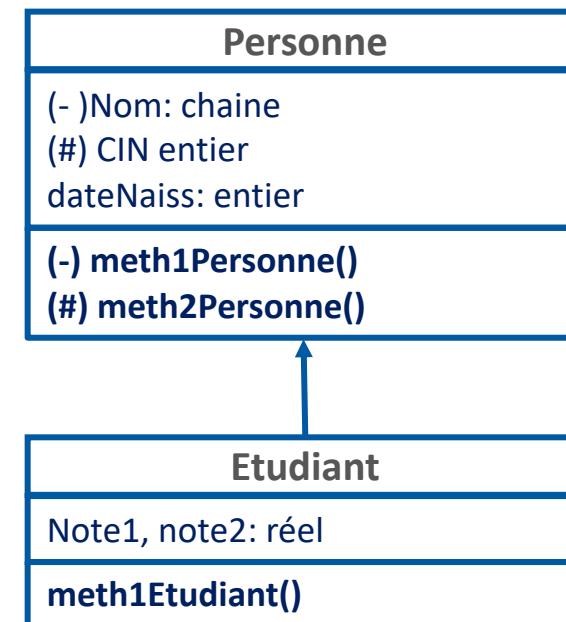
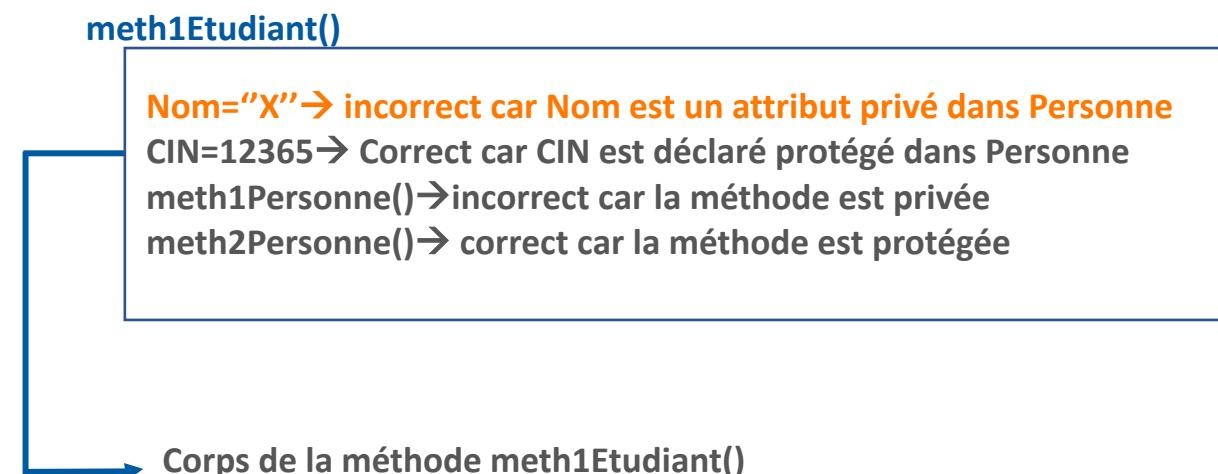
01 - Définir l'héritage

Visibilité des attributs et des méthodes de la classe fille

Visibilité des membres et héritage

- Si une classe mère définit certains (ou tous) de ses **attributs et méthodes**, comme **privés**, alors aucune de ses **classes filles** ne peut y accéder, ni les modifier.
- Pour permettre à une classe fille de lire ou modifier un attribut hérité, la classe mère doit les déclarer comme **protégés**.

Exemple :



CHAPITRE 2

Définir le polymorphisme



Ce que vous allez apprendre dans ce chapitre :

- Comprendre le principe de polymorphisme
- Savoir redéfinir et surcharger des méthodes



03 heures



CHAPITRE 2

Définir le polymorphisme

1. **Principe du polymorphisme**
2. Redéfinition des méthodes
3. Surcharge des méthodes

02 - Définir le polymorphisme

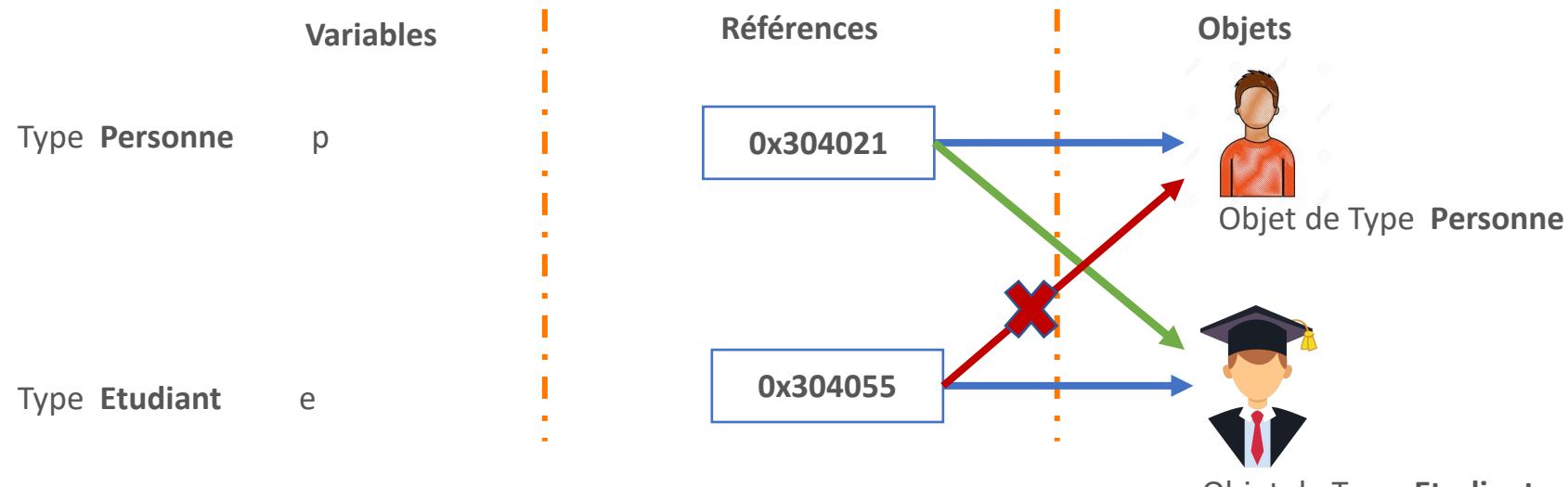
Principe du polymorphisme

Principe de polymorphisme

- Le polymorphisme désigne un concept de la théorie des types, selon lequel un nom d'objet peut désigner des instances de classes différentes issues d'une même arborescence.

Exemple :

- Une instance de Etudiant peut « être vue comme » une instance de Personne (**Pas l'inverse!!**)

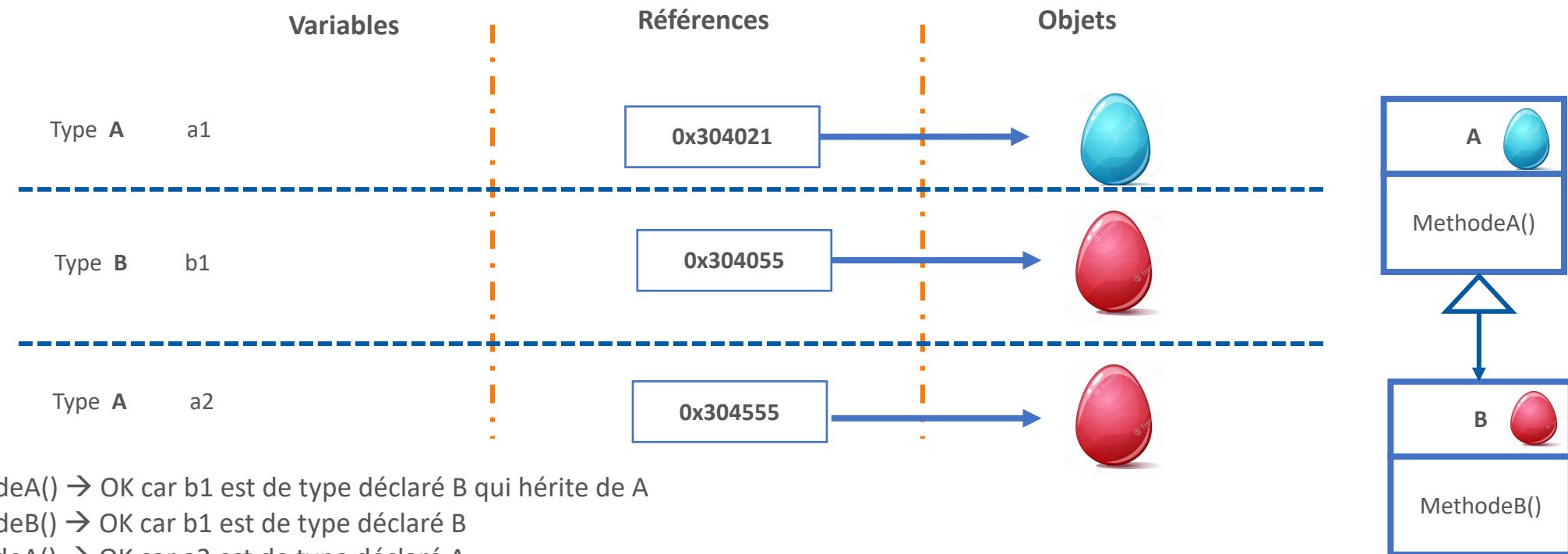


- = e → autorisé car le type de p est plus générique que le type de e
- e = p → non autorisé car le type de e est plus spécifique que celui de p

02 - Définir le polymorphisme

Principe de polymorphisme

- Le type de la variable est utilisé par le compilateur pour déterminer si on accède à un membre (attribut ou méthode) valide





CHAPITRE 2

Définir le polymorphisme

1. Principe du polymorphisme
2. **Redéfinition des méthodes**
3. Surcharge des méthodes

02 - Définir le polymorphisme

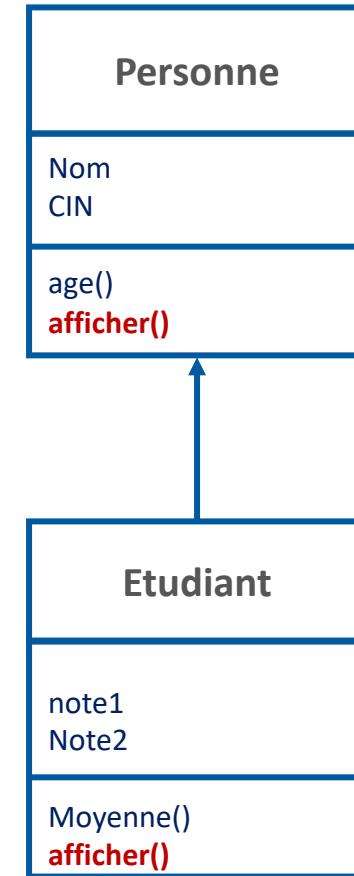
Redéfinition des méthodes

Redéfinition des méthodes héritées

- La redéfinition d'une méthode héritée est motivée par le fait que sa version, dans la classe mère, ne correspond pas aux besoins de la classe fille.
- La redéfinition permet alors de proposer **un code différent** à une méthode héritée tout en **gardant son entête**. Autrement, il ne peut s'agir d'une redéfinition mais d'une nouvelle méthode complètement différente de celle héritée.
- A partir d'une classe fille, il est possible, à tout moment, d'invoquer une méthode redéfinie, dans sa version initiale, déclarée dans la classe mère.

Exemple :

- Soit la classe Etudiant qui hérite de la classe Personne
- La méthode afficher() de la classe Personne affiche les attributs Nom, CIN d'une personne
- La classe Etudiant hérite la méthode afficher() de la classe Personne et
- la **redéfinit**, elle propose un nouveau code (la classe Etudiant **ajoute** l'affichage des attributs notes1 et note2 de l'étudiant)



02 - Définir le polymorphisme

Redéfinition des méthodes

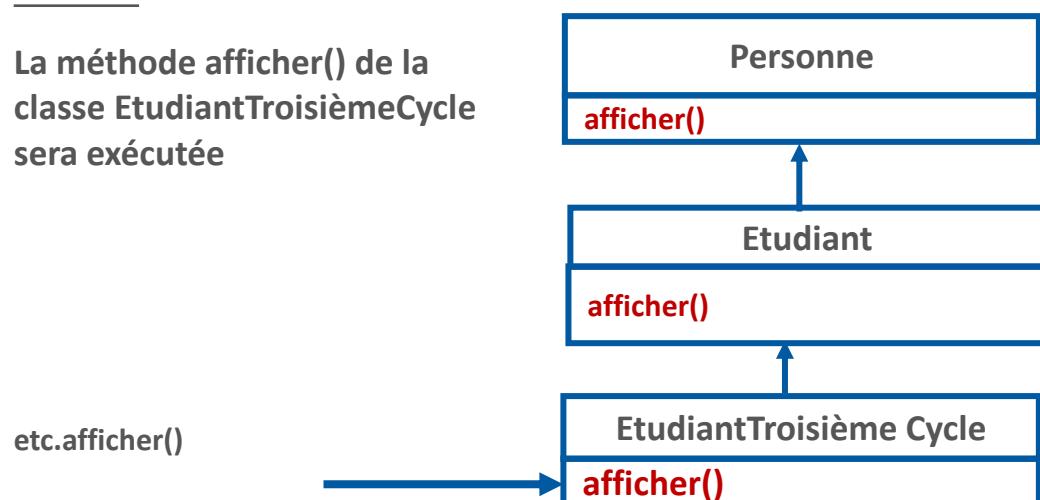
Mécanisme de la liaison retardée

- Soit 3 classes A, B et C, B hérite de A et C hérite de B.
- Soit o une instance (objet) de la classe A. On désire invoquer la méthode m() à partir de o, comme suit : « o.m() »
- o peut appeler m() qui représente la méthode appartenant à la classe A. La méthode peut être appelée par les classes filles de A à savoir B et C.
- Supposons que la classe C définit ou redéfinit la méthode m(). Si un objet de la classe C fait appel à la méthode m(), c'est celle de la classe C qui sera exécutée, en cas d'échec la recherche se poursuit au niveau de la classe B puis A et ainsi de suite.

Exemple : Soit etc un objet de type EtudiantTroisièmeCycle

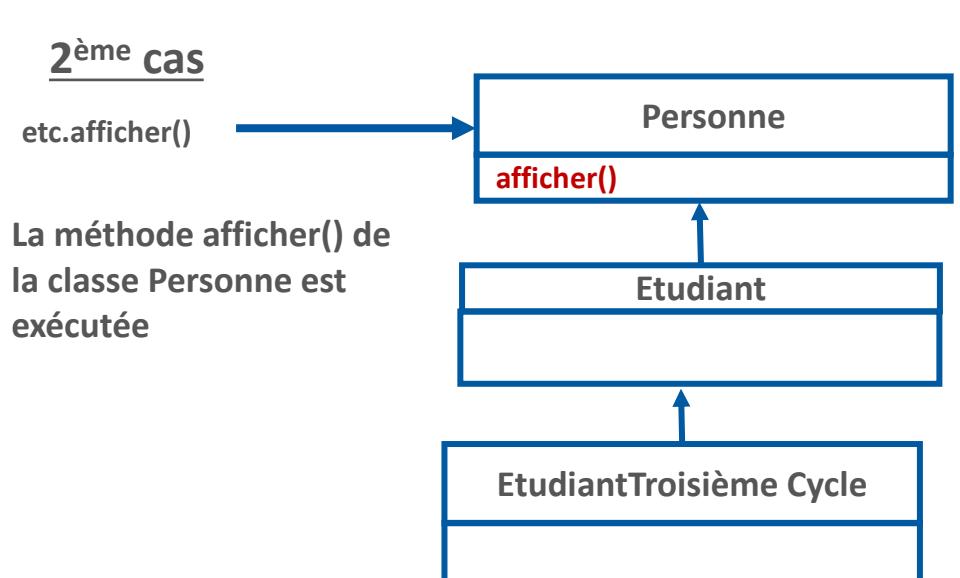
1^{er} cas

La méthode afficher() de la classe EtudiantTroisièmeCycle sera exécutée



2^{ème} cas

La méthode afficher() de la classe Personne est exécutée





CHAPITRE 2

Définir le polymorphisme

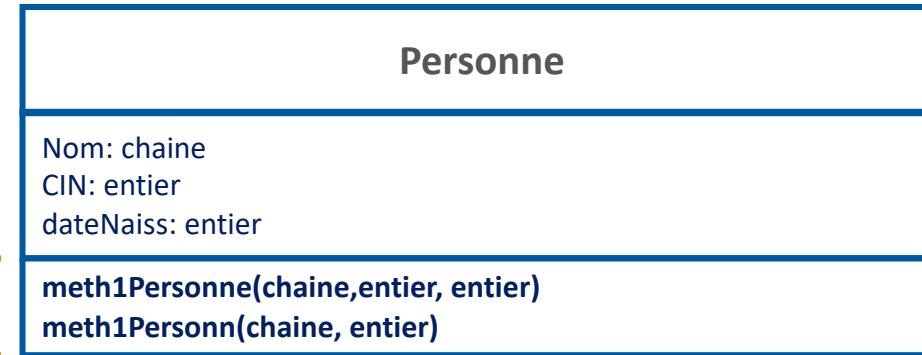
1. Principe du polymorphisme
2. Redéfinition des méthodes
3. **Surcharge des méthodes**

02 - Définir le polymorphisme

Surcharge des méthodes

- La surcharge d'une méthode permet de **définir plusieurs fois une même méthode avec des arguments différents.**
- Le compilateur choisit la méthode qui doit être appelée en fonction du nombre et du type des arguments.
- Une méthode est surchargée lorsqu'elle exécute des actions différentes selon le type et le nombre de paramètres transmis.

Surcharge de la
méthode
meth1Personne



CHAPITRE 3

Caractériser l'abstraction



Ce que vous allez apprendre dans ce chapitre :

- Comprendre le principe d'abstraction en POO
- Maîtriser les concepts de classes abstraites et méthodes abstraites



03 heures



CHAPITRE 3

Caractériser l'abstraction

1. **Principes**
2. Classes abstraites
3. Méthodes abstraites

03 - Caractériser l'abstraction

Principes



Principe de l'abstraction

- L'abstraction est un principe qui consiste à **ignorer certains aspects** qui ne sont pas importants pour le problème dans le but de **se concentrer sur ceux qui le sont**
- La POO permet, ainsi, de se focaliser sur **les caractéristiques importantes** d'un objet.
- Son objectif principal est de **gérer la complexité** en masquant les détails inutiles à l'utilisateur
 - Cela permet à l'utilisateur d'implémenter une logique plus complexe sans comprendre ni même penser à toute la complexité cachée



CHAPITRE 3

Caractériser l'abstraction

1. Principes
2. **Classes abstraites**
3. Méthodes abstraites

03 - Caractériser l'abstraction

Classes abstraites

- Contrairement à une classe "normale", **on ne peut créer des objets à partir d'une classe abstraite.**
- L'utilité d'une classe abstraite est essentiellement en tant que classe mère.
- En ce sens, une classe abstraite peut servir :
 - **Comme racine d'un héritage en cascade :**
 - Dans ce cas, chaque sous-classe spécifiera ses propres propriétés et méthodes.
 - Favorise le polymorphisme

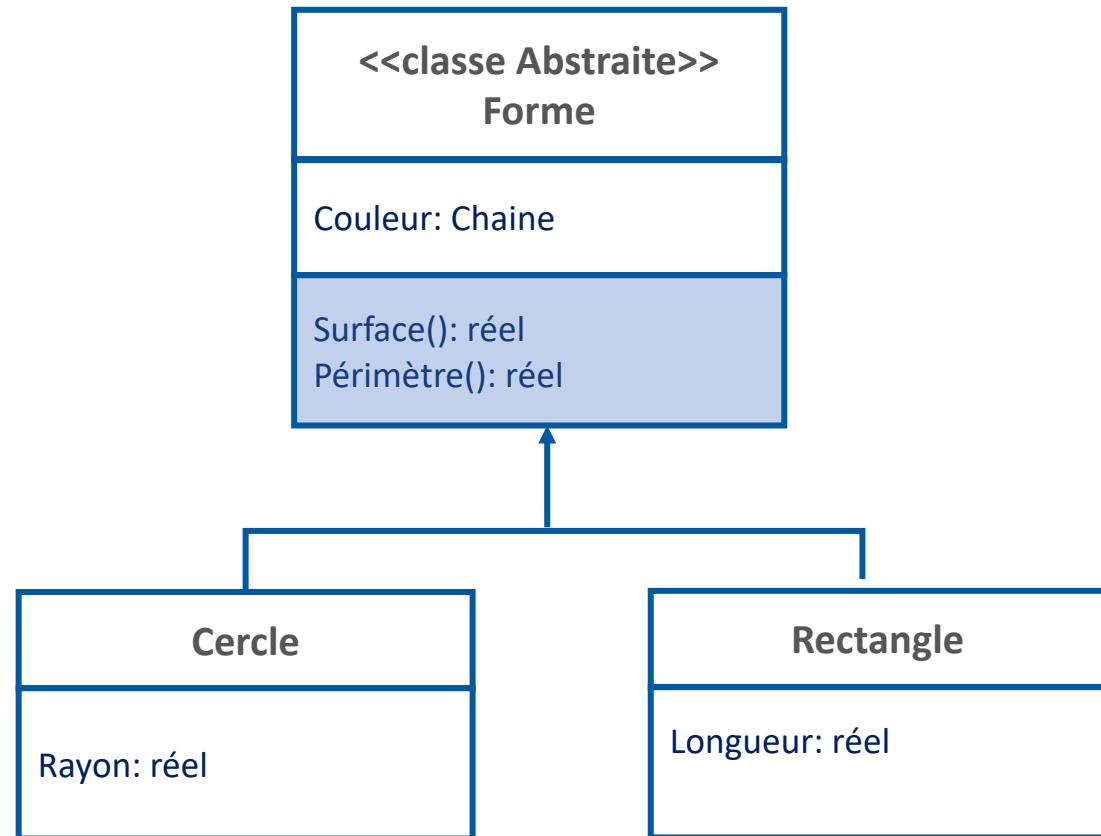
- Les classes abstraites permettent de définir des fonctionnalités (des méthodes) que **les sous-classes devront impérativement implémenter**
- Les utilisateurs des sous-classes d'une classe abstraite sont donc assurés de trouver toutes les méthodes définies dans la classe abstraite dans chacune des sous-classes concrètes
- Les classes abstraites constituent donc **une sorte de contrat** (spécification contraignante) qui garantit que certaines méthodes seront disponibles dans les sous-classes et qui oblige les programmeurs à les implémenter dans toutes les sous-classes concrètes

03 - Caractériser l'abstraction

Classes abstraites

Exemple :

- Sachant que toutes les formes possèdent les propriétés périmètre et surface, il est judicieux de placer les méthodes périmètre() et surface() dans la classe qui est à la racine de l'arborescence (Forme)
- Les méthodes surface() et périmètre() définies dans la classe Forme ne présentent pas de code et doivent impérativement être implémentées dans les classes filles
- Les méthodes surface() et périmètre() sont des méthodes abstraites





CHAPITRE 3

Caractériser l'abstraction

1. Principes
2. Classes abstraites
3. **Méthodes abstraites**

03 - Caractériser l'abstraction

Méthodes abstraites

- **Une méthode abstraite** est une méthode qui **ne contient pas de corps**. Elle possède simplement une signature de définition (pas de bloc d'instructions)
- Une méthode abstraite est déclarée dans **une classe abstraite**
- Une classe possédant une ou plusieurs méthodes abstraites devient obligatoirement une classe abstraite
- Il n'est pas indispensable d'avoir des méthodes abstraites dans une classe abstraite.

Les règles suivantes s'appliquent aux classes abstraites :

- Une sous-classe d'une classe abstraite ne peut être instanciée que si elle redéfinit chaque méthode abstraite de sa classe parente et qu'elle fournit une implémentation (un corps) pour chacune des méthodes abstraites
- Si une sous-classe d'une classe abstraite n'implémente pas toutes les méthodes abstraites dont elle hérite, cette sous-classe est elle-même abstraite (et ne peut donc pas être instanciée)

CHAPITRE 4

Manipuler les interfaces



Ce que vous allez apprendre dans ce chapitre :

- Comprendre le principe des interfaces et leurs utilités
- Maîtriser l'implémentation d'une interface



03 heures



CHAPITRE 4

Manipuler les interfaces

1. Définition des interfaces
2. Utilité des interfaces
3. Implémentation des interfaces

04 - Manipuler les interfaces

Définition des interfaces

- Comme une classe et une classe abstraite, une interface permet de définir un nouveau type (référence).
- Une interface est une forme particulière de classe où **toutes les méthodes sont abstraites**.

<<interface>>
imprimable

Imprimer()





CHAPITRE 4

Manipuler les interfaces

1. Définition des interfaces
2. **Utilité des interfaces**
3. Implémentation des interfaces

04 - Manipuler les interfaces

Utilité des interfaces

Utilité des interfaces

Les interfaces permettent de :

- Spécifier des propriétés qui peuvent être utilisées par les classes qui implémentent ces interfaces.
- Obliger les classes qui les implémentent de définir les méthodes abstraites déclarées dans les interfaces.
- Tirer profit du polymorphisme avec des instances dont les classes ne font pas partie de la même hiérarchie d'héritage.



CHAPITRE 4

Manipuler les interfaces

1. Définition des interfaces
2. Utilité des interfaces
3. **Implémentation des interfaces**

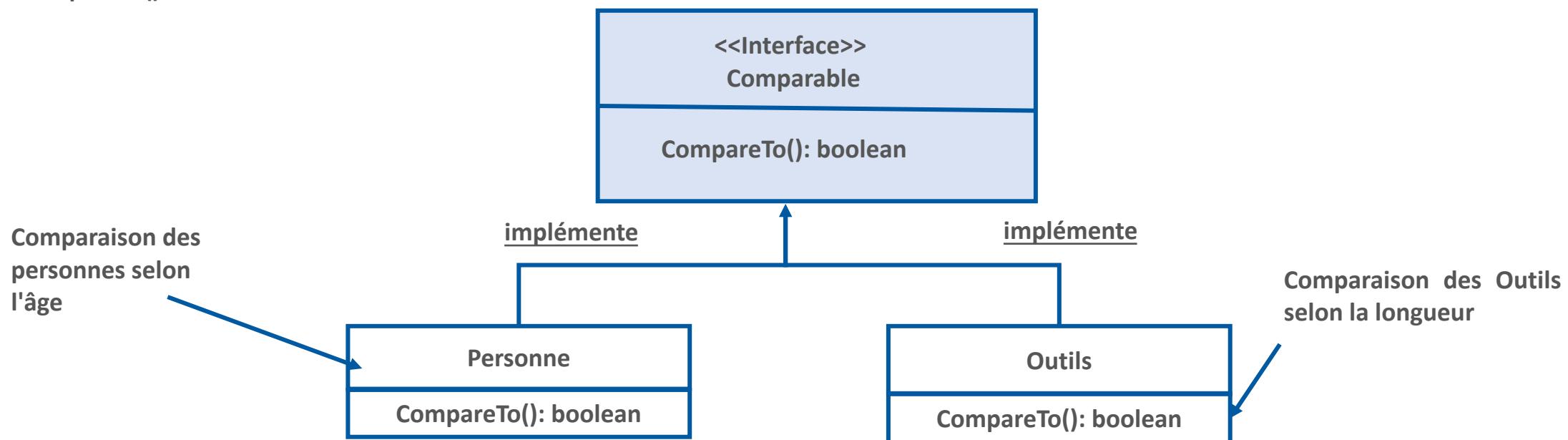
04 - Manipuler les interfaces

Implémentation des interfaces

- On dit qu'une **classe implémente une interface**, si elle fournit une implémentation (c'est-à-dire un corps) pour chacune des méthodes abstraites de cette interface.
- Si une classe implémente plus d'une interface, elle doit implémenter toutes les méthodes abstraites de chacune des interfaces.

Exemple 1 :

- Si l'on souhaite caractériser la fonctionnalité de comparaison qui est commune à tous les objets qui ont une relation d'ordre (plus petit, égal, plus grand), on peut définir l'interface Comparable.
- Les classes Personne et Outils qui implémentent l'interface Comparable **doivent présenter une implémentation de la méthode CompareTo()** sinon elles seront abstraites.



04 - Manipuler les interfaces

Implémentation des interfaces

Exemple 2 :

- Supposons que nous voulions que les classes dérivées de la classe Forme disposent toutes d'une méthode imprimer() permettant d'imprimer les formes géométriques.

Solution 1:

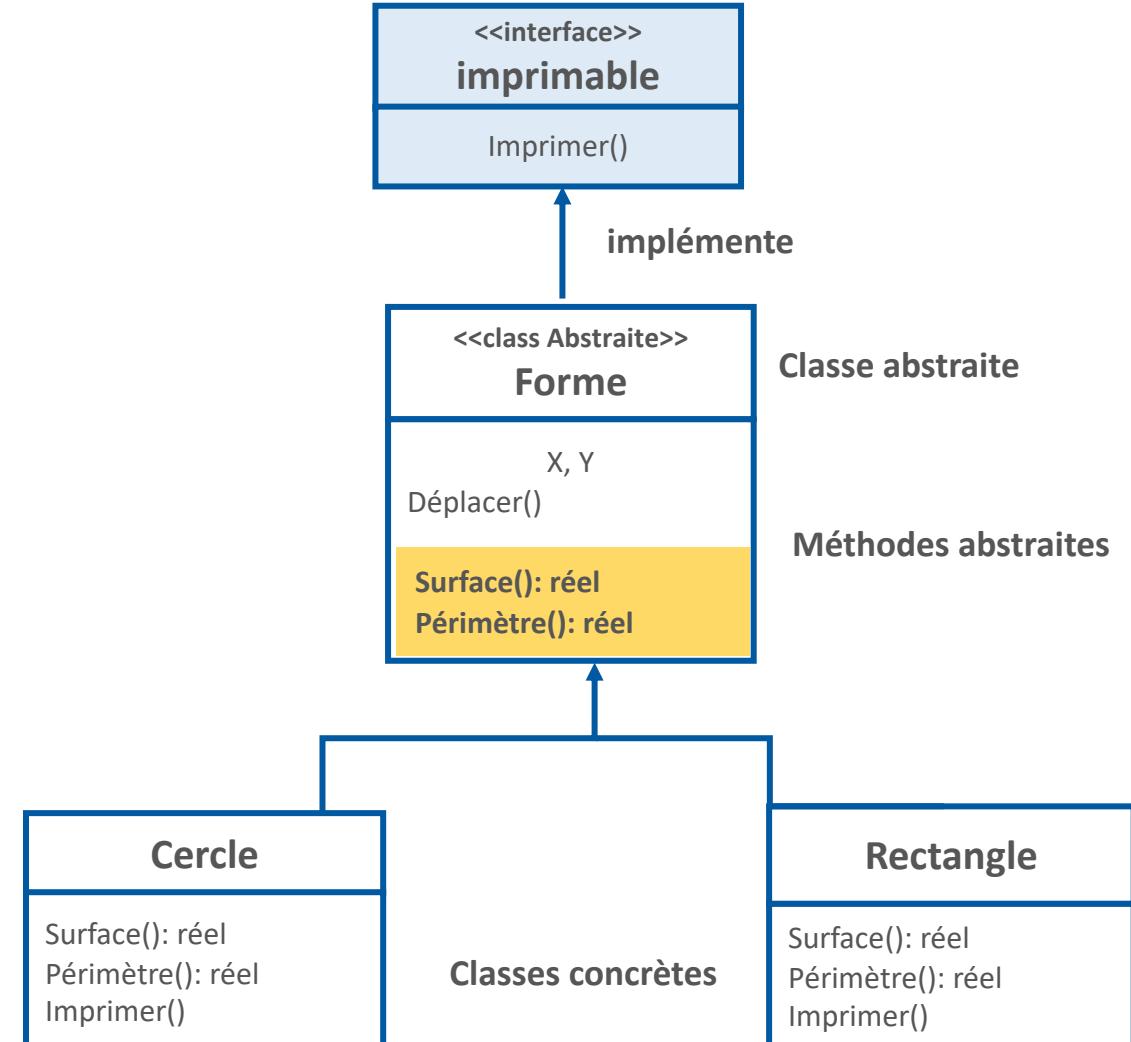
- Ajouter une méthode abstraite Imprimer() à la classe Forme et ainsi chacune des sous-classes concrètes devrait implémenter cette méthode.

→ Si d'autres classes (qui n'héritent pas de Forme) souhaitaient également disposer des fonctions d'impression, elles devraient à nouveau déclarer des méthodes abstraites d'impression dans leur arborescence.

Solution 2 :

La classe forme implémente l'interface imprimable

- ayant la méthode Imprimer()





PARTIE 3

Coder des solutions orientées objet

Dans ce module, vous allez :

- Explorer les concepts de base de l'orienté objet en langage Python
- Manipuler correctement des données en Python (collections, listes, fichiers)
- Manipuler correctement les expressions régulières en Python
- Gérer les exceptions en Python



 **50 heures**



CHAPITRE 1

Coder une solution orientée objet

Ce que vous allez apprendre dans ce chapitre :

- Maitriser le codage d'une classe en Python
- Maitriser le codage des principaux paliers de la POO en Python à savoir l'encapsulation, l'héritage, le polymorphisme et l'abstraction

25 heures



CHAPITRE 1

Coder une solution orientée objet

1. **Création d'un package**
2. Codage d'une classe
3. Intégration des concepts POO

01 - Coder une solution orientée objet

Création d'un package

- Un répertoire contenant des fichiers et/ou répertoires = package
- Pour créer votre propre package, commencez par créer dans le même dossier que votre programme principal (exemple main.py), un dossier portant le nom de votre package (exemple src).
- Le package src contient des fichiers sources classés dans des sous-packages
- Avant Python 3.3, un package contenant des modules Python doit contenir un fichier `__init__.py`

 src	2021-10-15 22:15	Dossier de fichiers
 main	2021-10-15 22:08	Python File

package src

 __init__	2021-10-15 22:08	Python File
 ExempleClass	2021-10-15 22:08	Python File

01 - Coder une solution orientée objet

Création d'un package



- Pour importer et utiliser les classes (classe1 et classe2) définies dans ExempleClass.py, il faut ajouter dans main.py la ligne suivante :

```
from src.ExempleClass import class1,class2
```

- Si ExempleClass.py est dans subpackage qui est défini dans src il faut ajouter dans main.py la ligne suivante :

```
from srcsubpackage.ExempleClass import class1,class2
```



CHAPITRE 1

Coder une solution orientée objet

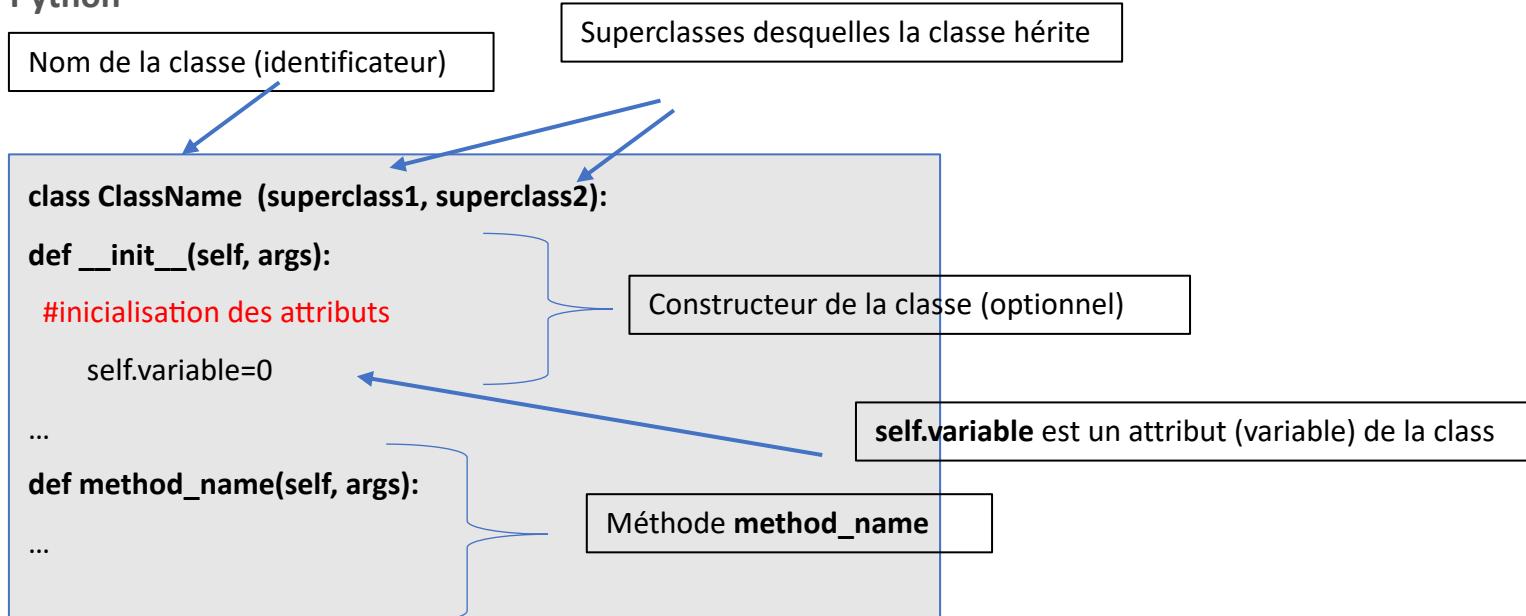
1. Création d'un package
2. **Codage d'une classe**
3. Intégration des concepts POO

01 - Coder une solution orientée objet

Codage d'une classe

Présentation d'une classe

- Définition de classe en Python



- Par convention le nom d'une classe commence par une majuscule
- Le mot-clé `pass` décrit le fait que la classe est vide.

```
class classe_vide:
    pass
```

01 - Coder une solution orientée objet

Codage d'une classe



Méthodes

- Les méthodes représentent des fonctions assignées directement à une classe. Elles accèdent automatiquement aux attributs de la même classe.
- Les méthodes sont des fonctions qui contiennent le paramètre explicite (self) qui représente l'instance de la classe courante. Ce paramètre permet l'utilisation des données de cette classe.

Déclaration d'une méthode

```
class nom_classe :  
    def nom_methode(self, param_1, ..., param_n):  
        # corps de la méthode...
```

Self désigne l'instance courante de la classe

Appel d'une méthode

```
cl = nom_classe() # variable de type nom_classe  
t = cl.nom_methode (valeur_1, ..., valeur_n)
```

01 - Coder une solution orientée objet

Codage d'une classe



Attributs

- Les attributs sont des variables assignées de manière directe à une classe.
- Les attributs de la classe sont des variables globales qui peuvent être utilisées dans toutes les méthodes de cette classe.

Déclaration d'un attribut

```
class nom_classe :  
    def nom_methode (self, param_1, ..., param_n) :  
        self.nom_attribut = param_1
```

Déclaration d'un attribut en précédant son nom par **self**

01 - Coder une solution orientée objet

Codage d'une classe



Méthodes et attributs

Exemple :

```
class exemple_classe:  
    « on cherche à générer un nombre aléatoire entre 0 et n1 »  
    def methode1(self,n):  
        self.rnd=42 #rnd est un attribut qui doit être précédé par self  
        self.rnd=397204094*self.rnd%2147483647  
        return self.rnd % n  
  
    nb=exemple_classe() #création d'une instance d'une classe  
    x=nb.methode1(100) #appel de la méthode  
    print(x) #affiche 19
```

01 - Coder une solution orientée objet

Codage d'une classe



Constructeur et instantiation

- Le constructeur d'une classe est une méthode dont le nom commence obligatoirement par `__init__`
- Le constructeur doit avoir comme premier paramètre l'attribut `self`. Il peut également contenir un ensemble de paramètres. Le constructeur ne doit jamais retourner de résultat.

Déclaration d'un constructeur

```
class nom_classe :  
    def __init__(self, param_1, ..., param_n):  
        # code du constructeur
```

Déclaration d'un constructeur à n paramètres

Appel d'un constructeur

```
x = nom_classe (valeur_1,...,valeur_n)
```

Exemple :

```
class classe1:  
    def __init__(self):  
        # pas de paramètres supplémentaires  
        print("constructeur de la classe classe1")  
        self.n = 1 # ajout de l'attribut n  
  
x = classe1() # affiche constructeur de la classe classe1  
print(x.n) # affiche 1
```

```
class classe2:  
    def __init__(self, a, b):  
        # deux paramètres supplémentaires  
        print("constructeur de la classe classe2")  
        self.n = (a + b) / 2 # ajout de l'attribut n
```

```
x = classe2(5, 9) # affiche constructeur de la classe classe2  
print(x.n) # affiche 7
```

01 - Coder une solution orientée objet

Codage d'une classe



Constructeur et instantiation

- Par défaut, toute classe en Python a un constructeur par défaut sans paramètre
- Le constructeur par défaut n'existe plus si la classe présente un constructeur à paramètre
- Contrairement à d'autres langues, la classe de Python n'a qu'un seul constructeur. Cependant, Python permet à un paramètre de prendre une valeur par défaut.
- Remarque : Tous les paramètres requis doivent précéder tous les paramètres qui ont des valeurs par défaut.

Exemple :

```
class Personne :  
  
    # Les paramètres d'âge et de sexe ont une valeur par défaut.  
    def __init__(self, nom, age = 1, sexe = "Male"):  
        self.nom = nom  
        self.age = age  
        self.sex = sexe  
  
    def afficherInfo(self):  
        print ("Name: ", self.nom)  
        print ("Age: ", self.age)  
        print ("sexe: ", self.sex)
```

```
if __name__=='__main__':  
    #Créez un objet de Personne.  
    aimee = Personne("Aimee", 21, "Femelle")  
    aimee.afficherInfo() #affiche Aimee 21 Femelle  
    print (" ----- ")  
  
    #age, sexe par défaut.  
    alice = Personne( "Alice" ) #affiche Alice 1 Male  
    alice.afficherInfo()  
    print (" ----- ")  
  
    #sexe par défaut.  
    tran = Personne("Tran", 37) #affiche Tran 37 Male  
    tran.afficherInfo()
```

01 - Coder une solution orientée objet

Codage d'une classe



Instanciation d'une classe

Affichage de l'instance

```
class Personne :  
    # Les paramètres d'âge et de sexe ont une valeur par défaut.  
    def __init__(self, nom, age = 1, sexe = "Male"):  
        self.name = nom  
        self.age = age  
        self.sex = sexe  
  
    if __name__=='__main__':  
        aimee =Personne("Aimee",21,« Femelle»)  
        print(aimee) #affiche <__main__.Personne object at 0x000002047E549070>
```

- La fonction `isinstance()` permet de s'assurer qu'une instance a été créé à partir d'une classe donnée.

```
class Personne :  
    # Les paramètres d'âge et de sexe ont une valeur par défaut.  
    def __init__(self, nom, age = 1, sexe = "Male"):  
        self. nom = nom  
        self.age = age  
        self.sex= sexe  
  
    if __name__=='__main__':  
        aimee =Personne("Aimee",21,"Female")  
        print(isinstance(aimee,Personne)) #affiche True car aimee est de type Personne
```

Destructeur

- Les **destructeurs** sont appelés lorsqu'un objet est détruit.
- les destructeurs ne sont pas nécessaires car Python dispose d'un ramasse-miettes assurant la gestion automatiquement de la mémoire.
- La méthode `__del__()` est une méthode appelée destructeur en Python. Il est appelé lorsque toutes les références à l'objet ont été supprimées, c'est-à-dire lorsqu'un objet est nettoyé.

Syntaxe de destructeur

```
def __del__(self):  
    # actions
```

01 - Coder une solution orientée objet

Codage d'une classe

Exemple :

```
class Personne :  
  
    def __init__(self, nom, age = "1", sexe = "Male"):  
        self.nom = nom  
        self.age = age  
        self.sex = sexe  
  
    def afficheInfo(self):  
        print(self.nom + " " + self.age + " " + self.sex)  
  
    def __del__(self): #definition d'un destructeur  
        print("je suis le destructeur")
```

```
if __name__ == '__main__':  
  
    aimee = Personne("Aimee", "21", "Femelle")  
    aimee.afficheInfo() #appel de la fonction showInfo()  
  
    del aimee #appel du destructeur  
  
    print(aimee) #affiche une erreur car l'objet a été détruit  
  
  
#Affichage:  
  
#Aimee 21 Femelle  
#je suis le destructeur  
#Traceback (most recent call last):  
#File "C:/Users/DELL/Desktop/poo/exemple.py", line 19, in <module>  
#    print(aimee)  
#NameError: name 'aimee' is not defined
```

01 - Coder une solution orientée objet

Codage d'une classe



Appart du langage Python

- Dans un code destiné à être réutilisé, il faut absolument définir dans le documentation ce que fait la classe et ses entrées et sorties
- L'attribut spécial `__doc__` affiche la documentation de la classe

```
class Vehicule():
    """
    Classe Véhicule avec option couleur
    """

    def __init__(self):
        self.couleur="rouge"

    def get_couleur(self):
        print ("Recuperation de la couleur")

    if __name__=='__main__':
        ma_voiture= Vehicule()
        print(ma_voiture.__doc__) # affiche : Classe voiture avec option couleur
```

01 - Coder une solution orientée objet

Codage d'une classe



Appart du langage Python

L'attribut spécial `__dict__`

- Cet attribut spécial donne les valeurs des attributs de l'instance :

```
if __name__=='__main__':
    ma_voiture= Vehicule()
    print( ma_voiture.__dict__) #affiche {'couleur': 'rouge'}
```

Fonction `dir`

- La fonction `dir` donne un aperçu des méthodes de l'objet :

```
if __name__=='__main__':
    ma_voiture= Vehicule()
    print(dir(ma_voiture)) #affiche les méthodes de la classe Voiture
```

01 - Coder une solution orientée objet

Codage d'une classe



Appart du langage Python

Affichage d'un objet

- `__str__` : vous donne la possibilité de redéfinir l'affichage d'un objet

```
class Vehicule():
    """
    Classe Vehicule avec option couleur
    """

    def __init__(self):
        self.couleur="rouge"
    def get_couleur(self):
        print ("Récupération de la couleur")

    def __str__(self): #définition de la fonction __str__
        return ("la couleur de la véhicule est :" + self.couleur)

if __name__=='__main__':
    ma_voiture= Vehicule()
    print( ma_voiture) #appel de __str__ et
                      # affiche: la couleur de la Véhicule est rouge
```

01 - Coder une solution orientée objet

Codage d'une classe



Apport du langage Python

Ajout d'un attribut d'instance

- Il est possible d'ajouter un attribut uniquement pour une instance donnée via la syntaxe suivante :
`instance.nouvel_attribut = valeur`

```
if __name__=='__main__':
    ma_voiture=Voiture()
    print("attribut de ma voiture")
    print(ma_voiture.__dict__)
    sa_voiture=Voiture()
    sa_voiture.matricule=12345 #ajout d'un nouveau attribut pour l'instance sa_voiture
    print("attribut de sa voiture")
    print(sa_voiture.__dict__)
#Affiche:
#attribut de ma voiture
#[{'couleur': 'rouge'}]
#attribut de sa voiture
#[{'couleur': 'rouge', 'matricule': 12345}]
```

Attribut de classe

- Les **attributs de classe** sont différents des attributs d'instance.
- Un attribut dont la valeur est la même pour toutes les instances d'une classe est appelé un attribut de classe. Par conséquent, la valeur de l'attribut de classe est partagée par tous les objets.
- Les attributs de classe sont définis au niveau de la classe plutôt qu'à l'intérieur de la méthode `__init__()`.
- Contrairement aux attributs d'instance, les attributs de classe sont accessibles à l'aide du nom de la classe ou le nom d'instance.

```
class Citron:  
    forme=""  
    def donnerForme(self, params):  
        Citron.forme = params
```

forme est un attribut statique

01 - Coder une solution orientée objet

Codage d'une classe



Exemple :

```
class Fruit:  
    nom='Fruit à manger' #nom est attribut statique  
    def __init__(self, couleur, poids_g):  
        print("J'adore les fruits.")  
        self.couleur = couleur  
        self.poids_g = poids_g  
  
    if __name__=='__main__':  
        pomme =Fruit("verte",100)  
        print(pomme.nom) #affiche fruit à manger  
        print(Fruit.nom) #affiche fruit à manger
```

01 - Coder une solution orientée objet

Codage d'une classe



Attribut de classe

- La modification de l'attribut de classe à l'aide du nom de la classe affectera toutes les instances d'une classe
- La modification de l'attribut de classe à l'aide de l'instance n'affectera pas la classe et les autres instances. Cela n'affectera que l'instance modifiée.

Exemple :

```
class Fruit:  
    nom='Fruit à manger' #nom est attribut statique  
  
    def __init__(self, couleur, poids_g):  
        print(" J'adore les fruits.")  
        self.couleur = couleur  
        self.poids_g = poids_g
```

```
if __name__=='__main__':  
  
    pomme =Fruit("verte",100) # affiche J'adore les fruits.  
    banane =Fruit("jaune",100) # affiche J'adore les fruits.  
    print(banane.nom) #affiche Fruit à manger  
    pomme.nom="fruit d'été"  
    print(pomme.nom) #affiche fruit d'été  
    print(Fruit.nom) #affiche Fruit à manger  
    print(banane.nom) #affiche Fruit à manger
```

01 - Coder une solution orientée objet

Codage d'une classe



Méthodes statiques

Rappel : Les méthodes statiques peuvent être appelées sans création d'instance au préalable.

- Une méthode statique peut être appelée sans instance. De ce fait, le paramètre Self est inutile.
- **La déclaration d'une méthode statique se fait avec le décorateur @staticmethod**

Exemple :

```
class nom_class :  
    @staticmethod  
    def nom_methode(params, ...):  
        # corps de la méthode
```

Nom_méthode est une méthode statique

La méthode statique **méthode** ne nécessite
aucune création d'instance pour être appelée

```
class essai_class:  
    @staticmethod  
    def methode(): #déclaration d'une méthode statique  
        print("méthode statique")  
  
    if __name__=='__main__':  
        essai_class.methode() #affiche méthode statique
```

Ajout de méthodes

- Supposons que nous avons déclaré une fonction statique à l'extérieur de la classe dans laquelle nous travaillons. Cette même fonction peut être appelée comme étant une méthode statique dans notre classe.
- Dans l'exemple ci-dessous, nous déclarons une fonction **methode**. A l'aide de `staticmethod` nous indiquons à la classe `essai_class` que c'est une méthode statique.

Exemple :

```
def methode():
    print("méthode statique")

class essai_class:
    pass

if __name__=='__main__':
    essai_class.methode = staticmethod(methode) #ajout d'une méthode statique
    essai_class.methode() #affiche: méthode statique
```

01 - Coder une solution orientée objet

Codage d'une classe

Modificateur d'accès public

- Les membres d'une classe déclarés publics sont facilement accessibles depuis n'importe quelle partie du programme.
- Tous les membres de données et les fonctions membres d'une classe sont publics par défaut.

```
class EtudiantGeek:  
  
    def __init__(self, nom, age):  
        self.Nom = nom  
        self.Age = age  
  
    def afficheAge(self):  
        print("Age: ", self.Age)
```

Nom et Age sont des attributs publics

Méthode publique

```
if __name__ == '__main__':  
    obj = EtudiantGeek("R2J", 20)  
    print(" Nom: ", obj.Nom)  
    obj.afficheAge()  
  
#affiche:  
  
#Name: R2J  
#Age: 20
```

01 - Coder une solution orientée objet

Codage d'une classe



Modificateur d'accès protégé

- Les membres d'une classe déclarés protégés ne sont accessibles qu'à une classe qui en dérive.
- Les données membres d'une classe sont déclarées protégées en ajoutant un seul symbole de soulignement «_» avant le membre de données de cette classe.

Exemple :

**_nom, _num, et
_branche** sont
des attributs
protégés

```
class Etudiant:  
    _nom = None  
    _num = None  
    _branche = None  
  
    def __init__(self, nom, num, branche):  
        self._nom = nom  
        self._num = num  
        self._branche = branche  
  
    def _affiche(self):  
        print("Num: ", self._num)  
        print("branche: ", self._branche)
```

```
if __name__=='__main__':  
  
    obj = Etudiant("R2J", 1706256, "Information Technology")  
  
    obj._affiche()  
  
#Affiche:  
  
#Roll: 1706256  
  
"branche: Information Technology"
```

- Si on considère que la classe EtudiantGeek est dérivée de la classe Etudiant alors:
- **_nom, _num, et _branche** sont des membres de données protégés et la méthode **_affiche()** est une méthode protégée de la super classe Etudiant.
- La méthode **afficheAge()** est une fonction membre publique de la classe EtudiantGeek qui est dérivée de la classe Etudiant, la méthode **afficheAge()** de la classe EtudiantGeek accède aux données membres protégées de la classe Etudiant.

01 - Coder une solution orientée objet

Codage d'une classe



Modificateur d'accès privé

- Les membres d'une classe qui sont déclarés privés sont accessibles uniquement dans la classe, le modificateur d'accès privé est le modificateur d'accès le plus sécurisé.
- Les données membres d'une classe sont déclarées privées en ajoutant un double trait de soulignement «__» avant le membre de données de cette classe.

Exemple :

```
class EtudiantGeek :  
    __nom = None  
    __num = None  
    __branche = None  
  
    def __init__(self, nom, num, branche):  
        self.__nom = nom  
        self.__num = num  
        self.__branche = branche  
  
    def __affiche(self):  
        print("Name: ", self.__nom)  
        print("Roll: ", self.__num)  
        print("Branch: ", self.__branche)  
  
    def accessPrivateFunction(self):  
        self.__affiche()
```

```
if __name__=='__main__':  
  
    obj = EtudiantGeek("R2J", 1706256, "Information Technology")  
    obj.accessPrivateFunction()  
  
#Affiche:  
#Name: R2J  
#Roll: 1706256  
#Branch: Information Technology
```

- __nom, __num, __branche sont des membres privés, __affiche() est une fonction membre privée (elle ne peut être accédée que dans la classe) et accessPrivateFunction() est une fonction membre publique de la classe EtudiantGeek et accessible de n'importe où dans le programme.
- La accessPrivateFunction() est une méthode qui accède aux membres privés de la classe EtudiantGeek.



CHAPITRE 1

Coder une solution orientée objet

1. Création d'un package
2. Codage d'une classe
3. **Intégration des concepts POO**

Getter et Setter en Python

- Les Getters et Setters en Python sont souvent utilisés pour éviter l'accès direct à un champ de classe, c'est-à-dire que les variables privées ne peuvent pas être accessibles directement ou modifiées par un utilisateur externe.

Utilisation de la fonction normale pour obtenir le comportement des Getters et des Setters

- Pour obtenir la propriété Getters et Setters, si nous définissons les méthodes `get()` et `set()`, cela ne reflètera aucune implémentation spéciale.
- Exemple :

```
class EtudiantGeek:  
    def __init__(self, age = 0):  
        self._age = age  
  
    def get_age(self): #déclarartion d'un getter  
        return self._age  
  
    def set_age(self, x): #declaration d'un setter  
        self._age = x
```

```
if __name__=='__main__':  
    raj = EtudiantGeek()  
    raj.set_age(21)  
    print(raj.get_age())  
    print(raj._age)  
#Affiche:  
#21  
#21
```

Getter et Setter avec `property()`

- En Python `property()` est une fonction intégrée qui crée et renvoie un objet de propriété.
- Un objet de propriété a trois méthodes, `getter()`, `setter()` et `delete()`.
- La fonction `property()` en Python a trois arguments **`property(fget, fset, fdel, doc)`**
 - **`fget`** est une fonction pour récupérer une valeur d'attribut
 - **`fset`** est une fonction pour définir une valeur d'attribut
 - **`fdel`** est une fonction pour supprimer une valeur d'attribut
 - **`doc`** est une chaîne contenant la documentation (docstring à voir ultérieurement) de l'attribut

01 - Coder une solution orientée objet

Intégration des concepts POO



- Exemple

```
class EtudiantGeek :  
    def __init__(self):  
        self._age = 0  
  
    def get_age(self):  
        print("getter est appelé")  
        return self._age  
  
    def set_age(self, a):  
        print("setter est appelé")  
        self._age = a  
  
    def del_age(self):  
        del self._age  
age = property(get_age, set_age, del_age)
```

appel

```
if __name__ == '__main__':  
  
    mark = EtudiantGeek()  
    mark.age = 10  
    print(mark.age)  
  
#affiche:  
#setter est appelé  
#getter est appelé  
#10
```

01 - Coder une solution orientée objet

Intégration des concepts POO



Utilisation de @property

- **@property** est un décorateur qui évite d'utiliser la fonction getter explicite
- **@attribut.setter** est un décorateur qui évite d'utiliser la fonction setter explicite

Exemple :

```
class EtudiantGeek:  
    def __init__(self):  
        self._age = 0  
  
    @property  
    def age(self):  
        print("getter method called")  
        return self._age  
  
    @age.setter  
    def age(self, a):  
        print("setter method called")  
        self._age = a
```

appel

```
if __name__=='__main__':  
    mark = EtudiantGeek()  
    mark.age = 19 #appel de setter  
    print(mark.age) #appel de getter  
  
#affiche:  
#setter method called  
#getter method called  
#19
```

01 - Coder une solution orientée objet

Intégration des concepts POO



Héritage en Python

- Rappel : l'héritage est défini comme la capacité d'une classe à dériver ou à hériter des propriétés d'une autre classe et à l'utiliser chaque fois que nécessaire.

Syntaxe en Python

Class Nom_classe(Nom_SuperClass)

Exemple :

```
class Enfant:  
    def __init__(self, nom):  
        self.nom = nom  
    def getNom(self):  
        return self.nom  
    def isEtudiant(self):  
        return False  
  
class Etudiant(Enfant):  
    def isStudent(self):  
        return True
```

```
if __name__=='__main__':  
    std = Enfant("Ram")  
    print(std.getNom(), std.isEtudiant())  
    std = Etudiant("Shivam")  
    print(std.getNom(), std.isEtudiant())  
  
#Affiche:  
#Ram False  
#Shivam <bound method Enfant.isEtudiant of <__main__.Etudiant object at  
0x0000024192C89250>
```

Classe Etudiant hérite de Enfant

01 - Coder une solution orientée objet

Intégration des concepts POO



Héritage unique

- **Rappel :** L'héritage unique permet à une classe dérivée d'hériter des propriétés d'une seule classe parente, permettant ainsi la réutilisation du code et l'ajout de nouvelles fonctionnalités au code existant.

Exemple :

```
class Parent:  
    def func1(self):  
        print("This function is in parent class.")  
  
class Enfant(Parent): #hérite de la classe Parent  
    def func2(self):  
        print("This function is in child class.")
```

```
if __name__=='__main__':  
    object = Enfant()  
    object.func1()  
    object.func2()  
  
#affiche:  
#This function is in parent class.  
#This function is in child class.
```

01 - Coder une solution orientée objet

Intégration des concepts POO



Héritage multiple

- Rappel : Lorsqu'une classe peut être dérivée de plusieurs classes de base, ce type d'héritage est appelé héritage multiple. Dans l'héritage multiple, toutes les fonctionnalités des classes de base sont héritées dans la classe dérivée.

Exemple :

```
class Mere:  
    nomMere = ""  
  
    def mere(self):  
        print(self.nomMere)  
  
class Pere:  
    nomPere = ""  
  
    def pere(self):  
        print(self.nomPere)  
  
class Enfant(Mere, Pere): #hérite de la classe Mere et Pere  
    def parents(self):  
        print("Father :", self.nomPere) # nomPere est attribut hérité  
        print("Mother :", self.nomMere) # nomMere est attribut hérité
```

```
if __name__ == '__main__':  
    s1 = Enfant()  
    s1.nomPere = "RAM"  
    s1.nomMere = "SITA"  
    s1.parents()  
  
#Affiche:  
#Father : RAM  
#Mother : SITA
```

01 - Coder une solution orientée objet

Intégration des concepts POO



Héritage en cascade

- Dans l'héritage en cascade, les fonctionnalités de la classe de base et de la classe dérivée sont ensuite héritées dans la nouvelle classe dérivée

```
class GrandPere:  
  
    def __init__(self, nomGrandPere):  
        self.nomGrandPere = nomGrandPere  
  
class Pere(GrandPere):  
  
    def __init__(self, nomPere, nomGrandPere):  
        self.nomPere = nomPere  
        GrandPere.__init__(self, nomGrandPere)  
  
class Enfant(Pere):  
  
    def __init__(self, nomEnfant, nomPere, nomGrandPere):  
        self.nomEnfant = nomEnfant  
        Pere.__init__(self, nomPere, nomGrandPere)  
  
    def affiche_name(self):  
        print('nom grand_père :', self.nomGrandPere)  
        print("nom_père:", self.nomPere)  
        print("nom enfant:", self.nomEnfant)
```

```
if __name__=='__main__':  
  
    s1 = Enfant('Prince', 'Rampal', 'Lal mani')  
    print(s1.nomGrandPere)  
    s1.affiche_name()  
  
#affiche:  
  
#Lal mani  
  
# nom grand_père : Lal mani  
# nom_père: Rampal  
# nom enfant: Prince
```

Chainage de constructeurs

- Le chaînage de constructeurs est une technique **d'appel d'un constructeur de la classe mère à partir d'un constructeur de la classe fille**

Exemple :

```
class GrandPere:  
  
    def __init__(self, nomGrandPere):  
  
        self.nomGrandPere = nomGrandPere  
  
class Pere(GrandPere):  
  
    def __init__(self, nomPere, nomGrandPere):  
  
        self.nomPere = nomPere  
  
        GrandPere.__init__(self, nomGrandPere)
```

Appel du constructeur de la classe mère pour initialiser l'attribut hérité
nomGrandPere

01 - Coder une solution orientée objet

Intégration des concepts POO



Chainage de constructeurs

```
class Enfant(Pere):  
  
    def __init__(self,nomEnfant, nomPere, nomGrandPere):  
  
        self. nomEnfant = nomEnfant  
  
        Pere.__init__(self, nomPere, nomGrandPere)
```



Appel du constructeur de la classe mère pour initialiser les attributs hérités **nomPere** et **nomGrandPere**

```
def affiche_name(self):  
  
    print('nom grand_père :', self. nomGrandPere)  
  
    print("nom_père:", self. nomPere)  
  
    print("nom enfant:", self. nomEnfant)
```

```
if __name__=='__main__':  
  
    s1 = Enfant('Prince', 'Rampal', 'Lal mani')  
  
    print(s1. nomGrandPere)  
  
    s1.affiche_name()  
  
#affiche:  
  
#Lal mani  
  
# nom grand_père : Lal mani  
  
# nom_père: Rampal  
  
# nom enfant: Prince
```

Surcharge des opérateurs en Python

- La surcharge d'opérateurs vous permet de redéfinir la signification d'opérateur en fonction de votre classe.
- Cette fonctionnalité en Python, qui permet à un même opérateur d'avoir une signification différente en fonction du contexte.

Opérateurs arithmétiques

```
class Point:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
  
    if __name__ == '__main__':  
        p1 = Point(2, 4)  
        p2 = Point(5, 1)  
        p3 = p1+p2 #erreur
```

Affiche:

```
Traceback (most recent call last):  
File "prog.py", line 10, in < module>  
p3 = p1+p2  
TypeError: unsupported operand type(s) for +: 'Point' and 'Point'
```

Erreur!!!

Python ne savait pas comment ajouter deux objets Point ensemble.

01 - Coder une solution orientée objet

Intégration des concepts POO



Surcharge des opérateurs en Python

- Pour surcharger l'opérateur +, nous devrons implémenter la fonction `__add__()` dans la classe

Exemple :

```
class Point:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
  
    def __str__(self):  
        return "({0},{1})".format(self.x, self.y)  
  
    def __add__(self, p): #surcharge de l'opérateur +  
        a = self.x + p.x  
        b = self.y + p.y  
        return Point(a, b)  
  
if __name__ == '__main__':  
    p1 = Point(2, 4)  
    p2 = Point(5, 1)  
  
    p3 = p1+p2  
    print(p3) #affiche (7,5)
```

01 - Coder une solution orientée objet

Intégration des concepts POO



Fonctions spéciales de surcharge de l'opérateur en Python

Opérateur	Expression	Interprétation Python
Addition	p1=p2	p1.__add__(p2)
Soustraction	p1-p2	p1.__sub__(p2)
Multiplication	pP1*p2	p1.__mul__(p2)
Puissance	p1**p2	p1.__pow__(p2)
Division	p1/p2	p1.__truediv__(p2)
Division entière	p1//p2	p1.__floordiv__(p2)
Le reste(modulo)	p1%p2	p1.__mod__(p2)
ET binaire	p1&p2	p1.__and__(p2)
OU binaire	p1 p2	p1.__or__(p2)
XOR	p1^p2	p1.__xor__(p2)
NON binaire	~p1	p1.__invert__(p2)

01 - Coder une solution orientée objet

Intégration des concepts POO



Surcharge des opérateurs en Python

Opérateurs de comparaison

- En Python, il est possible de surcharger les opérateurs de comparaison.

```
import math

class Point:

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

    def __lt__(self, p):
        m_self = math.sqrt((self.x ** 2) + (self.y ** 2))
        m2_p = math.sqrt((p.x ** 2) + (p.y ** 2))
        return m_self < m2_p
```

```
if __name__ == '__main__':
    p1 = Point(2, 4)
    p2 = Point(5, 1)
    if p1 < p2:
        print("p2 est plus loin que p1")

#affiche:
#p2 est plus loin que p1
```

Surcharge de l'opérateur inférieur

01 - Coder une solution orientée objet

Intégration des concepts POO



Fonctions spéciales de surcharge de l'opérateur en Python

Opérateur	Expression	Interprétation Python
Inférieur à	p1<p2	p1.__lt__(p2)
Inférieur ou égal	p1<=p2	p1.__le__(p2)
Egal	p1==p2	p1.__eq__(p2)
Different	p1!=p2	p1.__ne__(p2)
Supérieur à	p1>p2	p1.__gt__(p2)
Supérieur ou égal	p1>=p2	p1.__ge__(p2)

Polymorphisme et héritage

- En Python, le polymorphisme permet de définir des méthodes dans la classe enfant qui ont le même nom que les méthodes de la classe parent.
- En héritage, la classe enfant hérite des méthodes de la classe parent.
- Il est possible de modifier une méthode dans une classe enfant dont elle a hérité de la classe parent (redéfinition de la méthode).

Exemple :

```
class Oiseau:  
  
    def intro(self):  
        print("Il y a différents types d'oiseaux")  
  
    def vol(self):  
        print("il y a des oiseaux qui volent d'autres non")  
  
class Moineau(Oiseau):  
  
    def vol(self):  
        print("moineau peut voler.")  
  
class Autruche (Oiseau):  
  
    def vol(self):  
        print("autruche ne volent pas.")
```

```
if __name__=='__main__':  
  
    obj_bird = Oiseau()  
    obj_spr = Moineau()  
    obj_ost = Autruche()  
  
    obj_bird.intro() #affiche: Il y a différents types d'oiseaux  
    obj_bird.vol() #affiche: il y a des oiseaux qui volent d'autres non.  
    obj_spr.intro() #affiche: Il y a différents types d'oiseaux  
    obj_spr.vol() #affiche: moineau peut voler.  
    obj_ost.intro() #affiche : Il y a différents types d'oiseaux.  
    obj_ost.vol() #affiche: autruche ne volent pas.
```

Classe abstraite

- Rappel: Les classes abstraites sont des classes qui ne peuvent pas être instanciées, elles contiennent une ou plusieurs méthodes abstraites (méthodes sans code)
- Une classe abstraite nécessite des sous-classes qui fournissent des implémentations pour les méthodes abstraites sinon ces sous-classes sont déclarées abstraites
- Une classe abstraite hérite de la **classe abstraite de base – ABC**
- L'objectif principal de la classe de base abstraite est de fournir un moyen standardisé de tester si un objet adhère à une spécification donnée
- Pour définir une méthode abstraite dans la classe abstraite, on utilise un décorateur **@abstractmethod**

01 - Coder une solution orientée objet

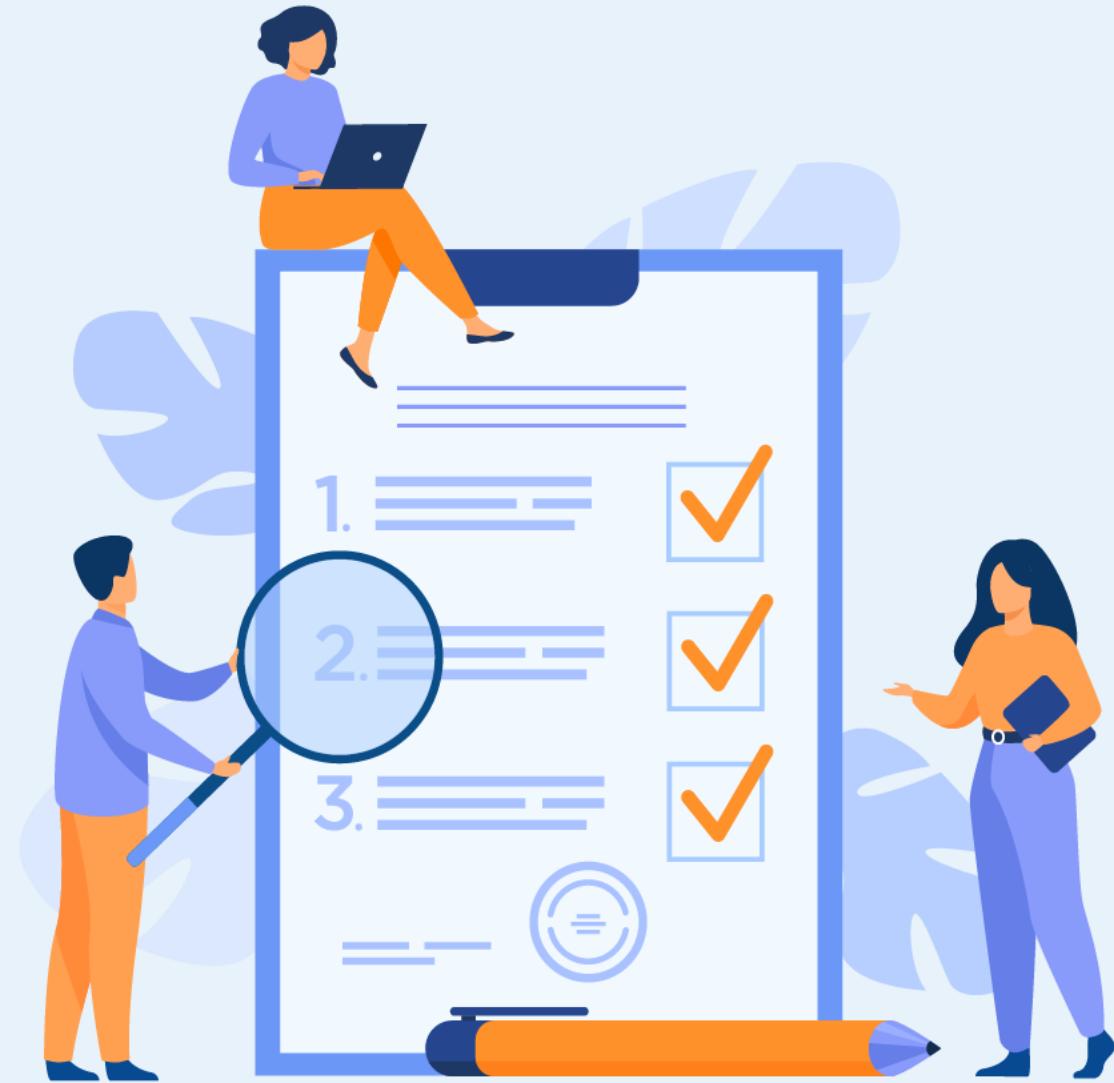
Intégration des concepts POO



Exemple

```
from abc import ABC, abstractmethod # abc est un module python intégré, nous importons ABC et  
abstractmethod  
  
class Animal(ABC): # hériter de ABC(Abstract base class)  
    @abstractmethod # un décorateur pour définir une méthode abstraite  
    def DonnerAManger(self):  
        pass  
  
class Cheval(Animal): #classe qui hérite de Animal  
    def autre_nom(self):  
        print(" Bonjour Cheval!")  
  
if __name__=='__main__':  
    po = Cheval() #instanciation de Cheval impossible car la méthode DonneAManger est abstraite elle doit être  
implémentée dans Cheval
```

Affiche:
Traceback (most recent call last):
File
"C:\Users\DELL\AppData\Local\Programs\Python
\Python39\ex0.py", line 14, in <module>
 po = Cheval() #instanciation de Cheval
impossible car la méthode nourrir est abstraite
elle doit être implémentée dans panda
TypeError: Can't instantiate abstract class Cheval
with abstract method DonnerAManger



CHAPITRE 2

Manipuler les données

Ce que vous allez apprendre dans ce chapitre :

- Manipuler les structures de données de Python à savoir les collections et les listes
- Manipuler les fichiers de données en Python



10 heures



CHAPITRE 2

Manipuler les données

1. Liste
2. Collections
3. Fichiers

02 - Manipuler les données

Listes



Listes

- Une **liste** est une **collection** qui est **ordonnée, modifiable et** qui peuvent contenir plusieurs fois la même valeur ;
- Une liste est déclarée comme suit :

Liste= [el1, elt2,...eln]

- **Liste[i]:** permet d'accéder à l'élément de la liste se trouvant à la i^{ème} position

```
cetteListe= ["apple","banana","cherry"]      #déclarer la liste
print(cetteListe)                            #afficher la liste
print(cetteListe[1])                          #afficher le premier élément de la liste
print(cetteListe[-1])                         #afficher la valeur de la position -1 (cycle)
cetteListe1= ["apple","banana","cherry","orange","kiwi","melon","mango"]
print(cetteListe1[2:5])                      #afficher les valeurs de la position 2 jusqu'à 5 (5 non inclus)
```

Affichage:
['apple', 'banana', 'cherry']
banana
cherry
['cherry', 'orange', 'kiwi']

- **Liste[i]= val:** permet de changer la valeur de l'élément de la liste se trouvant à la position i

```
cetteListe = ["apple","banana","cherry"]
cetteListe[1] ="blackcurrant"      #modifier la valeur de l'élément à la 1ère position
print(cetteListe) #affiche ['apple', 'blackcurrant', 'cherry']
```

Affichage:
['apple', 'blackcurrant', 'cherry']

02 - Manipuler les données

Listes



Listes

- Parcours une liste:

```
celleListe = ["apple", "banana", "cherry"]  
  
for x in celleListe :    # parcourir des éléments de la liste  
  
    print(x)      # afficher la valeur de x qui correspond à un élément de la liste
```

- Recherche d'un élément dans une liste:

```
celleListe= ["apple", "banana", "cherry"]  
  
if "apple" celleListe :      # vérifier si une chaîne est un élément de la liste  
  
    print("Yes, 'apple' appartient à la liste")
```

- Fonction len(Liste):** est une fonction permet de retourner la Longueur de la liste Liste

```
celleListe = ["apple", "banana", "cherry"]  
  
print(len(celleListe))  # afficher la longueur d'une liste
```

02 - Manipuler les données

Listes



Listes

- **Fonction append(élém):** est une fonction qui permet d'ajouter un élément élém à la fin de liste

```
 cetteListe = ["apple", "banana", "cherry"]
 cetteListe.append("orange") # ajouter de l'élément "orange" à la fin de la liste
 print(cetteListe)          # afficher ['apple', 'banana', 'cherry', 'orange']
```

- **Fonction insert(pos, élém)** est une fonction qui permet d'ajouter un élément élém à une position i de la liste

```
 cetteListe = ["apple", "banana", "cherry"]
 cetteListe.insert(1, "orange") # insérer l'élément "orange" à la deuxième position
 print(cetteListe)           # afficher les éléments de la liste ["apple", "orange", "banana", "cherry"]
```

- **Fonction pop()** est une fonction qui assure la suppression du dernier élément de la liste

```
 cetteListe = ["apple", "banana", "cherry"]
 cetteListe.pop() #supprimer le dernier de liste qui est cherry
 print(cetteListe) # afficher les éléments de la liste ["apple", "banana"]
```

02 - Manipuler les données

Listes



Listes

- **Fonction del(élém)** est une fonction qui permet de supprimer un élément particulier élém dans une liste

```
celleListe= ["apple", "banana", "cherry"]
Del(cetteListe[0]) #supprimer l'élément se trouvant à la première position
print(cetteListe) # affiche ["banana", "cherry"]
```

- **Fonction extend(Liste):** est une fonction qui permet de fusionner une liste

```
listA = ["a", "b" , "c"]
listB = [1, 8, 9]
listA.extend(listB) #fusionner le deux Listes Liste1 et Liste2
print(listA) #affiche ['a', 'b', 'c', 1,8, 9]
```

- **Fonction copy():** est une fonction qui permet de copier le contenu d'une liste dans une autre

```
cetteListe = ["apple", "banana", "cherry"]
malist= cetteListe.copy()
print(malist) #affiche ["apple", "banana", "cherry"]
```

02 - Manipuler les données

Listes



Listes

- **Fonction clear()** est une fonction qui permet de supprimer tous les éléments d'une liste

```
celleListe = ["apple", "banana", "cherry"]
celleListe.clear() #vider la liste
print(celleListe) #afficher une liste vide []
```

- **Fonction reverse()** est une fonction qui permet d'inverser une liste

```
celleListe = ["apple", "banana", "cherry"]
celleListe.reverse() #inverser la liste thislist
print(celleListe) #afficher la liste inversée ['cherry', 'banana', 'apple']
```



CHAPITRE 2

Manipuler les données

1. Liste
2. **Collections**
3. Fichiers

02 - Manipuler les données

Collections



- Le module collections contient des conteneurs de données spécialisés
- La liste des conteneurs est la suivante

Conteneur	Utilité
namedtuple	Une fonction permettant de créer une sous-classe de tuple avec des champs nommés
desque	Un conteneur ressemblant à une liste mais avec ajout et suppression rapide à chacun des bouts
ChainMap	Permet de linker entre eux plusieurs mappings ensemble pour les gérer comme tout
Counter	Permet de compter les occurrences d'objet hachable
OrderedDict	Une sous classe de dictionnaire permettant de savoir l'ordre des entrées
defaultdict	Une sous classe de dictionnaire permettant de spécifier une valeur par défaut dans le constructeur

Collections - namedtuple

- un tuple est une collection immuable de données souvent hétérogène.

```
t=(11,22)  
print(t) #affiche (11, 22)  
print(t[0]) #affiche 11  
print(t[1]) #affiche22
```

- La classe **namedtuple** du module collections permet d'ajouter des noms explicites à chaque élément d'un tuple pour rendre ces significations claires dans un programme Python

```
from collections import namedtuple #importation de la classe namedtuple du module collections  
  
Point = namedtuple('Point', ['x', 'y'])  
  
p = Point(11, y=22)      # instantiation par position ou en utilisant le nom du champs  
  
print(p[0] + p[1])       # les champs sont accessibles par leurs indexes affiche :33  
  
print(p.x + p.y)         # les champs sont accessibles par nom affiche 33  
  
print(p)                 # lisible dans un style nom=valeur affiche Point(x=11,y=22)
```

Collections - namedtuple

- La méthode `_asdict()` permet de convertir une instance en un dictionnaire.
- Appeler `p._asdict()` renvoie un dictionnaire mettant en correspondance les noms de chacun des deux champs de `p` avec leurs valeurs correspondantes.

```
from collections import namedtuple  
  
Point = namedtuple ('Point',['x','y'])  
  
p=Point(11,y=22)  
  
print(p._asdict()) # renvoie un dictionnaire, affiche {'x': 11, 'y': 22}
```

- La fonction `_replace(key=args)` permet de retourner une nouvelle instance de notre tuple avec une valeur modifiée.

```
from collections import namedtuple  
  
Point = namedtuple ('Point',['x','y'])  
  
p=Point(11,y=22)  
  
print(p._replace(x=4)) #changer la valeur de x x=4, affiche: Point(x=4, y=22)
```

Collections - namedtuple

- La fonction `_mytuple._fields` permet de récupérer les noms des champs de notre tuple. Elle est utile si on veut créer un nouveau tuple avec les champs d'un tuple existant.

```
from collections import namedtuple  
  
Point = namedtuple ('Point',['x','y'])  
  
print(Point._fields) #retourne les noms de champs affiche: ('x','y')  
  
Color =namedtuple ('Color', 'red green, blue')  
  
Pixel= namedtuple ('Pixel', Point._fields + Color._fields) #on crée un nouveau tuple avec les champs de point et de color  
  
print(Pixel(11,22,128,266,0)) #affiche: Pixel(x=11, y=22, red=128, green=266, blue=0)
```

Collections - deque

- Les listes Python sont une séquence d'éléments ordonnés, mutable ou modifiable.
- Python peut ajouter des listes en temps constant mais l'insertion au début d'une liste peut être plus lente (le temps nécessaire augmente à mesure que la liste s'agrandit).

Exemple :

```
favorite_list = ["Sammy", "Jamie", "Mary"]
favorite_list.insert(0, "Alice") # insérer "Alice" au début de favorite_fish_list
print(favorite_list) #affiche ['Alice', 'Sammy', 'Jamie', 'Mary']
```

- La classe **deque** du module collections est un objet de type liste qui permet d'insérer des éléments au début ou à la fin d'une séquence avec une performance à temps constant ($O(1)$).
- $O(1)$ performance signifie que le temps nécessaire pour ajouter un élément au début de la liste n'augmentera pas, même si cette liste a des milliers ou des millions d'éléments.

Collections - deque

- Les fonctions **append(x)**, **appendleft(x)**: **append** ajoute une seule valeur du côté droit du deque et **appendleft** du côté gauche

```
from collections import deque
favorite_deque = deque(["Sammy", "Jamie", "Mary"])
favorite_deque.appendleft("Alice") #ajout au début de favorite_fish_deque
favorite_deque.append("Bob") #ajout à la fin de favorite_fish_deque
print(favorite_deque) #affiche deque(['Alice', 'Sammy', 'Jamie', 'Mary', 'Bob'])
```

- Les fonctions **pop()**, **popleft()** et **clear()**: **pop()** et **popleft()** permettent de faire sortir un objet d'un deque et **clear()** le vide.

```
from collections import deque
favorite_deque = deque(['Alice', 'Sammy', 'Jamie', 'Mary', 'Bob'])
favorite_deque.pop() #affiche Bob
favorite_deque.popleft() #affiche: Alice
print(favorite_deque.clear()) #affiche None
```

Collections - ChainMap

- La classe **collections.ChainMap** permet de linker plusieurs mappings pour qu'ils soient gérés comme un seul.
- **class collections.ChainMap(*maps)** cette fonction retourne une nouvelle ChainMap. Si il n'y a pas de maps spécifiés en paramètres la ChainMap sera vide.

Exemple :

```
from collections import ChainMap

x = {'a': 1, 'b': 2}
y = {'b': 10, 'c': 11}
z = ChainMap(y, x)

for k, v in z.items(): #parcourir les éléments de z
    print(k, v)

#affiche:
#a 1
#b 10
C# 11
```

- Dans l'exemple précédent on remarque que la clé b a pris la valeur 10 et pas 2 car y est passé avant x dans le constructeur de ChainMap.

Collections - Counter

- La classe **collections.Counter** est une sous-classe de dict. qui permet de compter des objets hachable.
- En fait c'est un dictionnaire avec comme clé les éléments et comme valeurs leur nombre.
- class collections.Counter([iterable-or-mapping]) ceci retourne un Counter. L'argument permet de spécifier ce que l'on veut mettre dedans et qui doit être compté.

Exemple :

```
from collections import Counter

c = Counter()      #compteur vide
print(c)          #affiche: Counter()

c = Counter('gallahad') #compteur avec un iterable
print(c) #affiche Counter({'a': 3, 'l': 2, 'g': 1, 'h': 1, 'd': 1})

c = Counter({'red': 4, 'blue': 2}) #un compteur avec un mapping
print(c) #affiche: Counter({'red': 4, 'blue': 2})

c = Counter(cats=4, dogs=8)       #un compteur avec key=valeur
print(c) #affiche: Counter({'dogs': 8, 'cats': 4})
```

Collections - Counter

- Si on demande une valeur n'étant pas dans notre liste il retourne 0 et non pas KeyError

```
from collections import Counter
c = Counter(['eggs', 'ham'])
c['bacon'] # clé inconnue, affiche: 0
```

- La fonction **elements()**: retourne une liste de tous les éléments du compteur :

```
from collections import Counter
c = Counter(a=4, b=2, c=0, d=-2)
sorted(c.elements()) #affiche ['a', 'a', 'a', 'a', 'b', 'b']
```

- La fonction **most_common([n])**: retourne les n éléments les plus présents dans le compteur

```
from collections import Counter
c = Counter(a=4, b=2, c=0, d=-2)
print(Counter('abracadabra').most_common(3)) #affiche [('a', 5), ('b', 2), ('r', 2)]
```

Collections - Counter

- La fonction **subtract([iterable or mapping])** : permet de soustraire des éléments d'un compteur (mais pas de les supprimer)

```
from collections import Counter  
  
c = Counter(a=4, b=2, c=0, d=-2)  
d = Counter(a=1, b=2, c=3, d=4)  
  
print(c.subtract(d)) #affiche Counter({'a': 3, 'b': 0, 'c': -3, 'd': -6})
```

02 - Manipuler les données

Collections

Collections - OrdredDict

- Les **collections.OrderedDict** sont comme les dict. mais ils se rappellent l'ordre d'entrée des valeurs. Si on itère dessus les données seront retournées dans l'ordre d'ajout dans notre dict.
- La fonction **popitem(last=True)** : fait sortir une paire clé valeur de notre dictionnaire et si l'argument last est à 'True' alors les pairs seront retournées en LIFO sinon ce sera en FIFO.
- La fonction **move_to_end(key, last=True)** : permet de déplacer une clé à la fin de notre dictionnaire si last est à True sinon au début de notre dict.

Exemple :

```
from collections import OrderedDict
d=OrderedDict()
d['a']='1' #remplir d
d['b']='2'
d['c']='3'
d['d']='4'

d.move_to_end('b')
print(d) #affiche OrderedDict([('a', '1'), ('c', '3'), ('d', '4'), ('b', '2')])

d.move_to_end('b',last=False)
print(d) #affiche OrderedDict([('b', '2'), ('a', '1'), ('c', '3'), ('d', '4')])

print(d.popitem()) #affiche ('d', '4')

print(d) #affiche OrderedDict([('b', '2'), ('a', '1'), ('c', '3')])
```

Collections - defaultdict

- **defaultdict** du module de collections qui permet de rassembler les informations dans les dictionnaires de manière rapide et concise.
- **defaultdict** ne soulève jamais une **KeyError**.
- Si une clé n'est pas présente, **defaultdict** se contente d'insérer et de renvoyer une valeur de remplacement à la place.
- **defaultdict** se comporte différemment d'un dictionnaire ordinaire. Au lieu de soulever une **KeyError** sur une clé manquante, **defaultdict** appelle la valeur de remplacement sans argument pour créer un nouvel objet. Dans l'exemple ci-dessous, **list()** pour créer une liste vide.

```
from collections import defaultdict
my_defaultdict = defaultdict(list)
print(my_defaultdict["missing"]) #affiche []
```

Collections - defaultdict

```
from collections import defaultdict

def default_message():
    return "key is not there"

defaultdict_obj= defaultdict(default_message)
defaultdict_obj["key1"]="value1"
defaultdict_obj["key2"]="value2 »
print(defaultdict_obj["key1"]) #affiche: value1
print(defaultdict_obj["key2"]) #affiche: value2
print(defaultdict_obj["key3"]) #affiche: key is not there
```

Remplissage d'un defaultdict

Affichage des éléments d'un defaultdict

Si la clé n'existe pas alors appeler la fonction default_message

Utilisation de la fonction lambda

```
from collections import defaultdict

defaultdict_obj= defaultdict(lambda: "key is missing") #déclaration d'une fonction lambda pour afficher un message

defaultdict_obj["key1"]="value1"
defaultdict_obj["key2"]="value2"
print(defaultdict_obj["key1"]) #affiche: value1
print(defaultdict_obj["key2"]) #affiche: value2
print(defaultdict_obj["key3"]) #appel de la fonction lambda, affiche: key is missing
```

Collections - defaultdict

- Lorsque la classe int est fournie comme fonction par défaut, la valeur par défaut renvoyée est zéro.

```
from collections import defaultdict

defaultdict_obj = defaultdict(int)

defaultdict_obj["key1"] = "value1"
defaultdict_obj["key2"] = "value2"

print(defaultdict_obj["key1"]) #affiche: value1
print(defaultdict_obj["key2"]) #affiche: value2
print(defaultdict_obj["key3"]) #affiche: 0
```

- Nous pouvons également utiliser l'objet Set comme objet par défaut

```
from collections import defaultdict

defaultdict_obj = defaultdict(set)

defaultdict_obj["key1"] = "value1"
defaultdict_obj["key2"] = "value2"

print(defaultdict_obj["key1"]) #affiche: value1
print(defaultdict_obj["key2"]) #affiche: value2
print(defaultdict_obj["key3"]) #affiche: set()
```



CHAPITRE 2

Manipuler les données

1. Liste
2. Collections
3. Fichiers

Utilisation des fichiers

- Python a plusieurs fonctions pour créer, lire, mettre à jour et supprimer des fichiers texte.
- La fonction clé pour travailler avec des fichiers en Python est `open()`.
 - Elle prend deux paramètres; **nom de fichier et mode**.
 - Il existe quatre méthodes (modes) différentes pour ouvrir un fichier:
 - "r" -Lecture -Par défaut. Ouvre un fichier en lecture, erreur si le fichier n'existe pas
 - "a" -Ajouter -Ouvre un fichier à ajouter, crée le fichier s'il n'existe pas
 - "w" -Écrire -Ouvre un fichier pour l'écriture, crée le fichier s'il n'existe pas
 - "x" -Créer -Crée le fichier spécifié, renvoie une erreur si le fichier existe

- **Ouvrir et lire un fichier**

- Pour ouvrir le fichier, utilisez la fonction `open()` intégrée
- La fonction `open()` renvoie un objet fichier, qui a une méthode `read()` pour lire le contenu du fichier

```
f = open("demofile.txt", "r")
print(f.read())
```

- **Renvoyez les 5 premiers caractères du fichier :**

```
f = open("demofile.txt", "r")
print(f.read(5))
```

02 - Manipuler les données

Fichiers



Utilisation des fichiers

- Vous pouvez renvoyer une ligne en utilisant la méthode readline ():

```
f =open("demofile.txt","r")
print(f.readline())
```

- Ecrire dans un fichier

- Pour écrire dans un fichier existant, vous devez ajouter un paramètre à la fonction open():

```
f=open("demofile2.txt","a") # ouvrir le fichier en mode ajout c'est à dire il est possible d'ajouter du texte dedans
f.write("Le fichier contient plus de texte") #ajouter la phrase "Le fichier contient plus de texte"
f.close() #fermer le fichier
```

```
f=open("demofile3.txt","w") #ouvrir le fichier en mode écriture en effaçant son ancien contenu
f.write("domage!! l'ancien contenu est supprimé") #écrire la phrase domage!! l'ancien contenu est supprimé"
f.close() #fermer le fichier
```

```
f=open("demofile3.txt","r")
print(f.read()) # ouvrir et lire le fichier après l'ajout
```

Utilisation des fichiers

- Fermer un fichier
 - Il est recommandé de toujours fermer le fichier lorsque vous en avez terminé.

```
f =open("demofile.txt","r") #ouvrir le fichier en mode lecture  
print(f.readline()) #lire une line du fichier  
f.close() #fermer le fichier
```

- Supprimer d'un fichier
 - Pour supprimer un fichier, vous devez importer le module OS et exécuter sa fonction **os.remove ()**:

```
import os #importer la bibliothèque os  
os.remove("demofile.txt") #supprimer un fichier
```

02 - Manipuler les données

Fichiers

Format CSV

- Il existe différents formats standards de stockage de données. Il est recommandé de favoriser ces formats car il existe déjà des modules Python permettant de simplifier leur utilisation.
- Le fichier **Comma-separated values (CSV)** est un format permettant de stocker des tableaux dans un fichier texte. Chaque ligne est représentée par une ligne de texte et chaque colonne est séparée par un **séparateur** (virgule, point-virgule ...).
- Les champs texte peuvent également être délimités par des guillemets.
- Lorsqu'un champ contient lui-même des guillemets, ils sont doublés afin de ne pas être considérés comme début ou fin du champ.
- Si un champ contient un signe pouvant être utilisé comme séparateur de colonne (virgule, point-virgule ...) ou comme séparateur de ligne, les guillemets sont donc obligatoires afin que ce signe ne soit pas confondu avec un séparateur.

Données sous la forme d'un tableau

Nom	Prénom	Age
Dubois	Marie	29
Duval	Julien "Paul"	47
Jacquet	Bernard	51
Martin	Lucie;Clara	14

Données sous la forme d'un fichier CSV

```
Nom;Prénom;Age  
"Dubois";" Marie "; 29  
"Duval";"Julien ""Paul""";47  
Jacquet;Bernard;51  
Martin;"Lucie;Clara";14
```

Format CSV

- Le module csv de Python permet de simplifier l'utilisation des fichiers CSV
- Lecture d'un fichier CSV**
 - Pour lire un fichier CSV, il faut ouvrir un flux de lecture de fichier et ouvrir à partir de ce flux un lecteur CSV.

```
import csv #importer la bibliothèque csv
fichier = open("exempleCSV.csv", "r")
lecteurCSV = csv.reader(fichier,delimiter=";") # Ouverture du lecteur CSV en lui fournissant le caractère séparateur (ici ";")
for ligne in lecteurCSV: # parcourir les lignes du fichier CSV
    print(ligne) # afficher chaque ligne du fichier CSV
fichier.close() #fermer le fichier CSV
```

Affichage:
['Nom ', 'Prenom', 'Age']
['Dubois', 'Marie', '29']
['Duval', 'Julien ', '47']
['Jacquet', 'Bernard', '51']
['Martin', 'Lucie', '14']

- Il est également possible de lire les données et obtenir un dictionnaire par ligne contenant les données en utilisant **DictReader** au lieu de **reader**

```
import csv #importer la bibliothèque csv
fichier = open("noms.csv", "r")
lecteurCSV = csv.DictReader(fichier,delimiter=";")
for ligne in lecteurCSV:
    print(ligne)
fichier.close()
```

Affichage:
{'Nom ': 'Dubois', 'Prenom': 'Marie', 'Age': '29'}
{'Nom ': 'Duval', 'Prenom': 'Julien', 'Age': '47'}
{'Nom ': 'Jacquet', 'Prenom': 'Bernard', 'Age': '51'}
{'Nom ': 'Martin', 'Prenom': 'Lucie', 'Age': '14'}

02 - Manipuler les données

Fichiers



Format CSV

- Écriture dans un fichier CSV
 - À l'instar de la lecture, on ouvre un flux d'écriture (fonction `writer()`) et on ouvre un écrivain CSV à partir de ce flux (fonction `writerow()`) :

```
import csv #importer la bibliothèque csv

fichier = open("annuaire.csv", "w") #ouvrir un fichier en mode écriture
ecrivainCSV = csv.writer(fichier,delimiter=";") #ouvre un flux d'écriture
ecrivainCSV.writerow(["Nom","Prénom","Téléphone"]) #écrire une 1ère ligne dans le fichier annuaire.csv
ecrivainCSV.writerow(["Dubois","Marie","0198546372"]) #écrire une 2ème ligne dans le fichier annuaire.csv
ecrivainCSV.writerow(["Duval","Julien","0399741052"]) #écrire une 3ème ligne dans le fichier annuaire.csv
ecrivainCSV.writerow(["Jacquet","Bernard","0200749685"]) #écrire une 4ème ligne dans le fichier annuaire.csv
ecrivainCSV.writerow(["Martin","Julie","0399731590"]) #écrire une 5ème ligne dans le fichier annuaire.csv
fichier.close() #Fermer le fichier
```

Génération du fichier annuaire.csv

Nom	Prénom	Téléphone
Dubois	Marie	0198546372
Duval	Julien	0399741052
Jacquet	Bernard	0200749685
Martin	Julie	0399731590

02 - Manipuler les données

Fichiers



Format CSV

- Il est également possible d'écrire le fichier en fournissant un dictionnaire par ligne à condition que chaque dictionnaire possède les mêmes clés.
- La fonction DictWriter() produit les lignes de sortie depuis des dictionnaires . Il faut également fournir la liste des clés des dictionnaires avec l'argument **fieldnames**

```
import csv #importer la bibliothèque csv

bonCommande = [{"produit":"cahier", "reference":"F452CP", "quantite":41, "prixUnitaire":1.6},
               {"produit":"stylo bleu", "reference":"D857BL", "quantite":18, "prixUnitaire":0.95},
               {"produit":"stylo noir", "reference":"D857NO", "quantite":18, "prixUnitaire":0.95},
               ] #déclarer des dictionnaires

fichier = open("bon-commande.csv", "w") #ouvrir un fichier en écrire
ecrivainCSV = csv.DictWriter(fichier,delimiter=";",fieldnames=bonCommande[0].keys())
#produire des lignes de sortie depuis les dictionnaires,
# fieldnames contient les clés des dictionnaires
ecrivainCSV.writeheader()    # Ecrire la ligne d'en-tête avec le titre des colonnes
for ligne in bonCommande:   # Parcourir les dictionnaires
    ecrivainCSV.writerow(ligne) #Ecrire une ligne dans le fichier
fichier.close()
```

Génération du fichier bon-commande.csv

```
reference;quantite;produit;prixUnitaire
F452CP;41;cahier;1.6
D857BL;18;stylo bleu;0.95
D857NO;18;stylo noir;0.95
```

02 - Manipuler les données

Fichiers

Format JSON

- Le format JavaScript Object Notation (JSON) est issu de la notation des objets dans le langage JavaScript.
- Il s'agit aujourd'hui d'un format de données très répandu permettant de stocker des données sous une forme structurée.
- Il ne comporte que des associations **clés → valeurs** (à l'instar des dictionnaires), ainsi que des listes ordonnées de valeurs (comme les listes en Python).
- Une valeur peut être une autre association clés → valeurs, une liste de valeurs, un entier, un nombre réel, une chaîne de caractères, un booléen ou une valeur nulle.
- Sa syntaxe est similaire à celle des dictionnaires Python.

Exemple de fichier JSON : Définition d'un menu

il s'agit d'un objet composé de membres qui sont un attribut (Fichier) et un tableau (commandes)

qui contient d'autres objets: les lignes du menu. Une ligne de menu est identifiée par son titre et son action

Exemple de fichier JSON

```
{  
  "menu": "Fichier",  
  "commandes": [  
    {  
      "titre": "Nouveau",  
      "action": "CreateDoc"  
    },  
    {  
      "titre": "Ouvrir",  
      "action": "OpenDoc"  
    },  
    {  
      "titre": "Fermer",  
      "action": "CloseDoc"  
    }  
  ]  
}
```

Format JSON

- Lire un fichier JSON

- La fonction `loads(texteJSON)` permet de décoder le texte JSON passé en argument et de le transformer en dictionnaire ou une liste.

```
import json          #importer la bibliothèque json
fichier = open ( "exemple.json", "r" ) #ouvrir le fichier exemple.json en lecture
x=json.loads(fichier.read( ))      #décoder le texte et le transformer en dictionnaire.
print(x)
```

Affichage:

```
{'menu': 'Fichier', 'commandes': [{'titre': 'Nouveau', 'action': 'CreateDoc'}, {'titre': 'Ouvrir', 'action': 'OpenDoc'}, {'titre': 'Fermer', 'action': 'CloseDoc'}]}
```

- Écrire un fichier JSON

- la fonction `dumps(variable, sort_keys=False)` transforme un dictionnaire ou une liste en texte JSON en fournissant en argument la variable à transformer.

```
import json          #importer la bibliothèque json
quantiteFournitures= {"cahiers":134, "stylos":{"rouge":41,"bleu":74}, "gommes":85} #déclarer un dictionnaire
fichier=open ("quantiteFournitures.json","w") #ouvrir un fichier en écriture
fichier.write(json.dumps(quantiteFournitures)) #transformer le dictionnaire en texte json
fichier.close() #fermer le fichier
```



CHAPITRE 3

Utiliser les expressions régulières

Ce que vous allez apprendre dans ce chapitre :

- Connaitre le principe des expressions régulières
- Manipuler les fonctions sur les expressions régulières à savoir les fonctions de recherche, de division et de substitution

10 heures



CHAPITRE 3

Utiliser les expressions régulières

1. **Création des expressions régulières**
2. Manipulation des expressions régulières

03 - Utiliser les expressions régulières

Création des expressions régulières

Expressions régulières de base

- Les expressions régulières fournissent une notation générale permettant de décrire abstraitemment des éléments textuels
- Une expression régulière se lit de gauche à droite.
- Elle constitue ce qu'on appelle traditionnellement un motif de recherche.
- Elles utilisent **six** symboles qui, dans le contexte des expressions régulières, acquièrent les significations suivantes :
 1. **le point .** représente une seule instance de n'importe quel caractère sauf le caractère de fin de ligne.

Exemple : l'expression t.c représente toutes les combinaisons de trois lettres commençant par « t » et finissant par « c », comme tic, tac, tqc ou t9c

2. **la paire de crochets []** représente une occurrence quelconque des caractères qu'elle contient.

Exemple : [aeiouy] représente une voyelle, et Duran[dt] désigne Durand ou Durant.

- Entre les crochets, on peut noter un intervalle en utilisant le tiret.

Exemple : [0-9] représente les chiffres de 0 à 9, et [a-zA-Z] représente une lettre minuscule ou majuscule.

- On peut de plus utiliser l'accent circonflexe en première position dans les crochets pour indiquer le contraire de

Exemple : [^a-z] représente autre chose qu'une lettre minuscule, et [^"] n'est ni une apostrophe ni un guillemet.

3. **l'astérisque *** est un quantificateur, il signifie aucune ou plusieurs occurrences du caractère ou de l'élément qui le précède immédiatement.

Exemple : L'expression ab* signifie la lettre a suivie de zéro ou plusieurs lettres b, par exemple ab, a ou abbb et [A-Z]* correspond à zéro ou plusieurs lettres majuscules.



03 - Utiliser les expressions régulières

Création des expressions régulières

Expressions régulières de base

- l'accent circonflexe `^` est une ancre. Il indique que l'expression qui le suit se trouve en début de ligne.

Exemple : l'expression `^Depuis` indique que l'on recherche les lignes commençant par le mot Depuis.

- le symbole dollar `$` est aussi une ancre. Il indique que l'expression qui le précède se trouve en fin de ligne.

Exemple : L'expression suivante `:$` indique que l'on recherche les lignes se terminant par « suivante : ».

- la contre-oblique `\` permet d'échapper à la signification des métacaractères. Ainsi `\.` désigne un véritable point, `*` un astérisque, `\^` un accent circonflexe, `\$` un dollar et `\\\` une contre-oblique

Expressions régulières étendues

- Elles ajoutent cinq symboles qui ont les significations suivantes :

- la paire de parenthèses `()`:** est utilisée à la fois pour former des sous-motifs et pour délimiter des sous-expressions, ce qui permettra d'extraire des parties d'une chaîne de caractères.

Exemple : L'expression `(to)*` désignera to, tototo, etc.

- le signe plus `+`:** est un quantificateur comme `*`, mais il signifie une ou plusieurs occurrences du caractère ou de l'élément qui le précède immédiatement.

Exemple : L'expression `ab+` signifie la lettre a suivie d'une ou plus

- le point d'interrogation `?`:** il signifie zéro ou une instance de l'expression qui le précède.

Exemple : `écran(s)?` désigne écran ou écrans ;



03 - Utiliser les expressions régulières

Création des expressions régulières

Expressions régulières étendues

4. **la paire d'accolades { }** : précise le nombre d'occurrences permises pour le motif qui le précède.

Exemple : [0-9]{2,5} attend entre deux et cinq nombres décimaux.

- Les variantes suivantes sont disponibles : [0-9]{2,} signifie au minimum deux occurrences d'entiers décimaux et [0-9]{2} deux occurrences exactement

5. **la barre verticale |** : représente des choix multiples dans un sous-motif.

Exemple : L'expression Duran[d|t] peut aussi s'écrire (Durand|Durant). On pourrait utiliser l'expression (lu|ma|me|je|ve|sa|di) dans l'écriture d'une date.

- la syntaxe étendue comprend aussi une série de séquences d'échappement :
 - \ : symbole d'échappement ;
 - \e : séquence de contrôle escape ;
 - \f : saut de page ;
 - \n : fin de ligne ;
 - \r : retour-chariot ;
 - \t : tabulation horizontale ;
 - \v : tabulation verticale ;
 - \d : classe des nombres entiers ;
 - \s : classe des caractères d'espacement ;
 - \w : classe des caractères alphanumériques ;
 - \b : délimiteurs de début ou de fin de mot ;
 - \D : négation de la classe \d ;
 - \S : négation de la classe \s ;
 - \W : négation de la classe \w ;
 - \B : négation de la classe \b.





CHAPITRE 3

Utiliser les expressions régulières

1. Création des expressions régulières
2. **Manipulation des expressions régulières**

03 - Utiliser les expressions régulières

Manipulation des expressions régulières

- Le module `re` permet d'utiliser les expressions régulières dans les scripts Python.
- Les scripts devront donc comporter la ligne : `import re`
- **Fonction de compilation d'une regex en Python:**
 - Pour initialiser une expression régulière avec Python, il est possible de la compiler, surtout si vous serez amené à l'utiliser plusieurs fois tout au long du programme. **Pour ce faire, il faut utiliser la fonction `compile()`**

```
import re  
expression =re.compile(r'\d{1,3}')
```

Les options de compilation :

- Grâce à un jeu d'options de compilation, il est possible de piloter le comportement des expressions régulières. On utilise pour cela la syntaxe (...) avec les drapeaux suivants :
 - **a** : correspondance ASCII (Unicode par défaut) ;
 - **i** : correspondance non sensible à la casse ;
 - **L** : les correspondances utilisent la locale, c'est-à-dire les particularités du pays ;
 - **m** : correspondance dans des chaînes multilignes ;
 - **s** : modifie le comportement du métacaractère point qui représentera alors aussi le saut de ligne ;
 - **u** : correspondance Unicode (par défaut) ;
 - **x** : mode verbeux.

```
import re  
  
expression =re.compile(r"[a-z]+") #pilotage du comportement de  
l'expression régulière avec une sensibilité à la case  
  
expression =re.compile(r" (?i) [a-z]+") #pilotage du comportement  
de l'expression régulière sans une sensibilité à la case
```



03 - Utiliser les expressions régulières

Manipulation des expressions régulières



- Le module 're' propose un ensemble de fonctions qui nous permet de rechercher une chaîne pour une correspondance :

Fonction	Description
Findall	Renvoie une liste contenant toutes les correspondances
Search	Envoie un objet Match s'il existe une correspondance n'importe où dans la chaîne
split	Renvoie une liste où la chaîne a été divisée à chaque correspondance
sub	Remplace une ou plusieurs correspondances par une chaîne ordonnée

- La fonction findall()** renvoie une liste contenant toutes les correspondances. La liste contient les correspondances dans l'ordre où elles sont trouvées. Si aucune correspondance n'est trouvée, une liste vide est renvoyée.

Exemple :

```
import re

Nameage = '"Janice is 22 and Theon is 33
Gabriel is 44 and Joey is 21"'
ages = re.findall(r'\d{1,3}', Nameage) #chercher de un, deux ou trois entiers décimaux
print(ages) #affiche ['22', '33', '44', '21']

names = re.findall(r'[A-Z][a-z]*',Nameage) #chaines contenant des minuscules et des majuscules
Print(names) #affiche ['Janice', 'Theon', 'Gabriel', 'Joey']
```

03 - Utiliser les expressions régulières

Manipulation des expressions régulières

- La fonction search() recherche une correspondance dans la chaîne et renvoie un objet Match s'il existe une correspondance. S'il y a plus d'une correspondance, seule la première occurrence de la correspondance sera renvoyée:

Exemple : Extraction simple

- La variable expression reçoit la forme compilée de l'expression régulière, Puis on applique à ce motif compilé la méthode search() qui retourne la première position du motif dans la chaîne Nameage et l'affecte à la variable ages. Enfin on affiche la correspondance complète (en ne donnant pas d'argument à group())

```
import re
Nameage = """
Janice is 22 and Theon is 33
Gabriel is 44 and Joey is 21
"""

expression = re.compile(r'\d{1,3}') #chercher de un, deux ou trois entiers décimaux
ages=expression.search(Nameage) #retourne la première position du motif
print(ages) #affiche: <re.Match object; span=(11, 13), match='22'>
```

Position du résultat

Résultat =22

03 - Utiliser les expressions régulières

Manipulation des expressions régulières



Exemple : Extraction des sous-groupes

- Il est possible d'affiner l'affichage du résultat en modifiant l'expression régulière de recherche de façon à pouvoir capturer les éléments du motif

```
import re

motif_date = re.compile(r"(\d\d ?) (\w+) (\d{4})")
corresp = motif_date.search("Bastille le 14 juillet 1789")

print("corresp.group()", corresp.group())      #corresp.group() : 14 juillet 1789
print("corresp.group(1)", corresp.group(1))    #corresp.group(1) : 14
print("corresp.group(2)", corresp.group(2))    #corresp.group(2) : juillet
print("corresp.group(3)", corresp.group(3))    #corresp.group(3) : 1789
print("corresp.group(1,3)", corresp.group(1,3)) #corresp.group(1,3) : ('14', '1789')
print("corresp.groups()", corresp.groups())    #corresp.groups() : ('14', 'juillet', '1789')
```

03 - Utiliser les expressions régulières

Manipulation des expressions régulières



- Python possède une syntaxe qui permet de nommer des parties de motif délimitées par des parenthèses, ce qu'on appelle un motif nominatif :
 - syntaxe de création d'un motif nominatif : (?P<nom_du_motif>) ;
 - syntaxe permettant de s'y référer : (?P=nom_du_motif) ;

Exemple : Extraction des sous-groupes nommés

- la méthode **groupdict()** renvoie une liste comportant le nom et la valeur des sous-groupes trouvés (ce qui nécessite de nommer les sous-groupes).

```
import re

motif_date = re.compile(r"(?P<jour>\d\d ?) (?P<mois>\w+) (\d{4})")
corresp = motif_date.search("Bastille le 14 juillet 1789")

print(corresp.groupdict()) #{'jour': '14', 'mois': 'juillet'}
print(corresp.group('jour')) #14
print(corresp.group('mois')) #juillet
```

03 - Utiliser les expressions régulières

Manipulation des expressions régulières



- La fonction **split()** renvoie une liste où la chaîne a été divisée à chaque correspondance
- L'exemple suivant divise la chaîne à chaque espace trouvé. '\s' est utilisé pour faire correspondre les espaces.

```
import re
adresse="Rue 41 de la République"
liste=re.split("\s",adresse)
print(liste) #affiche: ['Rue', '41', 'de', 'la', 'République']
```

- Il est possible de contrôler le nombre d'occurrences en spécifiant le paramètre **maxsplit**:

Exemple : maxsplit=1

```
import re
adresse="Rue 41 de la République"
liste=re.split("\s",adresse,1)
print(liste) #affiche ['Rue', '41 de la République']
```

03 - Utiliser les expressions régulières

Manipulation des expressions régulières



- La fonction **sub()** remplace les correspondances par le texte de votre choix

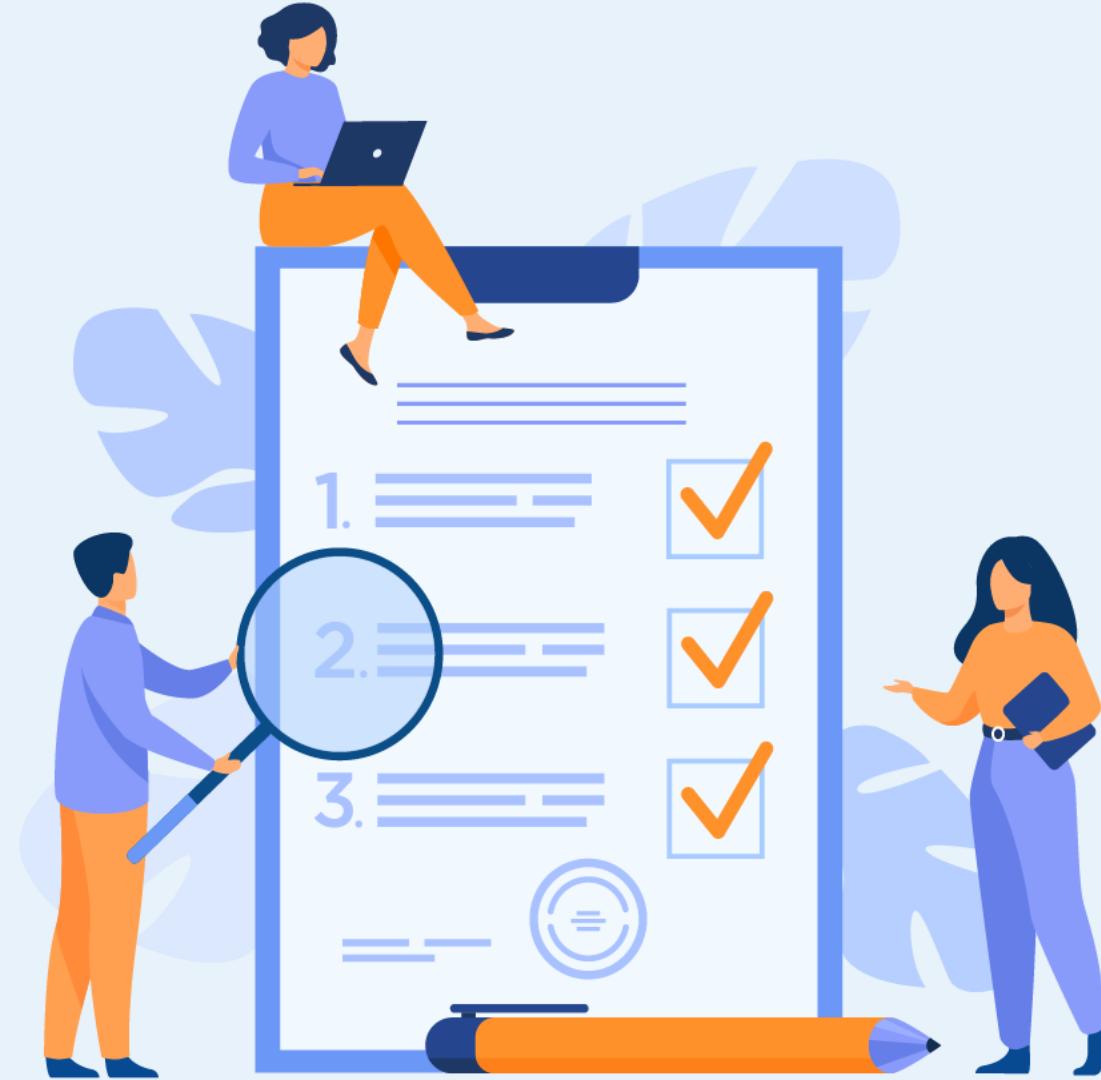
Exemple : Remplacer chaque espace par un tiret ‘-’:

```
import re
adresse="Rue 41 de la République"
liste=re.sub("\s","-",adresse) #affiche: Rue-41-de-la-République
print(liste)
```

- Il est possible de contrôler le nombre de remplacements en spécifiant le paramètre **count**:

Exemple : count=2

```
import re
adresse="Rue 41 de la République"
liste=re.sub("\s","-",adresse,2) #affiche Rue-41-de la République
print(liste)
```



CHAPITRE 4

Administre les exceptions

Ce que vous allez apprendre dans ce chapitre :

- Identifier les différents types d'erreurs
- Connaitre les principaux types d'exceptions en Python
- Maitriser la gestion des exceptions en Python

05 heures



CHAPITRE 4

Administrer les exceptions

1. **Types d'erreur et expérimentation**
2. Types des exceptions
3. Gestion des exceptions

Erreurs de syntaxe

- Les erreurs de syntaxe sont des erreurs d'analyse du code

Exemple :

```
>>> while True print('Hello world')
      File "<stdin>", line 1
          while True print('Hello world')
                  ^
SyntaxError: invalid syntax
```

- L'analyseur indique la ligne incriminée et affiche une petite « flèche » pointant vers le premier endroit de la ligne où l'erreur a été détectée.
- L'erreur est causée par le symbole placé avant la flèche.
- Dans cet exemple, la flèche est sur la fonction print() car il manque deux points (':') juste avant. Le nom du fichier et le numéro de ligne sont affichés pour vous permettre de localiser facilement l'erreur lorsque le code provient d'un script.

Exceptions

- Même si une instruction ou une expression est syntaxiquement correcte, elle peut générer une erreur lors de son exécution.
- Les erreurs détectées durant l'exécution sont appelées des exceptions et ne sont pas toujours fatales
- La plupart des **exceptions** toutefois ne sont pas prises en charge par les programmes, ce qui génère des messages d'erreurs comme celui-ci :

```
>>> 10 * (1/0)
Traceback (most recent call last):
ZeroDivisionError: division by zero
```

- La dernière ligne du message d'erreur indique ce qui s'est passé. Les exceptions peuvent être de différents types et ce type est indiqué dans le message : le type indiqué dans l'exemple est ZeroDivisionError



CHAPITRE 4

Administre les exceptions

1. Types d'erreur et expérimentation
2. **Types des exceptions**
3. Gestion des exceptions

- En Python, les erreurs détectées durant l'exécution d'un script sont appelées des exceptions car elles correspondent à un état "exceptionnel" du script
- Python analyse le type d'erreur déclenché
- Python possède de nombreuses classes d'exceptions natives et toute exception est une instance (un objet) créé à partir d'une classe exception
- La classe d'exception de base pour les exceptions natives est **BaseException**
- Quatre classes d'exception dérivent de la classe **BaseException** à savoir :

Exception

- Toutes les exceptions intégrées non-exit du système sont dérivées de cette classe
- Toutes les exceptions définies par l'utilisateur doivent également être dérivées de cette classe

SystemExit

- Déclenchée par la fonction `sys.exit()` si la valeur associée est un entier simple, elle spécifie l'état de sortie du système (passé à la fonction `exit()` de C)

GeneratorExit

- Levée lorsque la méthode `close()` d'un générateur est appelée

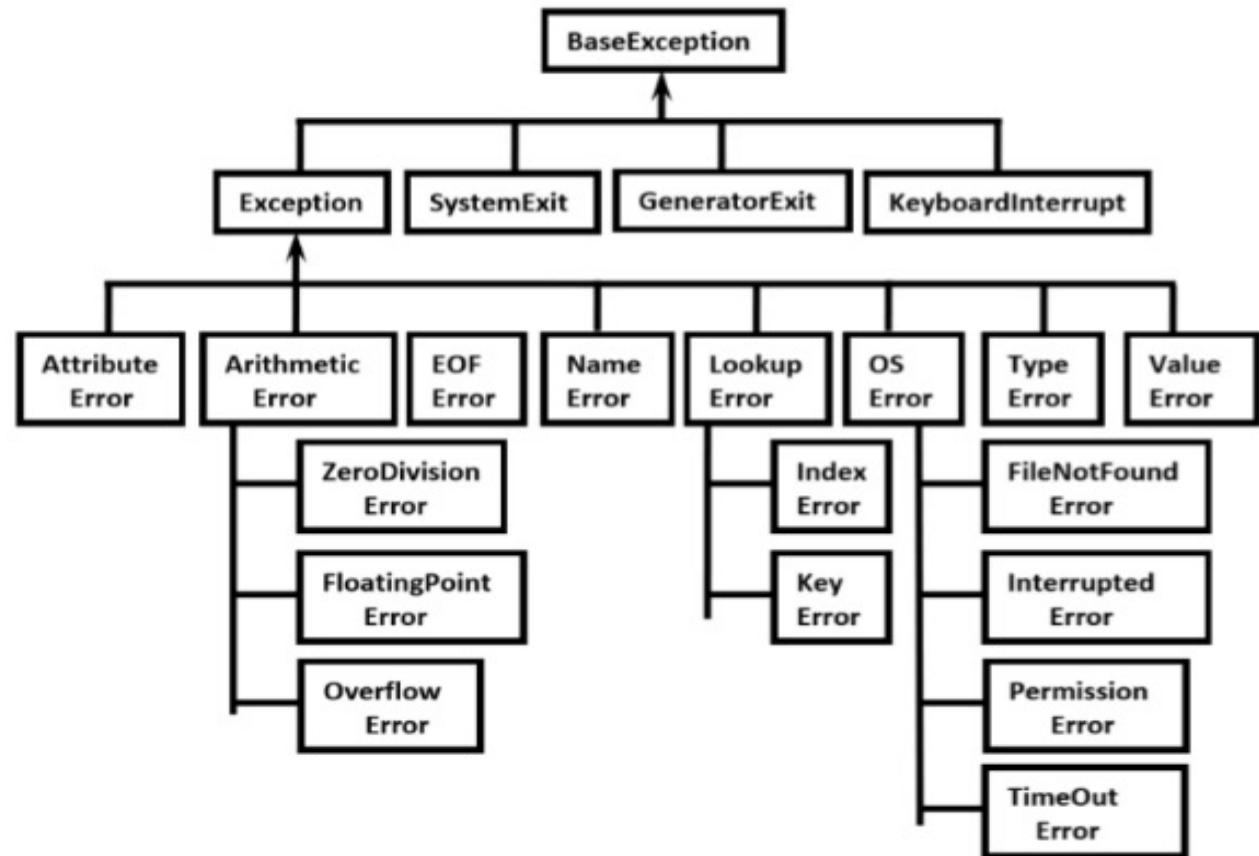
KeyboardInterrupt

- Levée lorsque l'utilisateur appuie sur la touche d'interruption (normalement Control-C ou Delete)

04 - Administrer les exceptions

Types des exceptions

- Il y a également d'autres classes **d'exception** qui dérivent de `Exception` telles que:
 - La classe **ArithmeticError** est la classe de base pour les exceptions natives qui sont levées pour diverses erreurs arithmétiques et notamment pour les classes **OverflowError**, **ZeroDivisionError** et **FloatingPointError** ;
 - La classe **LookupError** est la classe de base pour les exceptions qui sont levées lorsqu'une clé ou un index utilisé sur un tableau de correspondances où une séquence est invalide.
- De nombreuses classes dérivent ensuite de ces classes.
- En fonction de l'erreur rencontrée par l'analyseur Python, un objet exception appartenant à telle ou telle classe exception va être créé et renvoyé. Cet objet est intercepté et manipulé.





CHAPITRE 4

Administre les exceptions

1. Types d'erreur et expérimentation
2. Types des exceptions
3. **Gestion des exceptions**

Détection et traitement des exceptions en Python

- On peut détecter les exceptions en plaçant les instructions qui peuvent générer des exceptions dans un bloc **try**.
- Il existe 2 formes d'expressions try mutuellement exclusives (on ne peut en employer qu'une à la fois) : **try-except** et **try-finally**.
- Une instruction **try** peut être accompagnée d'une ou plusieurs clauses **except**, d'une seule clause **finally** ou d'une combinaison **try-except-finally**.

Exemple :

- On souhaite calculer l'âge saisi par l'utilisateur en soustrayant son année de naissance à 2016. Pour cela, il faut convertir la valeur de la variable birthyear en un int.
- Cette conversion peut échouer si la chaîne de caractères entrée par l'utilisateur n'est pas un nombre.

```
birthyear = input('Année de naissance ? ')

try:
    print('Tu as', 2016 - int(birthyear), 'ans.') #code risqué
except:
    print('Erreur, veuillez entrer un nombre.') #code à exécuter en cas d'erreur

print('Fin du programme.')
```

04 - Administre les exceptions

Gestion des exceptions



Instruction try-except

```
birthyear = input('Année de naissance ?')  
  
try:  
    print('Tu as', 2016 - int(birthyear), 'ans.') #code risqué  
  
except:  
    print('Erreur, veuillez entrer un nombre.') #code à exécuter en cas d'erreur  
  
print('Fin du programme.')
```

- Dans le premier cas, la conversion s'est passée normalement, et le bloc try a donc pu s'exécuter intégralement sans erreur.
- Dans le second cas, une erreur se produit dans le bloc try, lors de la conversion. L'exécution de ce bloc s'arrête donc immédiatement et passe au bloc except, avant de continuer également après l'instruction try-except.

1er cas

- Si l'utilisateur entre un nombre entier, l'exécution se passe sans erreur et son âge est calculé et affiché

```
Année de naissance ? 1994  
Tu as 22 ans.  
Fin du programme.
```

2ème cas

- Si l'utilisateur entre une chaîne de caractères quelconque, qui ne représente pas un nombre entier, un message d'erreur est affiché

```
Année de naissance ? deux  
Erreur, veuillez entrer un nombre.  
Fin du programme.
```

Type Exception

- Lorsqu'on utilise l'instruction **try-except**, le bloc **except** capture toutes les erreurs possibles qui peuvent survenir dans le bloc try correspondant.
- Une **exception** est en fait représentée par un objet, instance de la classe **Exception**.
- On peut récupérer cet objet en précisant un nom de variable après **except**.

```
try:  
    i = int(input('i ? '))      #code à risque  
    j = int(input('j ? '))      #code à risque  
    print(i, '/', j, '=', i / j) #code à risque  
  
except Exception as e:        # variable e de type Exception  
    print(type(e))            #afficher le type de l'exception  
    print(e)                  #afficher l'exception
```

- On récupère donc l'objet de type **Exception** dans la variable **e**.
- Dans le bloc **except**, on affiche son type et sa valeur.

Type Exception

Exemples d'exécution qui révèlent deux types d'erreurs différents :

- Si on ne fournit pas un nombre entier, il ne pourra être converti en **int** et une erreur de type **ValueError** se produit :

```
i ? trois  
  
#affiche:  
  
#<class 'ValueError'>  
  
#invalid literal for int() with base 10: 'trois'
```

- Si on fournit une valeur de 00 pour b, on aura une division par zéro qui produit une erreur de type **ZeroDivisionError** :

```
i ? 5  
  
j ? 0  
  
#affiche:  
  
#<class 'ZeroDivisionError'>  
  
#division by zero
```

Capture d'erreur spécifique

- Chaque type d'erreur est donc défini par une classe spécifique.
- Il est possible d'associer plusieurs blocs except à un même bloc try, pour exécuter un code différent en fonction de l'erreur capturée.
- Lorsqu'une erreur se produit, les blocs except sont parcourus l'un après l'autre, du premier au dernier, jusqu'à en trouver un qui corresponde à l'erreur capturée.

```
try:  
  
    i = int(input('i ? '))  
    j = int(input('j ? '))  
    print(i, '/', j, '=', i / j)  
  
except ValueError:  
    print('Erreur de conversion.')  
  
except ZeroDivisionError:  
    print('Division par zéro.')  
  
except:  
    print('Autre erreur.')
```

Exemple :

- Lorsqu'une erreur se produit dans le bloc try l'un des blocs except seulement qui sera exécuté, selon le type de l'erreur qui s'est produite.
- Le dernier bloc except est là pour prendre toutes les autres erreurs.
- **L'ordre des blocs except est très important et il faut les classer du plus spécifique au plus général, celui par défaut devant venir en dernier.**

Gestionnaire d'erreur partagé

- Il est possible d'exécuter le même code pour différents types d'erreur, en les listant dans un tuple après le mot réservé **except**.
- Si on souhaite exécuter le même code pour une erreur de conversion et de division par zéro, il faudrait écrire :

```
try:  
    i = int(input('i ?'))  
    j = int(input('j ?'))  
    print(i, '/', j, '=', i / j)  
  
except (ValueError, ZeroDivisionError) as e:  
    print('Erreur de calcul :', e) # exécuter le même code pour différents types d'exceptions  
  
except:  
    print('Autre erreur.')
```

Bloc finally

- Le mot réservé **finally** permet d'introduire un bloc qui sera exécuté soit après que le bloc **try** se soit exécuté complètement sans erreur, soit après avoir exécuté le bloc **except** correspondant à l'erreur qui s'est produite lors de l'exécution du bloc **try**.
- On obtient ainsi une instruction **try-except-finally**

```
print('Début du calcul.')

try:
    i = int(input('i ? '))
    j = int(input('j ? '))
    print('Résultat :', i / j)
except:
    print('Erreur.')
finally:
    print('Nettoyage de la mémoire.')
    print('Fin du calcul.)
```

- Si l'utilisateur fournit des valeurs correctes pour **a** et **b** l'affichage est le suivant :

```
Début du calcul.
i ? 2
j ? 8
Résultat : 0.25
Nettoyage de la mémoire.
Fin du calcul.
```

- Si une erreur se produit l'affichage est le suivant :

```
Début du calcul.
i ? 2
j ? 0
Erreur.
Nettoyage de la mémoire.
Fin du calcul.
```

Dans les 2 cas le bloc finally a été exécuté

Génération d'erreur

- Il est possible de générer une erreur dans un programme grâce à l'instruction `raise`.
- Il suffit en fait simplement d'utiliser le mot réservé `raise` suivi d'une référence vers un objet représentant une exception.

Exemple :

```
def factoriel(a):  
    if a < 0:  
        raise ArithmeticError() #signaler une erreur de type ArithmeticError si n<0  
    if a == 0:  
        return 1  
    return n * factoriel(a - 1)
```

- Le programme suivant permet de capturer spécifiquement l'exception de type `ArithmeticError` lors de l'appel de la fonction `fact`.

```
try:  
    a = int(input('Entrez un nombre : ')) #code à risque  
    print(factoriel(a)) #code à risque  
  
except ArithmeticError: #capturer de l'exception ArithmeticError  
    print('Veuillez entrer un nombre positif.') #afficher le message si l'exception  
                                                #ArithmeticError est capturée  
  
except: #capturer les autres types d'exception  
    print('Veuillez entrer un nombre.') #afficher le message si d'autres types  
#d'exceptions sont capturées
```

- Si `n` est strictement négatif, une exception de type `ArithmeticError` est générée.

Créer un type d'exception

- Il est parfois plus pratique et plus lisible de définir nos propres types d'exceptions
- Pour cela, il suffit de définir une nouvelle classe qui hérite de la classe Exception

Exemple :

```
class NoRootException(Exception):
    pass
```

- Cette classe est tout simplement vide puisque son corps n'est constitué que de l'instruction pass

Exemple :

- Définissons une fonction **trinomial** qui calcule et renvoie les racines d'un trinôme du second degré de la forme ax^2+bx+c et qui génère une erreur lorsqu'il n'y a pas de racine réelle :

```
from math import sqrt

def trinomial(a, b, c):
    delta = b ** 2 - 4 * a * c
    # Aucune racine réelle
    if delta < 0:
        raise NoRootException() #lever une exception de type NoRootException
    # Une racine réelle double
    if delta == 0:
        return -b / (2 * a)
    # Deux racines réelles simples
    x1 = (-b + sqrt(delta)) / (2 * a)
    x2 = (-b - sqrt(delta)) / (2 * a)
    return (x1, x2)
```

Exception paramétrée

- Lorsqu'on appelle la fonction **trinomial**, on va donc pouvoir utiliser l'instruction try-except pour attraper cette erreur, lorsqu'elle survient
- Essayons, par exemple, de calculer et d'afficher les racines réelles du trinôme $x+2$. Pour cela, on appelle donc la fonction **trinomial** en lui passant en paramètres 1, 0 et 2 puisque $x+2$ correspond à $a=1$, $b=0$ et $c=2$

```
try: #Attraper une exception avec le bloc try-except
    print(trinomial(1, 0, 2))
except NoRootException:
    print('Pas de racine réelle.')
```

- Pour l'exemple précédent, il pourrait être utile de connaître la valeur du discriminant
- Lorsqu'aucune racine réelle n'existe. Pour cela il faut ajouter une variable d'instance et
- Un accesseur à la classe **NoRootException**

```
class NoRootException(Exception):
    def __init__(self, delta): #définition d'un constructeur avec paramètre
        self.__delta = delta
    @property
    def delta(self):
        return self.__delta
```

Exception paramétrée

- Il est possible de récupérer la valeur du discriminant dans le bloc **except**, à partir de l'objet représentant l'exception qui s'est produite

```
from math import sqrt

def trinomial(a, b, c):
    delta = b ** 2 - 4 * a * c
    # Aucune racine réelle
    if delta < 0:
        raise NoRootException(delta) #lever une exception avec paramètre
    # Une racine réelle double
    if delta == 0:
        return -b / (2 * a)
    # Deux racines réelles simples
    x1 = (-b + sqrt(delta)) / (2 * a)
    x2 = (-b - sqrt(delta)) / (2 * a)
    return (x1, x2)
```

```
try:
    print(trinomial(1, 0, 2))
except NoRootException as e:
    print('Pas de racine réelle.')
    print('Delta =', e.delta) #récupérer la valeur du paramètre
```



PARTIE 4

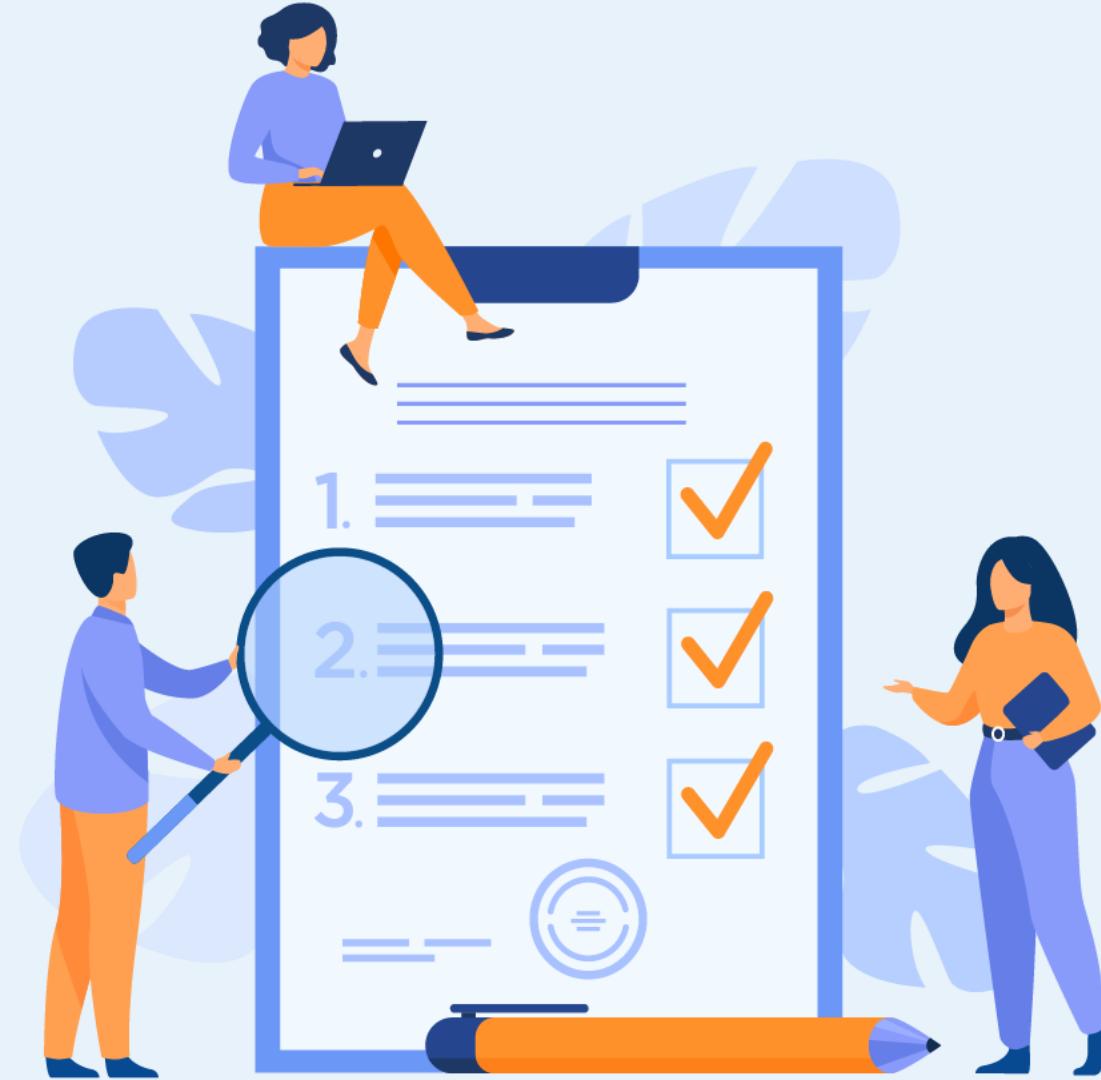
MANIPULER LES MODULES ET LES BIBLIOTHEQUES

Dans ce module, vous allez :

- Apprendre la création de modules en Python
- Maitriser l'importation des modules
- Installer des bibliothèques en Python
- Créer des bibliothèques en Python
- Importer des bibliothèques en Python



 30 heures



CHAPITRE 1

Manipuler les modules

Ce que vous allez apprendre dans ce chapitre :

- Créer des modules en Python
- Catégoriser les types de modules
- Maitriser l'importation absolue
- Maitriser l'importation relative



05 heures



CHAPITRE 1

Manipuler les modules

1. **Création des modules**
2. Importation des modules

01 - Manipuler les modules

Création des modules

Création de modules

- Un module est un fichier « .py » contenant un ensemble de **variables, fonctions et classes** que l'on peut importer et utiliser dans le programme principal (ou dans d'autres modules).
- Pour créer un module, il suffit de programmer les variables/fonctions et classes qui le constituent dans un fichier portant le nom du module, suivi du suffixe « .py ». Depuis un (autre) programme en Python, il suffit alors d'utiliser la primitive import pour pouvoir utiliser ces variables/fonctions/classes.
- **Les modules :**
 - Permettent la **séparation** du code et donc une **meilleure organisation du code**
 - Maximisent la **réutilisation**
 - **Facilitent le partage du code**

Exemple :

```
"""
exemple de module, aide associée
"""

exemple_variable = 3

def exemple_fonction():
    """exemple de fonction"""
    return 0
class exemple_classe:
    """exemple de classe"""

    def __str__(self):
        return "exemple_classe"
```

- L'exemple ci-dessus montre la définition d'un module contenant une variable, une fonction et une classe. Ces trois derniers peuvent être exploités par n'importe quel fichier important ce module.
- Le nom du module représente le nom du fichier sans son extension.



CHAPITRE 1

Manipuler les modules

1. Création des modules
2. Importation des modules

01 - Manipuler les modules

Importation des modules

- l'instruction **import nom_module** permet l'importation d'un module. L'importation doit se faire avant l'exploitation des composantes du module.
- En règle générale, toutes les importations sont faites au début du programme. Cependant, elles peuvent être faites dans n'importe quelle partie du programme.

```
import module_exemple #importation du module module_exemple

c = module_exemple.exemple_classe () #appel de la classe exemple_classe
print(c)
print(module_exemple.exemple_fonction()) #appel de la fonction exemple_fonction
```

Exemple :

- L'instruction **as** suivie de **alias** permet l'assignation d'un identificateur à un module. Cet identificateur peut être différent du nom du fichier dans lequel est défini le module.
- Cette méthode représente une autre manière de faire afin d'importer des modules.

```
import module_exemple as alias

c = alias.exemple_classe()
print(c)
print(alias.exemple_fonction())
```

- La syntaxe suivante n'est pas recommandée car elle masque le module d'où provient une fonction en plus de tout importer.



01 - Manipuler les modules

Importation des modules

- **import *** vous offre la possibilité d'importer toutes les composantes d'un module (classes, attributs et fonctions). Cependant, il est possible d'importer qu'une classe de ce module en écrivant **from module_exemple import exemple_class**
- Lorsqu'on importe un module, l'interpréteur Python le recherche dans différents répertoires selon l'ordre suivant :
 1. Le répertoire courant ;
 2. Si le module est introuvable, Python recherche ensuite chaque répertoire listé dans la variable shell PYTHONPATH ;
 3. Si tout échoue, Python vérifie le chemin par défaut (exemple pour windows \Python\Python39\Lib)

```
from module_exemple import * # importer toutes les classes, attributs, fonctions..  
from module_exemple import exemple_classe, exemple_fonction  
  
c = exemple_classe()  
print(c)  
print(exemple_fonction())
```



01 - Manipuler les modules

Importation des modules

Arborescence de modules

- Si vous avez plusieurs modules, il est préférable de répartir leurs fichiers dans des répertoires. Ces derniers doivent apparaître dans la liste `sys.path`.
- Python permet la définition de paquetage. Un répertoire est considéré comme étant un package qui contient tous les fichiers Python.
- Avant Python 3.3, un package contenant des modules Python doit contenir un fichier `__init__.py`

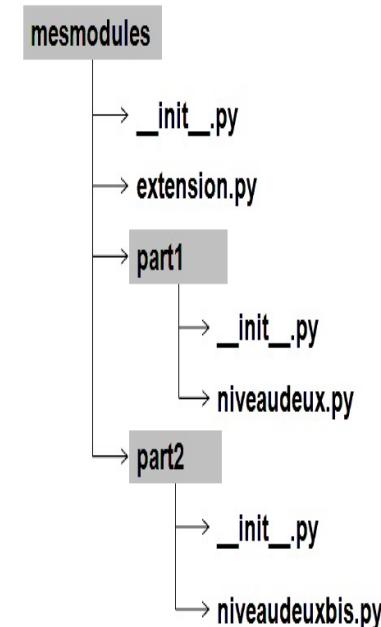
Importation absolue

- Une importation absolue spécifie la ressource à importer à l'aide de son chemin d'accès complet à partir du répertoire racine.

Exemple :

```
import mesmodules.extension  
  
import mesmodules.part1.niveaudeux  
  
import mesmodules.part2.niveaudeuxbis
```

- En exécutant l'instruction `import mesmodules.extension`, Python exécute le fichier `extension.py` mais également tout le contenu du fichier `__init__.py`



01 - Manipuler les modules

Importation des modules

Importation relative

- Une importation relative spécifie la ressource à importer par rapport à l'emplacement actuel, c'est-à-dire l'emplacement où se trouve l'instruction import.
 - Le symbole .** permet d'importer un module dans le même répertoire.
 - Le symbole ..** permet d'importer un module dans le répertoire parent.

Exemple :

- La fonction A** peut utiliser **la fonction B ou C** en les important de la façon suivante :

```
from .subpackage1 import B  
  
from .subpackage1.moduleX import C
```

- La fonction E** peut utiliser **la fonction F ou A ou C** en les important de la façon suivante :

```
from ..moduleA import F  
from .. import A  
from ..subpackage1.moduleX import
```

```
package/  
    __init__.py      # fonction A  
    subpackage1/  
        __init__.py  # fonction B  
        moduleX.py   # fonction C  
    subpackage2/  
        __init__.py  # fonction D  
        moduleY.py   # fonction E  
        moduleA.py    # fonction F
```

01 - Manipuler les modules

Importation des modules

PYTHONPATH

- Pour ajouter un dossier au PYTHONPATH en Python, il faut indiquer directement dans le code les lignes suivantes :

```
import sys  
sys.path.insert(0,"E:/exempleImport")
```

Chemin du module à emporter

- La fonction path du module sys permet de vérifier la variable PYTHONPATH

```
import sys  
print(sys.path)
```

- Remarque :** insert ne permet pas d'ajouter le dossier en question dans PYTHONPATH de façon permanente

```
['', 'C:\\\\Users\\\\DELL\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python39\\\\Lib\\\\idlelib', 'E:\\\\exempleImport', 'C:\\\\Users\\\\DELL\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python39\\\\python39.zip', 'C:\\\\Users\\\\DELL\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python39\\\\DLLs', 'C:\\\\Users\\\\DELL\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python39\\\\lib', 'C:\\\\Users\\\\DELL\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python39', 'C:\\\\Users\\\\DELL\\\\AppData\\\\Local\\\\Programs\\\\Python\\\\Python39\\\\lib\\\\site-packages']
```

CHAPITRE 2

Manipuler les bibliothèques



Ce que vous allez apprendre dans ce chapitre :

- Installer des bibliothèques standards en Python
- Manipuler la bibliothèque graphique Tinker
- Créer des bibliothèques en Python
- Importer des bibliothèques en Python

 25 heures



CHAPITRE 2

Manipuler les bibliothèques

1. Installation des bibliothèques externes (pip)
2. Création des bibliothèques
3. Importation des bibliothèques

02 - Manipuler les bibliothèques

Installation des bibliothèques externes (pip)



Bibliothèque en Python

- Dans Python, une bibliothèque est un ensemble logiciel de modules (classes (types d'objets), fonctions, constantes...) ajoutant des possibilités étendues à Python : calcul numérique, graphisme, programmation internet ou réseau, formatage de texte, génération de documents, etc.
- Il en existe un très grand nombre, et c'est d'ailleurs une des grandes forces de Python.
- La plupart est regroupée dans PyPI (Python Package Index) le dépôt tiers officiel du langage de programmation Python.

Bibliothèque standard

- La distribution standard de Python dispose d'une très riche bibliothèque de modules étendant les capacités du langage dans de nombreux domaines.
- La bibliothèque standard couvre un large éventail de fonctionnalités, notamment :
 - Modules de date et d'heure
 - Modules des interfaces graphiques
 - Modules numériques et mathématiques
 - Modules de système de fichiers
 - Modules de système d'exploitation
 - Modules pour la lecture et l'écriture de formats de données spécifiques tels que HTML, XML et JSON
 - Modules pour l'utilisation de protocoles Internet tels que HTTP, SMTP, FTP, etc.
 - Modules pour l'utilisation de données multimédias telles que les données audio et vidéo

Pip (Python Installer Package)

- Pip (Python Installer Package) est le manager de package pour Python
- Pip est un moyen d'installer et de gérer des packages et des dépendances supplémentaires qui ne sont pas encore distribués dans le cadre de la version standard du package
- Pip package est intégré dans l'installation du Python depuis les versions 3.4 pour Python3 et les versions 2.7.9 pour Python2, et utilisé dans nombreux projets du Python.
- En exécutant la commande ci-dessus, il est possible de vérifier que pip est disponible ou non

```
pip --version
```

- Résultat de l'exécution :

```
pip 21.2.4 from C:\Users\DELL\AppData\Local\Packages\PythonSoftwareFoundation.Python.3.9_qbz5n2kfra8p0\LocalCache\local-packages\Python39\site-packages\pip (python 3.9)
```

- La sortie permet d'afficher la version de **pip** dans votre machine, ainsi que l'emplacement de votre version du Python
- Si vous utilisez une ancienne version de Python qui n'inclut pas **pip**, vous devez l'installer

02 - Manipuler les bibliothèques

Installation des bibliothèques externes (pip)



Installation de Pip

1ère méthode :

- Téléchargez get-pip.py (<https://bootstrap.pypa.io/get-pip.py>) dans un dossier de votre ordinateur.
- Ouvrez l'invite de commande et accédez au dossier contenant le programme d'installation get-pip.py.
- Exécutez la commande suivante : **py get-pip.py**

2ème méthode :

- Voici la commande pour le télécharger avec l'outil **Wget** pour Windows

wget <https://bootstrap.pypa.io/get-pip.py>

- Pour se renseigner sur les commandes supportées par **pip** utilisez la commande suivante : **pip help**
- **Résultat de l'exécution :**

```
Usage: C:\Users\DELL\AppData\Local\Microsoft\WindowsApps\PythonSoftwareFoundation.Python.3.9_qbz5n2kfra8p0\python.exe -m pip <command> [options]
Commands:
install           Install packages.
download          Download packages.
uninstall         Uninstall packages.
freeze            Output installed packages in requirements format.
show              Show information on installed packages.
list               List installed packages.
check              Verify installed packages have compatible dependencies.
config             Manage local and global configuration.
search             Search package indexes.
cache              Inspect and manage pip's wheel cache.
index              Inspect information available from package indexes.
wheel              Build wheels from source distributions.
hash               Compute hashes of package archives.
completion        A helper command used for command completion.
debug             Print debugging information for pip.
help               Show help for commands.

General Options:
-h, --help          Show help.
--isolated         Run pip in an isolated mode, ignoring environment variables and user configuration.
-v, --verbose       Give more output. Option is additive, and can be used up to 3 times.
--quiet            Give less output. Option is additive, and can be used up to 3 times (corresponding to
                  levels 0 through 3).
--log <path>        Path to a verbose appending log.
--no-input         Disable prompting for input.
--proxy <proxy>     Use a proxy server. e.g. https://password@proxy-server:port.
--retries <retries> Maximum number of retries each connection should attempt (default 5 times).
--timeout <sec>      Set the socket timeout (default 15 seconds).
--exists-action <action> Define action when a path already exists: ('i')nitch, ('i')gnore, ('w')ipe, ('b')ackup,
                  ('a')bort.
--trusted-host <hostname> Mark this host or host:port pair as trusted, even though it does not have valid or any
                  SSL certificate.
--cert <path>        Path to PEM-encoded CA certificate bundle. If provided, overrides the default. See 'SSL
                  Certificate Verification' in pip documentation for more information.
```

02 - Manipuler les bibliothèques

Installation des bibliothèques externes (pip)



Installation des bibliothèques

- pip fournit une commande d'installation pour installer des packages/bibliothèques

```
py -m pip install sampleproject
```

```
Uninstalling sampleproject:  
[...]  
Proceed (y/n)? y  
Successfully uninstalled sampleproject
```

- Il est aussi possible de préciser une version minimum exacte directement depuis la ligne de commande
- Utiliser des caractères de comparaison tel que >, < ou d'autres caractères spéciaux qui sont interprétés par le shell, le nom du paquet et la version doivent être mis entre guillemets

```
py -m pip install SomePackage==1.0.4 # specific version  
py -m pip install "SomePackage>=1.0.4" # minimum version
```

- Normalement, si une bibliothèque appropriée est déjà installée, l'installer à nouveau n'aura aucun effet

02 - Manipuler les bibliothèques

Installation des bibliothèques externes (pip)



Installation des bibliothèques

- La mise à jour des bibliothèques existantes doit être demandée explicitement :

```
py -m pip install --upgrade SomePackage
```

- Pour désinstaller une bibliothèque, il faut utiliser la commande suivante

```
py -m pip uninstall sampleproject
```

02 - Manipuler les bibliothèques

Installation des bibliothèques externes (pip)

Bibliothèques graphiques

- La bibliothèque **matplotlib** (et sa sous-bibliothèque pyplot) sert essentiellement à afficher des graphismes. Son utilisation ressemble beaucoup à celle de Matlab.
- Pour installer **matplotlib**, il faut taper la commande :

```
py -m pip install matplotlib
```

```
PS C:\Users\DELL> pip install matplotlib
Collecting matplotlib
  Downloading matplotlib-3.4.3-cp39-cp39-win_amd64.whl (7.1 MB)
    |██████████| 7.1 MB 2.2 MB/s
Collecting numpy>=1.16
  Using cached numpy-1.21.3-cp39-cp39-win_amd64.whl (14.0 MB)
Collecting cycler>=0.10
  Downloading cycler-0.10.0-py2.py3-none-any.whl (6.5 kB)
Collecting kiwisolver>=1.0.1
  Downloading kiwisolver-1.3.2-cp39-cp39-win_amd64.whl (52 kB)
    |██████████| 52 kB 84 kB/s
Requirement already satisfied: pyparsing>=2.2.1 in c:\users\dell\appdata\local\packages\pythonsoftwarefoundation.python.3.9_qbz5n2kfra8p0\localcache\local-packages\python39\site-packages (from matplotlib) (2.4.7)
Collecting pillow>=6.2.0
  Downloading Pillow-8.4.0-cp39-cp39-win_amd64.whl (3.2 MB)
    |██████████| 3.2 MB 1.7 MB/s
Collecting python-dateutil>=2.7
  Downloading python_dateutil-2.8.2-py2.py3-none-any.whl (247 kB)
    |██████████| 247 kB 2.2 MB/s
Requirement already satisfied: six in c:\users\dell\appdata\local\packages\pythonsoftwarefoundation.python.3.9_qbz5n2kfra8p0\localcache\local-packages\python39\site-packages (from cycler>=0.10->matplotlib) (1.16.0)
```

02 - Manipuler les bibliothèques

Installation des bibliothèques externes (pip)



Bibliothèques graphiques

- PyQt est un module libre qui permet de créer des interfaces graphiques en Python.
- Pour installer PyQt, il faut taper la commande :

```
py -m pip install pyqt5
```

```
PS C:\Users\DELL> pip install pyqt5
Collecting pyqt5
  Downloading PyQt5-5.15.5-cp36-abi3-win_amd64.whl (6.7 MB)
    |██████████| 6.7 MB 2.2 MB/s
Collecting PyQt5-sip<13,>=12.8
  Downloading PyQt5_sip-12.9.0-cp39-cp39-win_amd64.whl (63 kB)
    |██████████| 63 kB 280 kB/s
Collecting PyQt5-Qt5>=5.15.2
  Downloading PyQt5_Qt5-5.15.2-py3-none-win_amd64.whl (50.1 MB)
    |██████████| 50.1 MB 1.1 MB/s
Installing collected packages: PyQt5-sip, PyQt5-Qt5, pyqt5
Successfully installed PyQt5-Qt5-5.15.2 PyQt5-sip-12.9.0 pyqt5-5.15.5
WARNING: You are using pip version 21.2.4; however, version 21.3.1 is available.
You should consider upgrading via the 'C:\Users\DELL\AppData\Local\Microsoft\WindowsApps\PythonSoftwareFoundation.Python.3.9_qbz5n2kfra8p0\python.exe -m pip install --upgrade pip' command.
```

Bibliothèques graphiques

- Tkinter est un module intégré à Python pour développer des applications graphiques.
- Ce module se base sur la bibliothèque graphique Tcl/Tk.
- Pour installer Tk, il faut taper la commande :

```
py -m pip install tk
```

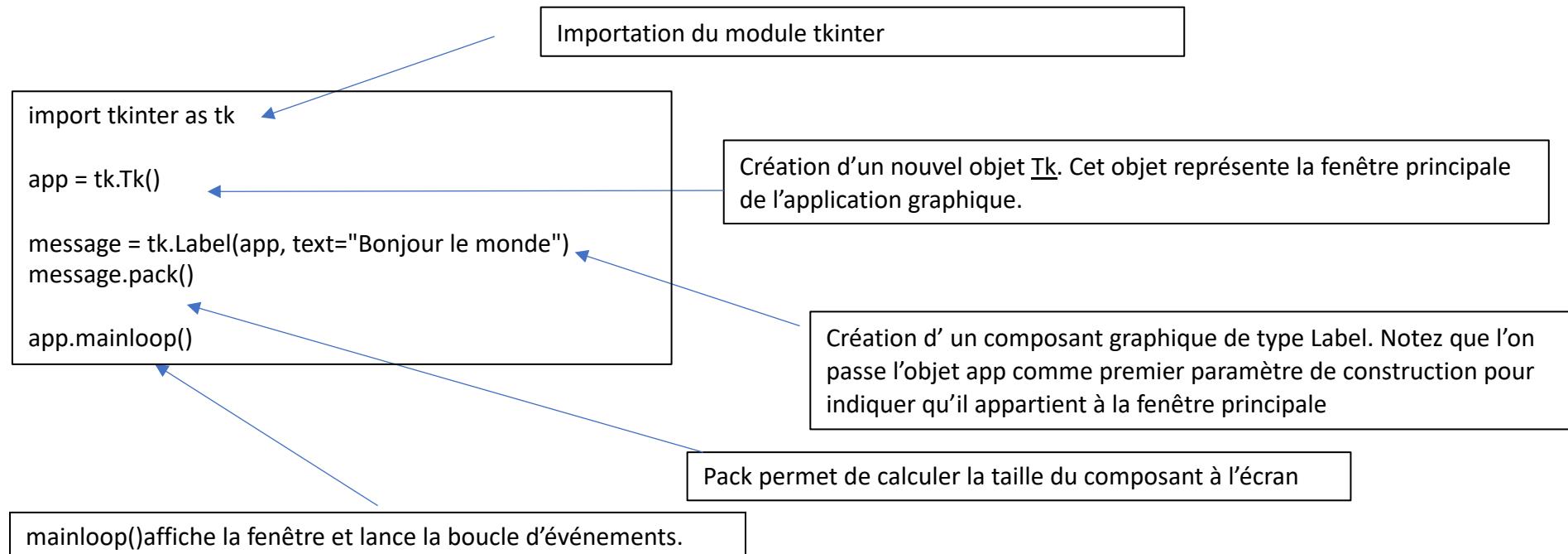
```
PS C:\Users\DELL> pip install tk
Collecting tk
  Downloading tk-0.1.0-py3-none-any.whl (3.9 kB)
Installing collected packages: tk
Successfully installed tk-0.1.0
WARNING: You are using pip version 21.2.4; however, version 21.3.1 is available.
You should consider upgrading via the 'C:\Users\DELL\AppData\Local\Microsoft\WindowsApps\PythonSoftwareFoundation.Python.3.9_qbz5n2kfra8p0\python.exe -m pip install --upgrade pip' command.
```

02 - Manipuler les bibliothèques

Installation des bibliothèques externes (pip)

Tkinter

- Le programme ci-dessous montre le principe de base de tkinter



02 - Manipuler les bibliothèques

Installation des bibliothèques externes (pip)

Tkinter - Lancer une fonction lorsqu'un bouton est pressé

```
import tkinter  
  
r = tkinter.Tk ()  
  
but = tkinter.Button (text = "changement légende")  
but.pack ()  
  
  
def change_legende () :  
    global but  
    but.config (text = "nouvelle légende")  
  
  
but.config (command = change_legende)  
root.mainloop ()
```

But est un bouton qui a pour fonction **change_legende**

fonction change_legende modifie la légende du bouton

Affectation du bouton but à la fonction change_legende



Command permet de lancer le programme et d'afficher la fenêtre

02 - Manipuler les bibliothèques

Installation des bibliothèques externes (pip)

Tkinter - Associer un événement à un objet

- Une fonction peut être exécutée lorsqu'on déplace la souris, on tape sur une touche du clavier ou à n'importe quel événement capté par l'interface graphique.
- La classe **Event** définit les événements. Le tableau suivant décrit les principaux événements relatifs aux mouvements de la souris et à la pression d'une touche.

char	Char contient le code de la touche pressée. Il ne prend pas en considération les touches muettes (return, suppr, shift, ctrl et alt)
keysym	Il contient le code de la touche pressée quelque soit la touche.
num	Cette attribut contient l'identificateur de l'objet qui a reçu l'événement.
x,y	Coordonnées relatives de la souris par rapport au coin supérieur gauche de l'objet ayant reçu l'événement.
x_root, y_root	Coordonnées absolues de la souris par rapport au coin supérieur gauche de l'écran.
widget	Identifiant permettant d'accéder à l'objet ayant reçu l'événement.

Tkinter - Associer un événement à un objet

- La méthode **bind** lance l'exécution d'une fonction lors de l'interception d'un évènement par un objet.
- La fonction exécutée possède comme paramètre un seul objet de type Event. Sa syntaxe est comme suit :

w.bind(ev, fonction)

- **w** représente l'ID de l'objet qui va intercepter l'évènement **ev**. Le Tableau suivant décrit les valeurs que peut prendre **ev**.
- **fonction** représente la fonction qui va être appelée lors de l'arrivé de l'évènement. Elle possède un seul paramètre de type Event.

<Key>	Capte n'importe quelle touche du clavier lorsqu'on presse dessus.
<Button-i>	Capte le clique de la souris. i doit être remplacé par 1,2,3.
<ButtonRelease-i>	capte le relâchement du clique de la souris. i doit être remplacé par 1,2,3.
<Double-Button-i>	Capte le double clique de la souris. i doit être remplacé par 1,2,3.
<Motion>	Capte le déplacement du curseur de la souris.
<Enter>	Lorsque le curseur de la souris entre dans la zone graphique de l'objet, <Enter> capte l'évènement associé.
<Leave>	Capte l'évènement associé lorsque le curseur de la souris sort de sa zone géographique.

02 - Manipuler les bibliothèques

Installation des bibliothèques externes (pip)

- L'exemple ci-dessous illustre l'utilisation de la méthode **bind**. Le bouton de la fenêtre doit capter n'importe quel mouvement du curseur lorsqu'il se trouve au-dessus de sa zone géographique (appuyer sur une touche du clavier ou cliquer sur un bouton de la souris).

```
import tkinter
root = tkinter.Tk()
b = tkinter.Button(text="appuyer sur une touche")
b.pack()

def affiche_touche_pressee (evt) :
    print("----- touche pressee")
    print("evt.char = ", evt.char)
    print("evt.keysym = ", evt.keysym)
    print("evt.num = ", evt.num)
    print("evt.x,evt.y = ", evt.x, ",", evt.y)
    print("evt.x_root,evt.y_root = ", evt.x_root, ",", evt.y_root)
    print("evt.widget = ", evt.widget)

    b.bind ("<Key>", affiche_touche_pressee)
    b.bind ("<Button-1>", affiche_touche_pressee)
    b.bind ("<Motion>", affiche_touche_pressee)
    b.focus_set ()  
  
    root.mainloop ()
```



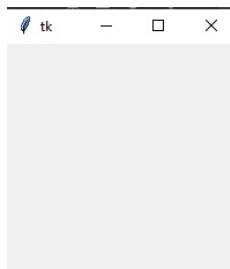
focus_set est l'objet qui capte les événements liés au clavier

Composants graphiques (Widgets)

Fenêtre principale

- Les interfaces graphiques sont composées d'objets ou widgets ou contrôles. Afin d'afficher cette fenêtre il faut écrire les lignes de codes suivantes :

```
root = tkinter.Tk ()  
# ici, il faut définir la spécification des objets et leurs positions  
root.mainloop ()
```



Boîte de message

- Tkinter fournit des fonctions simples pour afficher des boîtes de message à l'utilisateur. Ces fonctions prennent comme premier paramètre le titre de la fenêtre de dialogue et comme second paramètre le message à afficher.
 - Message d'information**

```
from tkinter import messagebox  
  
messagebox.showinfo("Message info", "Ceci est un message d'information")
```



02 - Manipuler les bibliothèques

Installation des bibliothèques externes (pip)

Composants graphiques (Widgets)

Boîte de message

- Message d'avertissement

```
from tkinter import messagebox
messagebox.showwarning("Message d'avertissement", "Ceci est un message d'avertissement")
```



- Message d'erreur

```
from tkinter import messagebox
messagebox.showerror("Message d'erreur", "Ceci est un message d'erreur")
```



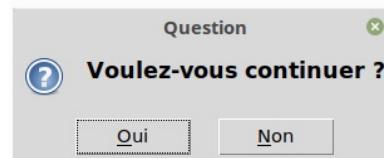
- Question à réponse ok / annuler

```
from tkinter import messagebox
reponse = messagebox.askokcancel("Question", "Voulez-vous continuer ?")
```



- Question à réponse oui / non

```
from tkinter import messagebox
reponse = messagebox.askyesnocancel("Question", "Voulez-vous continuer ?")
```

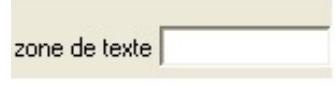


Composants graphiques (Widgets)

Zone de texte

- Une zone de texte est un label ou une légende qui se trouve en règle générale à côté d'une zone de saisie et qui indique à l'utilisateur ce qu'il faut écrire. Afin de créer une zone de texte, il faut écrire cette ligne de code:

```
zone_texte = tkinter.Label (text = "zone de texte")
```



- Afin de changer le texte qui se trouve dans la zone de texte, il faut appeler la méthode config comme suit :

```
zone_texte = tkinter.Label (text = "premier texte")
# ...
# pour changer de texte
zone_texte.config (text = "second texte")
```

Une zone de texte peut être à l'état **DISABLED**

- Pour ce faire, il faut ajouter la ligne de code suivante :
- Pour revenir à l'état normal, il faut écrire cette ligne de code :

```
zone_texte.config (state = tkinter.DISABLED)
```

```
zone_texte.config (state = tkinter.NORMAL)
```

Ces deux options peuvent être utilisées sur n'importe quel type d'objet d'une interface graphique

Composants graphiques (Widgets)

Bouton

- Un bouton est une composante graphique qui associe une fonction au clic de la souris. Afin de créer un bouton, il faut écrire le code suivant :

```
bouton = tkinter.Button (text = "zone de texte")
```

- Afin de modifier le texte qui apparaît au dessus d'un bouton, il faut faire appel à la méthode config comme suit :

```
bouton = tkinter.Button (text = "premier texte")
# ...
# pour changer de texte
bouton.config (text = "second texte")
```

Composants graphiques (Widgets)

Zone de saisie

- Une zone de saisie est une composante graphique qui offre à l'utilisateur la possibilité d'insérer des données.
- Afin de créer une zone de saisie, il faut faire appel à la classe Entry en écrivant la ligne de code suivante :

```
saisie = tkinter.Entry ()
```

- Afin de modifier le contenu de la zone de saisie, il faut faire appel à la méthode insert.

```
# le premier paramètre est la position  
# où insérer le texte (second paramètre)  
saisie.insert (pos, "contenu")
```

- Afin de récupérer le contenu d'une zone de saisie, il faut utiliser la méthode get :

```
contenu = saisie.get ()
```

- Pour supprimer le contenu de la zone de saisie, il faut utiliser la méthode delete.

```
# supprime le texte entre les positions pos1, pos2  
saisie.delete (pos1, pos2)
```

02 - Manipuler les bibliothèques

Installation des bibliothèques externes (pip)

Composants graphiques (Widgets)

Case à cocher

- Afin de créer une case à cocher, il faut faire appel à la classe Checkbutton.

```
# crée un objet entier pour récupérer la valeur de la case à cocher,  
# 0 pour non cochée, 1 pour cochée  
v = tkinter.IntVar()  
case = tkinter.Checkbutton(variable = v)
```



- Afin d'identifier la case qui a été sélectionnée, il faut exécuter l'instruction suivante :

```
v.get() # égal à 1 si la case est cochée, 0 sinon
```

- Les instructions ci-dessous permettent de cocher et de décocher une case:

```
case.select() # pour cocher  
case.deselect() # pour décocher
```

- Afin d'associer un texte à une case, il faut faire appel à la méthode config :

```
case.config(text = "case à cocher")
```

02 - Manipuler les bibliothèques

Installation des bibliothèques externes (pip)

Bouton radio

- Afin de créer un bouton radio, il faut faire appel à la classe **Radiobutton**.
- Leurs fonctionnent est similaires aux cases à cocher excepté le fait qu'elles fonctionnent en groupe. Pour créer un groupe de trois cases rondes, il suffit d'écrire le programme suivant :

```
# crée un objet entier partagé pour récupérer le numéro du bouton radio activé
v = tkinter.IntVar ()
case1 = tkinter.Radiobutton (variable = v, value = 10)
case2 = tkinter.Radiobutton (variable = v, value = 20)
case3 = tkinter.Radiobutton (variable = v, value = 30)
```

- v est une variable utilisée par les trois boutons radio. Value permet d'associer une valeur à chaque radio.
 - Si v == 10, le premier bouton radio sera sélectionné.
 - Si v == 20, le second bouton radio le sera.
- La méthode Get permet de retourner le numéro du bouton radio qui a été sélectionné :

```
v.get () # retourne le numéro du bouton radio coché (ici, 10, 20 ou 30)
```

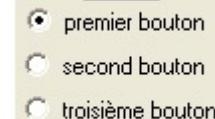
02 - Manipuler les bibliothèques

Installation des bibliothèques externes (pip)

Bouton radio

- Le programme suivant permet la sélection d'un des boutons radio :

```
v = tkinter.IntVar ()  
  
case1 = tkinter.Radiobutton (variable = v, value = 10)  
case2 = tkinter.Radiobutton (variable = v, value = 20)  
case3 = tkinter.Radiobutton (variable = v, value = 30)  
  
v.set (numero) # numéro du bouton radio à cocher  
# pour cet exemple, 10, 20 ou 30
```

- 
- premier bouton
 - second bouton
 - troisième bouton

- La méthode config permet l'insertion d'un texte à un bouton radio :

```
case1.config (text = "premier bouton")  
case2.config (text = "second bouton")  
case3.config (text = "troisième bouton")
```

02 - Manipuler les bibliothèques

Installation des bibliothèques externes (pip)

Liste

- Une liste est une composante graphique qui contient un ensemble d'items sélectionnables. Afin de créer une liste, il faut faire appel à la classe ListBox :

```
li = tkinter.Listbox ()
```

- L'instruction suivante permet de modifier les dimensions d'une liste :

```
# modifie les dimensions de la liste  
# width <-> largeur  
# height <-> hauteur en lignes  
li.config (width = 10, height = 5)
```



première ligne

- La méthode insert permet d'insérer un élément dans une liste déroulante :

```
pos = 0 # un entier, "end" ou tkinter.END pour insérer ce mot à la fin  
li.insert (pos, "première ligne")
```

02 - Manipuler les bibliothèques

Installation des bibliothèques externes (pip)

Liste

- La méthode **select_set** permet de sélectionner un élément de la liste.

```
pos1 = 0
li.select_set (pos1, pos2 = None)
# sélectionne tous les éléments entre les indices pos1 et
# pos2 inclus ou seulement celui d'indice pos1 si pos2 == None
```

- La méthode **curselection** retourne les ID des éléments sélectionnés.

```
sel = li.curselection ()
```

- La méthode **get** permet la récupération d'un élément de la liste. La méthode **codes{size}** retourne le nombre d'éléments.

```
for i in range (0, li.size()):
    print(li.get (i))
```

02 - Manipuler les bibliothèques

Installation des bibliothèques externes (pip)

Tableau (TreeView)

- Le widget Treeview permet d'afficher les données en lignes et en colonnes. La création d'un TreeView doit suivre les étapes suivantes

Etape 1 : Création du treeView

- Columns=ac définit les nom des colonnes, show='headings' signifie que la première ligne est l'en-tête du tableau, height=7 signifie que le tableau est de 7 lignes

```
tv=ttk.Treeview(root,columns=ac,show='headings',height=7)
```

Etape 2 : Définir les propriétés des colonnes

- a[i] est le nom de la colonne, width=70 est la taille de la colonne, anchor='e' est le type de l'alignement

```
tv.column(ac[i],width=70,anchor='e')
```

Etape 3 : Définition de la première ligne du tableau

- Text= area[i] présente le contenu de chaque colonne de la première ligne

```
tv.heading(ac[i],text=area[i])
```

Etape 4 : Remplissage du tableau

- Values=sales_data[i] définit le contenu de chaque ligne du tableau

```
tv.insert("",'end',values=sales_data[i])
```

02 - Manipuler les bibliothèques

Installation des bibliothèques externes (pip)

Tableau (TreeView)

Exemple :

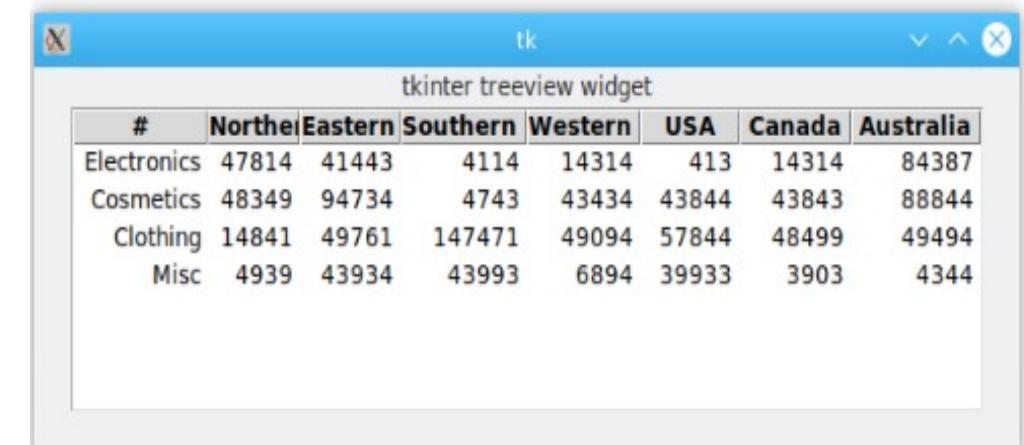
```
import tkinter as tk
from tkinter import ttk

root = tk.Tk()
root.geometry('320x240')
tk.Label(root,text='tkinter treeview widget').pack()
area=('#', 'Northern Europe', 'Eastern Europe', 'Southern Europe', 'Western Europe', 'USA', 'Canada', 'Australia')

ac=('all','n','e','s','ne','nw','sw','c')
sales_data=[('Electronics','47814','41443','4114','14314','413','14314','84387'),
           ('Cosmetics','48349','94734', '4743','43434','43844','43843','88844'),
           ('Clothing','14841','49761', '147471','49094', '57844','48499','49494'),
           ('Misc','4939','43934','43993','6894', '39933','3903','4344')
          ]

tv=ttk.Treeview(root,columns=ac,show='headings',height=7)
for i in range(8):
    tv.column(ac[i],width=70,anchor='e')
    tv.heading(ac[i],text=area[i])
tv.pack()

for i in range(4):
    tv.insert("",'end',values=sales_data[i])
root.mainloop()
```



#	Northe	Eastern	Southern	Western	USA	Canada	Australia
Electronics	47814	41443	4114	14314	413	14314	84387
Cosmetics	48349	94734	4743	43434	43844	43843	88844
Clothing	14841	49761	147471	49094	57844	48499	49494
Misc	4939	43934	43993	6894	39933	3903	4344

02 - Manipuler les bibliothèques

Installation des bibliothèques externes (pip)

Positionnement des objets dans une fenêtre

- Les méthodes pack, grid, place permettent de placer les objets graphiques dans une fenêtre.
- **pack, grid** : offrent la possibilité de positionner des objets sans se soucier de leurs dimensions et de leurs positions. La fenêtre adapte ces deux derniers automatiquement.
- **Place** : positionne les objets dans une fenêtre selon des coordonnées spécifiques.

Méthode pack

- **Pack** empile les objets graphiques d'une manière successive comme le montre l'exemple suivant :

```
I = tkinter.Label (text = "première ligne")
I.pack ()
s = tkinter.Entry ()
s.pack ()
e = tkinter.Label (text = "seconde ligne")
e.pack ()
```



- L'option Side permet de changer l'empilement des objets en les plaçant côté à côté à droite.

```
I = tkinter.Label (text = "première ligne")
I.pack (side = tkinter.RIGHT)
s = tkinter.Entry ()
s.pack (side = tkinter.RIGHT)
e = tkinter.Label (text = "seconde ligne")
e.pack (side = tkinter.RIGHT)
```

02 - Manipuler les bibliothèques

Installation des bibliothèques externes (pip)

Méthode grid

- La méthode grid préconise l'utilisation d'une grille dont chaque case peut contenir un objet graphique. L'exemple ci-dessous place trois objets dans les cases de coordonnées (0,0), (1,0) et (0,1).

```
I = tkinter.Label (text = "première ligne")
I.grid (column = 0, row = 0)
s = tkinter.Entry ()
s.grid (column = 0, row = 1)
e = tkinter.Label (text = "seconde ligne")
e.grid (column = 1, row = 0)
```



- La méthode grid possède plusieurs options telles que:
 - column** : colonne dans laquelle sera placé l'objet.
 - columnspan** : nombre de colonnes que doit occuper l'objet.
 - row** : ligne dans laquelle sera placé l'objet.
 - rowspan** : nombre de lignes que doit occuper l'objet.

02 - Manipuler les bibliothèques

Installation des bibliothèques externes (pip)

Méthode place

- La méthode place permet de positionner un objet graphique selon des coordonnées bien spécifique :

```
I = tkinter.Label(text="première ligne")  
I.place (x=10, y=50)
```





CHAPITRE 2

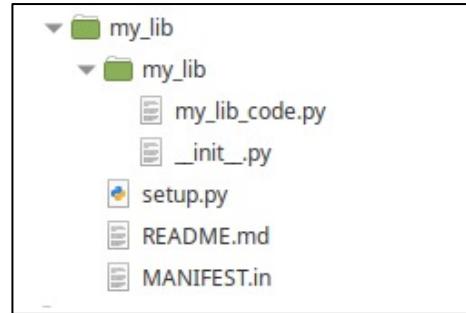
Manipuler les bibliothèques

1. Installation des bibliothèques externes (pip)
2. **Création des bibliothèques**
3. Importation des bibliothèques

02 - Manipuler les bibliothèques

Création des bibliothèques

- Parfois les bibliothèques standard ne suffisent pas et on espère créer une bibliothèque contenant des fonctions spécifiques.
- Pour mettre en place une bibliothèque, le projet doit respecter la structure suivante :



- Le dossier **my_lib**, qui contient l'ensemble de code de notre librairie, ainsi que le fichier **__init__.py** permettant à Python de considérer le dossier comme contenant des paquets.
- Le fichier **README.md** qui contient la description complète de la librairie
- Le fichier **MANIFEST.in** qui liste tous les fichiers non-Python de la librairie
- Le fichier **setup.py** qui contient la fonction setup permettant l'installation de la librairie sur le système

02 - Manipuler les bibliothèques

Création des bibliothèques

Le fichier setup.py

- Ce fichier contient la **méthode setup** qui permettra l'installation de la librairie sur le système.
- La fonction **setup** peut prendre en argument une trentaine d'argument, mais voici dans l'exemple ceux qui seront le plus souvent utilisés.
- Les premiers paramètres de cette méthode sont surtout des paramètres descriptifs de la librairie, de son auteur, sa licence, sa version, etc.

```
from setuptools import setup

setup(
    name='my_lib',
    version='1.0.0',
    author='Author Name',
    author_email='author.name@mail.com',
    description='Courte description de la librairie',
    license='Other/Proprietary License',
    keywords='lib',
    url='Lien vers la page officielle de la librairie',
    packages=[
        'my_lib'
    ],
    long_description=open('README.md').read(),
    classifiers=[
        'Development Status :: 4 - Beta',
        'Intended Audience :: Developers',
        'License :: Other/Proprietary License',
        'Natural Language :: French',
        'Operating System :: OS Independent',
        'Programming Language :: Python :: 3.7',
        'Topic :: Software Development',
        'Topic :: Software Development :: Libraries :: Python Modules',
        "Topic :: Utilities"
    ],
    install_requires=[
        'PyMongo==3.7.2'
    ],
    include_package_data=True
)
```

02 - Manipuler les bibliothèques

Création des bibliothèques

- Concernant les autres paramètres :
 - **package** : contient la liste de tous les packages de la librairie qui seront insérés dans la distribution
 - **classifier** : méta donnée permettant aux robots de correctement classer la librairie
 - **instal_requires** : cette partie contient toutes les dépendances nécessaires au bon fonctionnement de votre librairie
 - **include_package_data** : permet d'activer la prise en charge du fichier MANIFEST.in
- Une fois l'ensemble de ces éléments défini, la librairie peut être installée sur le système en utilisant la commande suivante (exécuter la commander dans le répertoire du projet) :

```
py setup.py install
```

02 - Manipuler les bibliothèques

Création des bibliothèques

Exemple :

- Création de la bibliothèque

Ce PC > Disque local (E:) > Nouvelle_Bib			
Nom	Modifié le	Type	Taille
Nouvelle_Bib	2021-10-25 12:42	Dossier de fichiers	
setup	2021-10-25 12:43	Python File	1 Ko

Ce PC > Disque local (E:) > Nouvelle_Bib > Nouvelle_Bib			
Nom	Modifié le	Type	Taille
__init__	2021-10-23 7:59	Python File	1 Ko
bonjour	2021-10-25 11:44	Python File	1 Ko

```
class Bonjour:
    def __init__(self,nom):
        self.nom=nom
    def affiche(self):
        print("Bonjour:"+self.nom)
```

```
from setuptools import setup

setup(
    name="Nouvelle_Bib",
    version="0.0.1",
    author="authorX",
    author_email="authorX@yahoo.fr",
    description="courte description de la librairie",
    url="lien vers la page officielle de la librairie",
    licence='Other/Proprietary Licence',
    packages={
        'Nouvelle_Bib',
    },
    classifiers=[
        "Programming Language :: Python :: 3",
        "License :: OSI Approved :: MIT License",
        "Operating System :: OS Independent",
    ],
    install_requires=[],
    include_package_data=True
)
```

02 - Manipuler les bibliothèques

Création des bibliothèques

Installation de la bibliothèque

- Exécution de la commande :

```
py setup.py install
```

```
PS E:\nouvelle_Bib> py setup.py install
C:\Users\DELL\AppData\Local\Programs\Python\Python39\lib\distutils\dist.py:259: UserWarning: 'licence' distribution option is deprecated; use 'license'
    warnings.warn(msg)
running install
running bdist_egg
running egg_info
creating Nouvelle_Bib.egg-info
writing Nouvelle_Bib.egg-info\PKG-INFO
writing dependency_links to Nouvelle_Bib.egg-info\dependency_links.txt
writing top-level names to Nouvelle_Bib.egg-info\top_level.txt
writing manifest file 'Nouvelle_Bib.egg-info\SOURCES.txt'
reading manifest file 'Nouvelle_Bib.egg-info\SOURCES.txt'
writing manifest file 'Nouvelle_Bib.egg-info\SOURCES.txt'
installing library code to build\bdist.win-amd64\egg
running install_lib
running build_py
creating build
creating build\lib
creating build\lib\Nouvelle_Bib
copying Nouvelle_Bib\bonjour.py -> build\lib\Nouvelle_Bib
copying Nouvelle_Bib\__init__.py -> build\lib\Nouvelle_Bib
creating build\bdist.win-amd64
creating build\bdist.win-amd64\egg
creating build\bdist.win-amd64\egg\Nouvelle_Bib
copying build\lib\Nouvelle_Bib\bonjour.py -> build\bdist.win-amd64\egg\Nouvelle_Bib
copying build\lib\Nouvelle_Bib\__init__.py -> build\bdist.win-amd64\egg\Nouvelle_Bib
byte-compiling build\bdist.win-amd64\egg\Nouvelle_Bib\bonjour.py to bonjour.cpython-39.pyc
byte-compiling build\bdist.win-amd64\egg\Nouvelle_Bib\__init__.py to __init__.cpython-39.pyc
creating build\bdist.win-amd64\egg\EGG-INFO
copying Nouvelle_Bib.egg-info\PKG-INFO -> build\bdist.win-amd64\egg\EGG-INFO
copying Nouvelle_Bib.egg-info\SOURCES.txt -> build\bdist.win-amd64\egg\EGG-INFO
copying Nouvelle_Bib.egg-info\dependency_links.txt -> build\bdist.win-amd64\egg\EGG-INFO
copying Nouvelle_Bib.egg-info\top_level.txt -> build\bdist.win-amd64\egg\EGG-INFO
zip_safe flag not set; analyzing archive contents...
creating dist
creating 'dist\Nouvelle_Bib-0.0.1-py3.9.egg' and adding 'build\bdist.win-amd64\egg' to it
removing 'build\bdist.win-amd64\egg' (and everything under it)
Processing Nouvelle_Bib-0.0.1-py3.9.egg
Copying Nouvelle_Bib-0.0.1-py3.9.egg to c:\users\dell\appdata\local\programs\python\python39\lib\site-packages
Adding Nouvelle-Bib 0.0.1 to easy-install.pth file

Installed c:\users\dell\appdata\local\programs\python\python39\lib\site-packages\nouvelle_bib-0.0.1-py3.9.egg
```

Bibliothèque bien installée!!

02 - Manipuler les bibliothèques

Création des bibliothèques



- L'installation de la bibliothèque entraîne la création d'un fichier : Nouvelle_Bib-0.0.1-py3.9.egg dans le dossier \Lib\site-packages de python

```
Installed c:\users\dell\appdata\local\programs\python\python39\lib\site-packages\nouvelle_bib-0.0.1-py3.9.egg
```

- Utilisation de la bibliothèque créée

```
from Nouvelle_Bib import bonjour #Importation du module bonjour de la bibliothèque Nouvelle_Bib  
  
e=bonjour.Bonjour("Meriam") #Appel de la classe Bonjour du module bonjour  
  
e.affiche()
```



CHAPITRE 2

Manipuler les bibliothèques

1. Installation des bibliothèques externes (pip)
2. Création des bibliothèques
3. **Importation des bibliothèques**

02 - Manipuler les bibliothèques

Importation des bibliothèques



Importer un module

- Pour utiliser un module Python dans la bibliothèque standard de Python ou un module d'une bibliothèque créée, il faut utiliser la syntaxe suivante :

```
import <module>
```

- Par exemple, le module random, on utilise la syntaxe suivante :

```
import random
```

- On identifiera les fonctions importées en préfixant leur nom par celui du module.

Par exemple :

```
import random
print(random.choice('aaioer'))
```

La fonction choice: choisit un élément au hasard dans une liste

02 - Manipuler les bibliothèques

Importation des bibliothèques



Importer une fonction particulière d'un module

- Pour importer une fonction d'un module, par exemple la fonction `choice()` du module random, on utilise la syntaxe suivante :

```
from random import choice
```

- On identifiera simplement la fonction importée par son nom.

Par exemple :

```
from random import choice
print(choice('aaioer')) #affiche r,une lettre aléatoire parmi celles proposées
```

- Les autres fonctions du module ne sont pas disponibles :

```
from random import choice
randint(12,42) #erreur

#affiche:
#Traceback (most recent call last):
#File "C:/Users/DELL/Desktop/poo/e3.py", line 3, in <module>
#randint(12,42)
#NameError: name 'randint' is not defined
```

Importer toutes les fonctions d'un module

- Pour importer l'ensemble des fonctions d'un module.

On utilise alors la syntaxe :

```
from random import*
```

- On identifiera simplement les fonctions importées par leur nom.

Par exemple :

```
from random import*
print(choice('uiو')) #affiche i
print(randint(12,42)) #affiche 27
```