Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Informatique

Cours et Travaux Dirigés

2ème Année (SP–SP–ST)

Année Universitaire 2024 – 2025

Enseignant
M. Anis SAIED

Contact : `anis.saied@ipein.ucar.tn`

Présentation du document

Ce recueil rassemble l'ensemble des supports de cours et de travaux dirigés (TD) destinés aux étudiants de deuxième année de l'*IPEIN* pour l'année universitaire 2024-2025.

L'objectif de ce document est de fournir un support unique, structuré et facile à consulter, couvrant les concepts fondamentaux de l'informatique théorique et appliquée, ainsi que les outils mathématiques nécessaires à la spécialité.

Organisation du contenu

Le contenu est divisé en huit chapitres couvrant les thèmes suivants :

- **Programmation :** Rappels de syntaxe et approfondissement de la **Programmation Orientée Objet (POO)**, concept clé pour le développement logiciel moderne.
- **Structures de Données :** Étude des structures linéaires fondamentales que sont les **Piles** et les **Files**.
- **Gestion des Données :** Introduction aux bases de données relationnelles, à l'algèbre relationnelle et à la maîtrise du langage **SQL**.
- **Analyse et Calcul Numérique :** Application de l'**Algèbre Linéaire** et des méthodes d'interpolation (Lagrange) au calcul scientifique.
- **Sciences du Numérique :** Introduction aux principes de la **Cryptographie** et aux techniques fondamentales du **Traitement d'Images**.

Guide de lecture

Pour faciliter la révision, ce document respecte une mise en page spécifique :

- Les **Cours** sont présentés sous forme de diapositives regroupées (2 par page) pour une vue d'ensemble synthétique.
- Les **Travaux Dirigés (TD)** sont présentés en pleine page afin de laisser l'espace nécessaire à la lecture des énoncés et à la résolution des exercices.

Bon travail à tous.

Ressources et Références

Ce document a été conçu pour offrir une expérience d'apprentissage complète. Les contenus s'appuient sur des standards académiques et des outils modernes de l'informatique.

Références du contenu

Les thématiques abordées dans ce recueil sont basées sur :

- Le **Programme officiel d'informatique** des classes préparatoires.
- La documentation officielle de **Python 3.x** pour la Programmation Orientée Objet.
- Les standards **SQL (ISO/IEC 9075)** pour la gestion des bases de données.
- Les bibliothèques scientifiques : **NumPy** (Calcul numérique) et **Pillow** (Traitement d'images).

Outils technologiques utilisés

La réalisation de ce support et les exercices pratiques reposent sur :

- **LaTeX** : Pour la mise en page structurée et professionnelle du document.
- **Graphviz (DOT)** : Pour la génération automatique de graphes et de schémas relationnels.
- **DB Browser for SQLite** : Pour l'exploration visuelle des bases de données.
- **Pyzo** / **VS Code** : Pour le développement et le test des scripts Python.

Accès numérique et Contact

- **Version numérique :** Ce document est mis à jour régulièrement. Vous pouvez télécharger la dernière version sur :
  `https://anis-saied.com/enseignment/`
- **Retour et Suggestions :** Si vous remarquez une erreur, une faute de frappe ou si vous souhaitez suggérer une amélioration, n'hésitez pas à me contacter par email :
  `anis.saied@ipein.ucar.tn`

Révisions et Annales (Examens & DS)

Pour approfondir votre préparation, j'ai développé une plateforme dédiée au partage des ressources pédagogiques :

- **Annales corrigées :** Vous y trouverez les anciens examens et devoirs surveillés (DS) avec leurs corrections détaillées.
- **Pluridisciplinarité :** Le site propose également des épreuves d'autres matières scientifiques.
- **Lien direct :** `https://ipei.tn`

Sommaire

Rappels

Chapitre 0 : Rappel

Séance 1: Notions de base en Python

Anis SAIED

IPEIN

2024/25

Plan de la séance

Table of contents

Python: Rappel général

- Documentation officielle
- Installer Python: Core, standard library
- Open source
- Utilisation: Web, API, Data Science,
- Emplois
- Certifications: 4C, Microsoft

Un programme Python

- EDI: IDLE, Spyder, Pyzo,
- Code source: instructions, blocs, commentaires
- byte code, interpreter
- Executer un programme: mode script
- 3 fichiers standards associés à chaque programme: std Input / std Output / std Error
- Exemple: Le **shell** (programme, execute des lignes de commandes)
  - std input: clavier
  - std output: écran
  - std error: écran
- IDLE > run > shell → Mode interatif
- IDLE > new > window/file → Mode script

Table of contents

Mots clés

Le choix de nom de variable de doit pas être l'un de ces mots clés.

Python keywords

for, in, range, while, continue, break, yield,
if, elif, else,
or, and, not,
open, close, print, input,
def, global, return, pass, global
lambda, filter, map,
try, except, finally, raise, assert
. . .

Type de variable

- Non typée
- type change selon son contenu: `type(var)`

Le types: int & float

- Opérateurs: $+, -, *, /$
- taille: limitée à l'espace mémoire disponible

```
n=1 ; n=n+1 ; n *=2 ; type(n) \# int
type(n)==int \# True
```

- Notation scientifique

```
f=1e-2 \#0.01
f=3. type(f)==float \# True
```


Le type bool

- Opérateurs: *and*, *or*, *not*
- Valeurs possibles: `True, False`

```
a=0 ; b=1; c=0. ; d=-1.
```

- *bool*(*expr*) evalue logiquement une expression.
- Toute variable non nulle/vide est évaluée comme True.

```
bool(a) ; bool(b) \#False True
```

Le type complex

- partie imaginaire, partie réelle

Conversion de types

- $\boxed{dir(int)}$ retourne les fonctions disponibles et applicables sur les valeurs de ce type.
- Convertir une valeur de type A en type B permet de profiter des fonctions offertes par le type B

  `a=1.0 ; a = int(a)` → passer de `float à int`

  `a = int("abc")` → `Erreur de conversion de type`
- `Rappel : Toute erreur arrête l'exécution du programme` → `Il faut prévoyer un traitement alternatif en cas d'erreur`

Gestion d'erreurs

- Toute instruction qui ne respecte pas les règles du langage Python est considérée comme une **Exception**
- Une Exception entraine l'arrêt immédiat du programme
- Eviter l'arrêt imprévisible du programme necessite
  1. Ecrire les instrcutions succeptibles d'arrêter le programme dans le bloc *try*
  2. Ecrire le traitement alternatif (qui corrige cette erreur) dans le bloc *except*
- Poursuivre la suite du programme après la sortie du bloc *try . . . except*

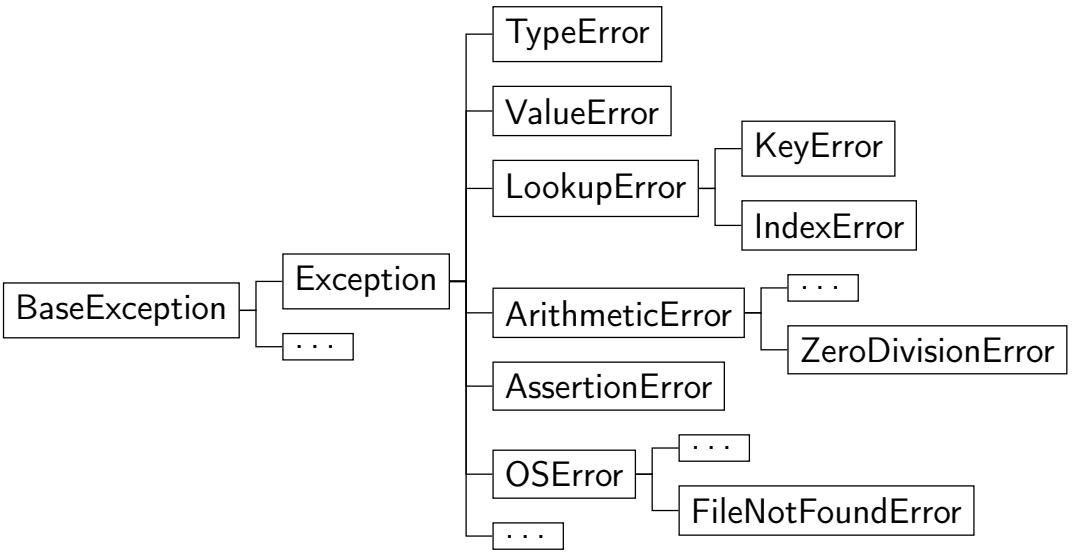
Exemple : voir suite

Exemple: try . . . except

Exemple :

```
try:
        x = 1.2
        a = int(x)
except ValueError:
        print("Erreur: type attendu 'int'")
except ZeroDivisionError:
        print("Erreur: Division par 0")
except:
        print("Erreur imprévue")
else:
        print("Pas d'erreur")
```

Principales Exceptions en Python



Liste complete :
https://docs.python.org/3/library/exceptions.html

raise Exception

- On peut décider qu'on a une exception si une expression ne respecte pas la logique métier (Les exigence de l'exercice).
- *raise* permet d'informer Python qu'on est en situation d'exception.
  Syntaxe: `raise <type d'exception>`
  Exemple

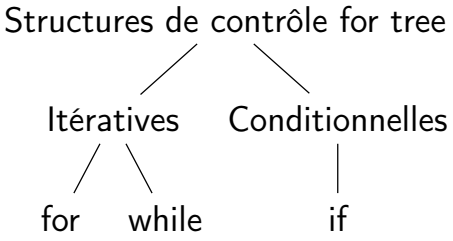
```
try:
        age = -1
        if age < 0:
                raise ValueError
except ValueError:
        print("Erreur: Valeur négative.")
else:
        print("Age correcte.")
```

Utilisation de assert en Python

- L'instruction *assert* est utilisée pour tester si une condition est vraie. Si la condition est fausse, elle lève une exception `AssertionError`.
- `Cette méthode est utile pour vérifier les hypothèses pendant le développement.`

```
def division(a, b):
        assert b != 0, "Le diviseur ne doit pas être zéro"
        return a / b

try:
        result = division(10, 0)
except AssertionError as error:
        print(f"Erreur: {error}")
else:
        print(f"Résultat: {result}")
```


Structures de contrôle



Les structures de contrôle conditionnelles

Exemple

```
x = 10

if x > 0:
    print("x est positif")
elif x == 0:
    print("x est zéro")
else: # si tout est faux.
    print("x est négatif")
```


Les structures de contrôle conditionnelles

Exemple

```
x = 10

if x > 0:
    print("x est positif")
elif x == 0:
    print("x est zéro")
else: # si tout est faux.
    print("x est négatif")
```

- Opérateur ternaire :

```
r = True if x>0 else False
```

Les structures de contrôle conditionnelles

Exemple

```
x = 10

if x > 0:
    print("x est positif")
elif x == 0:
    print("x est zéro")
else: # si tout est faux.
    print("x est négatif")
```

- Opérateur ternaire :

```
r = True if x>0 else False
r = "+" if x>0 else "-" if x<0 else "0"
```


Les structures de contrôle itératives

Répeter l'exécution d'un bloc d'instrcutions

- `for i in range(start, end, step)`
  - Utilisée lorsque le nombre d'itérations est connu à l'avance.
  - Idéale pour parcourir des séquences comme des listes, des tuples, et des chaînes de caractères.
- Boucle `while condtition`
  - Utilisée lorsque le nombre d'itérations dépend d'une condition.
  - Pratique pour les situations où l'on doit attendre qu'une condition change.

Boucle `for` : Exemple

Exemple : Calcul de la somme des nombres impairs entre 1 à 10.

```
s = 0;  for i in range(1, 10, 2): s += i
```

La fonction **range()** génère une séquence de nombres, et la boucle **for** itère sur cette séquence.

`range(start, stop, step)`

- **start**: Valeur initiale (incluse), 0 par défaut.
- **stop**: Valeur finale (exclue), la séquence s'arrête avant ce nombre
- **step**: Incrémentation (optionnel), la différence entre chaque nombre dans la séquence. Si omis, le pas par défaut est 1.

Autrement

```
s = sum(range(1, 10, 2))
s = 0
for i in range(1,10):
    if i % 2 == 0: continue
    s += i
```


Boucle `while` : Exemple

Exemple : Calcul de la somme des nombres impairs entre 1 à 10.

```
s = i = 0
while i<10 :
    i+=1
    if i%2==1: s += i
    #else: continue
```

```
s = i = 0
while 2 : #True
    if i>10: break
    if i%2 : s += i
    i += 1
```

```
s,i = 0,10;
while i>0:
    s += i if i%2 else 0
    i -= 1
```

Table of contents

La fonction *print()*.

Opérations Entrée/Sortie

» `help(print)`

```
print(*args, sep=' ', end='\n', file=None, flush=False)
```

Description :

- `objects` : Les objets à afficher, convertis en chaînes de caractères comme le fait `str()`.
- `sep` : Chaîne de séparation entre les objets (par défaut, un espace ‘ ’).
- `end` : Chaîne ajoutée à la fin. Par défaut, un saut de ligne ‘`\n`‘.
- `file` : Objet avec une méthode `write(string)` ; utilise `sys.stdout` par défaut.
- `flush` : Si `True`, force le vidage du flux.

La fonction *print()*.

Exemples

Exemple 1: Impression de plusieurs objets avec un séparateur personnalisé `print("Bonjour", "le monde", "!", sep="---")`

Exemple 2: Utilisation de 'end' pour personnaliser la fin de ligne

```
print("Début de la ligne", end=" ")
print("fin de la même ligne")
```

Exemple 3: Impression dans un fichier

```
with open("exemple.txt", "w") as file:
    print("Bonjour le monde", file=file)
```


La fonction *print()*.

Exemples: Suite

Exemple 4: Impression avec conversion de type

```
num = 123
print("Le nombre est " + str(num))
```

Exemple 5: Impression avec formatage

```
name = "Alice"
age = 30
print("Nom : " + name + ", Âge : " + str(age))
print("Nom : {}, Âge : {}".format(name, age))
print(f"Nom : {name}, Âge : {age}")

print("Nom : %s, Âge : %d" % (name, age))
#%s :chaine, %d: entier, %f:float

print("Valeur de pi : {:.2f}".format(3.14159))
#.2f affiche le nombre avec 2 décimales: 3.14

print("Grand nombre : {:,}".format(1234567))
#:, ajoute des séparateurs de milliers: 1,234,567
```

La fonction *input()*.

Aide et Explication

- `input(prompt='')` est une fonction intégrée de Python, utilisée pour lire une chaîne de caractères depuis l'entrée standard (habituellement, le clavier).
- Lorsque la fonction `input()` est appelée, elle affiche le texte du `prompt` (message qui demande d'entrer des informations) à l'utilisateur, sans ajouter de saut de ligne à la fin.
- Note: Toutes les entrées sont lues comme des chaînes de caractères. Pour traiter d'autres types de données, il est nécessaire de convertir les chaînes avec des fonctions comme `int()` ou `float()`.

La fonction *input()*.

Exemples

Exemple 1: Lecture d'une entrée utilisateur

```
nom = input("Entrez votre nom: ")
print(f"Bonjour, {nom}!")
```

Exemple 2: Lecture et conversion de données numériques

```
age = input("Entrez votre âge: ")
print(type(age)) # str
try:
    age = int(age)  # Conversion de chaîne à entier
except:
    print("Erreur de valeur")
else:
    print(f"Vous avez {age} ans.")
```

La fonction *input()*.

Exemples: Suite

Exemple 3: Utilisation d'un prompt avec des indications

```
reponse = input("Voulez-vous continuer ? (oui/non): ")
if reponse.lower() == "oui":
    print("Vous avez choisi de continuer.")
else:
    print("Fin du programme.")
```


Table of contents

Résumé des Types de Fonctions en Python

- **Fonctions intégrées** : Fonctions prédéfinies disponibles par défaut, comme `print()` et `len()`.
- **Fonctions définies par l'utilisateur** : Fonctions créées par les développeurs pour réutiliser du code.
- **Fonctions anonymes (lambda)** : Fonctions sans nom utilisées pour des opérations simples.
- **Fonctions d'ordre supérieur** : Fonctions prenant d'autres fonctions en argument ou renvoyant des fonctions.
- **Fonctions récursives** : Fonctions qui s'appellent elles-mêmes pour résoudre des problèmes divisibles.

Fonctions intégrées

- Disponibles sans importation.
- Exemples :
  - `print()` : Affiche du texte à l'écran.
  - `len()` : Retourne la longueur d'une séquence.

Exemple

```
texte = "Bonjour"
print(len(texte))  # Affiche 7
```

Fonctions définies par l'utilisateur

- Créées pour organiser et réutiliser le code.
- Utilisent le mot-clé `def`.

Exemple

```
def addition(a, b):
    return a + b

print(addition(3, 5))  # Affiche 8
```

C'est quoi le mot-clé `return`?

- `return` termine l'exécution d'une fonction et renvoie une valeur à l'appelant.

Quand utiliser `return`?

- Utiliser `return` pour renvoyer le résultat d'une opération ou le contenu d'une fonction.
- Permet de récupérer des données traitées par la fonction.

Fonctions anonymes (lambda)

- Utilisées pour des opérations simples.
- Définies avec le mot-clé `lambda`.

Exemple

```
addition = lambda x, y: x + y
print(addition(3, 5))  # Affiche 8
```

Fonctions d'ordre supérieur

- Prennent d'autres fonctions en argument ou les renvoient.
- Exemples : `map()`, `filter()`.

Exemple

```
def appliquer_fonction(f, liste):
    return [f(x) for x in liste]

résultats = appliquer_fonction(lambda x: x * 2, [1, 2, 3])
print(résultats)  # Affiche [2, 4, 6]
```


Fonctions récursives

- S'appellent elles-mêmes pour résoudre des sous-problèmes.
- Utile pour des problèmes comme le calcul de factorielle.

Exemple

```
def factorielle(n):
    if n == 0:
        return 1
    else:
        return n * factorielle(n - 1)

print(factorielle(5))  # Affiche 120
```

Paramètres, Arguments, et Valeurs par Défaut

Paramètres:

- Ce sont les variables définies dans la signature d'une fonction.
- Ils représentent les valeurs que la fonction attend de recevoir lors de son appel.

Exemple:

```
def greet(name, age):
print(f"Bonjour {name}, vous avez {age} ans!")
```

Arguments:

- Ce sont les valeurs réelles passées à une fonction lors de son appel.

Exemple:

```
greet("Alice", 30)  # "Alice" et 30 sont des arguments
```


Paramètres, Arguments, et Valeurs par Défaut

Suite

Valeurs par défaut:

- Permettent de spécifier une valeur par défaut pour un paramètre si aucun argument n'est fourni lors de l'appel de la fonction.
- Utile pour simplifier les appels de fonction lorsque des valeurs courantes sont souvent utilisées.

Exemple:

```
def greet(name, age=25):
print(f"Bonjour {name}, vous avez {age} ans!")

greet("Bob") # age utilise la valeur par défaut de 25
greet("Carol", 40) # age est explicitement défini à 40
```

Les fonctions: Passage de paramètres

Passage par valeur

Passage par valeur:

- Les valeurs des variables sont copiées dans les paramètres de la fonction.
- Les modifications apportées aux paramètres dans la fonction n'affectent pas les variables d'origine.
- Exemple:

```
def increment(value):
    value += 1
    return value

x = 10
increment(x) # est différent de x = increment(x)
print(x)  # x est toujours 10
```


Les fonctions: Passage de paramètres

Passage par adresse

Passage par adresse

- Les variables passent des références aux objets, pas les objets eux-mêmes.
- Les modifications apportées à l'objet affectent l'objet d'origine.

Exemple:

```
def append_element(lst, elem):
    lst.append(elem)

my_list = [1, 2, 3]
append_element(my_list, 4)
print(my_list)  # my_list est maintenant [1, 2, 3, 4]
```

Variables locales et globales

Variables locales:

- Déclarées à l'intérieur d'une fonction.
- Valables uniquement dans le scope (portée) de cette fonction.
- Exemple:

```
def my_function():
    local_var = 5  # Variable locale
    print(local_var)

my_function()
print(local_var) #Erreur: local_var n'est pas définie ici
```


Variables locales et globales

Variables globales:

- Déclarées à l'extérieur de toute fonction.
- Valables dans tout le script, y compris dans les fonctions.

Exemple:

```
global_var = 10  # Variable globale
def my_function():
    print(global_var) # Peut accéder à la variable globale

my_function()
print(global_var) # Accessible ici aussi
```

Variables locales et globales

Suite

Utilisation de `global`

- Permet de modifier une variable globale à l'intérieur d'une fonction.

Exemple:

```
global_var = 10
def modify_global():
    global global_var
    global_var = 20

modify_global()
print(global_var)  # 20
```

Sans **global**, on aura deux variables différentes(locale et globale) de même nom mais avec des valeurs différentes.

Identificateurs (ID)

Fonction `id()` en Python:

- Retourne l'identifiant (adresse mémoire) d'un objet.
- Peut être utilisé pour vérifier si deux variables font référence au même objet en mémoire.

Exemple :

```
def verifier_id(a, b):
    print(f"Identifiant de a: {id(a)}")
    print(f"Identifiant de b: {id(b)}")
    if id(a) == id(b):
        print("a et b font référence au même objet.")
    else:
        print("a et b ne font pas référence au même objet.")
# Exemple d'utilisation
x = [1, 2, 3];    y = x ;    z = [1, 2, 3]
verifier_id(x, y) # a et b sont identiques, mêmes identifiants.
verifier_id(x, z) # a et b sont différents objets en mémoire,
                  # donc leurs identifiants seront différents.
```

TD : Rappel général

Objectif : se rappeler des principales notions de base du langage de programmation `Python` vues en 1ère année.

Exercice 1: Crible d'Eratosthène

Un nombre premier est un entier qui n'est divisible que par 1 et par lui-même. On se propose d'écrire un programme qui établit la liste de tous les nombres premiers compris entre 2 et 100, en utilisant la méthode du *crible d'Eratosthène* :

- Créer une liste de 100 éléments, chacun initialisé à la valeur 1 ;
- Parcourir cette liste à partir de l'élément d'indice 2: si l'élément analysé possède la valeur 1, mettez à zéro tous les autres éléments de la liste, dont les indices sont les multiples de l'indice auquel vous êtes arrivé.

Lorsque vous aurez parcouru ainsi toute la liste, les indices des éléments qui seront restés à 1 seront les nombres premiers recherchés.
En effet, à partir de l'indice 2, vous annulez tous les éléments d'indices pairs $4, 6, 8, \ldots$
Avec l'indice 3, vous annulez les éléments d'indices $6, 9, \ldots$
Et ainsi de suite. Seul resteront à 1 les éléments dont les indices sont effectivement des nombres premiers.
Travail à faire :

1. Écrire une fonction `init()` permettant de retourner une liste de 100 éléments initialisés à 1 .
2. Écrire une fonction `multiple(L, i)` permettant de mettre à zéro les éléments dont l'indice est un multiple d'un entier donné comme paramètre.
3. Écrire une fonction `suivant(L, i)` qui à partir d'un indice donné, retourne le premier indice dont l'élément correspondant est différent de zéro.
4. Écrire une fonction `crible()` permettant d'appliquer la méthode présentée par la crible d'Eratosthène pour retourner une liste qui contient les nombres premiers (il suffit d'étirer jusqu'à la racine entière de 100).
5. Écrire une fonction récursive `crible_recursive` permettant d'appliquer la méthode présentée pour retourner la liste des nombres premiers.
6. Donner le schéma d'exécution de la fonction précédente (si on veut déterminer les entiers premiers qui sont inférieur à 10).
7. Écrire un programme qui fait appel aux fonctions déjà définies afin d'afficher les entiers premiers compris entre 1 et 100.

Exercice 2: Fonction passée en paramètre

Écrire les fonctions Python suivantes :

1. `carre(n)` : calcule et retourne le carré d'un entier n passé en paramètre.
2. `somme_fonction(f, n)` : calcule et retourne la somme $\sum_{i=0}^{n} f(i)$, où f est une fonction passée en paramètre.
3. Utiliser la fonction `somme_fonction` pour calculer la somme des carrés des entiers $i \in [1, 10]$.

Exercice 3: Recherche dichotomique

On suppose que `L` est une liste triée de réels distincts et que x est un réel vérifiant $L[0] \leq x < L[n-1]$ où n est la taille de la liste.

Écrire une fonction `dicho(x,L)` retournant l'unique entier i vérifiant $L[i] \leq x < L[i+1]$ en faisant une recherche dichotomique.

Exercice 4: Récursivité

Une chaine est palindrome si elle se lit de la même manière de gauche vers la droite et de droite vers la gauche.
Exemples : ELLE, NON, RADAR.
Écrire une fonction récursive qui permet de vérifier si une chaine donnée est palindrome

Exercice 5: Suite de polynômes

On définit une suite de polynôme SP par $\begin{cases} SP_0(x) & = P(x) \\ SP_1(x) & = -P^{'}(x) \\ SP_{k+2}(x) & = Q(x) * SP_{k+1}(x) + SP_k(x) \end{cases}$

Avec :

- P un polynôme de degré n $(n \neq 0)$ tel que : $P(x) = \sum_{i=0}^{n} a_i x^i$
- $P^{'}$ le polynôme dérivé de degré $n-1$ tel que : $P^{'}(x) = \sum_{i=0}^{n-1} (i+1) a_{i+1} x^i$
- Q est le produit des deux polynômes SP_{k+1} et SP_k.

Tout polynôme de degré n sera représenté sous forme d'une liste de taille $n+1$ tel que les coefficients sont les éléments et les degrés sont les indices.

Exemple :
La liste correspondante au polynôme $P(x) = 1 + x - 5x^2 + x^3 - 2x^4 + 6x^6$ est : `L=[1,1,-5,1,-2,0,6]`

On désire déterminer à partir de la suite ci-dessus le polynôme SP_m avec $m \geq 2$.

Travail à faire :

1. Écrire une fonction appelée `saisie_deg()` qui permet de saisir un entier strictement positif. Ajouter un bloc `try-except` qui gère une exception de type `ValueError`, ce qui pourrait se produire si l'utilisateur saisie une valeur non numérique. Imprimer un message d'erreur si cela se produit.
2. Écrire une fonction appelée `saisie_poly(n)` permettant de saisir la représentation d'un polynôme $P(x)$ de degré n dans une liste.
3. Écrire une fonction appelée `derive(P)` permettant de déterminer la représentation du polynôme dérivé d'un polynôme P représenté par une liste, le résultat sera une liste.
4. Écrire une fonction appelée `opp_poly(P)`permettant de déterminer la représentation du polynôme opposé d'un polynôme P représenté par une liste, le résultat sera une liste.
5. Écrire une fonction appelée `add_poly(P1,P2)` qui prend comme paramètres deux polynômes $P1$ et $P2$ et retourne la représentation du polynôme $P1 + P2$.

6. Écrire une fonction appelée `mul_poly(P1, P2)` qui prend comme paramètres deux polynômes $P1$ et $P2$ et retourne la représentation du polynôme $P1 * P2$.
   Sachant que si $P1(x) = \sum_{i=0}^{n1} a_i x^i$ et $P2(x) = \sum_{i=0}^{n2} b_i x^i$ alors le produit est $P1(x) * P2(x) = \sum_{k=0}^{n1+n2} c_k x^k$ avec $c_k = \sum_{i+j=k} a_i b_j$.

7. Écrire le programme principal permettant de :
   (a) Saisir le degré n d'un polynôme.
   (b) Saisir la liste qui représente un polynôme P.
   (c) Déterminer et afficher la liste relative au $i^{\text{ème}}$ terme de la suite du polynôme SP.

Exercice 6: Manipulation de fichiers

Vous travaillerez avec un fichier texte qui stocke des informations sous forme de lignes, chaque ligne contenant un enregistrement (des noms et des âges séparés par une virgule).
Par exemple: `"'Ali', 20"`
Écrivez les fonctions Python suivantes :

1. `ecrire_fichier(fichier, donnees)` : écrit une liste de chaînes de caractères dans un fichier, chaque chaîne représentant un enregistrement. Si le fichier existe, son contenu est remplacé.

2. `chercher_fichier(fichier, mot_cle)` : recherche un enregistrement contenant le mot clé passé en paramètre et retourne l'enregistrement correspondant.

Exercice 7: Dictionnaire, set, tuple, fichier ...

Etant donné un ensemble de points du plan, on veut identifier le couple de points les plus proches au sens de la distance euclidienne (utile dans les transports, aériens ...). Nous représenterons chaque **Point** par un tuple de flottants (x, y) représentant ses coordonnées. L'ensemble des Points sera stocké dans une structure de type **set**.

1. Écrire une fonction **dist(p, q)** qui retourne la distance euclidienne entre deux Points **p** et **q**.

2. Écrire une fonction **plus_proches_voisins(ensPts)** qui prend pour argument l'ensemble (set) des Points et retourne un couple de Points situés à une distance minimale l'un de l'autre. Pour simplifier, il suffit de faire une recherche exhaustive (complexité $O(n^2)$ avec `n = len(ensPts)`).

3. Écrire une fonction **loadCities** permettant de récupérer les coordonnées de villes à partir d'un fichier texte où chaque ligne est de la forme : `nom_ville:abscisse:ordonnée`

Exemple :

`cities.txt`

```
Sousse:10.63699:35.82539
Douz:9.038601:33.456535
Sidi Bouzid:9.48494:35.03823
Sfax:10.766163:34.747847
Gabes:10.097522:33.888077
Touzeur:8.122933:33.918534
```

Cette fonction prend en paramètres le nom du fichier et renvoie un dictionnaire `Dcities` contenant les données des villes récupérées à partir du fichier. Le dictionnaire `Dcities` a le format suivant :

- Chaque `clé` est un *Point* représentant les coordonnées d'une ville;
- Chaque `valeur` est une chaine (str) représentant le nom d'une ville.

4. Écrire une fonction **plus_proches_villes(Dcities)** qui prend pour argument le dictionnaire `Dcities` construit précédemment, et retourne le couple des noms des villes situées à une distance minimale l'une de l'autre. (Important : pensez à utiliser la fonction `plus_proches_voisins`)
5. Écrire la fonction **writeCities(Dcities, nomF)** qui prend en paramètre le dictionnaire `Dcities` et permet d'écrire les coordonnées des villes extraites de `Dcities` dans le fichier `nomF` où chaque ligne est de la forme `nom_ville:abscisse:ordonnée`.

ANNEXE - Quelques fonctions Python

`source.split(motif)` retourne une liste formée par des chaînes de caractères résultantes du découpage de la chaîne source autour de la chaîne motif.

`f=open(nomFichier, 'r' ou 'w' ou 'a')` ouvre le fichier en mode *'r' : lecture*, *'w' : écriture* ou
'a' : ajout.

`f.close()` permet de fermer un fichier.

`f.readlines()` permet de lire et retourner le contenu de toutes les lignes d'un fichier dans une liste

`f.write(ch)` écrit la chaîne `ch` à partir de la position courante du fichier `f`.

On rappelle les opérations utiles sur un dictionnaire `D` :

- `D[k]` retourne la valeur associée à la clef `k` du dictionnaire `D`. Si la clef `k` n'existe pas dans `D`, cette syntaxe génère une erreur.
- `D.keys()` retourne un itérable formé par les clefs du dictionnaire `D`.
- `D.values()` retourne un itérable formé par les valeurs du dictionnaire `D`.

- `D.items()` retourne un itérable formé par des couples `(k,v)` où `k` est une clef du dictionnaire `D` et `v` est la valeur associée.

On rappelle les opérations utiles sur un ensemble `S` :

- `S.add(element)` ajoute l'`element` à l'ensemble `S`. Si l'`element` existe déjà dans `S`, il ne sera pas ajouté à nouveau.
- `S.remove(element)` supprime l'`element` de l'ensemble `S`. Si l'`element` n'existe pas dans `S`, cette syntaxe génère une erreur.
- `S.discard(element)` supprime l'`element` de l'ensemble `S` sans générer d'erreur si l'`element` n'existe pas.
- `S.pop()` supprime et retourne un élément arbitraire de l'ensemble `S`. Génère une erreur si `S` est vide.
- `S.union(T)` retourne un nouvel ensemble contenant tous les éléments de `S` et `T`.
- `S.intersection(T)` retourne un nouvel ensemble contenant tous les éléments communs à `S` et `T`.
- `S.difference(T)` retourne un nouvel ensemble contenant tous les éléments de `S` qui ne sont pas dans `T`.

01

Programmation Orientée Objet

POO en Python

Syntaxe de Base

Partie 1/3

Anis SAIED

IPEIN

2024/25

Plan de la séance

Introduction

Objectif d'étudier la programmation orientée objets (POO) en Python :

- Comprendre comment les types en Python fonctionnent
- Créer de nouveaux types (**classes**)
- Reutiliser le code grâce à l'**héritage** et la **composition**
- Maîtriser les concepts d'**encapsulation** et le **polymorphisme**

Introduction

Exemple : Comprendre les types en Python

```
x = 2
print(type(x)) # <class 'int'>

y = 3.14
print(type(y)) # <class 'float'>

z = "Bonjour"
print(type(z)) # <class 'str'>

a = [1, 2, 3]
print(type(a)) # <class 'list'>
```

Introduction aux objets en Python

En Python, toutes les variables (entiers, chaînes, listes, etc.) sont des instances (objets) d'une classe spécifique.

Exemple :

```
x = 2
print(isinstance(x, int)); type(x)==int  # True

y = "Bonjour"
print(isinstance(y, str))  # True

a = [1, 2, 3]
print(isinstance(a, list))  # True
```

Grâce aux **classes**, nous pouvons créer des types de données plus complexes.

Qu'est-ce qu'une classe en programmation ?

- Une **classe** est une structure de données qui définit un *modèle* pour créer des objets.
- Une classe encapsule des *données* (appelées **attributs**) et des *comportements* (appelés **méthodes**) liés à ces données.
- Elle permet de créer des **objets** réutilisables et d'organiser le code de manière modulaire.

Exemple :

- Une classe "Voiture" peut avoir des attributs tels que "marque" et "année".
- Elle peut également avoir des méthodes pour démarrer, arrêter et accélérer la voiture.

Qu'est-ce qu'un objet en programmation ?

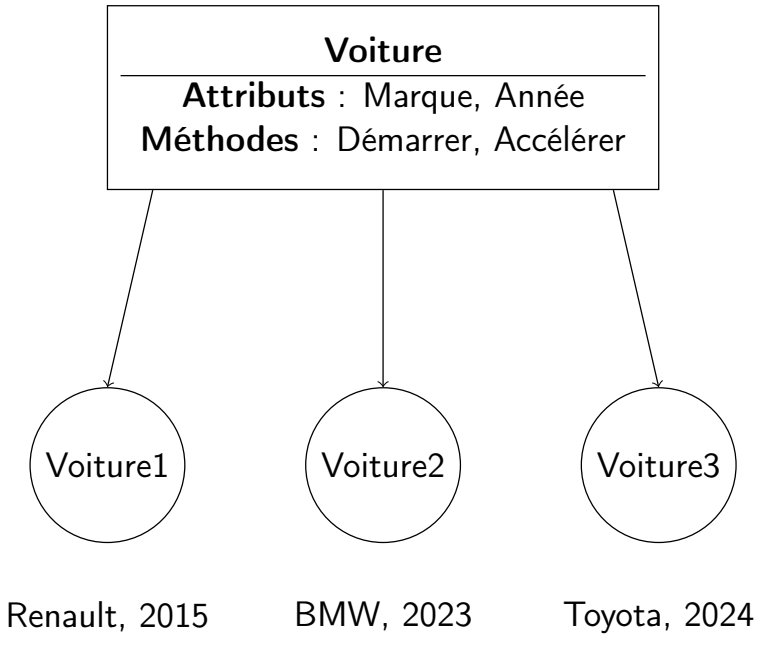
Un **objet** est une instance concrète créée à partir d'une classe.

Exemple :
Supposez que nous ayons une classe "Voiture" avec des attributs comme "marque" et "année".

- Un objet spécifique créé à partir de cette classe pourrait être une "Peugeot 1980".
- Cet objet aurait des valeurs spécifiques pour les attributs "marque" et "année".

Classe & Objtes

Exemples



Comment créer une classe en Python ?

- Pour créer une classe en Python, utilisez le mot-clé **class**.
- Voici la syntaxe de base :

```
1 class Voiture:
2     # Attributs : Marque, Année
3     # Méthodes: Démarrer, Accélérer
```


Comment créer un objet ?

- Pour créer un objet à partir d'une classe, utilisez la syntaxe suivante :

```
1 # Création d'un objet à partir d'une classe
2 nom_objet = NomDeLaClasse()
3 v1 = Voiture()
```

- Remplacez 'NomDeLaClasse' par le nom de la classe à partir de laquelle vous souhaitez créer un objet.

Exemple de création de classe et d'objet

```python
class Voiture:
    def __init__(self):
        self.marque = ""
        self.annee = 0

# Création d'un objet de la classe Voiture
v1 = Voiture()
```

- Dans cet exemple, nous avons créé une **classe** *Voiture* avec deux attributs *marque* et *année* (associés au mot clé **self**).
- Nous avons également créé un **objet** (instance) *v1* de cette classe.
- En Python, **self** est un paramètre spécial utilisé pour faire référence à l'instance actuelle de la classe.

L'utilisation de `self`

En Python, **self**

- fait référence à l'instance actuelle de la classe.
- utilisé pour accéder aux attributs d'instance et aux méthodes de l'objet en cours.
- est le premier paramètre dans la liste des paramètres de méthode.
- n'est pas un mot réservé en Python, mais il est largement utilisé par convention.

Note : Dans une méthode (fonction définie dans une classe)

- une variable associée à self est un **attribut d'instance**.
- une variable non associée à self est une **variable locale**

Attributs

Les attributs peuvent être de deux types principaux :

Les attributs d'instance

- Chaque objet (instance) de la classe possède sa propre copie de ces attributs.
- Définis dans le **constructeur** (__init__) de la classe.

Les attributs de classe

- Partagés par toutes les instances de la classe.
- Définis à l'intérieur de la classe, mais en dehors des méthodes.

Attributs d'Instance

- Spécifiques à chaque objet (instance) de la classe.
- Définis dans le **constructeur** (__init__) de la classe.
- Accessibles via ***self*** à l'intérieur de la classe et via le nom de l'instance en dehors de la classe.

```
1 class Voiture:
2     def __init__(self, marque, annee):
3         self.marque = marque
4         self.annee = annee
5
6 # Exemple d'utilisation
7 v1 = Voiture("Toyota", 2024)
8 v2 = Voiture("Renault", 2015)
9 print(v1.marque) # Affiche Toyota
10 print(v2.marque) # Affiche Renault
```

Attributs de Classe

- Partagés par toutes les instances de la classe.
- Définis à l'intérieur de la classe, mais en dehors des méthodes.
- Accessibles via le nom de la classe depuis n'importe où.

```python
class Voiture:
    compteur = 0 # Attribut de classe
    def __init__(self, marque, annee):
        self.marque = marque
        self.annee = annee
        Voiture.compteur += 1

# Exemple d'utilisation
v1 = Voiture("Toyota", 2024)
v2 = Voiture("Renault", 2015)

#Affiche le nombre total de voitures créés
print(Voiture.compteur)
```


Attributs d'Instance Additionnels

- Parfois, il est nécessaire d'ajouter à certaines instances des attributs spécifiques qui ne sont pas définis dans le constructeur (__init__).
- Exemple : Vous avez une classe "Voiture" avec des attributs de base, mais pour certaines voitures spécifiques, vous devez ajouter des attributs personnalisés tels que "couleur" ou "options".

```python
class Voiture:
    def __init__(self, marque, annee):
        self.marque = marque
        self.annee = annee

# Création d'une voiture standard
voiture1 = Voiture("Toyota", 2020)
voiture2 = Voiture("Honda", 2022)
# Attributs supplémentaires
voiture2.couleur = "Rouge"
voiture2.options = ["Toit ouvrant", "Sièges en cuir"]
```

Méthodes

Syntax

Une **méthode** est une fonction définie dans une classe, dont sa signature commence *self*.

```
class MaClasse:
    def ma_methode(self, parametres):
        # Corps de la méthode
```

- 'self' : premier paramètre de méthode, faisant référence à l'instance actuelle de la classe sur laquelle la méthode est appliquée .
  Exemple : `v1.__init__("Toyota", 2020)`
- Vous pouvez définir des méthodes avec des paramètres comme toute autre fonction.

Méthodes

Exemple

```
class Voiture:
    def __init__(self, marque, annee):
        self.marque = marque
        self.annee = annee
    def afficher(self):
        print(f"Marque:{self.marque}, année :{self.annee}")

# Création d'un objet de la classe Voiture
voiture1 = Voiture("Toyota", 2020)
voiture1.afficher()  # Appel de la méthode
```

Appeler une méthode sur un objet :

- **objet.méthode(paramètres)** → `v1.afficher()`
- **classe.méthode(objet,paramètres)** → `voiture.afficher(v1)`

`__init__()`

Constructeur

La méthode `__init__()` est le **constructeur** d'une classe en Python.

- Automatiquement appelée lors de la création d'un objet à partir de la classe.
- Permet d'initialiser les attributs de l'objet.

Syntaxe :

```
class MaClasse:
    def __init__(self, parametres):
        # Initialiser les attributs par les paramètres
```

Exemple :

```
class Voiture():
    def __init__(self, marque, annee):
        self.marque = marque
        self.annee = annee

# Création et initialisation d'un objet de la classe Voiture
voiture1 = Voiture("Toyota", 2020)
```


`__del__()`

Destructeur

La méthode `__del__()` est le **destructeur** d'une classe en Python.

- Automatiquement appelée lorsque l'objet est détruit ou n'est plus référencé.
- Permet de nettoyer les ressources associées à l'objet.

Syntaxe :

```
class MaClasse:
    def __del__(self):
        # Code de nettoyage ici
```

`__del__()`

Destructeur

Exemple :

```python
class Voiture:
    def __init__(self, marque):
        self.marque = marque
    def __del__(self):
        print(f"La voiture {self} a été détruite.")

# Création d'un objet de la classe Voiture
voiture1 = Voiture("Toyota")

# Destruction de l'objet
del voiture1
# La voiture <__main__.Voiture object at 0x7f085e8d8560>
# a été détruite.
```


Exercices

Exercice 1

1. Créez une classe **Rectangle** avec des attributs **longueur** et **largeur** (attributs d'instance).
2. Ajoutez une méthode **calculer_surface** qui calcule et retourne la surface du rectangle.
3. Créez ensuite deux objets de cette classe :
   - Rectangle 1 : longueur = 5, largeur = 3
   - Rectangle 2 : longueur = 4, largeur = 6
4. Calculez et affichez la surface pour chaque objet.

Solution Exercice 1

```python
class Rectangle:
    def __init__(self,larg=0, long=0):
        self.long = long
        self.larg = larg
    def calcluler_surface(self):
        return self.long * self.larg

r1 = Rectangle(4,6)
r2 = Rectangle(3,5)
r3 = Rectangle()

#meth 1 : nom_classe.nom_methode(obj, parametres)
Rectangle.calcluler_surface(r1) # list.pop(l)

#meth 2 : nom_objet.nom_methode(parametres)
r1.calcluler_surface() # l.pop()
```


Exercices

Exercice 2

1. Créez une classe **Point** avec trois attributs **x**, **y** et **nom** pour représenter des coordonnées et le nom d'un point.
2. Instanciez plusieurs objets de type *Point* avec différentes coordonnées et stockez-les dans une liste.
   Exemple:
   - A = Point(1, 2,"A")
   - B = Point(3, 4,"B")
   - C = Point(0, 0,"C")
   - L = [A, B, C]
3. Créer une fonction **calculer_distance(p1, p2)** qui permet de calculer la distance entre deux points **p1** et **p2**.
4. Calculer et afficher la distance entre chaque paire de points dans la liste.
   Exemple:
   - Distance entre A et B : 2.828
   - Distance entre A et C : 2.236
   - Distance entre B et C : 4.472

Solution Exercice 2

```
1 from math import *
2 from random import *
3 class Point:
4  def __init__(self, x, y):
5      self.x, self.y = x, y
6 # Création des objets Point
7 n = randint(5,10); L=[]
8 for i in range(n):
9     x, y, nom=uniform(100), uniform(100), 'P'+str(i)
10    L.append(Point(x, y, nom))
11 def calculer_distance(p1, p2):
12    return sqrt((p1.x - p2.x)**2+ (p1.y - p2.y)**2)
13 # Affichage des distances
14 ch = "Distance entre {} et {} : {}"
15 for i in range(n-1):
16    for j in range(i+1, n):
17        d = calculer_distance(L[i], L[j])
18        print(ch.format(L[i].nom, L[j].nom, d))
```


C'est qu'on doit retenir

- Définir un nouveau type c'est définir une nouvelle **classe**.
- Une **classe** est un modèle pour créer des objets, définissant à la fois des attributs (données) et des méthodes (comportements).
- On peut créer des **instances** (**objets**) à partir d'une classe.
- Les **attributs** peuvent être de trois types : de classe, d'instance (**self.attribut**), ou associés à l'objet après son instanciation.
- Les **méthodes** sont des fonctions définies dans une classe pour effectuer des opérations sur les objets.
- Le **constructeur (__init__)** est une méthode spéciale pour initialiser les attributs lors de la création d'un objet.
- Le **destructeur (__del__)** est une méthode spéciale pour nettoyer les ressources associées à un objet lors de sa destruction.

POO en Python

Héritage & Polymorphisme

Partie 2/3

Anis SAIED

IPEIN

2024/25

Héritage entre classes

L'**héritage**

- Est une caractéristique fondamentale de la programmation orientée objet (POO).
- Permet de créer de nouvelles classes basées sur des classes existantes.
- Favorise la réutilisation du code et la structuration des classes en hiérarchies.
- En Python, l'héritage s'effectue en spécifiant la **classe parente** entre parenthèses lors de la définition de la **classe enfant**.

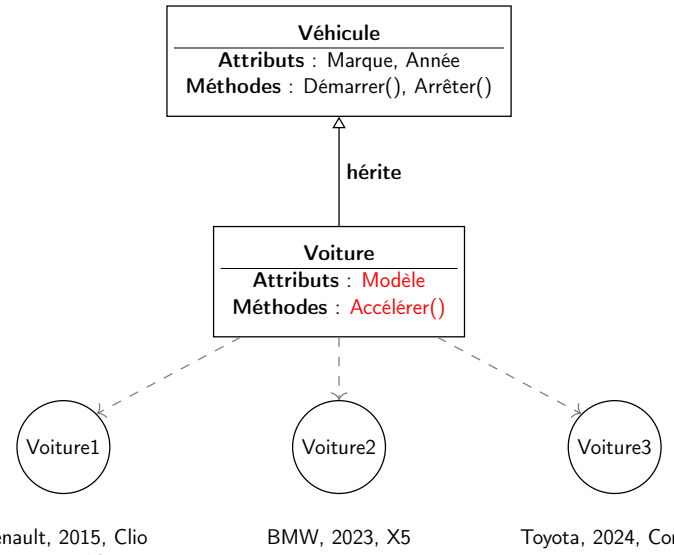
```
1 class Parent:
2     # Attributs et méthodes de la classe parente
3
4 class Enfant(Parent):
5     # Attributs et méthodes spécifiques à la classe enfant
```

Héritage entre Classes

Exemple 1

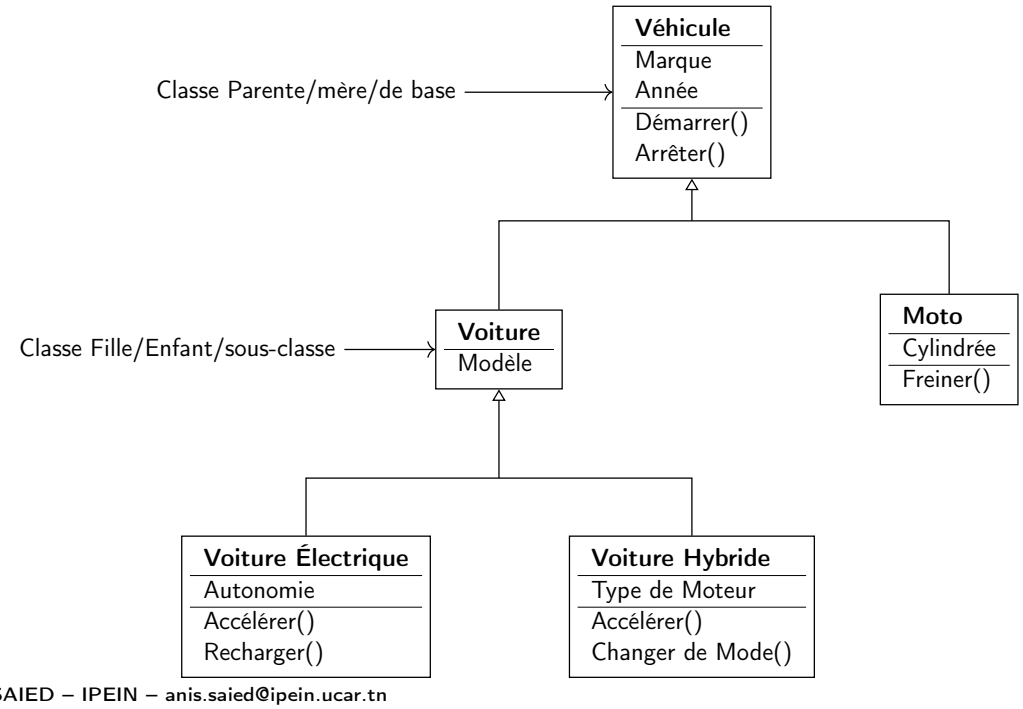
Considérons une **classe parente** appelée `Véhicule` avec les attributs `marque` et `année`.

La **classe enfant** appelée `Voiture` **hérite** la classe `Véhicule` et ajoute un attribut `modèle`.



Héritage entre Classes

Exemple 2: Diagramme de classes



Héritage entre classes

Implémentation

```python
class Véhicule: # classe de base
    def __init__(self, marque, annee):
        self.marque = marque
        self.annee = annee

class Voiture(Véhicule): # classe dérivée
    def __init__(self, marque, annee, modele):
        # Reutilisation de code
        super().__init__(marque, année) # self est passé implicitement
        # Ajouter des attributs spécifiques
        self.modele = modele

# Créez une instance de Voiture et affichez ses attributs
voiture1 = Voiture("Toyota", 2022, "Camry")
print("Marque:", voiture1.marque)
print("Année:", voiture1.annee)
print("Modèle:", voiture1.modele)
```

super() :

- fait référence à la classe parent (ou mère) la plus proche.
- permet d'appeler une méthode de la classe parente depuis une classe enfant pour étendre ou modifier son comportement.

Exercice 1

1. Définissez une classe **Forme** avec une méthode **aire** qui renvoie 0.
2. Définissez la classe **Point**, qui est une sous-classe de Forme. Ajoutez des attributs **x** et **y** pour représenter les coordonnées d'un point et une méthode **afficher** qui affiche les coordonnées.
3. Créez une classe **Cercle**, qui est une sous-classe de Point. Ajoutez un attribut **rayon** pour représenter le rayon du cercle et une méthode **aire** pour calculer l'aire du cercle.
4. Créez une classe **Rectangle**, qui est également une sous-classe de Point. Ajoutez des attributs **largeur** et **hauteur** pour représenter les dimensions du rectangle et une méthode **aire** pour calculer l'aire du rectangle.
5. Créez un objet de chaque classe (Point, Cercle et Rectangle) et utilisez les méthodes `aire` et `afficher` pour afficher leurs aires respectives et leurs coordonnées.

Exercice 1 I

Solution

```python
class Forme:
    def aire(self):
        return 0 # valeur par défaut

class Point(Forme):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def afficher(self):
        print("Point : ({}, {})".format(self.x, self.y))

import math

class Cercle(Point):
    def __init__(self, x, y, rayon):
        super().__init__(x, y)
        self.rayon = rayon

    def aire(self):
        return math.pi * self.rayon ** 2
```


Exercice 1 II

Solution

```python
class Rectangle(Point):
    def __init__(self, x, y, largeur, hauteur):
        super().__init__(x, y)
        self.largeur = largeur
        self.hauteur = hauteur
    def aire(self):
        return self.largeur * self.hauteur

point = Point(2, 3)
cercle = Cercle(0, 0, 5)
rectangle = Rectangle(1, 1, 4, 3)

point.afficher()
print("Aire du point : {:.2f}".format(point.aire()))

cercle.afficher()
print("Aire du cercle : {:.2f}".format(cercle.aire()))

rectangle.afficher()
print("Aire du rectangle : {:.2f}".format(rectangle.aire()))
```

Relations "Est-un" & "A-un"

Définition

En programmation orientée objet (POO), deux concepts clés pour modéliser les relations entre objets sont l'**héritage** et la **composition** :

- L'héritage permet de créer des classes dérivées (enfants) à partir de classes de base (parents), établissant ainsi une relation "**est un**" où la classe dérivée est un type spécialisé de la classe de base.
  - Exemple : Une "Voiture" **est un** "Véhicule".
- La relation "**a un**" est implémentée via la composition, qui consiste à inclure des objets d'une classe dans une autre, où la classe contenant l'objet est responsable de sa création et de sa gestion.
  - Exemple : Une "Voiture" **a un** "Moteur".

Relations "Est-un" & "A-un"

Syntaxe

Héritage (Relation "Est-un") :

```
class ClasseParent:
    # Attributs et méthodes de la classe parente
class ClasseEnfant(ClasseParent):
    # Attributs et méthodes spécifiques à la classe enfant
```

Composition (Relation "A-un") :

```
class ObjetContenu:
    def __init__(self, nom):
        self.nom = nom
    # Autres attributs et méthodes de la classe ObjetContenu

class ClasseContenant:
    def __init__(self):
        self.objet = ObjetContenu("nom_de_l_objet")
    # Autres attributs et méthodes de la classe ClasseContenant
```

Relations "Est-un" & "A-un"

Exemple

```python
class Vehicule:
    def __init__(self, nom):
        self.nom = nom

class Moteur:
    def __init__(self, type):
        self.type = type

class Voiture(Vehicule): # Héritage (est-un)
    def __init__(self, nom, moteur):
        super().__init__(nom)
        self.moteur = moteur # Composition (a-un)

    def afficher_info(self):
        print(self.nom + "a un moteur de type" + self.moteur.type)

moteur_voiture = Moteur("Essence")
ma_voiture = Voiture("Sedan", moteur_voiture)
ma_voiture.afficher_info()
```


Exercice 2

Soient les classes suivantes :

```python
class Véhicule: # classe de base
    def __init__(self, marque, annee):
        self.marque = marque
        self.annee = annee

class Voiture(Véhicule): # classe dérivée
    def __init__(self, marque, annee, modele):
        super().__init__(marque, année)
        self.modele = modele
```

1. Créez une nouvelle classe **MoteurElectrique** avec un attribut **batterie** pour représenter le type de batterie (par exemple, "lithium-ion").
2. Créez une sous-classe de Voiture nommée **VoitureElectrique**, qui inclue un attribut **moteur** de type MoteurElectrique.
3. Créez une instance de la classe *VoitureElectrique*.
   Afficher l'attribut batterie du moteur de la VoitureElectrique.

Exercice 2 I

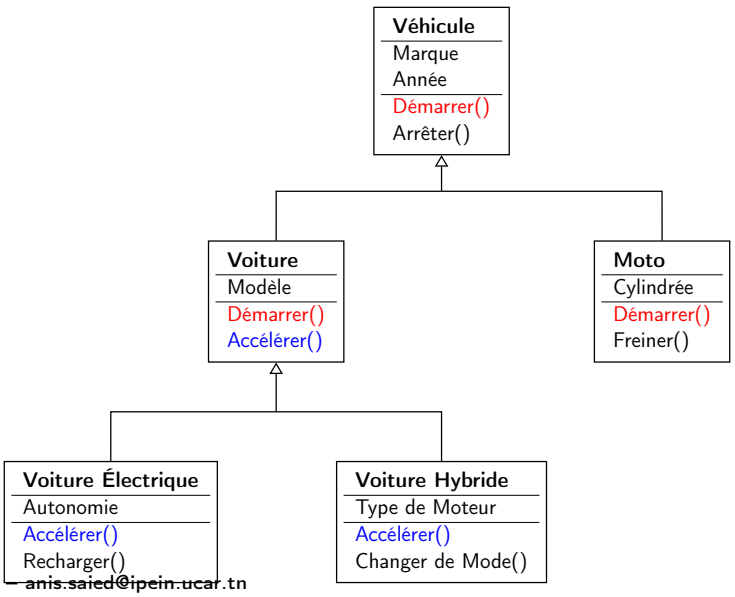
Solution

```python
class MoteurElectrique:
    def __init__(self, batterie):
        self.batterie = batterie

class VoitureElectrique(Voiture): # la relation est-un
    def __init__(self, marque, annee, modele, moteur):
        Voiture. __init__(self, marque, annee, modele)

        # la relation A-un
        assert isinstance(moteur, MoteurElectrique)
        self.moteur = moteur

# Création d'instances
moteurElect = MoteurElectrique("Lithium-ion")
voitureElect = VoitureElectrique("BMW", 2023, "X1", moteurElect)

# La batterie du(.) moteur de(.) la voitureElectrique
print(voitureElect.moteur.batterie)
```


Polymorphisme

Définition

Le **polymorphisme** se manifeste à travers l'implémentation de *méthodes avec le même nom dans différentes classes*.
Exemple :



Polymorphisme

Le polymorphisme permet à des objets de différentes classes de répondre à une même méthode ou opération de manière adaptée à leur propre classe.

- Si vous avez une méthode `démarrer()` dans les classes `Véhicule`, `Voiture`, et `Moto`, chaque classe peut implémenter cette méthode de manière différente.
- Appeler `démarrer()` sur une instance de `Voiture` affichera un message comme "La voiture démarre" tandis que pour une instance de `Moto`, cela pourrait afficher "Le moto démarre".

→ Le **Polymorphisme** permet d'utiliser une **interface uniforme** tout en conservant des **comportements spécifiques à chaque classe**.

Polymorphisme I

Exemple 1

```python
class Vehicule: # Classe parente
    def __init__(self, marque, annee):
        self.marque = marque
        self.annee = annee
    def demarrer(self):
        return "Le véhicule démarre"

class Voiture(Vehicule): # Classe enfant Voiture
    def __init__(self, marque, annee, modele):
        super().__init__(marque, annee)
        self.modele = modele
    def demarrer(self):
        return "La voiture démarre avec une clé"

class Moto(Vehicule): # Classe enfant Moto
    def __init__(self, marque, annee, cylindree):
        super().__init__(marque, annee)
        self.cylindree = cylindree
    def demarrer(self):
        return "La moto démarre en appuyant sur un bouton"

```

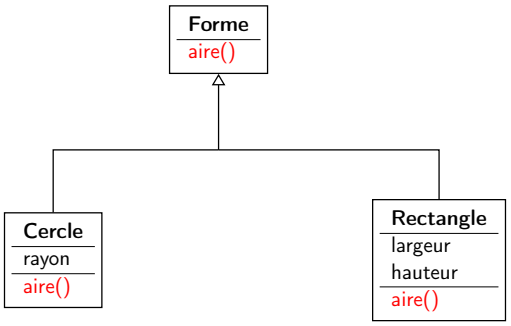
Polymorphisme II

Exemple 1

```
22      def freiner(self):
23          return "La moto freine"
24
25  # Création d'objets et démonstration du polymorphisme
26  # 1. Création d'objets
27  vehicules = [
28      Voiture("Toyota", 2020, "Corolla"),
29      Moto("Yamaha", 2019, "600cc")
30  ]
31
32  # 2. Utilisation polymorphique de la méthode `demarrer()`
33  for vehicule in vehicules:
34      # Affiche le message spécifique de démarrage selon le type de véhicule
35      print(vehicule.demarrer())
36
37
```


Polymorphisme

Exemple 2: Diagramme de classes



- Les classes Cercle et Rectangle héritent de la classe abstraite Forme, ce qui leur permet d'implémenter leur propre méthode aire(), illustrant ainsi le principe du polymorphisme où une méthode peut avoir différentes implémentations selon le type d'objet qui l'appelle.

Polymorphisme I

Exemple 2: Implémentation

```python
from math import pi
class Forme:
    def aire(self):
        pass # Traitement générique

class Cercle(Forme):
    def __init__(self, rayon):
        self.rayon = rayon
    def aire(self):
        return pi * (self.rayon ** 2)

class Rectangle(Forme):
    def __init__(self, largeur, hauteur):
        self.largeur = largeur
        self.hauteur = hauteur
    def aire(self):
        return self.largeur * self.hauteur

formes = [Cercle(5), Rectangle(4, 6)]
for forme in formes:
    print(f"Aire: {forme.aire()}")
```


Polymorphisme

Exercice 1

- Créez une classe `Polynome` avec une méthode `evaluer(x)` qui renvoie la valeur du polynôme pour une valeur donnée `x`.
- Créez ensuite des classes `PolynomeLineaire` (Degré 1, forme $ax + b$) et `PolynomeQuadratique` (Degré 2, forme $ax^2 + bx + c$) qui héritent de `Polynome`.
  Redéfinissez la méthode `evaluer(x)` dans chaque classe pour renvoyer les valeurs spécifiques des polynômes linéaires et quadratiques.

Polymorphisme I

Exercice 1: Solution

```python
# Classe de base pour un polynôme
class Polynome: # classe abstraite
    def evaluer(self, x):
        raise NotImplementedError("La méthode 'evaluer' non implémentée")

# Classe pour un polynôme linéaire
class PolynomeLineaire(Polynome):
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def evaluer(self, x):
        return self.a * x + self.b

# Classe pour un polynôme quadratique
class PolynomeQuadratique(Polynome):
    def __init__(self, a, b, c):
        self.a = a
        self.b = b
        self.c = c

```


Polymorphisme II

Exercice 1: Solution

```python
    def evaluer(self, x):
        return self.a * x**2 + self.b * x + self.c

# Tester le polymorphisme
# 1. Création d'objets
polynomes = [
    PolynomeLineaire(2, 3),        # Représente 2x + 3
    PolynomeQuadratique(1, -2, 1)  # Représente x^2 - 2x + 1
]

# 2. Évaluation des polynômes pour x = 5
for polynome in polynomes:
    print(f"Valeur du polynôme pour x = 5 : {polynome.evaluer(5)}")

```

À retenir

Héritage

- L'héritage est un mécanisme en POO qui permet de créer de nouvelles classes (enfants) basées sur des classes existantes (parents).
- Les classes enfants héritent des attributs et des méthodes des classes parentes.
- Cela favorise la réutilisation du code et la structuration des classes en hiérarchies.

Polymorphisme

- Le polymorphisme est la capacité à traiter des objets de différentes classes de manière uniforme.
- Souvent implémenté via la **surcharge** des méthodes (redéfinir une méthode de la classe mère dans la classe fille).
- Il permet de réutiliser des méthodes ou des opérateurs sur des objets de différentes classes, simplifiant ainsi le code.

POO en Python

Classe Object
Méthodes spéciales et surcharge des opérateurs
Encapsulation

Partie 3/3

Anis SAIED

IPEIN

2024/25

Introduction

Rappel:

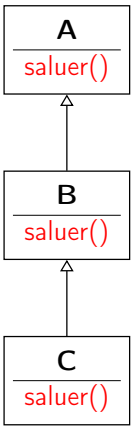
Définir une sous-classe: `class Enfant(Parent)`

`super()` fait référence à la classe parent la plus proche.

Le classe Enfant peut rédéfinir une méthode de la classe Parent.

Exemple:

```
class A:
    def saluer(self):
        print("Bonjour de A")
class B(A):
    def saluer(self):
        print("Bonjour de B")
        super().saluer()
class C(B):
    def saluer(self):
        print("Bonjour de C")
        super().saluer()
c = C()
c.saluer()
```



Ordre de recherche des méthodes

• L'attribut **__mro__ (Method Resolution Order)** : Affiche l'ordre dans lequel Python recherche les méthodes d'une classe lors de l'appel d'une méthode.

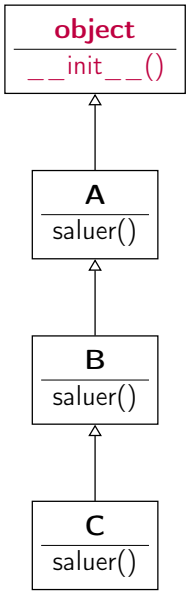
```
c = C()
c.saluer()
print(C.__mro__)
```

Affiche:

```
(<class 'C'>, <class 'B'>, <class 'A'>, <class 'object'>)
```

Python commence par chercher l'attribut/méthode dans la classe C, puis si elle n'existe pas, il le cherche dans la classe B, et ainsi de suite jsuqu'à arriver à la classe de base object.

→ Toute classe en Python hérite directement ou indirectement la classe `object`.



La classe object

• L'attribut **__bases__** retourne un tuple contenant les classes de base directes d'une classe donnée.

```
print(B.__bases__)# (<class 'A'>,)
```

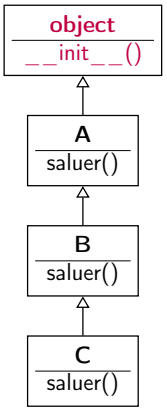
• Pour chaque nouvelle classe créée, Python lui affecte par défaut la classe object comme une classe de base.

• Les déclarations suivantes sont équivalentes:

```
class A:
```
```
class A():
```
```
class A(object):
```

La classe **object** est

- La classe de base de toutes les classes en Python.
- Implicitement héritée par toutes les classes.
- Au sommet de la hiérarchie de l'héritage des classes.
- Définit des méthodes spéciales et attributs prédéfinis.

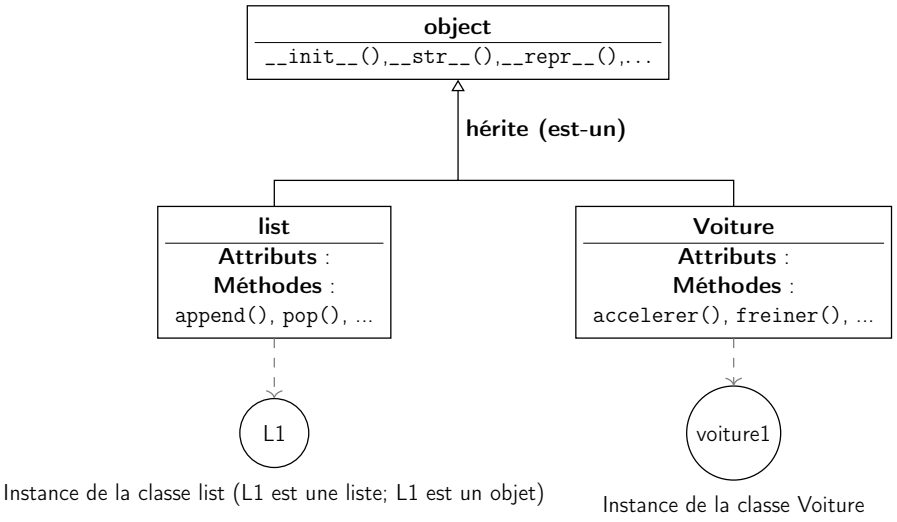


Tout est object

Relation entre la Classe `object` et les autres classes en Python

La classe **object** est la classe de base de toutes les classes en Python.

Les classes prédéfinies et les classes utilisateur héritent directement ou indirectement de la classe `object` → **Tout est Objet**



Méthodes et attributs de la classe `object`

`dir(object)` retourne les méthodes et les attributs prédéfinis de la classe object.

```
>>> print(dir(object))
[..., '__class__', '__delattr__', '__dir__','__doc__', '__eq__',
'__format__', '__ge__', '__getattribute__', '__gt__', '__hash__',
'__init__', '__le__', '__lt__', '__ne__', '__new__', '__repr__',
'__setattr__', '__str__', ...]
```

Exemples:

`__init__(self)` : Appelée lors de la création d'une instance de la classe pour initialiser ses attributs.

`__str__(self)` : Appelée lorsqu'un objet est converti en chaîne de caractères, par exemple avec `str()` ou `print()`, pour fournir une représentation lisible de l'objet.

`__repr__(self)` : Appelée par `repr()` pour obtenir une représentation détaillée et non ambiguë de l'objet, souvent utilisée pour le débogage. Si vous ne définissez pas __str__() dans une classe, Python utilise __repr__() par défaut lorsque vous faites un print() de l'objet.

`__eq__(self, other)` : Appelée pour comparer deux objets via l'opérateur ==, en vérifiant si les instances sont égales en termes de contenu.

Méthodes Spéciales de la Classe object I

Exemples d'utilisation

Les méthodes spéciales (magiques) de la classe `object` peuvent être appelées de manière spécifique pour personnaliser le comportement des objets.

Exemples:

```
1  class Voiture(object):
2
3      # La méthode __init__ initialise l'objet après sa création
4      def __init__(self, marque, modele):
5          print(f"Appel de __init__ pour {self}")
6          self.marque = marque
7          self.modele = modele
8
9      # Définit la représentation en chaîne de l'objet
10     def __str__(self): # toujours retourne une chaine
11         return f"Voiture: {self.marque} {self.modele}"
12
13     # représentation souvent utilisée pour le débogage
14     def __repr__(self): # toujours retourne une chaine
15         return f"Voiture(marque='{self.marque}', modele='{self.modele}')"
```


Méthodes Spéciales de la Classe object II

Exemples d'utilisation

```
16     # La méthode __eq__ compare deux objets pour l'égalité
17     def __eq__(self, other):
18         if not isinstance(other, Voiture):
19             raise TypeError("Comparaison des objets de types différents")
20         return (self.marque==other.marque and self.modele==other.modele)
21     # Vous pouvez également redéfinir d'autres méthodes spéciales
22     # comme __lt__ pour <, __gt__ pour >, etc.
23 # Exemples d'utilisation
24 voiture1 = Voiture("Toyota", "Corolla",)
25 voiture2 = Voiture("Toyota", "Corolla")
26 voiture3 = Voiture("Tesla", "Model 3")
27
28 str(voiture1)  # Appel à __str__
29 print(voiture1)  # Appel à __str__
30
31 print(repr(voiture3))  # Appel à __repr__
32 >>>voiture3  # Appel à __repr__
33
34 # Comparaison des objets
35 print(voiture1 == voiture2)  # Appel à __eq__
36 print(voiture1 < voiture3)  # Appel à __lt__
```

Encapsulation en Python

Principes et Avantages

Encapsulation :

Principe clé de la programmation orientée objet.

Regroupe les données et les méthodes dans une même classe.

Protège les données sensibles contre les modifications non souhaitées.

Avantages :

Protection des Données : Limite l'accès direct aux attributs.

Validation des Données : Permet de contrôler les modifications via des méthodes.

Encapsulation en Python

Exemple sans Encapsulation

```python
class Personne:
    def __init__(self, nom, age):
        # Attribut publiques:  accessibles et modifiables directement.
        self.name = name
        self.age = age

# Utilisation de la classe sans encapsulation
p = Personne("Ali", 30)
print(p.nom)  # Ali
print(p.age)   # 30

p.age = -5     # Modification directe de l'attribut
print(p.age)   # -5 (l'âge est maintenant invalide)
```

Risques d'utilisation des attributs publiques: Les données peuvent devenir invalides ou incohérentes sans validation, car il n'y a pas de contrôle sur les modifications des attributs.

Solution : marquer les attributs sensibles comme privés et controller la nouvelle valeur à chaque tentative de modification.

Encapsulation en Python

Attributs privés

Mise en œuvre en Python :
En Python, l'encapsulation est réalisée en définissant des attributs et des méthodes comme **privés** en les préfixant par un **double underscore** __

Syntaxe Générale :

Attributs privés : `self.__attribut`

Tout accès à un attribut privé doit se faire via les méthodes (publiques) suivantes:

```
def get_attribut(self):
def set_attribut(self, value):
```

On peut définir une propriété qui simplifie l'accès à l'attribut privé:
`nom_propriete = property(get_attribut,set_attribut)`

Exemple d'tilisation :
Soit une classe Etudiant a un attribut note déclaré comme attribut privé.
Soit e une instance de la classe Etudiant.

Déclarer un attributs privé : `self.__note = 0`

Accès en lecture: `e.get_note()`

Modification: `e.set_note(15)`

Accès en lecture via la propriété note: `e.note`

Modification via la propriété note: `e.note=15`

Encapsulation en Python I

Attributs privés: Exemple

```python
class Personne:
    def __init__(self, nom, age):
        self.nom = nom # Attribut publique
        self.__age = age # Attribut privé : protégé contre les accès directs.

    # Méthodes d'Accès: contrôlent l'accès et la modification des données.
    def get_age(self):
        return self.__age
    def set_age(self, age):
        if age > 0:  self.__age = age
        else:  print("Age doit être positif.")
    # Définir une propriété age
    age = property(get_age, set_age)

# Utilisation de la classe avec encapsulation
p = Personne("Ali", 30)
print(p.nom) # Ali
print(p.__age) # Erreur
print(p.get_age()) # 30
p.set_age(-5);  p.age = -10 # Affiche "Age doit être positif."
print(p.age) # 30 (l'âge n'a pas été modifié)
```

TD : Programmation Orientée Objet

Classe, Instances, Atributs, Méthodes, Encapsulation
Héritage, Composition, Polymorphisme
Object, Méthodes spéciales

1 Classe, Objets, instanciation, Attributs et Méthodes

Exercice 1

1. Créer une *classe* Python nommée **Point**, définie par deux attributs d'instance, x et y, représentant les coordonnées d'un point.
2. Créez deux *instances* **p1** et **p2** de la classe **Point** avec les mêmes coordonnées, $x = 1$ et $y = 2$.
3. Essayez les opérations suivantes :
   - `print(isinstance(p1, Point)); print(type(p2) == Point)` : Vérifiez si **p1** et **p2** sont des instances de la classe **Point**.
   - Testez l'égalité des objets **p1** et **p2**.
   - `p1 = p2; p1.x = 7.0; print(p2.x)`
4. Créer une classe Python nommée **Rectangle**, définie par :
   - **coinSupG** : un objet de type **Point** représentant la position du coin supérieur gauche du rectangle.
   - **largeur** : la largeur du rectangle.
   - **hauteur** : la hauteur du rectangle.
5. Créer une instance nommée **boite** de la classe **Rectangle**.
6. Définir une fonction Python **trouver_centre** qui prend en argument un objet de type **Rectangle** et renvoie un objet de type **Point** représentant les coordonnées du centre du rectangle.
7. Appelez cette fonction en utilisant comme argument l'objet **boite** défini précédemment.
8. Affichez les coordonnées du centre trouvé.

Exercice 2

Créer et manipuler une classe en Python pour gérer une liste de tâches :

1. **Définir une classe** `Tache` avec les attributs suivants :
   - `titre` : le titre de la tâche.
   - `description` : une brève description de la tâche.
   - `complete` : un booléen indiquant si la tâche est complétée ou non.
2. **Méthodes de la classe** :
   - `__init__` : Méthode d'initialisation qui permet de définir le `titre`, la `description` et l'état `complete` lors de la création de l'objet.
   - `marquer_complete` : Méthode qui marque la tâche comme complétée.

- `marquer_incomplete` : Méthode qui marque la tâche comme non complétée.
- `afficher_details` : Méthode qui affiche les détails de la tâche (titre, description, état).

3. **Définir une classe** `ListeDeTaches` qui contiendra une liste d'objets `Tache`. Cette classe doit avoir les méthodes suivantes :
   - `ajouter_tache` : Méthode pour ajouter une nouvelle tâche à la liste.
   - `supprimer_tache` : Méthode pour supprimer une tâche par son titre.
   - `afficher_taches` : Méthode pour afficher toutes les tâches de la liste.
4. **Tester les classes** en créant des instances de `Tache` et de `ListeDeTaches`, et en utilisant les méthodes définies pour gérer la liste des tâches.

2 Héritage et polymorphisme

Exercice 3

Vous allez créer une hiérarchie de classes pour gérer une bibliothèque en utilisant les concepts d'héritage et de polymorphisme.

1. Créez une classe de base appelée `Livre` avec les attributs suivants :
   - `titre` : le titre du livre
   - `auteur` : l'auteur du livre
   - `année_de_publication` : l'année de publication du livre

   Ajoutez également un attribut de classe pour suivre le nombre total de livres :
   - `nombre_total_livres` : nombre total de livres dans la bibliothèque

   Méthodes à inclure :
   - `__init__(self, titre, auteur, année_de_publication)` : constructeur pour initialiser les attributs d'instance.
   - `description(self)` : méthode pour retourner une description générique du livre.
   - `nombre_total_livres(cls)` : méthode de classe pour retourner le nombre total de livres.
2. Créez deux sous-classes :
   - **LivrePapier** (hérite de `Livre`) : Ajoutez un attribut supplémentaire pour le nombre de pages :
     - `nombre_de_pages` : nombre total de pages du livre papier.

     Méthodes supplémentaires :
     - `description(self)` : redéfinissez cette méthode pour fournir une description spécifique des livres papier.
   - **LivreNumerique** (hérite de `Livre`) : Ajoutez un attribut supplémentaire pour la taille du fichier :
     - `taille_fichier_mo` : taille du fichier en mégaoctets pour les livres numériques.

     Méthodes supplémentaires :

- `description(self)` : redéfinissez cette méthode pour fournir une description spécifique des livres numériques.

3. Créez une classe `Bibliotheque` pour gérer une collection de livres :
   - **Attribut :**
     - `livres` : une liste pour stocker les objets `Livre`, `LivrePapier`, et `LivreNumerique`.
   - **Méthodes :**
     - `ajouter_livre(self, livre)` : ajoute un livre à la bibliothèque.
     - `supprimer_livre(self, titre)` : supprime un livre de la bibliothèque en fonction de son titre.
     - `afficher_livres(self)` : affiche la liste de tous les livres dans la bibliothèque en utilisant la méthode `description()`.
4. Créez quelques instances de livres (à la fois `LivrePapier` et `LivreNumerique`) avec des titres, auteurs, années de publication, et les attributs spécifiques aux sous-classes.
5. Ajoutez ces livres à une instance de la classe `Bibliotheque`.
6. Affichez la liste des livres disponibles dans la bibliothèque.

3 Classe Object et surcharge des méthodes spéciales

Exercice 4

L'objectif de cet exercice est la manipulation des polynômes creux à une seule variable. Un polynôme creux est un polynôme dont certains coefficients sont nuls. Un polynôme est construit à partir de monômes. Un monôme est une expression de la forme ax^n où a ($a \neq 0$) est le coefficient du monôme et $n (n > 0)$ son degré.
Un monôme est représenté par un dictionnaire à un élément dont la clé est le degré n et la valeur est le coefficient a.
Exemple : Le monôme $8x^2$ est représenté par le dictionnaire $\{2 : 8\}$.
Un polynôme creux est alors défini comme une association de monômes de degrés différents.
Exemples :
Le polynôme $-x^4 + 8x^2 - 5x$ est représenté par le dictionnaire $\{2 : 8, 1 : -5, 4 : -1\}$.
Le dictionnaire $\{0 : 1, 5 : 1, 8 : 1\}$ représente le polynôme $x^8 + x^5 + 1$.
On se propose de construire la classe PolynomeCreux à coefficients réels dont le squelette (à compléter) est défini par :

```
class PlynomeCreux :
    """Manipulation des polynômes creux à une seule variable """
    def __init__(self):
        self.data = {} #initialisation à un polynôme nul

    def ajout_monome(self, monome ={}):
        """
        Cette méthode ajoute un monôme saisi au clavier si le paramètre
        monome est nul ou ajoute le monôme nommé sinon
        """
        if len(monome) == 0 :
```

```
            #réponse à la question 1
        else : #si monome est non vide
            degre = list(monome.keys())[0] # extraction du degré
            coeff = list(monome.values())[0] # extraction du coefficient
            try :
                assert degre >= 0
                assert type(degre) == int
                assert type(coeff) == int or type(coeff) == float
                assert len(monome) == 1
                self.data.update(monome) # self.data[degre] = coeff
            except :
                print ("Erreur d'ajout monome")

    def degree(self):           #réponse à la question 2
    def __call__(self, x0):    #réponse à la question 3
    def __add__(self, other):  #other est un polynôme creux
    #réponse à la question 4
    def __mul__(self, other):  #other est un polynôme creux
    #réponse à la question 5
    def __str__(self):      #réponse à la question 6
    def primitive(self):    #réponse à la question 7
```

Travail demandé :

1. Compléter le script de la méthode **ajout_monome**. On rappelle que cette méthode ajoute un monôme saisi au clavier (en faisant les contrôles nécessaires) si le paramètre monome est nul ou ajoute le monôme nommé *monome* sinon.

2. Écrire le script de la méthode, nommée **degree**, qui retourne le degré du polynôme.

3. Écrire le script de la méthode, nommée **__call__** qui retourne la valeur du polynôme pour un réel $x0$ donné.

4. Écrire le script de la méthode, nommée **__add__** , qui retourne le polynôme somme de deux polynômes.
   Remarque : aucun monôme nul ne doit apparaître dans le polynôme résultat.

5. Écrire le script de la méthode, nommée **__mul__** , qui retourne le polynôme produit de deux polynômes.
   Remarque : aucun monôme nul ne doit apparaître dans le polynôme résultat.

6. Écrire le script de la méthode, nommée **__str__**, qui retourne la chaîne représentant l'expression du polynôme ordonné par ordre décroissant.
   Pour le polynôme représenté par $\{4 : 4, 0 : 4, 12 : 6, 9 : 1, 7 : -1\}$, la chaîne retournée est : "6*x**12 + x**9 - x**7 + 4*x**4 + 4"

7. Écrire le script de la méthode, nommée **primitive**, qui retourne le polynôme représentant la primitive. On suppose que la constante d'intégration est nulle.

8. On définit, l'intégrale d'un polynôme creux P en x entre les bornes a et b, par: $S = \int_a^b P dx$.
   Écrire le script de la fonction, nommée **integrale**, permettant de retourner la valeur de S à partir d'un polynôme P, de type PolynomeCreux, et des bornes d'intégration a et b réels.

Exercice 5: Gestion des Fonctions Mathématiques

Créez un système en Python pour représenter et manipuler des fonctions mathématiques. Vous devrez définir plusieurs classes pour représenter les fonctions et effectuer des opérations sur celles-ci.

1. **Classe** `Fonction` : Cette classe représente une fonction mathématique de base. Elle doit inclure les attributs suivants :

   - `nom` : Le nom de la fonction.
   - `fonction` : Une fonction lambda ou une fonction définie qui représente l'expression mathématique.

   La classe doit inclure :

   - Un constructeur pour initialiser les attributs.
   - Une méthode `__str__()` pour retourner une représentation en chaîne de caractères de la fonction.
   - Une méthode `evaluer(x)` qui retourne la valeur de la fonction pour un argument donné `x`.

2. **Classe** `FonctionComposee` : Cette classe représente la composition de deux fonctions. Elle doit hériter de la classe `Fonction` et inclure :

   - Les attributs `fonction1` et `fonction2` : Deux instances de la classe `Fonction`.

   La classe doit inclure :

   - Un constructeur pour initialiser les attributs et créer une fonction composée.
   - Une méthode `__str__()` pour retourner une représentation en chaîne de caractères de la fonction composée.
   - Une méthode `evaluer(x)` qui évalue la fonction composée pour un argument donné `x`.

3. **Classe** `FonctionInverse` : Cette classe représente l'inverse d'une fonction. Elle doit hériter de la classe `Fonction` et inclure :

   - Une méthode `calculer_inverse(x)` pour calculer l'inverse de la fonction pour un argument `x` donné .
   - Un constructeur pour initialiser les attributs et créer une fonction inverse.
   - Une méthode `__str__()` pour retourner une représentation en chaîne de caractères de la fonction inverse.
   - Une méthode `evaluer(x)` qui évalue la fonction inverse pour un argument donné `x`.

Exemple d'utilisation :

```
# Définir des fonctions
f1 = Fonction("f(x) = x^2", lambda x: x**2)
f2 = Fonction("g(x) = x + 1", lambda x: x + 1)

# Composer des fonctions
f_composée = FonctionComposee(f1, f2)

# Calculer et afficher des valeurs
print(f_composée)  # Affiche la composition de f et g
print(f_composée.evaluer(2))  # Affiche la valeur de (x^2 + 1) pour x = 2

# Calculer l'inverse d'une fonction
f_inverse = FonctionInverse(f1, lambda y: y**0.5)
print(f_inverse)  # Affiche l'inverse de f
print(f_inverse.evaluer(4))  # Affiche la valeur inverse de x^2 pour x = 4
```

4 Encapsulation

Exercice 6

Concevoir une hiérarchie de classes en Python pour modéliser une réaction chimique, en utilisant les concepts de réactifs et de produits.

1. **Définir une classe** `Reactif` avec les attributs suivants :
   - `nom` : le nom du réactif (par exemple, "H_2O").
   - `quantite` : la quantité du réactif (exprimée en moles).
2. **Définir une classe** `Produit` héritant de la classe `Reactif` avec un attribut supplémentaire :
   - `rendement` : le rendement du produit (exprimé en pourcentage, 0.05 pour 5%), indiquant l'efficacité de la conversion des réactifs en produits.
3. **Définir une classe** `ReactionChimique` avec les attributs suivants :
   - `reactifs` : une liste d'objets `Reactif` impliqués dans la réaction.
   - `produits` : une liste d'objets `Produit` résultants de la réaction.
4. **Encapsulation des attributs** :
   - Les attributs `reactifs` et `produits` doivent être privés. Ils ne doivent pas être accessibles directement depuis l'extérieur de la classe. Fournir des méthodes publiques pour obtenir et modifier ces attributs.
   - Les modifications des listes de réactifs et de produits doivent être validées pour garantir qu'elles contiennent des objets valides de type `Reactif` ou `Produit`.
5. **Méthodes de la classe** `ReactionChimique` :
   - `get_reactifs` : Retourne la liste des réactifs.

- `set_reactifs` : Permet de définir la liste des réactifs, uniquement si les objets sont valides.
- `get_produits` : Retourne la liste des produits.
- `set_produits` : Permet de définir la liste des produits, uniquement si les objets sont valides.
- `verifier_equilibre` : Vérifie si le nombre total de moles des réactifs est égal au nombre total de moles des produits.
- `__str__` : Retourne une représentation textuelle de la réaction chimique pour faciliter l'affichage des informations. Cette représentation doit inclure :
  - La liste des réactifs avec leur nom et quantité.
  - La liste des produits avec leur nom et quantité.
  - L'état de l'équilibrage de la réaction (équilibrée ou non équilibrée).

6. **Tester les classes** :
   - Créez des instances des classes `Reactif` et `Produit`.
   - Créez une instance de la classe `ReactionChimique` en utilisant les instances des réactifs et des produits.
   - Utilisez les méthodes de la classe `ReactionChimique` pour vérifier leur fonctionnalité.

Exercice 7

Soit les classes A et B suivantes. Quelle sera la sortie du programme suivant ?

```
class A:
   i = 1
   k = 3
   def __init__(self):
     self.i = 2
   def f(self):
     return self.g()
   def g(self):
     return 'A'

class B(A):
   def g(self):
     return 'B'
```

```
a = A(); b = B() ;
type(a) ; help(a) ;
print(a.g(), b.g()) # affiche : ?
print(a.f(), b.f()) # affiche : ?
print(A.i) # affiche ?
print(A().i) # affiche ?
a.j = 0 #Ajouter un attributobjet a
print(a.j) # affiche ?
print(b.j) # affiche ?
b = a ; print(b.j) ;
b.j = -1 ; print(a.j) # affiche ?
```

02

Structures de données avancées

Les structures de données avancées
Piles & Files

Anis SAIED

IPEIN

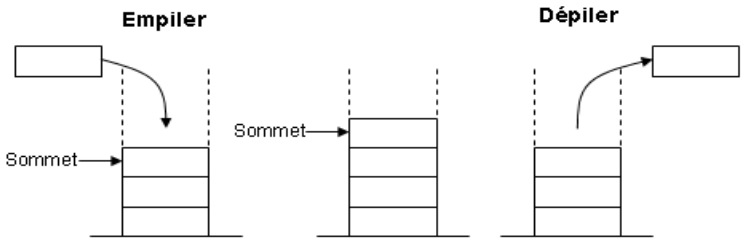
2024/25

Les Piles
Définition

Une Pile (**stack**) est une structure de données qui met en oeuvre le principe : **dernier entré, premier sorti** ou **LIFO** (**L**ast - **I**n - **F**irst - **O**ut).

Cette structure permet de "stocker" provisoirement des éléments et de récupérer les données les plus récentes.

Les seules opérations dont on a besoin sont EMPILER (INSERER ou **push**en anglais) et DEPILER (SUPPRIMER ou **pop** en anglais).



Le sommet de la pile est le dernier élément empilé. C'est celui qui sera le premier dépilé.

Les Piles

Exemples de cas d'utilisations

- Gestion de l'historique dans un navigateur web:
  Les deux boutons « page suivante » et « page précédente» de votre navigateur, chacun utilise une pile.
  Les différentes pages visitées sont stockées dans une pile. Lorsqu'on clique sur le bouton de retour en arrière, on dépile la pile associée au bouton précédent.
- Exécution d'une fonction récursive. Chaque appel récursif est stocké dans la pile d'exécution jusqu'à atteindre un cas terminal, puis les fonctions sont dépilées.

Les Piles I

Exemples: Pile d'exécution d'une fonction récursive

Une pile d'exécution est une structure de données qui stocke les appels de fonctions en cours d'exécution.

Chaque appel de fonction crée un nouveau cadre (frame) dans la pile, contenant les **variables locales** et l'**adresse de retour**.

Les cadres sont empilés lors des appels et dépilés lorsque les fonctions se terminent.

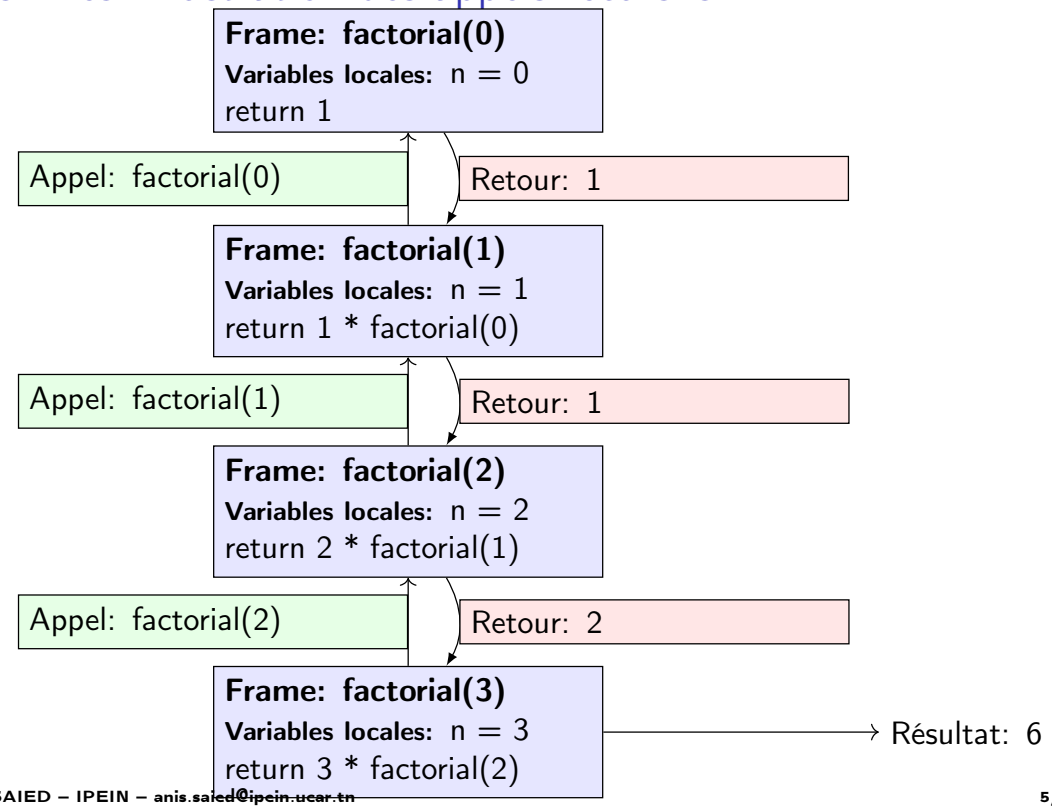
Le premier appel est empilé au bas de la pile, et chaque appel successif est ajouté au-dessus.

La pile d'exécution fonctionne de bas en haut, où le fond de la pile représente le premier appel.

Une fois le cas terminal est atteint, les fonctions sont dépilées en sens inverse, avec les valeurs de retour propagées du haut vers le bas.

Exemple : voir schéma d'exécution de `factorial(3)`

Les Piles: Illustration des appels récursifs



Les Piles

Mise en œuvre d'une pile avec une liste

Il y a beaucoup de façons de concevoir une pile : listes, tableaux,.... La plus simple consiste à utiliser un conteneur de type **list**.

Programmer une classe nommée `Pile` contenant les méthodes suivantes :

- `__init__(self)` : initialise une pile vide non bornée (dont la taille maximale n'est pas définie).
- `pile_vide(self)` : retourne `True` si la pile p est vide et `False` sinon.
- `sommet(self)` : renvoie l'élément du sommet de la pile p, sans le supprimer.
- `taille(self)` : renvoie la taille de la pile p.
- `empiler(self,x)` : ajoute l'élément x au sommet de la pile p.
- `depiler(self)` : supprime le sommet de la pile p et le retourne. Que se passe-t-il quand on essaye de dépiler une pile vide ?

Les Piles

Exercices d'application

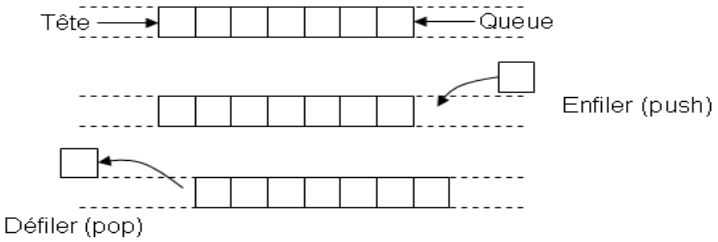
Exercice 1: conversion de base 10 en base 2
Écrire une fonction `conversion(n)` qui retourne la représentation en base 2 du nombre n représenté en base 10 à l'aide de piles.
Indication : La fonction python `bin(10)` retourne `'0b1010'`.

Exercice 2: Permutations circulaires
Écrire une fonction `permut_circ(p,n)` qui reçoit en argument une pile `p` et un entier `n` et effectue sur la pile `n` permutations circulaires successives.
Dans cet exercice, c'est la pile `p` elle-même qui sera modifiée.
Par exemple avec $n = 2$:
`p = [103, 2, 98, 11, 7]` $\longrightarrow$ `p = [11, 7, 103, 2, 98]`

Les Files

Définition

Une File (**queue**) est une structure de données qui met en œuvre le principe : **premier entré, premier sorti** ou **FIFO** (**F**irst - **I**n - **F**irst - **O**ut). La première valeur de la file est la tête ou sommet de la file et la dernière est la queue de la file.
Les opérations principales sont ENFILER (**enqueue/push** en anglais) et DEFILER (**dequeue/pop** en anglais).



Lorsque nous enfilons un élément, il est ajouté à la fin de la file.
Quand nous défilons un élément, le premier élément est supprimé de la file.
Ce processus continue, le premier élément ajouté étant toujours le premier à être retiré.

Les Files

Exemples de cas d'utilisation

En Informatique, les files sont utilisées pour :

- Gestion de la file d'attente d'impressions : Les documents sont enfilés dans une file et imprimés dans l'ordre où ils ont été ajoutés.
- Gestion des processus dans un système d'exploitation: Les tâches sont enfilées en attente de traitement par le processeur, chacune traitée dans l'ordre d'arrivée.
- Gestion des appels téléphoniques dans un centre d'appel: Les appels sont enfilés et traités dans l'ordre où ils ont été reçus.

Les Files

Mise en œuvre d'une file avec une liste et une taille maximale globale

Le type `list` intègre déjà toutes les méthodes pour réaliser cette structure de données.

Programmer une classe nommée `File` contenant les méthodes suivantes :

- TAILLE_MAX = 40 : attribut de classe définissant la taille maximale par défaut pour la file.
- __init__(self) : initialise une file vide bornée, en utilisant la taille maximale définie par l'attribut `TAILLE_MAX`.
- file_vide(self) : retourne `True` si la file f est vide et `False` sinon.
- enfiler(self, x) : ajoute l'élément x à la fin de la file f si celle-ci n'est pas pleine.
- defiler(self) : retire et retourne l'élément en début de file f.
- sommet(self) : renvoie l'élément au début de la file f sans le retirer.
- taille(self) : renvoie la taille de la file f.
- est_pleine(self) : retourne `True` si la file a atteint sa capacité maximale définie par `TAILLE_MAX`.

Exercices d'application

Exercice 3: Gestion d'une file de priorité
Écrire une fonction `gestion_priorite(f, obj)` qui insère un objet `obj` dans une file `f` en fonction de son attribut `priorite` (un nombre entier). Les objets ayant une priorité élevée doivent être en début de file pour être traités en premier. La file doit respecter la limite de taille définie par `TAILLE_MAX`.
Indication : Assurez-vous que la file n'est pas pleine avant d'ajouter l'objet, et placez chaque nouvel objet de manière à ce que les priorités les plus hautes soient toujours en début de file.

Exercice 4: Gestion d'une file circulaire
Écrire une fonction `rotation_file(f, n)` qui effectue une rotation circulaire de la file `f` par n positions. La rotation circulaire déplace les éléments en début de file vers la fin, et ainsi de suite. Ce processus doit modifier la file elle-même.

Solution - Exercice 3 : Gestion d'une file de priorité

```python
from file import *

def gestion_priorite(f, obj):
    """Insère un élément dans la file en fonction de sa priorité."""
    if est_plein(f):
        print("La file est pleine!")
        return
    # Insertion en fonction de la priorité
    index = 0
    while index < taille(f) and f[index].priorite >= obj.priorite:
        index += 1
    f.insert(index, obj)

# Exemple d'utilisation
class Tache:
    def __init__(self, nom, priorite):
        self.nom = nom
        self.priorite = priorite

tacheA = Tache("Tâche A", 3); tacheB = Tache("Tâche B", 1)
tacheC = Tache("Tâche C", 5)
file = creer_file()
gestion_priorite(file,tacheA); gestion_priorite(file, tacheB)
gestion_priorite(file, tacheC)
```

Solution - Exercice 4 : Gestion d'une file circulaire

```python
from file import *

def rotation_file(f, n):
    """Effectue une rotation circulaire de n positions sur la file."""
    if file_vide(f):
        print("La file est vide!")
        return

    n = n % len(f) # Gérer les rotations supérieures à la taille de la file
    for i in range(n):
        enfiler(f, defiler(f))

# Exemple d'utilisation
f = creer_file()
for i in range(5): # Remplir la file avec des objets
    f.enfiler(Tache("A", i))
rotation_file(file, 2)
for i in range(taille(f)): # Affiche la file après rotation
    elt = defiler(f); print(elt); enfiler(f, elt)
```

Anis SAIED – IPEIN – anis.saied@ipein.ucar.tn

TD : Structures de données avancées

Les Piles (LIFO : Last In, First Out)

Exercice 1: Mise en oeuvre d'une pile avec une liste

Il y a beaucoup de façons de concevoir une pile : listes, tableaux,.... La plus simple consiste à utiliser un conteneur de type **list**. Programmer le module `pile.py` contenant les fonctions suivantes :

1. `creer_pile()` : crée et retourne une pile vide non bornée (dont la taille maximale n'est pas définie, donc on peut y empiler autant d'éléments que l'on veut, dans la limite de la mémoire de l'ordinateur).
2. `pile_vide(p)` : retourne `True` si la pile p est vide et `False` sinon.
3. `sommet(p)` : renvoie l'élément du sommet de la pile p, sans le supprimer.
4. `taille(p)` : renvoie la taille de la pile p.
5. `empiler(p,x)` : ajoute l'élément x au sommet de la pile p.
6. `depiler(p)` : supprime le sommet de la pile p et le retourne.
   Que se passe-t-il quand on essaye de dépiler une pile vide ?

Exercice 2: conversion de base 10 en base 2

Écrire une fonction `conversion(n)` qui retourne la représentation en base 2 du nombre n représenté en base 10 à l'aide de piles.
Indication : La fonction python `bin(10)` retourne `'0b1010'`.

Exercice 3: Évaluation d'une expression arithmétique

Une des tâches fondamentales d'un interpréteur comme celui de Python (ou d'un compilateur dans un autre langage de programmation) est d'attribuer une valeur à une expression arithmétique. Une expression arithmétique est formé des 10 chiffres «$0, 1, \ldots, 9$», des signes «$+, -, \times, /$» et des parenthèses ouvrante «(» et fermante «)».
Par exemple : $(\ 3 + (4 \times 5)\ )/2$ est une expression arithmétique dont la valeur est 11,5.

En général, l'utilisateur tape cette expression à l'aide de son clavier, ce qui génère une chaîne de caractères
ch = "$(3 + (4 * 5))/2$". Le problème est de traduire cette chaîne en un nombre, tout en gérant les priorités des opérations et les parenthèses : l'interpréteur doit se livrer pour cela à une **analyse syntaxique**.

Q 1. **Vérification des parenthèses dans une expression**
Écrire une fonction `verif_parenthese(expression)` qui reçoit une expression mathématique sous la forme d'une chaîne de caractères et renvoie **False** si l'expression n'est pas bien parenthésée et la liste des couples d'indices des paires de parenthèses dans le cas contraire. Par exemple pour '((2+5)*4(+1*3' la fonction renverra **False** et pour '((2+5)*4+1)*3' elle renverra **[(0,10),(1,5)]**.

Indication : empiler les indices des parenthèses ouvrantes dans une pile.

Une même expression arithmétique a au moins trois représentations naturelles :

- La représentation **infixée** : C'est celle qu'on utilise tout le temps. Exemple : $(3 + 2) \times 5$
- La représentation **préfixée** : où on place tous les opérateurs avant les opérandes.
  Exemples : $2 + 5 \longrightarrow +25$; $(3 + 2) \times 5 \longrightarrow \times + 325$; $3 + (2 \times 5) \longrightarrow +3 \times 25$
- On peut également placer les opérateurs après les opérandes : c'est la représentation **postfixée**. Dans ce cas $2 + 5$ s'écrirait : $25+$ et si on partait de $(3 + 2) \times 5$ on aurait : $32 + 5\times$. Aucun besoin de parenthèses là non plus.

Q 2. Donner les deux autres formes des expressions suivantes et indiquer à chaque fois leurs valeurs :
a) $2 + (3 - (5 + 1) \times 2)$; b) $2345 + -6 \times /$

L'évaluation d'une expression **postfixée** peut se faire de façon très simple, en une seule lecture de la gauche vers la droite à l'aide d'une pile qui stocke les résultats intermédiaires.

Sur l'exemple suivant, le caractère « ▶ » a été rajouté pour indiquer la limite de la partie déjà traitée de l'expression.
Partons de la chaîne postfixée : $ch =$ "$234 + -5 \times$ "

▶ 2 3 4 + - 5 ×	pile P : []
2 ▶ 3 4 + - 5 ×	pile P : [2]
2 3 ▶ 4 + - 5 ×	pile P : [2, 3]
2 3 4 ▶ + - 5 ×	pile P : [2, 3, 4]
2 3 4 + ▶ - 5 ×	pile P : [2, 7]
2 3 4 + - ▶ 5 ×	pile P : [−5]
2 3 4 + - 5 ▶ ×	pile P : [−5, 5]
2 3 4 + - 5 × ▶	pile P : [−25] $\longrightarrow$ **Résultat** $= -25$

Q 3. Écrire une fonction `calc_exp_arith(ch)` qui prend en paramètre une expression arithmétique (chaîne de caractères en notation postfixée) et qui renvoie sa valeur numérique.
Indication : L'expression contiendra uniquement des entiers positifs (pas de nombre négatifs, ni de nombres à virgule) et des signes des opérations séparés par des espaces.

Exercice 4: Permutations circulaires

Écrire une fonction `permut_circ(p,n)` qui reçoit en argument une pile `p` et un entier `n` et effectue sur la pile `n` permutations circulaires successives. Dans cet exercice, c'est la pile `p` elle-même qui sera modifiée.
Par exemple avec $n = 2$:
`p = [103, 2, 98, 11, 7]` $\longrightarrow$ `p = [11, 7, 103, 2, 98]`

Exercice 5: Somme des éléments d'une pile

Écrire une fonction récursive `somme(p)` qui prend en paramètre une pile `p`, contenant éventuellement des sous-piles, de profondeur quelconques, dont tous les composants élémentaires sont des nombres, et calcule la somme de tous ses éléments.
Par exemple : `somme([ [[1,2], [3,4,5]], [6], [7,8], [9,[10]], 11])` $\longrightarrow$ `66`

TD : Structures de données avancées

Les Files (FIFO : First In, First Out)

Exercice 1: Mise en oeuvre d'une pile avec une liste

Il existe plusieurs façons d'implémenter une file : à l'aide de listes, de tableaux, etc. La solution la plus simple consiste à utiliser un conteneur de type **list**.

Cependant, bien que les listes puissent servir à représenter une file (*First In, First Out*), elles ne sont pas optimales dans ce cas. En effet, si les ajouts et suppressions en fin de liste sont rapides, les insertions ou retraits en début de liste sont coûteux, car tous les autres éléments doivent être décalés.

Indications pour la réalisation en Python

Le type `'list'` intègre déjà toutes les méthodes pour réaliser cette structure de données :

- `list.append(x)` qui ajoute un élément à la fin de la liste.
- `list.insert(i, x)` qui insère un élément à la position indiquée. Le premier argument est la position de l'élément courant avant lequel l'insertion doit s'effectuer, donc `a.insert(0, x)` insère l'élément en tête de la liste, et `a.insert(len(a), x)` est équivalent à `a.append(x)`.
- `list.remove(x)` qui supprime de la liste le premier élément dont la valeur est x. Une exception est levée s'il n'existe aucun élément avec cette valeur.
- `list.pop([i])`, qui enlève de la liste l'élément situé à la position indiquée, et le retourne. Si aucune position n'est indiqué, `a.pop()` enlève et retourne le dernier élément de la liste

Programmer le module `file.py` contenant les fonctions suivantes :

1. `creer_file()` : crée et retourne une file vide non bornée (dont la taille maximale n'est pas définie, donc on peut y emfiler autant d'éléments que l'on veut, dans la limite de la mémoire de l'ordinateur).
2. `file_vide(f)` : retourne `True` si la file `f` est vide et `False` sinon.
3. `sommet(f)` : renvoie l'élément du sommet (tête) de la file `f`, sans le supprimer.
4. `taille(f)` : renvoie la taille de la file `f`.
5. `enfiler(f,x)` : ajoute l'élément x au queue de la file `f`.
6. `defiler(f)` : supprime le l'élément située à la tête de la file `f` et le retourne .

Exercice 2

On pourra éventuellement utiliser une ou des piles/files temporaires, on utilisera les primitives `creer_pile()` qui renvoie une pile vide et `creer_file()` qui renvoie une file vide.
Écrivez les fonctions suivantes :

1. `afficher(f)` : cette fonction affiche tous les éléments de la file `f`.
2. `defilerJusqua(f,x)` : cette méthode défile la file `f` jusqu'à l'élément x. L'élément x n'est pas défilé. Si l'élément x n'appartient pas à la file, alors la méthode défile tous les éléments de la file.
3. `appartient(f,x)` : cette fonction s'applique sur une file `f`. Elle renvoie **True** si l'élément x appartient à la file, et **False** sinon. Attention : il est important que la file ne change pas.
4. `inverser_file(f)` : cette fonction inverse les éléments de la file `f`. On a le droit d'utiliser des files ou des piles temporaires.

Exercice 3

Les *nombres de Hamming* sont les entiers naturels non nuls dont les seuls facteurs premiers éventuels sont 2, 3 et 5.
Exemples : 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, 16, 18, 20, 24, 25, 27, 30, ...
Le but de cet exercice est de les générer de manière croissante. Évidemment, on peut parcourir un à un tous les entiers en testant à chaque fois si ceux-ci sont des entiers de Hamming, mais cette démarche montre vite des limites (Le 2000e entier de Hamming est égal à 8 100 000 000 et le 2001e à 8 153 726 976 : il faudrait tester plus de 53 millions de nombres avant d'augmenter notre liste d'un élément !).
On adopte donc la démarche suivante : on utilise trois files `f2` ,`f3` et `f5` contenant initialement le nombre 1, et on suit la démarche suivante :

1. On détermine le plus petit des trois têtes des files, que l'on note k et que l'on imprime à l'écran ;
2. On retire cet élément des files où il se trouve ;
3. On insère en queue des files `f2` ,`f3` et `f5` les entiers $2k$, $3k$ et $5k$.

Cette démarche utilise le fait que tout nombre de Hamming différent de 1 est le produit par 2, 3 ou 5 d'un nombre de Hamming plus petit.
Travail à faire : Rédiger une fonction Python permettant l'affichage des n premiers nombres de Hamming.

Exercice 4

Une structure de **File** fonctionne sur le principe **premier entré-premier sorti** (comme les files d'attentes à un guichet). Une façon d'implémenter une File contenant au plus n éléments est de créer $F = [queue, tete, [0, 0, \ldots, 0]]$ où la sous-liste $F[2]$ de F est initialisée avec n zéros. Les éléments `tete` et `queue` sont deux entiers.
Le fonctionnement est le suivant :

- À la création de la File `F` : `tete` et `queue` sont initialisées à zéros.
- Quand on insère un élément `x` dans la File `F`, on le met dans `F[2][queue]`, puis `queue` est augmenté d'une unité.
- Si on retire un élément de la File `F`, on le retire de `F[2][tete]`, puis on augmente `tete`

d'une unité. La fonction qui gère cela doit retourner la valeur retirée.

- Si `tete` ou `queue` atteignent la fin de `F[2]`, ils doivent pouvoir revenir à 0 : la gestion des indices `tete` et `queue` se fait donc de manière circulaire.
- La File `F` est vide lorsque `tete` égale à `queue`.
- La File `F` est pleine lorsque la différence entre la `tete` et la `queue` égale à n.

Écrire les fonctions Python suivantes :

1. `creer_file(n)`, `taille(f)`, `file_vide(f)`, `sommet(f)`.
2. `enfiler(f, x)` qui insère l'élément `x` dans la file `f` si celle-ci n'est pas pleine.
3. `defiler(f)` qui retire l'élément correct (sommet) de la file `f`,si celle-ci n'est pas vide, et le renvoie.

03

Bases de Données

Introduction aux Bases de Données

De la gestion des fichiers aux bases de données
Partie 1/2

Anis SAIED

IPEIN

2024/25

Introduction

Le but de ce chapitre est

- Assimiler les fondements des Bases de Données Relationnelles (BDR)
- Avantages des BD par rapport aux fichiers ?

Contexte

Considérons une application de gestion de bibliothèque qui utilise des fichiers texte pour stocker les informations des **livres**, des **étudiants** et leurs **emprunts**.

Problématique

Comment Modéliser le système d'informations

On commence par se poser les questions suivantes:
- Comment modéliser ce problème ? comment passer du monde réel au monde virtuel ? Quelles informations une bibliothèque (simplifiée) doit-elle gérer ?

- Des **Livres**
- Des etudiants : **Emprunteurs** ;
- Des emprunts de livres par des emprunteurs : **Emprunts** ;

→On doit avoir des informations sur ces trois types d'objets/concepts :

- Livre : numéro, titre, auteur, année,...
- Emprunteur : nom, prénom, de quoi l'identifier, ...
- Emprunt : qui a emprunté quoi, quand ? Quand le document doit-il être rendu ? ...

Comment mettre en oeuvre le système d'information de la bibliothèque ?

→Solution disponible : **Fichiers texte**

Les Fichiers texte I

Exemple

Livres.txt :
```
ISBN, Titre, Auteur, Année
978-3-16-148410-0, Python 3, A. Jamel, 2020
978-0-13-110362-7, Intro BD, B. Med., 2018
```

Etudiants.txt :
```
ID, Nom, Prénom, Adresse
001, B.A., Ali, 12 Rue Liberté
002, D.S., Bacem, 46 Avenue Tunis
```

Emprunts.txt :
```
ID Emprunteur, ISBN Livre, Titre Livre, Date Emprunt, Date Retour
001, 978-3-16-148410-0, Python 3, 2024-08-01, 2024-08-15
002, 978-0-13-110362-7, Intro BD, 2024-08-05, 2024-08-19
```

Ce model de données textuel présente plusieurs inconvénients !

Limites des fichiers texte

1. Redondance des données

Les mêmes informations sont **dupliquées** dans plusieurs fichiers ce qui entraîne : Du gaspillage d'espace; Un risque élevé d'erreurs lors des mises à jour; Une difficulté à garantir l'unicité des données...

Exemple : Si le titre ou le ISBN d'un livre change, quels fichiers doivent être **mis à jour** ? sachant que toute omission entraîne une incohérence.

Livres.txt :

```
ISBN, Titre, Auteur, Année
978-3-16-148410-0, Python 3, A. Jamel, 2020
978-0-13-110362-7, Intro BD, B. Med., 2018
```

Emprunts.txt :

```
ID Emprunteur, ISBN Livre, Titre Livre, Date Emprunt
001, 978-3-16-148410-0, Python 3, 2024-08-01, 2024-08-15
002, 978-0-13-110362-7, Intro BD, 2024-08-05, 2024-08-19
```

→ Si le titre ou le ISBN du livre "Python 3" est modifié, il faut le mettre à jour dans *Livres.txt* et *Emprunts.txt*.

Limites des fichiers texte

2. Incohérence des données

Une incohérence apparaît lorsque des fichiers censés contenir la même information ne sont plus synchronisés.

Exemple : La suppression d'un livre dans `Livres.txt` n'est pas suivie automatiquement (par erreur humaine ou technique) par la suppression des emprunts associés dans `Emprunts.txt`.

Livres.txt :

```
ISBN, Titre, Auteur, Année
978-3-16-148410-0, Python 3, A. Jamel, 2020(supprimé)
978-0-13-110362-7, Intro BD, B. Med., 2018
```

Emprunts.txt :

```
ID Emprunteur, ISBN Livre, Titre Livre, Date Emprunt
001, 978-3-16-148410-0, Python 3, 2024-08-01, 2024-08-15
002, 978-0-13-110362-7, Intro BD, 2024-08-05, 2024-08-19
```

→ Le livre "Python 3" a été supprimé de *Livres.txt*, mais il est toujours présent dans *Emprunts.txt*, créant ainsi une incohérence.

Limites des fichiers texte

3. Sécurité des données

Il est difficile d'affiner le contrôle d'accès aux données spécifiques dans chaque fichier et de sécuriser les données sensibles.

Exemple :

Comment empêcher un utilisateur non autorisé d'accéder ou de modifier (ajouter, modifier ou supprimer) une ligne du fichier `Emprunts.txt` ?

Livres.txt :

```
ISBN, Titre, Auteur, Année
978-3-16-148410-0, Python 3, A. Jamel, 2020
978-0-13-110362-7, Intro BD, B. Med., 2018
```

Emprunts.txt :

```
ID Emprunteur, ISBN Livre, Titre Livre, Date Emprunt, Date Retour
001, 978-3-16-148410-0, Python 3, 2024-08-01, 2024-08-15
002, 978-0-13-110362-7, Intro BD, 2024-08-05, 2024-08-19
```

→ Si le fichier *Emprunts.txt* est stocké en clair et accessible à tous les utilisateurs, des personnes non autorisées pourraient y accéder ou le modifier, compromettant ainsi la **confidentialité** et l'**intégrité des données**.

→ Il est difficile de mettre en place des **contrôles d'accès** *granulaires* pour chaque fichier individuel.

Limites des fichiers texte

4. Couplage fort entre le contenu des fichiers texte et le code source

Le **contenu** des fichiers texte est étroitement **lié au code source**, ce qui rend les **modifications difficiles** et **sensible aux erreurs**.

Exemple :

Modifier le format de *Emprunts.txt* nécessite des changements dans le code qui lit ce fichier.

Format initial de *Emprunts.txt* :

```
ID Emprunteur, ISBN Livre, Titre Livre, Date Emprunt, Date Retour
001, 978-3-16-148410-0, Python 3, 2024-08-01, 2024-08-15
```

Format modifié de *Emprunts.txt* :

```
ID Emprunteur, ISBN, État Livre, Titre, Date Emprunt, Date Retour
001, 978-3-16-148410-0, Bon état, Python 3, 2024-08-01,
```

→ Ajouter un champ **État Livre** pour indiquer l'état du livre nécessite une mise à jour du code source pour gérer ce nouveau champ.

→ Sans ajustement du code, le traitement des données dans le fichier peut être incorrect ou échouer.

Limites des fichiers texte

5. Recherche inefficace

La **recherche d'informations** spécifiques ou le **tri des données** devient rapidement **complexe** et **lent**.

Exemple :
Comment trouver tous les livres empruntés par un même emprunteur sans parcourir manuellement tous les enregistrements ?

→ Trouver tous les livres empruntés par un emprunteur nécessite de parcourir le fichier *Emprunts.txt* en entier (parcours, strip, split, conversion de types, comparaisons, faire attention aux index ...).

→ La recherche et le tri des données deviennent longs et peu pratiques avec des fichiers texte volumineux.

Autres limites des fichiers texte

De nombreuses autres limites apparaissent selon les contextes d'utilisation :

- Absence de contrôle d'intégrité *(ex. impossibilité d'imposer l'unicité d'un ID)* ;
- Difficulté à gérer les accès concurrents *(ex. deux utilisateurs modifient simultanément le même fichier)* ;
- Absence de transactions *(ex. une mise à jour échoue au milieu et laisse les données incohérentes)* ;
- Risques élevés de perte des données *(ex. suppression accidentelle d'un fichier)* ;
- Faible capacité d'évolution du modèle *(ex. ajout d'un champ nécessitant de modifier tous les fichiers)*.

Conclusion : Ces limites justifient la nécessité d'un **modèle de données structuré**, comme celui proposé par les Bases de Données Relationnelles (BDR).

Solution fiable

Base de données

Fichier texte

Modèle textuel

↓

BDR

Modèle relationnel

Du fichier à la base de données I

Modèle Textuel → Modèle Relationnels

Données en Fichiers Texte

Les données enregistrées dans des fichiers texte sont souvent organisées en lignes, chaque ligne représentant un enregistrement avec des valeurs séparées par des délimiteurs.

Exemple :

Etudiants.txt :

```
ID, Nom, Prénom, Adresse
001, B.A., Ali, 12 Rue Liberté
002, D.S., Bacem, 46 Avenue Tunis
```

Du fichier à la base de données II

Modèle Textuel → Modèle Relationnels

Modélisation en Relations

En base de données relationnelle (BDR), les mêmes données peuvent être modélisées sous forme de Relations : **ensemble** d'enregistrements.

Chaque **enregistrement** est représenté comme un ensemble de couples (paires clé-valeur), où chaque **clé** est un **attribut** et chaque **valeur** est la **donnée** associée.

Exemple : **Relation Etudiants**

	Clé	:	**Valeur**
Enregistrement	ID	:	001
	Nom	:	B.A.
	Prénom	:	Ali
	Adresse	:	12 Rue Liberté
Enregistrement	ID	:	002
	Nom	:	D.S.
	Prénom	:	Bacem
	Adresse	:	46 Avenue Tunis

Du fichier à la base de données III

Modèle Textuel → Modèle Relationnels

Lignes de Texte

```
001, B.A. , Ali, 12 Rue Liberté
002, D.S. , Bacem, 46 Avenue Tunis
```

Transformation

Couples Clé-Valeur

```
{ID:001 , Nom:B.A , Prénom:Ali , Adresse:12 Rue Liberté}
{ID:002 , Nom:D.S , Prénom:Bacem , Adresse:46 Avenue Tunis}
```

Conversion

Relation / Table Relationnelle

ID	Nom	Prénom	Adresse
001	B.A.	Ali	12 Rue Liberté
002	D.S.	Bacem	46 Avenue Tunis

Modèle Relationnel

Relation

Une **Relation** est une table représentant un ensemble de tuples.

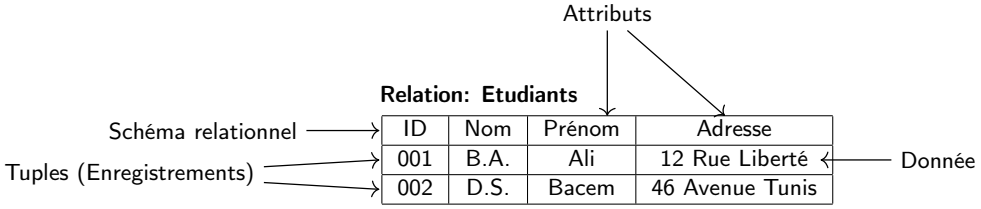


Schéma relationnel S : Ensemble d'attributs décrivant les données d'une relation, noté $S = (A_1, \ldots, A_n)$, avec $i \neq j \Rightarrow A_i \neq A_j$.

Pour un **attribut** A donné, $D(A)$ est le **domaine** (type) de A: c'est l'ensemble des *valeurs* qui peuvent être prises par l'attribut A.
Ex: $D(\text{ID}) = \text{entier}$
Exemples courants : Entier, Réel, chaine, booléen, Date, ...

On note aussi $S = ((A_1, D(A_1)), \ldots, (A_n, D(A_n))$
Exemple : Etudiants (ID, Nom, Prénom, Adresse)

Modèle Relationnel

Définition

Le modèle relationnel est un modèle de données où les informations sont organisées sous forme de **relations** (ou tables).

Chaque relation est constituée de lignes (**tuples**) et de colonnes (**attributs**).

Exemple :

> Une table "Etudiants" avec des colonnes comme ID, Nom, Prénom, et Adresse.

Une base de données organisée selon un modèle relationnel est dite **base de données relationnelle (BDR)** et est gérée à l'aide d'un **SGBD Relationnel (SGBDR)**.

Modèle Relationnel

Clés

Clé Primaire :

Une **clé primaire** est un attribut (ou une combinaison d'attributs) dont le contenu est unique pour chaque enregistrement, permettant d'identifier un et un seul enregistrement dans une table.

Exemple :

Dans la table "Etudiants", la colonne "ID" peut être la clé primaire car chaque étudiant a un ID unique.

Notation : Etudiant(**ID**, Nom, Prenom, Adresse)

Clé Étrangère :

Une **clé étrangère** est un attribut d'une table qui fait référence à la clé primaire d'une autre table, établissant un lien entre les deux tables.

Exemple :

Dans la table "Emprunts", la colonne "ID_Emprunteur" est une clé étrangère qui fait référence à l'ID de la table "Etudiants".

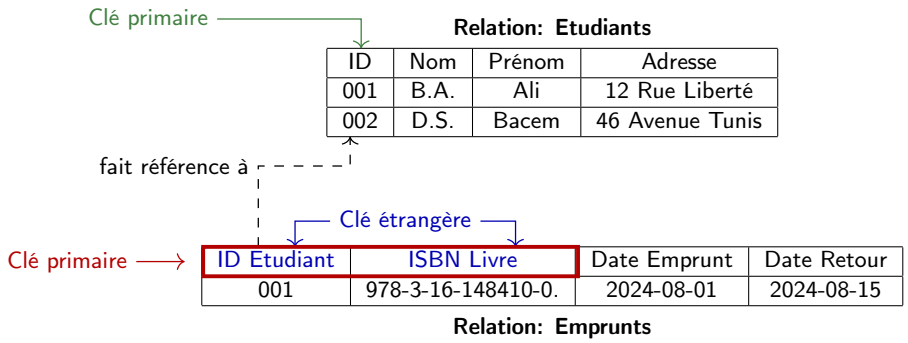
Notation : Emprunts(#**ID_Emprunteur**, ...)

Modèle Relationnel

Intégrité Référentielle

L'**intégrité référentielle** garantit que toutes les clés étrangères pointent vers des enregistrements existants dans la table référencée, assurant ainsi la cohérence des données.
Une « *violation d'intégrité référentielle* » se produit lorsque cette condition n'est pas respectée.
Exemple :

Si un enregistrement est supprimé dans "Etudiants", toutes les emprunts associées dans "Emprunts" doivent être supprimées ou mises à jour (*NULL* à la place des ID supprimés).



BDR

Manipulation de données

Une base de données organisée selon un modèle de données de type relationnel, est dite **base de données relationnelle** (BDR).

Manipulation Théorique :

Les opérations sur les bases de données relationnelles sont définies formellement par l'**algèbre relationnelle**.

Manipulation Pratique :

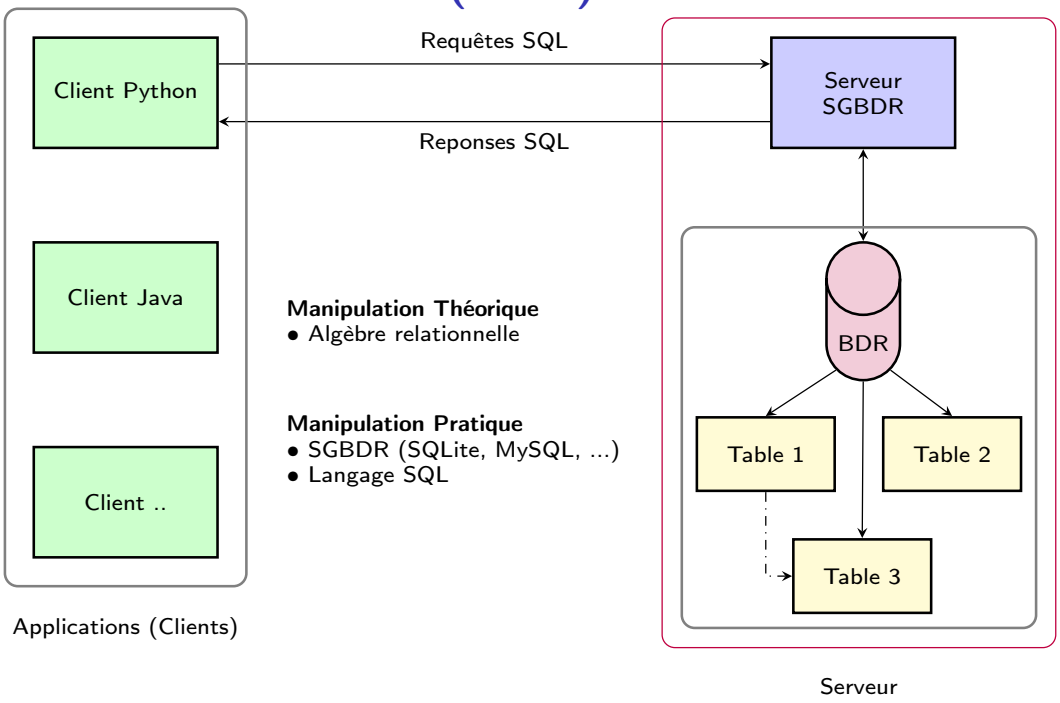
En pratique, les bases de données sont manipulées par les **Systèmes de Gestion de Bases de Données (SGBDR)** à l'aide du langage **SQL** (Structured Query Language).

Ces systèmes (ex: **SQLite**, MySQL, PostgreSQL, ...) permettent de gérer les tables, d'ajouter, modifier, et supprimer des données, et de garantir l'intégrité des données.

Vérification des acquis

1. Quel est le principal problème des fichiers texte ?
   A. Ils sont difficiles à lire
   B. Ils entraînent de la redondance
   C. Ils ne peuvent pas être ouverts sous Windows
2. Dans une table, un tuple correspond à :
   A. Une colonne
   B. Une ligne
   C. Une base de données
3. Une clé étrangère sert à :
   A. Garantir l'unicité
   B. Relier deux tables
   C. Accélérer les recherches

Architecture Client-Serveur pour une Base de Données Relationnelle (BDR)



Algèbre Relationnelle

Définition

L'**algèbre relationnelle** est un ensemble d'opérations formelles qui permet de manipuler et interroger les relations (tables).

Elle se base sur l'utilisation d'opérateurs pour manipuler les relations (tables) afin de produire de nouvelles relations comme résultat.

Syntaxe générale d'une opération :

```
R = Opérateur Relation
R = Relation1 Opérateur Relation2
```

Opérateurs: Les symboles comme $\cup$, $\cap$, σ, π, $\bowtie$, etc., sont utilisés pour représenter des opérations en algèbre relationnelle.

Exemple :

$R3 = R1 \cup R2$
Union (R3 = Tous les éléments de R1 et R2, sans répétition)

Les opérateurs de l'algèbre relationnelle sont classés en deux catégories : **unaires** et **binaires**.

Opérateurs de l'AR

Opérateurs Unaires

Les opérateurs unaires sont appliqués sur une seule relation :

Sélection (σ) : Filtre les lignes d'une relation qui satisfont un prédicat spécifique.
Exemple : $\sigma_{\text{condition}}(R)$

Projection (π) : Sélectionne un sous-ensemble de colonnes d'une relation.
Exemple : $\pi_{\text{colonnes}}(R)$

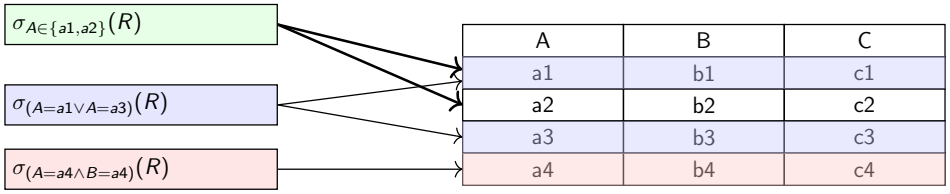
Renommage (ρ) : Renomme les attributs d'une relation.
Exemple : $\rho_{B \to A}(R)$

Opérateur de Sélection (σ)

L'opérateur de sélection (σ) est utilisé pour filtrer les lignes d'une relation qui satisfont un prédicat spécifique. Cet opérateur est unaire et s'applique donc à une seule relation.

Syntaxe: $\mathrm{R} = \sigma_{\text{condition}}(\text{Relation})$

Où `condition` est une expression booléenne qui filtre les tuples de la relation, et dans laquelle on peut utiliser les opérateurs logiques suivants : **AND** ($\wedge$), **OR** ($\vee$), **NOT** ($\neg$), **IN** ($\in$), et **LIKE**



Exemple : Sélectionner les emprunts où la **Date de Retour** est le 2024-08-15.

$$R = \sigma_{\text{Date Retour}=\text{'2024-08-15'}}(\text{Emprunts})$$

R :

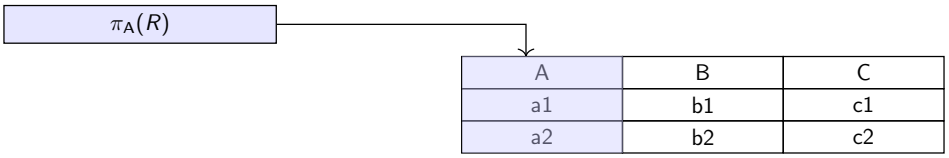
ID Emprunteur	ISBN Livre	Titre Livre	Date Emprunt	Date Retour
001	978-3-...	Python 3	2024-08-01	2024-08-15

Opérateur de Projection (π)

L'opérateur de projection (π) est utilisé pour sélectionner des colonnes spécifiques d'une relation. Cet opérateur est aussi unaire et s'applique donc à une seule relation.

Syntaxe: $R = \pi_{\text{colonnes}}(\text{Relation})$

Où `colonnes` est une liste des colonnes que vous souhaitez conserver dans le résultat.



Exemple : Projeter les colonnes **ID Emprunteur** et **Date Retour** d'une table d'emprunts. $R = \pi_{\text{ID Emprunteur,Date Retour}}(\text{Emprunts})$

R :

ID Emprunteur	Date Retour
001	2024-08-15
002	2024-08-19

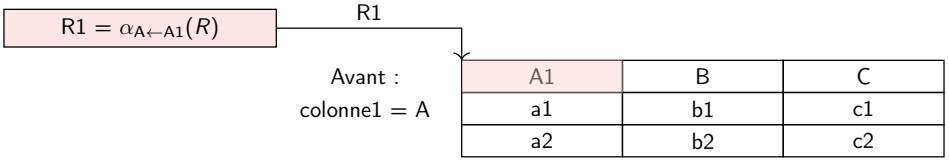
Renommage (α)

Le renommage est un opérateur unaire qui produit une relation R' ayant les mêmes tuples/enregistrements et le même schémaque la relation R de départ, mais dans lequel un attribut a été renommé.

But : résoudre des problèmes de compatibilité entre noms d'attributs de deux relations opérandes d'une opération binaire.

Notation :

$$R' = \alpha_{\text{nomAttribut} \leftarrow \text{nouveauNom}}(R)$$



Exercice

Application simple: Opérateurs unaires

Question

Afficher l'écrivan du livre "Pyhon 3".

Relation: Livres

ISBN	Titre	Auteur	Année
978-3-16-148410-0	Python 3	A. Jamel	2020
978-0-13-110362-7	Intro BD	B. Med.	2018

Réponse

$R1 = \sigma_{Titre='Python3'}(Livres)$
$R2 = \pi_{Auteur}(R1)$
$R3 = \alpha_{Auteur \leftarrow Ecrivan}(R2)$

Relation: R3

Ecrivan
A. Jamel

Opérateurs Binaires

Les opérateurs binaires sont appliqués sur deux relations. Voici les principaux :

Union ($\cup$) : Combine les lignes de deux relations, sans les dupliquer.
Exemple : $R \cup S$

Intersection ($\cap$) : Sélectionne les lignes communes entre deux relations.
Exemple : $R \cap S$

Différence ($-$) : Sélectionne les lignes qui sont dans la première relation mais pas dans la deuxième.
Exemple : $R - S$

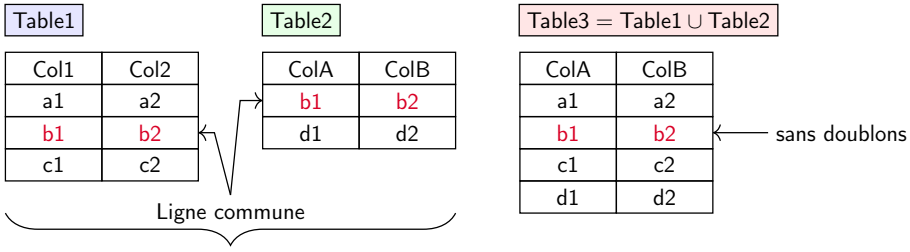
Produit Cartésien ($\times$) : Combine chaque ligne de la première relation avec chaque ligne de la seconde relation.
Exemple : $R \times S$

Jointure ($\bowtie$) : Combine les lignes de deux relations en fonction d'une condition de jointure.
Exemple : $R \bowtie_{\text{condition}} S$

Opérateur d'Union (∪)

L'opérateur d'union (∪) est utilisé pour combiner les lignes de deux relations ayant le même schéma. Cet opérateur est également binaire et s'applique donc à deux relations.Les doublons sont automatiquement éliminés.

Syntaxe : $R = \text{Relation1} \cup \text{Relation2}$



Même schéma: même nombre de colonnes et mêmes types

Exemple: Les noms de tous les étudiants.

$R3 = \pi_{Nom}(R1) \cup \pi_{Nom}(R2)$

R1: Étudiants 1ère année

Nom	ID Étudiant
Ahmed	E1
Fatima	E2
Sami	E3

R2: Étudiants 2ème année

Nom	ID Étudiant
Ahmed	E4
Hala	E5
Sami	E6

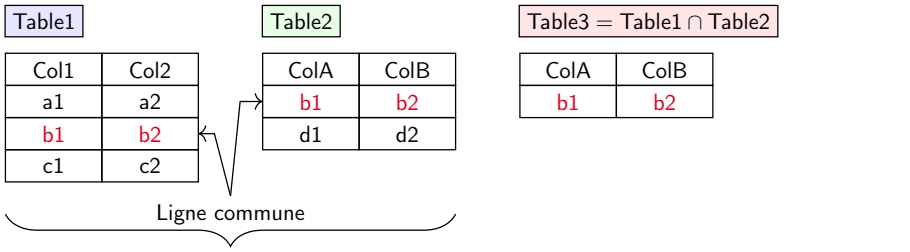
R3: Liste de tous les étudiants

Nom
Ahmed
Fatima
Sami
Hala

Opérateur d'Intersection (∩)

L'opérateur d'intersection (∩) est utilisé pour trouver les lignes communes entre deux relations ayant le même schéma.

Syntaxe : $R = \text{Relation1} \cap \text{Relation2}$



Même schéma: même nombre de colonnes et mêmes types

Exemple: Les noms communs des étudiants dans les deux années.

$R3 = \pi_{Nom}(R1) \cap \pi_{Nom}(R2)$

R1: Étudiants 1ère année

ID	Nom
E1	Ali
E2	Ahmed
E3	Hela

R2: Étudiants 2ème année

ID	Nom
E2	Ali
E4	Sami
E5	Chaima

R3: les nom des étudiants communs

Nom
Ali

Opérateur de Différence ($-$)

L'opérateur de différence ($\setminus$ ou $-$) est utilisé pour trouver les lignes présentes dans la première relation mais pas dans la deuxième relation ayant le même schéma.

Syntaxe : $R = \text{Relation1} - \text{Relation2}$

Table1

Col1	Col2
a1	a2
b1	b2
c1	c2

Table2

ColA	ColB
b1	b2
d1	d2

Ligne commune

Table3 = Table1 − Table2

ColA	ColB
a1	a2
c1	c2

Même schéma: même nombre de colonnes et mêmes types

Exemple: Les ID des produits non commandés.

$R3 = \pi_{ID}(R1) - \pi_{IdProd}(R2)$

R1: Produits

ID	Nom
P1	Orange
P2	Pomme
P3	Banane

R2: Commandes

IdProd	Quantité
P1	2.5
P3	3

R3: ID des Produits non commandés

ID
P2

Opérateur de Produit Cartésien ($\times$)

L'opérateur de produit cartésien ($\times$) est utilisé pour combiner chaque ligne de la première relation avec chaque ligne de la deuxième relation. Le résultat est une nouvelle relation contenant toutes les combinaisons possibles des lignes des deux relations.

Syntaxe : $R = \text{Relation1} \times \text{Relation2}$

Table1

Col1	Col2
a1	a2
b1	b2
c1	c2

Table2

ColA	ColB
x1	x2
y1	y2

Table3 = Table1 × Table2

Col1	Col2	ColA	ColB
a1	a2	x1	x2
a1	a2	y1	y2
b1	b2	x1	x2
b1	b2	y1	y2
c1	c2	x1	x2
c1	c2	y1	y2

Produit Cartésien ($\times$)

Exemple

Imaginons que nous ayons deux tables : une pour les clients et une autre pour les produits. Nous voulons obtenir toutes les combinaisons possibles de clients et de produits pour générer des commandes potentielles.

Clients

ClientID	Nom
C1	Ali
C2	Bacem

Produits

ProduitID	Produit
P1	Livre
P2	Stylo

Commandes = Clients $\times$ Produits

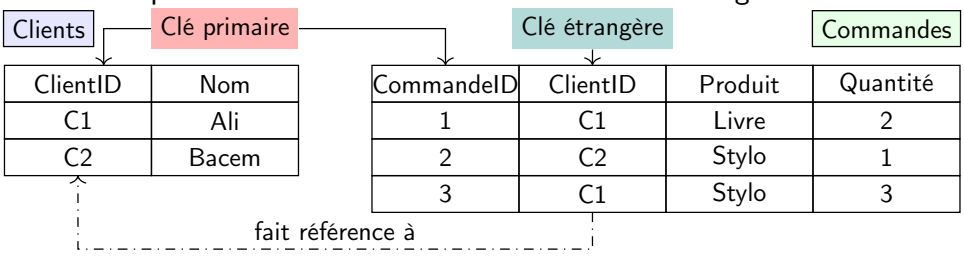
ClientID	Nom	ProduitID	Produit
C1	Ali	P1	Livre
C1	Ali	P2	Stylo
C2	Bacem	P1	Livre
C2	Bacem	P2	Stylo

Résultat : Cette table montre toutes les commandes potentielles où chaque client est associé à chaque produit.

Jointure ($\bowtie$)

$R3 = R1 \underset{\text{condition}}{\bowtie} R2$

La jointure est utilisée pour combiner des lignes de deux ou plusieurs tables en fonction d'une colonne commune entre elles. Elle permet de récupérer des données de plusieurs tables et de les associer de manière significative.



Exemple : Associer chaque commande à son client.
Principe : Il s'agit de combiner chaque ligne de la table 'Commandes' avec une ligne de la table 'Clients' où le 'ClientID' correspond.

$$\textit{Clients} \underset{\text{Clients.ClientID} = \text{Commandes.ClientID}}{\bowtie} \textit{Commandes}$$

1. On commence par réaliser le produit cartésien entre les tables 'Clients' et 'Commandes'.
2. Ensuite, on sélectionne les lignes où les 'ClientID' correspondent pour obtenir la jointure finale.

Jointure

Étape 1 : Produit Cartésien

Le produit cartésien combine chaque ligne de la table 'Clients' avec chaque ligne de la table 'Commandes', produisant ainsi une table intermédiaire où chaque combinaison possible de lignes est présente.

Produit Cartésien: Clients × Commandes

ClientID	Nom	CommandeID	ClientID	Produit	Quantité
C1	Ali	1	**C1**	Livre	2
C1	Ali	2	C2	Stylo	1
C1	Ali	3	**C1**	Stylo	3
C2	Bacem	1	C1	Livre	2
C2	Bacem	2	**C2**	Stylo	1
C2	Bacem	3	C1	Stylo	3

Le tableau ci-dessus représente le produit cartésien des tables 'Clients' et 'Commandes', combinant chaque ligne de 'Clients' avec chaque ligne de 'Commandes'.

Jointure

Étape 2 : Sélection

La sélection de la jointure consiste à filtrer le produit cartésien pour ne conserver que les lignes où les 'ClientID' correspondent (les valeurs des clés étrangères égales aux valeurs de clés primaires).

Selection : $\sigma_{Clients.ClientID=Commandes.ClientID}(Clients \times Commandes)$

ClientID	Nom	CommandeID	ClientID	Produit	Quantité
C1	Ali	1	C1	Livre	2
C1	Ali	2	C1	Stylo	1
C2	Bacem	3	C2	Stylo	3

Le tableau ci-dessus représente la table résultante après avoir appliqué la condition de jointure 'Clients.ClientID = Commandes.ClientID'.
Cette étape réduit le produit cartésien à la jointure correcte des deux tables, affichant les colonnes 'ClientID' deux fois.

Jointure

Clients $\underset{\text{Clients.ClientID} = \text{Commandes.ClientID}}{\bowtie}$ *Commandes*

Table Clients

ClientID	Nom
C1	Ali
C2	Bacem

Table Commandes

CommandeID	ClientID	Produit	Quantité
1	C1	Livre	2
2	C2	Stylo	1
3	C1	Stylo	3

Table intermédiare: Produit Cartésien

ClientID	Nom	CommandeID	ClientID	Produit	Quantité
C1	Ali	1	C1	Livre	2
C1	Ali	2	C2	Stylo	1
C1	Ali	3	C1	Stylo	3
C2	Bacem	1	C1	Livre	2
C2	Bacem	2	C2	Stylo	1
C2	Bacem	2	C1	Stylo	3

Table : Résultat de la jointure

ClientID	Nom	CommandeID	ClientID	Produit	Quantité
C1	Ali	1	C1	Livre	2
C1	Ali	3	C1	Stylo	3
C2	Bacem	2	C2	Stylo	1

Jointure

Exercice

Q. Afficher le nom du client de la commande N° 1.

R.

$$R1 = \text{Clients} \underset{\text{Clients.ClientID} = \text{Commandes.ClientID}}{\bowtie} \text{Commandes}$$

ou bien

$$R1 = \text{Clients} \bowtie \text{Commandes}$$

$$R2 = \sigma_{\text{CommandeID}=1}(R1)$$

$$R3 = \pi_{\text{Nom}}(R2)$$

Autrement:

$$\pi_{\text{Nom}}\left(\text{Clients} \underset{\substack{\text{Clients.ClientID} = \text{Commandes.ClientID} \\ \wedge \text{CommandeID} = 1}}{\bowtie} \text{Commandes}\right)$$

À Retenir

Modèle Relationnel :

Modèle des données basé sur des tables (relations).

Chaque table est composée de lignes (tuples) et de colonnes (attributs).

Clés primaires et étrangères assurent l'intégrité des données.

Algèbre Relationnelle :

Langage formel pour manipuler et interroger des bases de données relationnelles.

Opérations principales : sélection (σ), projection (π), jointure ($\bowtie$), union ($\cup$), intersection ($\cap$), différence ($-$).

Permet de définir des requêtes de manière abstraite avant leur implémentation en SQL.

SQL (Structured Query Language) :

Langage standard pour la gestion des bases de données relationnelles.

Implémente les concepts théoriques de l'algèbre relationnelle pour manipuler les données.

Evaluation — Opérations de l'AR I

1. Que fait l'opérateur de sélection σ ?

(A) Il sélectionne des colonnes

(B) Il filtre des lignes selon un prédicat (condition ou une expression logique)

(C) Il combine deux relations

2. Que retourne l'opérateur de projection π ?

(A) Un sous-ensemble de colonnes

(B) Un sous-ensemble de lignes

(C) Une jointure

3. La projection peut-elle produire des doublons ?

(A) Oui

(B) Non

(C) Seulement si la relation contient une clé primaire

4. Quelle opération combine toutes les lignes de deux relations ?

(A) Union

(B) Produit cartésien

Evaluation — Opérations de l'AR II

(C) Jointure naturelle

5. La condition nécessaire pour faire une union :

(A) Même nombre de lignes

(B) Même schéma

(C) Aucune condition

6. L'opération $R - S$ retourne :

(A) Les tuples communs à R et S

(B) Les tuples de R uniquement

(C) Les tuples de S uniquement

7. Une jointure équivaut souvent à :

(A) Un produit cartésien + une sélection

(B) Une union + une différence

(C) Une projection + une intersection

8. Quelle opération peut augmenter très fortement le nombre de tuples ?

(A) Projection

Evaluation — Opérations de l'AR III

(B) Produit cartésien

(C) Différence

9. Quelle opération garde uniquement les tuples présents dans les deux relations ?

(A) Intersection

(B) Union

(C) Jointure extérieure

10. Dans $R \bowtie S$, le symbole $\bowtie$ correspond à :

(A) Une sélection

(B) Une jointure

(C) Une projection

Références utiles

Livres :

- R. Elmasri , S. Navathe — Fundamentals of Database Systems
- C. Date — Introduction to Database Systems
- Ramakrishnan , Gehrke — Database Management Systems

Sites web :

- https://en.wikipedia.org/wiki/Relational_model
- https://sqlbolt.com
- https://w3schools.com/sql
- https://db-engines.com

Bases de Données

De L'AR à SQL
partie 2/2

Anis SAIED

IPEIN

2024/25

SQL

Structured Query Language

Langage universel pour gérer les bases de données relationnelles.

Définit, manipule et interroge les données.

Utilisé pour structurer les tables et gérer les accès.

Principales catégories de commandes SQL :

DDL (Data Definition Language) : Création et modification de la structure des bases de données (ex : 'CREATE', 'ALTER', 'DROP').

DML (Data Manipulation Language) : Manipulation des données à l'intérieur des tables (ex : 'SELECT', 'INSERT', 'UPDATE', 'DELETE').

Commande SQL: CREATE

Définition et Syntaxe

Définition :

La commande `CREATE` est utilisée pour créer une nouvelle table dans une base de données.

Elle définit la structure de la table, y compris les colonnes et leurs types.

Syntaxe Générale :

```
CREATE TABLE TableName (
  Column1 DataType [constraints],
  Column2 DataType [constraints],
  ...
);
```

Types Possibles de Colonnes :

`INT` : Entier

`Text / VARCHAR(n)` : Chaîne de caractères de longueur variable (n est la longueur maximale)

`DATE` : Date

`REAL` : (ou FLOAT) Nombre à virgule flottante

`BOOLEAN` : Valeur booléenne (vrai/faux)

Commande SQL: CREATE

Contraintes sur les Colonnes

`PRIMARY KEY`
Assure l'unicité de la colonne et identifie chaque enregistrement de manière unique. Ex: `ID INT PRIMARY KEY`

`FOREIGN KEY`
Crée une relation entre deux tables en pointant vers une clé primaire d'une autre table. Ex: `FOREIGN KEY (ClientID) REFERENCES Clients(ID)`

`NOT NULL`
Empêche qu'une colonne ait une valeur `NULL`.
Ex: `Nom VARCHAR(100) NOT NULL`

`UNIQUE`
Garantit que tous les éléments d'une colonne sont uniques.
Ex : `Email VARCHAR(255) UNIQUE`

`CHECK`
Implique une condition qui doit être vraie pour chaque enregistrement.
Ex: `Age INT CHECK (Age > 0)`

`DEFAULT`
Définit une valeur par défaut pour une colonne si aucune valeur n'est spécifiée lors de l'insertion. Ex: `CreatedDate DATE DEFAULT CURRENT_DATE`

Commande SQL: CREATE

Exemple 1

Donner la commande SQL permettant de créer la table `Etudiant` du schéma relationnel:

`Etudiant (ID, Nom, Age, Email)`

Requête SQL :

```
CREATE TABLE Etudiant (
  ID INT PRIMARY KEY,
  Nom TEXT NOT NULL,
  Age INT CHECK (Age >= 18),
  Email VARCHAR(100) UNIQUE
);
```

ID	Nom	Age	Email

Table 1: Table `Etudiant` (Vide)

Commande SQL: CREATE

Exemple 2

```
Departements (ID, Nom)
Cours (ID, Titre, Description, #DepartementID, DateDebut)
```

Requêtes SQL :

```
CREATE TABLE Departements (
  ID INT PRIMARY KEY,
  Nom VARCHAR(100) NOT NULL
);

CREATE TABLE Cours (
  ID INT PRIMARY KEY,
  Titre VARCHAR(100) NOT NULL,
  Description TEXT,
  DepartementID INT,
  DateDebut DATE DEFAULT CURRENT_DATE,
  FOREIGN KEY (DepartementID) REFERENCES Departements(ID)
);
```

Commande SQL: DROP

Définition et Syntaxe

Définition :

La commande `DROP` est utilisée pour supprimer une table existante dans une base de données.

Cette action est irréversible, ce qui signifie que les données et la structure de la table seront définitivement supprimées.

Syntaxe Générale :

```
DROP TABLE TableName;
```


Commande SQL: DROP

Exemple 1

Donner la commande SQL permettant de supprimer la table `Etudiant` du schéma relationnel:

`Etudiant (ID, Nom, Age, Email)`

Requête SQL :

```
DROP TABLE Etudiant;
```

→ Table `Etudiant` supprimée, n'existe plus.

Commande SQL: ALTER TABLE

Définition et Syntaxe

Définition :

La commande `ALTER TABLE` est utilisée pour modifier la structure d'une table existante dans une base de données.

Elle permet d'ajouter, de supprimer ou de modifier des colonnes ainsi que de changer les contraintes sur les colonnes.

Syntaxe Générale :

```
ALTER TABLE TableName
  ADD ColumnName DataType [constraints];
  DROP COLUMN ColumnName;
  RENAME COLUMN OldColumnName TO NewColumnName;
```


Commande SQL: ALTER TABLE

Exemple 1

Modifier la table `Etudiant` en ajoutant une colonne `Adresse` et en supprimant la colonne `Age`:

De
`Etudiant (ID, Nom, Age, Email)`
à
`Etudiant (ID, Nom, Email, Adresse)`

Requêtes SQL :

```
ALTER TABLE Etudiant
  ADD Adresse VARCHAR(255);
ALTER TABLE Etudiant
  DROP COLUMN Age;
```

ID	Nom	Email	Adresse

Table 2: Table `Etudiant` modifiée

Commande SQL: ALTER TABLE

Exemple 2

Modifier la table `Cours` pour ajouter une colonne `HeureDebut` de type `TIME`, supprimer la colonne `Description`, et renommer la colonne `Titre` en `NomCours` :

`Cours (ID, Titre, #DepartementID, DateDebut)`

Requêtes SQL :

```
ALTER TABLE Cours
  ADD HeureDebut TIME;

ALTER TABLE Cours
  DROP COLUMN Description;

ALTER TABLE Cours
  RENAME COLUMN Titre TO NomCours;
```


Exercice

Mise en pratique des commandes SQL: CREATE, ALTER, DROP

CREATE : Créez les tables Livres et Etudiants avec les colonnes et contraintes appropriées :

`Livres (ID, Titre, Auteur, Code_Livre, Prix)`

`Etudiants (ID, Nom, Prenom, Adresse, Email)`

`Emprunts (ID, #ID_Livre, #ID_Etudiant, Date_Emprunt, Date_Retour)`

ALTER : Modifiez les tables créées :

Ajoutez une colonne `DateNaissance` à la table `Etudiants`.

Supprimez la colonne `Prix` de la table `Livres`.

Renommez la colonne `Code_Livre` en `ISBN` dans la table `Livres`.

DROP : Supprimez la table `Emprunts`.

Solution

Réponses aux commandes SQL

```
CREATE TABLE Livres (
ID INT PRIMARY KEY,
Titre VARCHAR(255) NOT NULL,
Auteur VARCHAR(255) NOT NULL,
Code_Livre INT UNIQUE,
Prix REAL
);
```

```
CREATE TABLE Etudiants (
ID INT PRIMARY KEY,
Nom VARCHAR(255) NOT NULL,
Prenom VARCHAR(255) NOT NULL,
Adresse TEXT,
Email VARCHAR(255) UNIQUE
);
```

- Ajouter une colonne `DateNaissance` à la table `Etudiants` :
  `ALTER TABLE Etudiants ADD DateNaissance DATE;`
- Supprimer la colonne `Prix` de la table `Livres` :
  `ALTER TABLE Livres DROP COLUMN Prix;`
- Renommer la colonne `Code_Livre` en `ISBN` dans la table `Livres` :
  `ALTER TABLE Livres RENAME COLUMN Code_Livre TO ISBN;`
- Supprimer la table `Emprunts` :
  `DROP TABLE Emprunteurs;`

Commande SQL: INSERT

Définition et Syntaxe

Définition :

La commande `INSERT INTO` est utilisée pour insérer des enregistrements dans une table.

Elle ajoute une ou plusieurs lignes de données à une table existante.

Syntaxe Générale :

```
INSERT INTO TableName (Column1, Column2, ...)
VALUES (Value1, Value2, ...);
```

Exemple :

```
INSERT INTO Etudiant (ID, Nom, Age, Email)
VALUES (1, 'Ali', 20, 'ali@ipei.tn');
```

Commande SQL: INSERT

Exemples

```
Table :  Etudiant (ID, Nom, Age, Email)
Requête SQL :
INSERT INTO Etudiant (ID, Nom, Age, Email)
VALUES (1, 'Ali', 20, 'ali@ipei.tn');

Table :
Cours (ID, Titre, Description, DepartementID, DateDebut)
Requête SQL :
INSERT INTO Cours
VALUES (101, 'Introduction à SQL', 'Cours de base sur
SQL', 1, '2024-09-01');

Table :  Livres (ID, Titre, Auteur, ISBN)
Requête SQL :
INSERT INTO Livres (ID, Titre, ISBN)
VALUES (1, 'Python 3', '978-0451524935');
```


Commande SQL: UPDATE

Définition et Syntaxe

Définition :

La commande `UPDATE` est utilisée pour modifier les enregistrements existants dans une table.

Elle permet de mettre à jour une ou plusieurs colonnes pour une ou plusieurs lignes.

Syntaxe Générale :

```
UPDATE TableName
SET Column1 = Value1, Column2 = Value2, ...
WHERE condition;
```

Exemple :

```
UPDATE Etudiant
SET Email = 'nouveau_email@ipei.tn'
WHERE ID = 1;
```

Commande SQL: UPDATE

Exemples

Exemple 1 :

Table : Cours (ID, Titre, Description, DepartementID, DateDebut)

Mettre à jour la description du cours avec l'ID 101 pour qu'elle soit plus détaillée :

```
UPDATE Cours
SET Description = 'Introduction complète à SQL'
WHERE ID = 101;
```

Exemple 2 :

Table : Livres (ID, Titre, Auteur, AnneePublication, ISBN)

Mettre à jour l'année de publication et l'auteur du livre avec l'ID 1 :

```
UPDATE Livres
SET AnneePublication = 2023, Auteur = 'Ali'
WHERE ID = 1;
```


Commande SQL: DELETE

Définition et Syntaxe

Définition :

La commande DELETE est utilisée pour supprimer des enregistrements d'une table.

Elle permet de supprimer une ou plusieurs lignes de données en fonction d'une condition.

Syntaxe Générale :

```
DELETE FROM TableName
WHERE condition;
```

Exemple :

```
DELETE FROM Etudiant
WHERE ID = 1;
```

Commande SQL: DELETE

Exemples

Exemple 1 :

Cours (ID, Titre, Description, DepartementID, DateDebut)

Supprimer le cours avec l'ID 101 :

```
DELETE FROM Cours
WHERE ID = 101;
```

Exemple 2 :

Livres (ID, Titre, Auteur, AnneePublication, ISBN)

Supprimer les livres publiés avant 2000 et écrits par 'Ali' :

```
DELETE FROM Livres
WHERE AnneePublication < 2000 AND Auteur = 'Ali';
```


Commande SQL: SELECT

Définition et Syntaxe

Définition :

La commande SELECT est utilisée pour interroger une base de données et extraire des données des tables.

Elle permet de spécifier les colonnes à afficher et de filtrer les résultats avec des conditions.

Syntaxe Générale :

```
SELECT Column1, Column2, ...
FROM TableName
WHERE condition;
```

Exemple :

```
SELECT Nom, Email
FROM Etudiant
WHERE Age > 18;
```

Commande SQL: SELECT

Opérateur Projection $\pi \rightarrow$ SELECT

L'opérateur Projection π en algèbre relationnelle, qui permet de sélectionner certaines colonnes d'une table, est traduit en SQL par la commande :

```
SELECT Column1, Column2, ...
FROM TableName
```

Exemple :

Table : Etudiant (ID, Nom, Age, Email)

Projection en AR :

$\pi_{Nom,Email}(Etudiant)$

En SQL :

```
SELECT Nom, Email
FROM Etudiant
```


Commande SQL: SELECT

Clause DISTINCT

La clause DISTINCT est utilisée en SQL pour éliminer les doublons dans les résultats d'une requête. Elle s'applique à la sélection des colonnes spécifiées dans la commande SELECT.

Syntaxe :

```
SELECT DISTINCT Column1, Column2, ...
FROM TableName
```

Exemple :

Table : Etudiant (ID, Nom, Age, Email)

Pour afficher des noms distincts des étudiants :

```
SELECT DISTINCT Nom
FROM Etudiant;
```

Note :
La clause DISTINCT est utile lorsque vous souhaitez obtenir des valeurs uniques dans le résultat.

Commande SQL: SELECT

Opérateur Renommage $\alpha \rightarrow$ AS

L'opérateur Renommage α en algèbre relationnelle permet de renommer les colonnes ou les tables dans une requête. En SQL, cet opérateur est traduit par la clause AS.

Syntaxe en Algèbre Relationnelle :

$$R1 = \alpha_{AncienNom \leftarrow NouveauNom}(Relation)$$

Syntaxe SQL :

```
SELECT ColumnName AS AliasName
FROM TableName AS AliasName
```

Exemple :

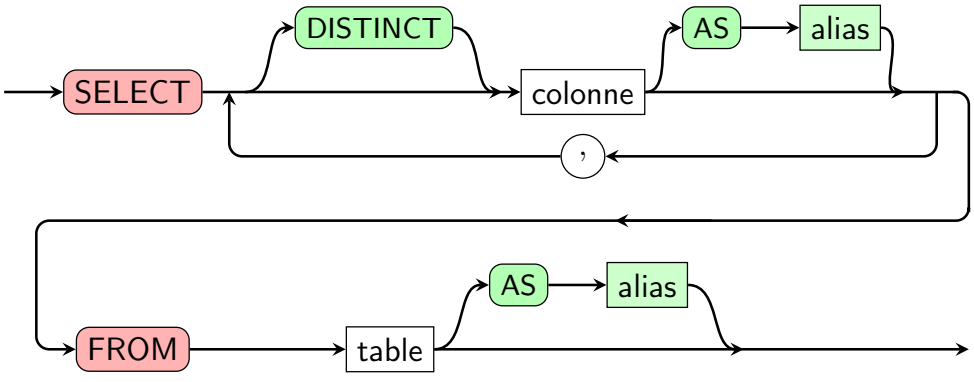
Table : Etudiant (ID, Nom, Age, Email)

Renommer la colonne Nom en NomEtudiant et la table Etudiant en E :

```
SELECT Nom AS NomEtudiant
FROM Etudiant AS E;
```


Commande SQL: SELECT

SELECT ... FROM



Commande SQL: SELECT

Opérateur Sélection σ

L'opérateur Sélection σ en algèbre relationnelle permet de filtrer les lignes d'une table en fonction de conditions spécifiques.

Correspondance SQL :
`WHERE condition`

Exemple :

Table : Etudiant (ID, Nom, Age, Email)

Sélection en AR:
$\sigma_{Age>18}(Etudiant)$

Sélection en SQL :

```
SELECT *
FROM Etudiant
WHERE Age > 18;
```


Commande SQL: SELECT

Exemple : WHERE & LIKE

Sélectionner les titres des cours qui ont débuté après le 1er janvier 2023 et dont le titre contient le mot "Python".

Table :
Cours (ID, Titre, Description, DepartementID, DateDebut)

Requête SQL :

```
SELECT C.Titre
FROM Cours C
WHERE C.DateDebut > '2023-01-01'
AND C.Titre LIKE '%Python%';
```

Note sur l'opérateur LIKE :

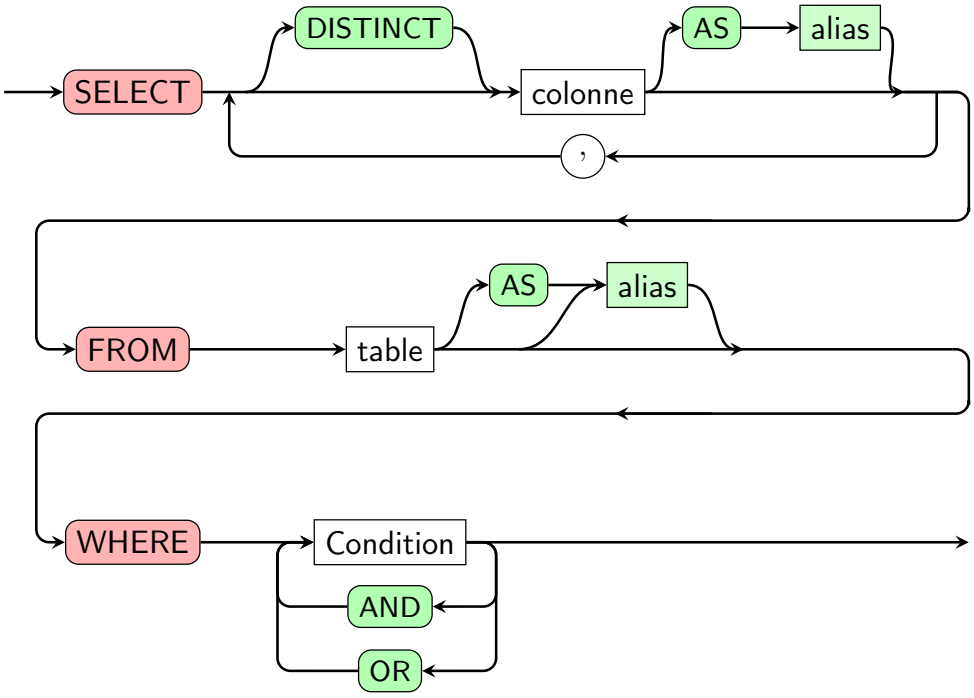
L'opérateur LIKE est utilisé pour rechercher une correspondance dans une colonne de type texte.

Les caractères spéciaux utilisés avec LIKE sont :

- % : Représente zéro ou plusieurs caractères.
- _ : Représente un seul caractère.

Commande SQL: SELECT

`SELECT ... FROM ... WHERE`



Commande SQL: SELECT

Opérateur Jointure ⋈

L'opérateur `Jointure` ⋈ en algèbre relationnelle combine les tuples de deux relations en fonction d'une condition de jointure.

Correspondance SQL :

```
Table1 JOIN Table2 ON Table1.Column = Table2.Column
```

Exemple :

```
Table :  Cours (ID, Titre, #DepartementID)
Table :  Departement (ID, Nom)
```

Jointure en AR:

$$Cours \underset{\text{Cours.DepartementID} = \text{Departement.ID}}{\bowtie} Departement$$

En SQL :

```sql
SELECT *
FROM Cours C JOIN Departement D ON C.DepartementID = D.ID;
```

Exemple de Jointure

Utilisation de `JOIN`

`Employes` (id, nom, #departementID)
`Departement` (id, nom)

Question : Afficher le département de chaque employé.

Requête SQL :

```sql
SELECT E.Nom AS Employe, D.Nom AS Departement
FROM Employes E
JOIN Departements D ON E.DepartementID = D.ID;
```

Table Employes

ID	Nom	DepartementID
1	Ali Ben Salem	101
2	Mohamed Trabelsi	102
3	Amel Saidi	103
4	Amina Bouaziz	101

Table `Departements`

ID	Nom
101	Informatique
102	Ressources Humaines
103	Comptabilité

Jointure: Combiner une ligne de Employes à une ligne de Departements

ID	Nom	DepartementID	ID	Nom
1	Ali Ben Salem	101	101	Informatique
2	Mohamed Trabelsi	102	102	Ressources Humaines
3	Amel Saidi	103	103	Comptabilité
4	Amina Bouaziz	101	101	Informatique

Employe	Departement
Ali Ben Salem	Informatique
Mohamed Trabelsi	Ressources Humaines
Amel Saidi	Comptabilité
Amina Bouaziz	Informatique

Exemple de Jointure avec Trois Tables

Utilisation de `JOIN` multiples

`Employes` (id, nom, #departementID)
`Departements` (id, nom)
`Projets` (id, nom, #employeID)

Question : Afficher le département et le projet de chaque employé.

```sql
SELECT E.nom Employe, D.nom Departement, P.nom Projet
FROM Employes E
JOIN Departements D ON E.DepartementID = D.ID
JOIN Projets P ON E.ID = P.EmployeID;
```

Table Employes

ID	Nom	DepartementID
1	Ali Ben Salem	101
2	Mohamed Trabelsi	102
3	Amel Saidi	103
4	Amina Bouaziz	101

Table `Departements`

ID	Nom
101	Informatique
102	Ressources Humaines
103	Comptabilité

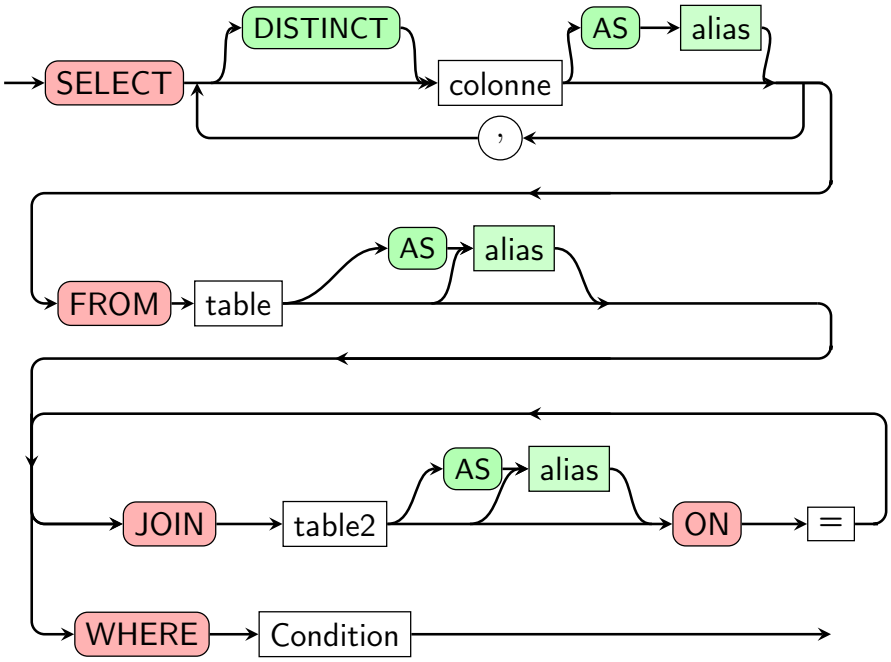
Table `Projets`

ID	Nom	EmployeID
1	Projet A	1
2	Projet B	2
3	Projet C	3
4	Projet D	1

E.ID	E.nom	E.DepartementID	D.id	D.nom	P.ID	P.nom	P.EmployeID
1	Ali Ben Salem	101	101	Informatique	1	Projet A	1
1	Ali Ben Salem	101	101	Informatique	4	Projet D	1
2	Mohamed Trabelsi	102	102	Ressources Humaines	2	Projet B	2
3	Amel Saidi	103	103	Comptabilité	3	Projet C	3

Commande SQL: SELECT

`SELECT ... FROM ... JOIN ... ON`



Commande SQL: SELECT

La clause `GROUP BY`

La clause `GROUP BY` en SQL permet de regrouper les lignes d'une table en fonction des valeurs d'une ou plusieurs colonnes, souvent utilisé en combinaison avec des **fonctions d'agrégation** comme `COUNT`, `SUM`, `AVG`, etc.

Syntaxe SQL :

```
GROUP BY colonne1, colonne2, ...
```

Exemple :

`Table : Cours (ID, Titre, DepartementID, DateDebut)`

Requête SQL : Nombre de cours par département

```
SELECT DepartementID, COUNT(*) AS NombreDeCours
FROM Cours
GROUP BY DepartementID;
```

Exemple de regroupement avec `GROUP BY`

La clause `GROUP BY`: Exemple

Table `Cours (ID, Titre, DepartementID, DateDebut)` :

ID	Titre	DepartementID	DateDebut
1	Mathématiques	1	2024-01-10
3	Informatique	1	2024-01-15
5	Statistiques	1	2024-01-22
2	Physique	2	2024-01-12
7	Économie	2	2024-01-30
4	Chimie	3	2024-01-20
6	Biologie	3	2024-01-25

Requête SQL: Nombre de cours par département.

```
SELECT DepartementID, COUNT(*) AS NombreDeCours
FROM Cours
GROUP BY DepartementID;
```

Résultat :

DepartementID	NombreDeCours
1	3
2	2
3	2

Fonctions d'agrégation en SQL

Principales fonctions d'agrégation

Les fonctions d'agrégation en SQL permettent de réaliser des calculs sur un ensemble de valeurs et de retourner un résultat unique. Elles sont couramment utilisées en combinaison avec la clause `GROUP BY` pour regrouper et résumer des données.

Principales fonctions d'agrégation :

- `COUNT()`: Compte le nombre de lignes dans un groupe.
- `SUM()`: Calcule la somme des valeurs numériques d'une colonne.
- `AVG()`: Calcule la moyenne des valeurs numériques d'une colonne.
- `MIN()`: Retourne la plus petite valeur dans un groupe.
- `MAX()`: Retourne la plus grande valeur dans un groupe.

Exemple d'utilisation :

- Trouver le nombre total d'étudiants par département.
- Calculer la moyenne des salaires par service.
- Déterminer le chiffre d'affaires total par produit.

Commande SQL: SELECT

Exemple : `GROUP BY` & `HAVING`

La clause `HAVING` est utilisée pour filtrer les groupes après l'application de `GROUP BY`, souvent avec des fonctions d'agrégation.

Table: `Cours (ID, Titre, #DepartementID, DateDebut)`
Requête SQL : Les départements ayant plus que 2 cours.

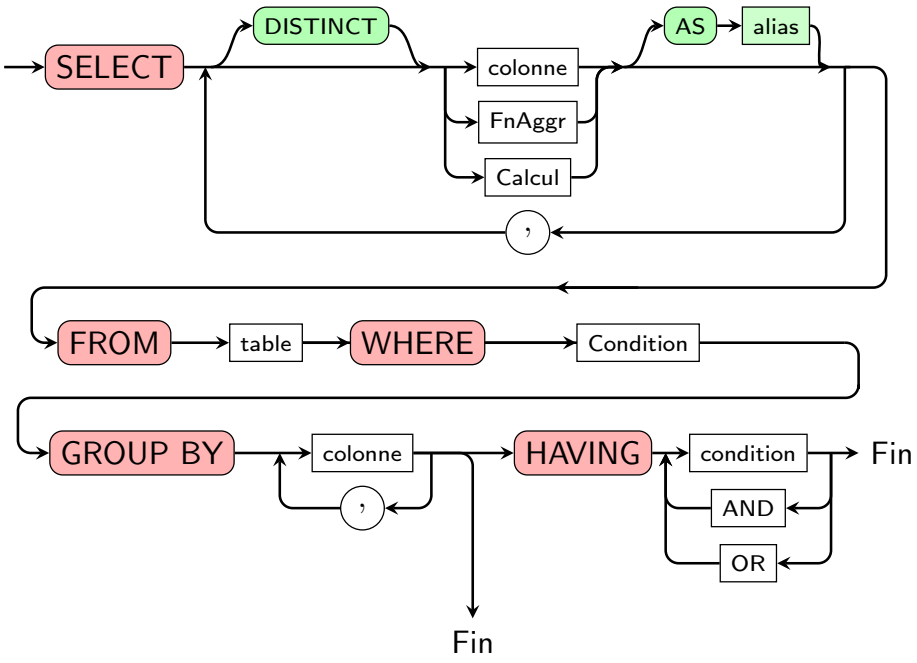
```
SELECT DepartementID, COUNT(*) AS NombreDeCours
FROM Cours
GROUP BY DepartementID HAVING NombreDeCours > 2;
```

ID	Titre	DepartementID	DateDebut
1	Mathématiques	1	2024-01-10
2	Informatique	1	2024-01-15
3	Statistiques	1	2024-01-22
4	Physique	2	2024-01-12
5	Économie	2	2024-01-30
6	Chimie	3	2024-01-20
7	Biologie	3	2024-01-25

DepartementID	NombreDeCours
1	3

Commande SQL: SELECT

`SELECT ... FROM ... GROUP BY ... HAVING`



Exercice d'application

Utilisation de `GROUP BY` et `HAVING`

Exercice : Vous avez une table **Ventes** qui contient les informations suivantes :

ID	ProduitID	Quantité	DateVente
1	101	10	2024-01-05
2	102	15	2024-01-07
3	101	20	2024-01-10
4	103	25	2024-01-12
5	101	5	2024-01-15
6	102	10	2024-01-18
7	103	30	2024-01-20
8	101	15	2024-01-25

1. Écrivez une requête SQL pour calculer la quantité totale vendue par produit.
   a. Utilisez la clause `GROUP BY` pour regrouper les ventes par `ProduitID`.
   b. Affichez les résultats avec les colonnes `ProduitID` et `QuantitéTotale`.
2. Ajoutez une clause `HAVING` pour ne garder que les produits pour lesquels la quantité totale vendue est supérieure à 30.

Solution de l'exercice

Utilisation de `GROUP BY` et `HAVING`

Etape 1 : Regroupe les ventes par identifiant de produit.

ID	ProduitID	Quantité	DateVente
1	101	10	2024-01-05
2	102	15	2024-01-07
3	101	20	2024-01-10
4	103	25	2024-01-12
5	101	5	2024-01-15
6	102	10	2024-01-18
7	103	30	2024-01-20
8	101	15	2024-01-25

ID	ProduitID	Quantité	DateVente
1	101	10	2024-01-05
3	101	20	2024-01-10
5	101	5	2024-01-15
8	101	15	2024-01-25
2	102	15	2024-01-07
6	102	10	2024-01-18
4	103	25	2024-01-12
7	103	30	2024-01-20

Etape 2: Calculer la quantité totale par groupe.

ProduitID	QuantitéTotale
101	50
102	25
103	55

Etape 3: Filtre les groupes pour ne garder que ceux dont la quantité totale dépasse 30.

Résultat final :

ProduitID	QuantitéTotale
101	50
103	55

Requête SQL :

```
SELECT ProduitID, SUM(Quantité) AS QuantitéTotale
FROM Ventes GROUP BY ProduitID HAVING QuantitéTotale > 30;
```

Tri et Pagination

Clauses SQL: `ORDER BY`, `LIMIT`, `OFFSET`, `ASC`, **et** `DESC`

Les clause de tri et de pagination

`ORDER BY` : Trie les résultats d'une requête SQL par une ou plusieurs colonnes.

`LIMIT` : Limite le nombre de résultats retournés par la requête.

`OFFSET` : Sauter un certain nombre de résultats avant de commencer à renvoyer les résultats.

`ASC` : Ordre croissant (par défaut).

`DESC` : Ordre décroissant.

Syntaxe :

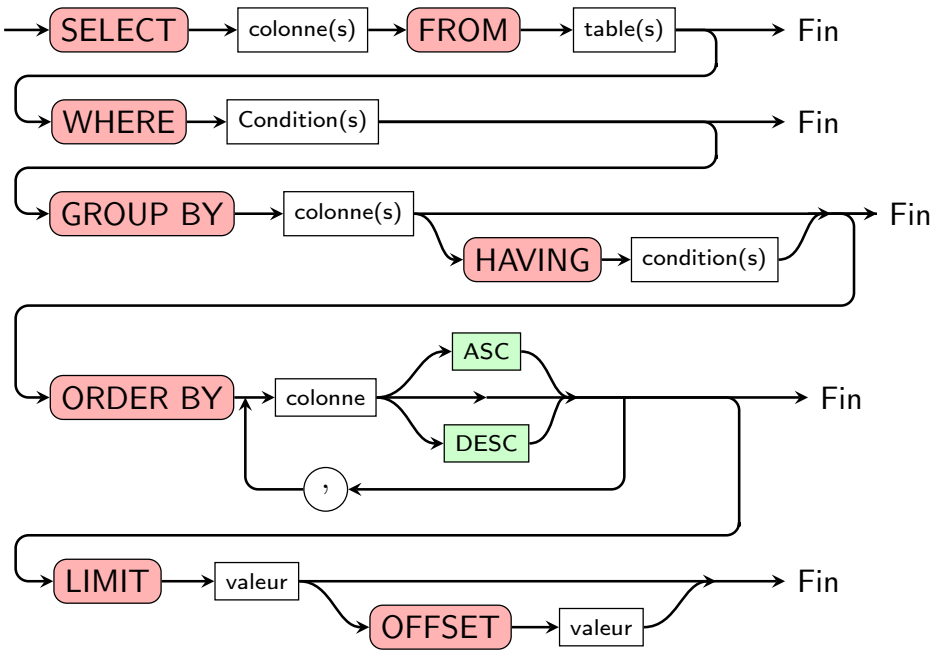
```
SELECT colonne1, colonne2, ...
FROM table
ORDER BY colonne1 [ASC|DESC], colonne2 [ASC|DESC], ...
LIMIT nombre_lignes
OFFSET nombre_lignes;
```

Exemple: Le deuxième produit le plus cher.

```sql
SELECT * FROM Produits
ORDER BY prix DESC
LIMIT 1 OFFSET 1;
```


Commande SQL: SELECT

`SELECT ... FROM ... GROUP BY ... HAVING`



Commande SQL: SELECT

Opérateur Union ∪

L'opérateur `Union` ∪ en algèbre relationnelle combine les résultats de deux requêtes distinctes en une seule. Par défaut, il élimine les doublons. Pour conserver les doublons, on utilise `UNION ALL`.

Correspondance SQL :

```
Requête1 UNION Requête2
Requête1 UNION ALL Requête2
```

Exemple :

```
Table :  Cours (ID, Titre, #DepartementID)
Table :  AnciensCours (ID, Titre, #DepartementID)
```

Union en AR:

$$Cours \cup AnciensCours$$

En SQL :

```sql
SELECT Titre FROM Cours
UNION
SELECT Titre FROM AnciensCours;
```


Exemple d'Union

Utilisation de `UNION` et `UNION ALL`

`Table Cours` (id, titre, #departementID)
`Table AnciensCours` (id, titre, #departementID)

Question : Afficher les titres de cours actuels et anciens.

Requête SQL :

```sql
SELECT Titre FROM Cours
UNION ALL
SELECT Titre FROM AnciensCours;
```

Table Cours

ID	Titre
1	Mathématiques
2	Informatique
3	Physique

Table AnciensCours

ID	Titre
4	Chimie
5	Biologie
6	Informatique

Résultat: Combinaison des titres des deux colonnes Titre

Titre
Mathématiques
Informatique
Physique
Chimie
Biologie
Informatique

Exercice : Utilisation de l'opérateur `UNION` avec projection

Exercice pratique

Énoncé :
Vous disposez de deux tables : `EtudiantsActuels` et `AnciensEtudiants`. Les tables ont les schémas suivants :

`EtudiantsActuels` (id, nom, prenom, #departementID, annee)
`AnciensEtudiants` (id, nom_complet, #departement)

Question :
Écrivez une requête SQL pour obtenir une liste des noms complets de tous les étudiants (actuels et anciens).

Corrigé :

```sql
SELECT CONCAT(nom, ' ', prenom) AS nom_complet
FROM EtudiantsActuels
UNION
SELECT nom_complet
FROM AnciensEtudiants;
```


Fonctions de manipulation de chaînes en SQL

Présentation des principales fonctions (Partie 1)

`CONCAT`
Combine plusieurs chaînes en une seule.
Syntaxe : `CONCAT`(chaîne1, chaîne2, ...)
Requête SQL :

```sql
SELECT CONCAT(Prenom, ' ', Nom) AS NomComplet
FROM etudiants;
```

Résultat :

NomComplet
Aymen zaied

`SUBSTRING`
Extrait une sous-chaîne d'une chaîne de caractères.
Syntaxe : `SUBSTRING`(chaîne, position, longueur)
Requête SQL :

```sql
SELECT SUBSTRING(Nom, 1, 3) AS NomPartiel
FROM etudiants;
```

Résultat :

NomPartiel
Aym

Fonctions de manipulation de chaînes en SQL

Présentation des principales fonctions (Partie 2)

`LENGTH`

Retourne la longueur d'une chaîne de caractères.

Syntaxe : `LENGTH`(chaîne)

Requête SQL :

```
SELECT LENGTH(Nom) AS LongueurNom
FROM etudiants;
```

Résultat :

LongueurNom
5

`UPPER` **et** `LOWER`

Convertit une chaîne en majuscules ou minuscules.

Syntaxe : `UPPER`(chaîne), `LOWER`(chaîne)

Requête SQL :

```
SELECT UPPER(Nom) AS NomMajuscules, LOWER(Prenom) AS
    PrenomMinuscules
FROM etudiants;
```

Résultat :

NomMajuscules	PrénomMinuscules
ZAIED	aymen

Commande SQL: SELECT

Opérateur Intersection ∩

L'opérateur `Intersection` ∩ en algèbre relationnelle retourne uniquement les enregistrements communs entre deux requêtes.

Correspondance SQL :

`Requête1 INTERSECT Requête2`

Exemple :

`Table : EtudiantsActuels (ID, Nom, #DepartementID)`

`Table : AnciensEtudiants (ID, Nom, #DepartementID)`

`Intersection en AR:`

$$EtudiantsActuels \cap AnciensEtudiants$$

`En SQL :`

```
SELECT Nom FROM EtudiantsActuels
INTERSECT
SELECT Nom FROM AnciensEtudiants;
```

Exemple d'Intersection

Utilisation de `INTERSECT`

`Table EtudiantsActuels` (id, nom, #departementID)
`Table AnciensEtudiants` (id, nom, #departementID)

Question : Trouver les étudiants inscrits actuellement et qui ont également été inscrits dans le passé.

Requête SQL :

```
1 SELECT Nom FROM EtudiantsActuels
2 INTERSECT
3 SELECT Nom FROM AnciensEtudiants;
```

Table EtudiantsActuels

ID	Nom
1	Ali
2	Bacem
3	Amira

Table AnciensEtudiants

ID	Nom
2	Ali
3	Aymen
4	Bacem

Résultat: Noms communs entre les deux colonnes Nom

Nom
Ali
Bacem

Exercice : Utilisation de l'opérateur `INTERSECT`

Exercice pratique

Énoncé :
Vous disposez de deux tables : `ProduitsEnStock` et `ProduitsCommandes`. Les tables ont les schémas suivants :

`ProduitsEnStock` (id, nom_produit, categorie)

`ProduitsCommandes` (id, nom_produit, categorie)

Question :
Écrivez une requête SQL pour obtenir une liste des produits qui sont à la fois en stock et ont été commandés.

Corrigé :

```
1 SELECT nom_produit
2 FROM ProduitsEnStock
3 INTERSECT
4 SELECT nom_produit
5 FROM ProduitsCommandes;
```

Commande SQL: SELECT

Opérateur Différence –

L'opérateur Différence – en algèbre relationnelle retourne les enregistrements présents dans la première requête mais pas dans la seconde.

Correspondance SQL :

```
ProduitsEnStock EXCEPT ProduitsCommandes
```

Exemple :

Table : ProduitsEnStock (ID, Nom_Produit, #Categorie)
Table : ProduitsCommandes (ID, Nom_Produit, #Categorie)

Différence en AR:

ProduitsEnStockExceptProduitsCommandes

En SQL :

```
1 SELECT Nom_Produit FROM ProduitsEnStock
2 EXCEPT
3 SELECT Nom_Produit FROM ProduitsCommandes;
```


Exemple de Différence

Utilisation de EXCEPT

Table ProduitsEnStock (id, nom_produit, #categorie)
Table ProduitsCommandes (id, nom_produit, #categorie)

Question : Trouver les produits qui sont en stock mais n'ont pas été commandés.

Requête SQL :

```
1 SELECT Nom_Produit FROM ProduitsEnStock
2 EXCEPT
3 SELECT Nom_Produit FROM ProduitsCommandes;
```

Table ProduitsEnStock

ID	Nom_Produit
1	Télévision
2	Radio
3	Smartphone

Table ProduitsCommandes

ID	Nom_Produit
2	Télévision
3	Radio
4	Ordinateur

Résultat: Noms des produits en stock mais non commandés

Nom_Produit
Smartphone

Exercice : Utilisation de l'opérateur EXCEPT

Exercice pratique

Énoncé :
Vous disposez de deux tables : `Clients` et `Commandes`. Les tables ont les schémas suivants :

`Clients` (id, nom_client, ville)

`Commandes` (id, #id_client, produit)

Question :
Écrivez une requête SQL pour obtenir une liste des clients qui n'ont passé aucune commande.

Corrigé :

```sql
SELECT *
FROM Clients
WEHERE id IN (SELECT id
              FROM Clients
              EXCEPT
              SELECT id_client
              FROM Commandes);
```


Types de Requêtes Imbriquées

Sous-requêtes en SQL

Il existe plusieurs types de sous-requêtes en SQL, chacune ayant son propre usage :

- Sous-requête qui retourne une seule valeur.
- Sous-requête qui retourne plusieurs valeurs ou lignes.
- Sous-requête qui dépend de la requête principale pour chaque ligne.

Exemple :Trouver le nom de l'employé ayant le salaire le plus élevé.

```sql
SELECT nom
FROM   Employes
WHERE  salaire = (SELECT MAX(salaire) FROM Employes);
```

Exemple de Sous-Requêtes

Exemple 1

Trouver les employés dont le salaire est supérieur au salaire moyen de leur département.

```sql
SELECT nom, salaire
FROM Employes e1
WHERE salaire > (
                SELECT AVG(salaire)
                FROM Employes e2
                WHERE e1.departement_id = e2.departement_id)
```

Explication : La sous-requête calcule le salaire moyen pour chaque département, et la requête principale sélectionne les employés dont le salaire est supérieur à cette moyenne.

Sous-Requêtes dans la clause `FROM`

Exemple 2

Les sous-requêtes peuvent également être utilisées dans la clause `FROM` pour créer une table temporaire qui sera utilisée dans la requête principale.

Exemple : Trouver les départements avec un salaire total supérieur à 100 000.

```sql
SELECT departement_id, SUM(salaire) AS total_salaire
FROM (  SELECT departement_id, salaire
        FROM Employes ) AS sous_table
GROUP BY departement_id
HAVING SUM(salaire) > 100000;
```

Explication : La sous-requête crée une table temporaire avec les départements et leurs salaires, puis la requête principale regroupe les résultats et applique la condition.

Exercice : Sous-Requête dans une clause `WHERE`

Exercice pratique

Énoncé :
Vous avez deux tables : `Produits` et `Ventes`. Les tables ont les schémas suivants :

`Produits` (id, nom_produit, prix)

`Ventes` (id, #produit_id, quantite)

Question :
Écrivez une requête SQL pour trouver les produits qui ont été vendus plus de 100 fois.

Corrigé :

```sql
SELECT nom_produit FROM Produits
WHERE id IN (
            SELECT produit_id
            FROM Ventes
            GROUP BY produit_id
            HAVING COUNT(*) > 100
            );
```

TD : LES BASES DE DONNÉES RELATIONNELLES

AR & SQL

Exercice 1: Gestion d'une Bibliothèque

On considère le schéma relationnel de la base « bibliothèque.db » :

- **Livre**(CodeLivre, Titre, Genre, Auteur, Collection)
- **Exemplaire**(CodeEx, *#CodeLivre*, DateAchat, Etat)
- **Abonne**(NumCarte, Nom, Prénom, Téléphone)
- **Pret**(*#CodeEx*, *#Numcarte*, DatePret)

Travail demandé :

Exprimer en **algèbre relationnelle** les requêtes permettant de :

1. Afficher les différents genres des livres.
2. Afficher toutes les informations sur les livres dont le genre est « Parascolaire ».
3. Afficher tous les titres livres de la collection « Ellipse ».
4. Afficher les titres des livres qui ont des exemplaires dont l'état est « abimé ».
5. Afficher les titres des livres achetés en 2019.
6. Afficher la collection du livre dont le titre est « INFORMATIQUE AVEC PYTHON ».
7. Pour les exemplaires en cours de prêt, afficher le nom et le prénom de l'abonné ainsi que le code de l'exemplaire et le titre du livre.
8. Afficher les codes des exemplaires qui ne sont pas en cours de prêt.
9. Pour les exemplaires qui ne sont pas en cours de prêt, afficher leurs codes et les titres de livres correspondants.

Exercice 2: Système de Consultation Médicale

La base de données **"Consultation.db"** gère les interactions entre médecins (RPPS) et patients (numSS).
Schéma relationnel :

- **Medecin** (numRPPS, nomM, prenomM, specialite, dateNaiss, adresse)
- **Patient** (numSS, nomP, prenomP, dateNaiss, *#numMT*)
- **Ordonnance** (numOrd, dateOrd, Type)
- **Consulte** (*#numMC*, *#numSS*, dateC, diagnostic, *#numOrd*)
- **Medicament** (idMed, nomMed, prix, categorie)
- **Prescription** (*#numOrd*, *#idMed*, nbBoites)

Partie 1 : Algèbre relationnelle

1. Décrire le résultat obtenu pour les requêtes suivantes :
   (a) $R = \pi_{numRPPS}(Medecin) - \pi_{numMC}(\sigma_{Type='Acte'}(Consulte \bowtie_{numOrd} Ordonnance))$
   (b) $R = \pi_{nomP,prenomP,nomM,prenomM}(Patient \bowtie_{numMT=numRPPS} Medecin)$
2. Exprimer en algèbre relationnelle :
   (a) Numéros des ordonnances où on a prescrit le médicament « AMLODIS 5mg ».
   (b) Numéros SS des patients dont toutes les consultations ont donné lieu à une ordonnance.

Partie 2 : SQL

1. Écrire la requête SQL de création de la table **Patient** (incluant les clés et contraintes).
2. Déterminer les médecins, triés par nom et prénom (ordre alphabétique).
3. Augmenter de 10% le prix des médicaments de la catégorie « Pneumologie ».
4. Noms et prénoms des généralistes ayant prescrit des ordonnances de type 'Acte'.
5. Patients diagnostiqués avec 'Rhume commun' ou 'Asthme' le 01/02/2024.
6. Informations du médecin cardiologue le plus âgé.
7. Nombre de médicaments dont le prix est supérieur au prix moyen.
8. Coût total de l'ordonnance numéro 300.
9. Numéros SS des patients ayant passé plus de 10 consultations en 2023.
10. Identifiants des patients ayant consulté au moins trois médecins différents.

Partie 3 : Programmation `sqlite3`

1. Écrire une fonction `AjoutMedicament(cur, ...)` avec vérification des contraintes (prix > 0, nom commençant par une majuscule).
2. Écrire une fonction `PourcentageSpecialite(cur, sp)` retournant le % de médecins et le % de consultations pour une spécialité donnée.
3. Écrire le script Python complet pour se connecter à la base et afficher les statistiques par spécialité.

Exercice 3: Base de données universitaire

Une université possède la base de données (simplifiée) relatives à ses étudiants suivantes :

— **ETUDIANTS**(ne, nom, prenom, #id_formation, date_naissance, adresse, salarie)
— **FORMATION**(id_formation, libelle_f, type)
— **MODULE**(id_module, #id_formation, libelle_m)
— **RESULTAT**(#ne, #id_module, note, annee)

Indications sur les attributs :

- ne : numéro étudiant, type : INT
- nom : nom de l'étudiant, type : VARCHAR
- prenom : prenom de l'étudiant, type : VARCHAR
- date_naissance : date de naissance de l'étudiant, type : DATE
- adresse : adresse de l'étudiant, type : VARCHAR
- id_module :réference du module, type : INT
- annee : annee d'obtention du module, type : INT
- id_formation : réference de la formation, type : INT
- libelle_m : libellé du module , type : VARCHAR
- libelle_f : libellé de la formation , type : VARCHAR
- type : type de formation , type : VARCHAR
- salarie : indique si l'étudiant est salarié (=1) ou non (=0), type : INT
- note : note de l'étudiant à l'examen relatif au module, type : INT

Autres Indications :

- Pour simplifier, tous les modules d'une formation ont même coefficient.
- La référence d'une formation est unique.
- Les types de formation sont « L1 », « L2 », « L3 », « M1 », « M2 ».
- Une formation dure un semestre.
- Une formation comprend plusieurs modules.
- Pour simplifier, un module ne peut pas être partagé par plusieurs formations.
- La formation est validée si la moyenne des modules relatifs à cette formation est supérieure ou égal à 10.
- Un module est validée si la note obtenu à l'examen est supérieur ou égale à 10, même si la formation dans son ensemble n'est pas validée.
- On peut donc valider sa formation sans valider chacun des modules.
- Seuls les modules non validés sont à repasser l'année suivante.
- Les notes sont entières et comprises entre 0 et 20.
- L'adresse d'un étudiant est donnée uniquement par le nom de sa ville de résidence.
- Pour simplifier, on suppose que chaque étudiant ne suit qu'une seule formation par année.

Questions :

Les requêtes seront exprimés en langage SQL.

1. Repérer et donner une clé primaire pour chaque relation de la base de données.
2. Repérer et donner la ou les clés étrangères pour chaque relation de la base de données.

3. Requête affichant les noms et prénoms des étudiants salariés.
4. Requête affichant la note de chacun des étudiants ayant suivi le module 1402 en 2016. Les étudiants seront identifiés par leurs numéros d'étudiant.
5. Requête affichant les numéros étudiant, les noms et prénoms des étudiants de L1.
6. Requête affichant l'identifiant et le libellé de tous les modules de licence (« L1 »,« L2 »,« L3 »).
7. Requête affichant les noms et prénoms des étudiants inscrits en M1 et ayant suivi le module « Informatique ».
8. Requête affichant l'identifiant du module 4526 ainsi que les notes moyenne, minimale et maximale obtenues dans ce module en 2016.
9. Requête affichant le nombre d'étudiants salariés suivant la formation dont le libellé est « Physique de rêve ».
10. Requête affichant le libellé et le nombre d'étudiants dans chaque formation.
11. Requête affichant le numéro d'étudiant, le nom, le prénom et la moyenne générale de Manel sayadi en 2014.
12. Requête affichant la liste (numéro d'étudiants, noms, prénoms) ainsi que les moyennes générales de tous les étudiants ayant validé leur année en 2015, quelque soit la formation.

Exercice 4: Gestion des livraisons (FPJ)

Soit la base de données relationnelle **FPJ** suivante :

— **FOURNISSEUR** (codfrs, nomfrs, villefrs, telfrs)

— **PROJET** (codproj, nomproj, villeproj, budjetproj, *#coddirecteur*)

— **DIRECTEUR** (coddirecteur, nomdirecteur)

— **PIECE** (codpiece, nompiece, couleurpiece, poidspiece, villepiece)

— **FPJ** (*#codfrs, #codpiece, #codproj*, qtelivree, dateliv)

Travail à faire :

1. Créer la base de données ci-dessus tout en créant toutes les contraintes d'intégrités nécessaires.
2. Insérer trois lignes dans chaque table.
3. Formuler les requêtes suivantes en **SQL** :
   a) Donner les numéros des pièces destinées à tout projet se déroulant dans la même ville que celle où se situe le fournisseur de ces mêmes pièces.
   b) Donner les numéros des projets dont au moins un des fournisseurs ne se trouve pas dans la même ville que celle où le projet se déroule.
   c) Donner les numéros des projets utilisant au moins une des pièces fournies par 'F3'.
   d) Quels sont les projets dont la deuxième lettre de leur nom est 'E' ?
   e) Quels sont les projets dont le nom de leur directeur se termine par 'A' ?
   f) Combien de fois chaque pièce a-t-elle été livrée ?

g) Combien de livraisons ont été effectuées entre le 01/01/19 et 01/01/20 par le fournisseur 'F2'?
h) Quelle est la pièce qui a la plus grande quantité livrée pour le projet 'P1'?
i) Quel est le poids de la pièce qui a été livrée le plus de fois?
j) Quelle sont les pièces qui n'ont été jamais livrées à des projets se déroulant à 'Tunis'?
k) Quels sont les projets auxquels on a livré toutes les pièces?
l) Ajouter 1000 aux budgets des projets se déroulant à Tunis et qui ont reçu plus de 10 livraisons de fournisseurs n'habitant pas Tunis.
m) Changer les couleurs de toutes les pièces rouges en orange.
n) Supprimer tous les projets pour lesquels il n'y a pas de livraison.
o) Augmenter de 10% toutes les livraisons effectuées par les fournisseurs de pièces détachées rouges.

Exercice 5: Gestion Bancaire et Python

Soit la base de données relationnelle **« Compte-bancaire »** suivante :

1. **CLIENT** (numcli, nomcli, prencli, adr)
2. **PERSONNEL** (numpre, nompres, prenpers, manager, sal) — *avec sal ≥ 1250*
3. **TYPCOMPT** (numTypC, nomTypC)
4. **COMPTE** (numcpt, *#numcli*, *#numTypC*, dateOuv, *#numPers*)
5. **TYPOPER** (numTypO, nomTypO)
6. **OPERATION** (numop, *#numcpt*, *#numcli*, *#numTypO*, dateOp, montant) — *montant ≠ 0*
7. **PERSONNE** (numP, nomP, PrenP, *#numP_pere*, *#numP_mere*)

Travail à faire (Scripts Python) :

1. Écrire un script Python permettant de copier les 20 premiers enregistrements de la table `Personne` dans la table `Client`.
2. Écrire un script Python permettant d'ouvrir un « compte courant » pour chaque client et d'enregistrer un « versement » de 500 dinars.
3. Écrire une fonction Python qui reçoit, comme paramètres, 2 numéros de personnes et retourne `True` s'ils sont frères/sœurs et `False` sinon.
4. Un client peut avoir plusieurs comptes. Écrire un script Python qui affiche, pour chaque client, son numéro, son nom, son prénom et la liste des comptes qu'il possède (`NumCpt`, `DateOuv`).

04

Algèbre Linéaire

Algèbre linéaire

Résolution des systèmes linéaires
Méthode du pivot de Gauss

Anis SAIED

IPEIN

2024/25

Plan

Introduction

Ce document constitue un rappel du module `NumPy` de Python. Il a pour objectif de présenter l'essentiel de la bibliothèque à travers des explications et illustrations claires des concepts et fonctionnalités.

Remarque : Tout ce qui est présenté ici rappelle le cours d'Algèbre linéaire de 1ère année. Pour plus de détails, se référer au cours correspondant.

Plan

Pourquoi NumPy ?

Le calcul scientifique nécessite la manipulation de grands ensembles de données.
`NumPy` (**Num**erical **Py**thon) est une bibliothèque Python pour travailler efficacement avec ces données.

- Tableaux homogènes (int, float, etc.)
- Taille fixe à la création

Installation et sous-modules

Ce module n'est pas fourni avec Python par défaut.
`http://docs.scipy.org/doc/numpy-1.10.1/user/install.html`

- `numpy.linalg` : algèbre linéaire
- `numpy.random` : générateurs aléatoires

Charger le module :

```
import numpy as np
```

Plan

Création de tableaux

```python
a = np.array([[1, 2, 3], [2, 3, 4], [0, 1, 0]]) # Création d'une matrice 3x3
print(type(a))     # Affiche le type de l'objet (numpy.ndarray)
print(a.dtype)     # Affiche le type des éléments (ex: int64)
b = a.tolist()     # Convertit le tableau numpy en liste Python standard
```

Vecteurs spéciaux

```python
t0 = np.arange(0, 10, 2) # [0, 2, 4, 6, 8] (début, fin exclue, pas)
t1 = np.arange(3)        # [0, 1, 2] (entiers par défaut)
t2 = np.arange(3.)       # [0., 1., 2.] (flottants car '3.' est float)
t3 = np.array([2]*5)     # [2, 2, 2, 2, 2] (répétition de liste)
b = np.linspace(0, 10, 5)# 5 points équirépartis entre 0 et 10 inclus
```


Matrices spéciales

```python
a = np.ones((3,5))              # Matrice 3x5 remplie de 1.0 (float)
b = np.zeros((3,5), dtype=int) # Matrice 3x5 de 0 (entiers)
c = np.eye(3)                   # Matrice identité d'ordre 3
d = np.diag([1,2,3])            # Matrice diagonale avec 1, 2, 3
AB0 = np.concatenate((a,b),axis=0) # Fusion verticale (empilement)
AB1 = np.concatenate((a,b),axis=1) # Fusion horizontale (côte à côte)
```

Tableaux aléatoires

```
a = np.random.rand(6)          # 6 valeurs entre 0 et 1 (loi uniforme)
b = np.random.randint(0, 10, (2,5)) # Matrice 2x5 d'entiers entre 0 et 9
a1 = np.random.choice(a)       # Tire un élément au hasard dans 'a'
a2 = np.random.choice(a, 10)   # Tire 10 éléments au hasard (vecteur)
a3 = np.random.choice(a, (2,10)) # Tire des éléments pour remplir une matrice 2x10
```


Lecture et écriture dans les tableaux

```
v = np.array(range(0,160,10))  # Vecteur de 0 à 150 par pas de 10
v[4:11]                        # Slicing : éléments de l'indice 4 à 10
v[::-1]                        # Inverse l'ordre du vecteur
a = np.arange(10)              # [0, 1, 2, ..., 9]
a[7] = 21                      # Modifie la valeur à l'indice 7
a[[7,2,5,3]] = (7777,2222,5555,3333) # Modification multiple par indexation
```

Copies et vues

```python
a = np.array([[0,1,2,3],[10,11,12,13],[20,21,22,23]])
b = a         # Simple étiquette (si on modifie b, a est modifié !)
b[2,1] = 9    # Modifie la ligne 2, colonne 1 dans b ET dans a
c = a.copy() # Création d'un objet distinct en mémoire (indépendant)
```


Opérations, Fonctions et Vectorisation

Opérations élémentaires (Ufuncs) :

```python
a = np.random.randint(0,10,(2,5))
b = np.random.randint(1,10,(2,5))
# Opérations terme à terme (addition, soustraction, produit, etc.)
print(a+b, a-b, a*b, a/b, a**b)
np.negative(a)                  # Inverse le signe de chaque élément
```

Vectorisation de fonctions personnalisées :

```python
def f(x):
return (x+np.sin(x))/(x+np.cos(x)) # Fonction définie pour un scalaire

a = np.arange(1,7)      # Vecteur [1, 2, 3, 4, 5, 6]
vf = np.vectorize(f)    # Transforme f pour qu'elle accepte des tableaux NumPy
resultat = vf(a)        # Applique f sur chaque élément de 'a' simultanément
```

Méthodes utiles

```python
a.max()    # Valeur maximale du tableau
a.min()    # Valeur minimale
a.sum()    # Somme de tous les éléments
a.prod()   # Produit de tous les éléments
a.mean()  # Moyenne arithmétique
# Applique la fonction 'sum' sur l'axe 0 (somme des colonnes) :
np.apply_along_axis(np.sum, 0, a)
```


Plan

Système d'équations linéaires

Considérons le système suivant :

$$\begin{array}{rcrcrcr} x_1 & - & 2x_2 & + & x_3 & = & 0 \\ 2x_1 & + & x_2 & - & 3x_3 & = & -1 \\ 3x_1 & - & x_2 & + & 2x_3 & = & 3 \end{array}$$

- Système linéaire à trois inconnues
- Résolution par la méthode du **pivot de Gauss**

Plan

Principe de la méthode

La méthode du pivot de Gauss consiste à :

1. Transformer le système en système triangulaire
2. Utiliser des opérations élémentaires sur les lignes
3. Appliquer la substitution arrière

Étape 1 : Pivot x1

$$\begin{array}{rcrcrcr} 1x_1 & - & 2x_2 & + & x_3 & = & 0 \\ 2x_1 & + & x_2 & - & 3x_3 & = & -1 \\ 3x_1 & - & x_2 & + & 2x_3 & = & 3 \end{array}$$

Étape 2 : Élimination x1

$$\begin{array}{rcrcrcr} 1x_1 & - & 2x_2 & + & x_3 & = & 0 \\ 0x_1 & + & 5x_2 & - & 5x_3 & = & -1 \\ 0x_1 & + & 5x_2 & - & x_3 & = & 3 \end{array}$$

Étape 3 : Pivot x2

On divise la deuxième équation par 5 :

$$\begin{array}{rcrcrcr} 1x_1 & - & 2x_2 & + & x_3 & = & 0 \\ 0x_1 & + & 1x_2 & - & x_3 & = & -\frac{1}{5} \\ 0x_1 & + & 5x_2 & - & x_3 & = & 3 \end{array}$$

Plan

Pivot de Gauss en Python

```python
import numpy as np

# Matrice augmentée [A|B]
A = np.array([[1,-2,1,0], [2,1,-3,-1], [3,-1,2,3]], dtype=float)
n = len(A) # Nombre de lignes

# --- ÉTAPE 1 : Élimination (Triangularisation) ---
for i in range(n):
A[i] = A[i] / A[i][i] # Normalisation : le pivot devient 1
for j in range(i+1, n):
# Soustrait la ligne pivot aux lignes suivantes pour faire des 0
A[j] = A[j] - A[j][i] * A[i]

# --- ÉTAPE 2 : Substitution arrière (Calcul des inconnues) ---
x = np.zeros(n) # Initialisation du vecteur solution
for i in range(n-1, -1, -1): # On remonte de la dernière ligne à
↪  la première
# Calcul de x[i] en isolant l'inconnue
x[i] = A[i][-1] - np.dot(A[i][i+1:n], x[i+1:n])

print(x) # Affiche les solutions [x1, x2, x3]
```

Plan

Conclusion

- La méthode du pivot de Gauss est systématique
- Elle permet de résoudre tout système linéaire compatible
- Facilement implémentable en Python avec un style clair et lisible

TD : Résolution des systèmes linéaires

Pivot du Gauss

Exercice 1 : Méthode du pivot de Gauss

Cet exercice a pour objectif de **mettre en pratique la méthode du pivot de Gauss** pour résoudre des systèmes linéaires inversibles de taille $n \times n$ avec Python. ¿ **Remarque :** Tout ce qui est présenté ici est un rappel des notions d'algèbre linéaire de 1re année. Pour plus de détails, se référer au cours correspondant.

Le système linéaire à résoudre

On considère un système linéaire S de la forme :

$$S : \begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n \end{cases}$$

Soit sous forme matricielle :

$$Ax = b, \quad A = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{pmatrix}, \quad x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \quad b = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix}$$

Présentation de la méthode du Pivot de Gauss

Principe

Le pivot de Gauss consiste à transformer la **matrice augmentée** $M = (A|B)$ en matrice triangulaire supérieure $M' = (A'|B')$, puis à résoudre $M'x = B'$ par **substitutions successives**.

Étape 1 : Triangularisation

Opérations élémentaires sur les lignes :

— Permuter deux lignes : $L_i \leftrightarrow L_j$

— Multiplier une ligne par un scalaire non nul : $L_i \leftarrow \lambda L_i$

— Ajouter un multiple d'une ligne à une autre : $L_i \leftarrow L_i + \lambda L_j$

Exemple à appliquer :

$$S = \begin{cases} x - 3y - 2z = 6 \\ 2x - 4y - 3z = 8 \\ -3x + 6y + 8z = -5 \end{cases}$$

Matrice augmentée correspondante :

$$M = \begin{pmatrix} 1 & -3 & -2 & 6 \\ 2 & -4 & -3 & 8 \\ -3 & 6 & 8 & -5 \end{pmatrix}$$

Instructions :

1. Choisir le premier pivot non nul a_{11}, permuter si nécessaire.
2. Éliminer tous les coefficients sous le pivot à l'aide de :

$$L_i \leftarrow L_i - \frac{a_{i1}}{a_{11}} L_1, \quad i = 2, \ldots, n$$

3. Répéter le processus sur les sous-matrices.

Étape 2 : Remontée

Après triangularisation, on résout le système triangulaire supérieur S' de bas en haut :

$$x_n = \frac{b'_n}{a'_{nn}}, \quad x_i = \frac{1}{a'_{ii}} \left(b'_i - \sum_{j=i+1}^{n} a'_{ij} x_j \right), \quad i = n-1, \ldots, 1$$

Questions

Vous allez programmer en Python les fonctions suivantes :

1. **taille()** : saisie d'un entier $n \geq 2$.
2. **saisie_Vect()** : saisie des coefficients d'un vecteur B (matrice colonne $(n, 1)$).
3. **saisie_Mat()** : saisie des coefficients d'une matrice carrée A (n, n).
4. **triangulaire(M)** : transforme la matrice augmentée $M = (A|B)$ en matrice triangulaire supérieure M'.
5. **remontee(T)** : résout un système triangulaire supérieur T par substitutions successives.
6. **pivot_Gauss(A,B)** : combine **triangulaire** et **remontee** pour résoudre $AX = B$.

Exemple de système à résoudre

$$A = \begin{pmatrix} 2 & -5 & 1 & 3 \\ 4 & 7 & 8 & 2 \\ 3 & 1 & 1 & 6 \\ 4 & 1 & 7 & 9 \end{pmatrix}, \quad B = \begin{pmatrix} 5 \\ 10 \\ 2 \\ 6 \end{pmatrix}$$

Trouver X solution de $AX = B$ avec votre script Python.

Remarques

— Utiliser `numpy` et `np.copy()` pour ne pas altérer les données initiales.
— Ce travail consiste à réimplémenter des fonctions déjà disponibles dans `numpy.linalg`, mais l'objectif est de comprendre la méthode.

Exercice 2 : Addition de matrices creuses

Un enjeu scientifique et technologique actuel est de savoir traiter des problèmes mettant en jeu un nombre très important de données. Un point crucial est souvent de pouvoir manipuler des matrices de très grandes dimensions, ce qui est a priori très coûteux, en temps de calcul et en mémoire. On peut cependant souvent considérer que les matrices manipulées ne contiennent que "peu" d'éléments non nuls : c'est ce que l'on appelle les matrices *creuses*.

Nous nous intéressons ici à l'implémentation de deux algorithmes d'addition de matrices : l'un pour une représentation classique des matrices, l'autre pour une représentation des matrices creuses.

Soit $n \in \mathbb{N}^*$, on note $\mathtt{M}_n(\mathbb{R})$ l'ensemble des matrices carrées d'ordre n, à coefficients dans $\mathbb{R}$.

Représentation des matrices en Python

- Classique : un tableau à double entrée (listes de listes) de longueur n représentant les lignes de la matrice.
- Creuse : on ne décrit que les cases non nulles par un tableau de triplets (i, j, x), où x est l'élément de la matrice situé à la i-ème ligne et j-ème colonne. Les éléments non nuls sont décrits ligne par ligne.

Exemple : La matrice

$$\begin{pmatrix} 1 & 0 & 5 \\ 0 & -2 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

sera représentée classiquement par le tableau '[[1,0,5], [0,-2,0], [0,0,0]]' et de manière creuse par le tableau '[(0,0,1), (0,2,5), (1,1,-2)]'.

Questions

Q1 Écrire une fonction `add(M,N)` prenant en argument deux représentations classiques M et N de deux matrices carrées d'ordre n et renvoyant la représentation classique de $M + N$. Optimiser la complexité spatiale et temporelle.

Q2 Écrire une fonction `add_creuse(M,N)` prenant en argument deux représentations creuses M et N de matrices carrées et renvoyant la représentation creuse de $M + N$. Optimiser la complexité spatiale et temporelle. On ne justifiera pas l'ordre des éléments.

Q3 Pour des matrices classiques d'ordre n, évaluer asymptotiquement la complexité temporelle de `add(M,N)`.

Q4 Pour des matrices creuses contenant au plus p éléments non nuls, évaluer asymptotiquement la complexité temporelle de `add_creuse(M,N)`.

Q5 Discuter de la représentation pertinente à utiliser selon la densité des matrices.

05

Cryptographie

Cryptographie

Anis SAIED

IPEIN

2024/25

Plan de la séance

Introduction au codage

- Le codage permet de transformer des données pour pouvoir ensuite les décoder.
- Le but : confidentialité d'un message, compréhensible uniquement par le destinataire.
- Ce cours présente trois méthodes de codage :
  1. Codage de César
  2. Codage de Vigenère
  3. Codage de Hill

Principe du codage de César

`cesar(m,d)` → `message codé avec décalage d`

- On décale chaque lettre du message d'un entier d (clé).
- Exemple avec $d = 3$: A → D, B → E, ..., X → A, Y → B, Z → C.
- Message clair :
  `LE TRESOR EST CACHE DANS LE CHATEAU DE: SUITE AU PROCHAIN MESSAGE!`
- Message codé :
  `OH WUHVRU HVW FDFKH GDQV OH FKDWHDX GH: VXLWH DX SURFKDLQ PHVVDJH!`
- Pour décoder, il suffit de décaler de $-d$.

Décodage et cryptanalyse

`decesar(m,d)` → décodage

- Pour décoder si la clé est connue : décalage inverse.
- Pour casser le code (cryptanalyse) :
  - Identifier la lettre la plus fréquente dans le message codé.
  - Comparer avec la lettre E en français pour deviner la clé.

Exercices - Codage de César

- Écrire une fonction `cesar(m,d)` qui code un message m avec un décalage d.
- Écrire une fonction `decesar(m,d)` qui décode un message codé avec la clé d.
- Écrire une fonction `cassecesar(m)` qui décode un message codé sans connaître la clé.
- Tester avec : `"OVRA, NIRP IBHF, IREPVATRGBEVK NHENVG CRHG-RGER TNTAR!"`.

Principe du code de Vigenère

`vigenere(m,cle) → message codé`

- Chaque lettre est décalée selon la lettre correspondante de la clé.
- Exemple : Clé = `CAE`, Message = `TRESOR`
  - T + C(3) = W, R + A(1) = S, E + E(5) = J, S + C(3) = V, ...
- Décodage : mêmes décalages mais en négatif.

Cryptanalyse du code de Vigenère

`cassevigenere(m) → message décodé sans clé`

- Plus difficile que le code de César.
- Étapes possibles pour casser la clé :
  - Identifier les séquences répétées pour estimer la longueur de la clé.
  - Découper le message en sous-séquences codées comme César.
  - Analyser la fréquence des lettres pour chaque sous-séquence.

Exercices - Vigenère

- Écrire `rang(c)` : renvoie le rang d'une lettre.
- Écrire `lettre(r)` : renvoie la lettre correspondant au rang.
- Écrire `vigenere(m,cle)` : code un message *m* avec une clé *cle*.
- Écrire `devigenere(m,cle)` : décode un message *m* avec une clé *cle*.
- Écrire `cassevigenere(m)` : décode un message sans connaître la clé.

Principe du codage de Hill

Codage matriciel `hill_encode(m,M)`

- Utilise une matrice de chiffrement M et un alphabet.
- Découpe le texte en n-grammes (n = taille de la matrice).
- Exemple :

$$M = \begin{pmatrix} 2 & 3 \\ 5 & 7 \end{pmatrix}, \quad \text{Message} = \text{"DCODE"}$$

- Découpage en bigrammes : DC, OD, EZ (ajout de Z pour compléter).
- Conversion en rangs, multiplication par la matrice modulo 26.

Exercices - Hill

- Écrire une fonction qui transforme une chaîne en vecteurs numériques selon l'alphabet.
- Écrire une fonction qui code un message avec une matrice de chiffrement.
- Écrire une fonction qui décode un message codé avec Hill.

À retenir

- Les méthodes de codage sont utiles pour la confidentialité.
- Le codage de César est simple mais vulnérable.
- Vigenère augmente la sécurité grâce à la clé répétitive.
- Hill est basé sur les mathématiques et nécessite une clé matricielle.
- La cryptanalyse permet de décoder sans connaître la clé en utilisant des méthodes analytiques.

06 Traitement d'images

La simulation numérique

Traitement d'images

Anis SAIED

IPEIN

2024/25

Introduction

- Une image numérique peut être en couleurs, en noir et blanc ou en niveaux de gris.
- Une image peut être 2D, 3D (ex : IRM) ou 3D+Temps (film).
- Nous nous concentrons uniquement sur les images 2D enregistrées sur support numérique.

Deux types d'images numériques :

1. **Image vectorielle :** représentée par des formes géométriques (ex : droite, cercle). Agrandissement sans perte de qualité.
2. **Image matricielle :** représentée par une matrice de pixels.

Image matricielle

- Une image matricielle est représentée par une matrice de pixels.

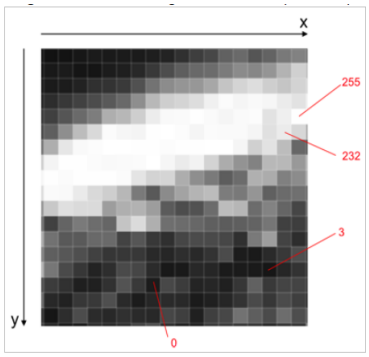


Image en niveaux de gris zoomée

- x : largeur de l'image ou nombre de colonnes
- y : hauteur de l'image ou nombre de lignes

Pixel

- pixel = picture element
- Plus petit élément d'une image
- Contient des informations sur une zone de l'espace numérisé
- Chaque pixel code une couleur codée sur 1, 8 ou 24 bits
- Définit la **quantification des couleurs** ou le **mode de l'image**

Mode d'image

Le mode de l'image est exprimé en nombre de bits par pixel (bpp) :

- **1 bit :** Noir et Blanc (0 = noir, 1 = blanc), mode «1» en Python
- **8 bits :** Niveaux de gris (0 = noir, 255 = blanc), mode «L» en Python
- **24 bits :** Couleurs RVB, chaque composante 0–255, mode «RGB» en Python

Définition et Résolution

- **Définition (Taille de l'image) :** C'est le nombre total de pixels composant l'image ($L \times H$).
  - *Exemple :* Une image Full HD a une définition de $1920 \times 1080 = 2\,073\,600$ pixels (soit environ 2 Megapixels).
- **Résolution (Finesse des détails) :** C'est la densité de pixels sur une unité de mesure physique.
  - Elle s'exprime en **ppp** (points par pouce) ou **dpi** (dots per inch).
  - 1 pouce $= 2,54$ cm.
  - Une résolution plus élevée signifie une image plus nette à l'impression.
- **Poids d'une image non compressée :** Il se calcule en multipliant la définition par la profondeur de codage.

$$\text{Poids (bits)} = \text{Largeur} \times \text{Hauteur} \times \text{bpp}$$

  - Pour obtenir le poids en **Octets**, on divise le résultat par 8.

Module Python de traitement d'image

- **Bibliothèque PIL / Pillow :**
  - `PIL` (*Python Imaging Library*) est la bibliothèque historique.
  - Elle est aujourd'hui maintenue sous le nom **Pillow**.
  - C'est l'outil de référence pour la manipulation simple de fichiers pixels.
- **Installation et Importation :**
  - Installation via console : `pip install Pillow`
  - Dans le script : `from PIL import Image`
- **Lien avec NumPy :** Pillow permet de convertir facilement une image en **tableau NumPy** pour effectuer des calculs matriciels complexes.

Ouverture et affichage d'une image

```
1  from PIL.Image import *
2
3  # Ouverture d'un fichier image
4  tiger = open('tiger.jpg')
5  tiger.show()
```

Modifier mode d'image

```python
# Conversion en niveaux de gris
Img_gris = tiger.convert('L')
Img_gris.show()
Img_gris.save('tigergris.bmp')
```

Manipulation des données et des pixels

1. Création et transfert de données :

```python
# Création d'une image vide : 'RGB', même taille que tiger, noir par défaut
nouveau = new('RGB', tiger.size)

# getdata() extrait tous les pixels sous forme de séquence (objet interne PIL)
# list() convertit cette séquence en une liste Python standard
# putdata() injecte cette liste de pixels d'un coup dans la nouvelle image
nouveau.putdata(list(tiger.getdata()))
nouveau.show()
```

2. Accès et modification individuelle (Pixel par Pixel) :

```python
# load() crée un objet d'accès direct aux pixels (plus rapide en mémoire)
Tabpixel = Img_gris.load()

# getpixel((x,y)) : Récupère la valeur du pixel à la colonne 0, ligne 1
val_p = Img_gris.getpixel((0,1))

# putpixel((x,y), valeur) : Modifie le pixel (0,1) en blanc (255 pour le mode 'L')
Img_gris.putpixel((0,1), 255)

# Alternative rapide avec Tabpixel :
Tabpixel[0, 1] = 255 # Équivalent à putpixel mais optimisé pour les boucles for

Img_gris.show()
```

Copier, redimensionner, transposer et faire pivoter

```python
# Copie
tiger1 = tiger.copy()

# Redimension
tiger1.resize((120,120)).show()

# Transposer horizontal
Img_gris.transpose(FLIP_LEFT_RIGHT).show()

# Rotation 45°
Img_gris.rotate(45).show()
```


Exercices - Niveau 1 : Manipulation d'images

1. **Inversion des niveaux de gris (Effet Négatif) :**
   - Charger une image en mode 'L'. Parcourir chaque pixel et remplacer sa valeur p par son complémentaire $(255 - p)$.
   - *Compétence :* Utilisation de doubles boucles `for` et de `putpixel` ou `load()`.

2. **Ajout d'une bordure d'épaisseur e :**
   - Ajouter une bordure colorée autour d'une image sans écraser le contenu original.
   - *Algorithme :* 1. Calculer les nouvelles dimensions : $(W + 2e, H + 2e)$. 2. Créer une nouvelle image vide de cette taille. 3. "Coller" l'image originale aux coordonnées (e, e).
   - *Compétence :* Gestion du système de coordonnées et création d'un nouveau canevas.

Exercices - Niveau 2 : Dessin Algorithmique

1. **Génération d'un motif en "X" :**
   - Créer une image blanche de 100×100. Tracer deux lignes noires de 1 pixel d'épaisseur reliant les quatre coins.
   - *Logique mathématique :*
     - Diagonale descendante : les pixels où $i = j$.
     - Diagonale ascendante : les pixels où $j = (Taille - 1) - i$.

2. **Défi supplémentaire : Le damier (Optionnel)**
   - Générer un damier de 8×8 carrés alternant noir et blanc.
   - *Indice :* Utiliser l'opérateur modulo % sur les indices de boucle pour alterner les couleurs.

TD : Traitement d'images numériques

PIL & Numpy

Objectifs

- Maîtriser le Type Abstrait *Image* et ses modes (1, L, RGB).
- Implémenter des transformations géométriques et algorithmiques.
- Comprendre les filtres de voisinage (Moyenneur, Convolution, Dilatation).

Exercice 1: Analyse de base et comptage

Écrire un script Python permettant d'ouvrir une image et de compter le nombre de pixels parfaitement noirs (valeur 0) et le nombre de pixels parfaitement blancs (valeur 255) dans celle-ci.

Exercice 2: Création graphique

Écrire un script Python permettant de créer une image carrée de taille 100×100 présentant une croix noire « $\times$ » sur fond blanc joignant les coins opposés.

Exercice 3: Le Damier

Écrire une fonction Python `damier(n, p)` qui affiche l'image d'un damier de dimensions $n \times p$ où les pixels sont alternativement noirs et blancs le long des lignes et des colonnes.

Exercice 4: Transformation de niveaux de gris

Écrire un script permettant d'inverser les niveaux de gris d'une image (calcul du négatif).

Exercice 5: Ajout de bordure

Écrire un script Python permettant d'ajouter autour d'une image une bordure d'épaisseur e pixels. La bordure ne doit pas écraser les pixels existants mais agrandir la taille totale de l'image.

Exercice 6: Agrandissement par duplication

Écrire un script permettant d'agrandir une image d'un facteur 2. Chaque pixel original doit être remplacé par un bloc de 2×2 pixels identiques.

Exercice 7: Compression RLE (Run-Length Encoding)

1. Créer une fonction `compression_RLE(img)` qui renvoie une liste de tuples `(nombre, valeur)` selon la redondance des pixels.
2. Créer la fonction inverse `decompression_RLE(L)` pour reconstruire l'image.

Exercice 8: Filtre moyenneur et Convolution

1. Implémenter `filtre_moyenneur(image, k)` utilisant une fenêtre glissante de taille $(2k+1)$.
2. Écrire une fonction `appFiltreConv` appliquant un masque de convolution 3×3 (ex : filtre de Prewitt pour la détection de contours).

Annexe : Documentation du module PIL (classe Image)

Importation : *`from PIL.Image import *`*

Méthode / Attribut	Description
`open(chemin)`	Ouvre un fichier image et crée un objet Image.
`img.show()`	Affiche l'image dans une fenêtre.
`img.convert(m)`	Change le mode ('1' : B&W, 'L' : Gris, 'RGB' : Couleur).
`new(m, s, c)`	Crée une nouvelle image vide ou colorée.
`img.getpixel((x,y))`	Retourne la valeur du pixel à la position (x, y).
`img.putpixel((x,y),v)`	Modifie le pixel à la position (x, y) avec la valeur v.
`img.size`	Tuple `(largeur, hauteur)` de l'image.
`img.resize((w, h))`	Retourne une copie redimensionnée.
`img.transpose(OP)`	Retourne une copie (ex : `FLIP_LEFT_RIGHT`).
`img.rotate(angle)`	Rotation de l'image en degrés.

À RETENIR

→ **Système de coordonnées :** Le pixel $(0, 0)$ est en haut à gauche.

→ **Dimensions :**

— En haut à droite : $(\text{largeur} - 1, 0)$

— En bas à gauche : $(0, \text{hauteur} - 1)$

— En bas à droite : $(\text{largeur} - 1, \text{hauteur} - 1)$

→ **Codage :**

— *Niveaux de gris :* Un entier entre 0 (noir) et 255 (blanc).

— *Couleurs (RVB) :* Un tuple $(p[0], p[1], p[2])$ représentant le Rouge, Vert, Bleu.

Interaction avec Numpy :

→ `tab = np.array(img)` : Convertit une image PIL en tableau Numpy.

→ `img = Image.fromarray(tab)` : Convertit un tableau Numpy en image PIL.

07

Interpolation de Lagrange

TD : Interpolation polynomiale de Lagrange

Introduction : Interpoler une fonction

Si on a le problème suivant : on dispose d'un tableau de nombres issus d'un calcul, et on veut connaître la fonction qui lie ces points. L'interpolation peut être une solution à ce problème.

Que veut dire interpolation ?

Supposons donc que les couples de mesures dont nous disposons (x_i, y_i) soient des points représentatifs de la courbe d'une fonction f que nous cherchons à déterminer. Nous allons interpoler cette fonction f, c'est-à-dire l'approcher par un polynôme, en s'arrangeant pour que la courbe représentative de ce polynôme passe par tous les points (x_i, y_i) de notre tableau.

Il existe de nombreuses méthodes d'interpolation polynomiale. Nous étudions les polynômes de Lagrange.

Principe de l'interpolation polynomiale de Lagrange

Pour construire les points de la courbe approchée de f par l'interpolation polynomiale à partir de $n+1$ points $M_i(x_i, y_i)$, $\forall i,\ 0 \leq i \leq n$ (avec les x_i distincts deux à deux), il suffit de calculer à chaque fois la valeur d'un polynôme en un x donné $p(x)$.

Comment le faire ?

1. Il existe un unique polynôme P de degré inférieur ou égal à n qui aux abscisses x_i prend les valeurs y_i :

$$P(x_i) = y_i \qquad \forall i,\ 0 \leq i \leq n$$

   — P : polynôme d'interpolation aux points x_i pour les mesures y_i ;
   — x_i : nœuds ou points d'interpolation ;
   — y_i : $y_i = f(x_i)$ représentent les valeurs interpolées.

2. P au point x s'écrit sous la forme :

$$P(X) = \sum_{i=0}^{n} y_i\, l_i(x) \quad \text{avec} \quad l_i(x) = \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{x - x_j}{x_i - x_j}$$

Les l_i sont les polynômes d'interpolation de Lagrange de degré n pour tout i :

$$l_i(X) = \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{X - x_j}{x_i - x_j} = \frac{(X - x_0)\cdots(X - x_{i-1})(X - x_{i+1})\cdots(X - x_n)}{(x_i - x_0)\cdots(x_i - x_{i-1})(x_i - x_{i+1})\cdots(x_i - x_n)}$$

Ainsi, le polynôme de Lagrange au point x se calcule par la formule générale suivante :

$$P(X) = \sum_{i=0}^{n} y_i \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{X - x_j}{x_i - x_j}$$

Exemple

Pour les points $(x_0 = 1, y_0 = 3)$, $(x_1 = -1, y_1 = 2)$, $(x_2 = 2, y_2 = -1)$, on calcule d'abord les polynômes de Lagrange :

$$l_0(X) = \frac{(X+1)(X-2)}{(1+1)(1-2)} = -\frac{1}{2}(X^2 - X - 2)$$

$$l_1(X) = \frac{(X-1)(X-2)}{(-1-1)(-1-2)} = \frac{1}{6}(X^2 - 3X + 2)$$

$$l_2(X) = \frac{(X-1)(X+1)}{(2-1)(2+1)} = \frac{1}{3}(X^2 - 1)$$

Puis on calcule la fonction polynomiale passant par ces points :

$$P(X) = 3l_0(X) + 2l_1(X) - l_2(X)$$

$$P(X) = -\frac{3}{2}X^2 + \frac{1}{2}X + 4$$

Algorithme python de l'interpolation de Lagrange

Données de l'algorithme :

— Le vecteur $X = (x_0, \ldots, x_n)$ formé des points d'interpolation ;

— Le vecteur $Y = (y_0, \ldots, y_n)$ formé des valeurs d'interpolation ;

— Le point réel x auquel le polynôme se calcule.

Résultat : estimation au point x.

Travail à faire

1. Écrire une fonction python `produit` qui retourne le produit des éléments d'une séquence S donnée.
2. Écrire une fonction python `interpolation_Lagrange` qui permet d'implémenter la méthode de Lagrange et retourne une estimation du polynôme de Lagrange au point x donné.
3. Lors d'une expérience, les mesures de la masse volumique de l'eau pure en fonction de la température ont donné les résultats suivants :

Température (°C)	0	5	10	15	20
Masse volumique (kg/m^3)	999.87	999.99	999.73	999.13	998.23

Stocker ces mesures dans deux tableaux : `Temperature` et `MasseVol`.

4. Calculer les estimations du polynôme de Lagrange pour x variant entre 0° et 20° avec un pas égal à 0.1 par deux méthodes différentes :
   (a) En utilisant la fonction implémentée.
   (b) En utilisant la fonction prédéfinie `lagrange(vectx, vecty)` du module `scipy.interpolate`.
5. Tracer sur le même graphique avec grille :
   (a) La courbe des mesures en rouge avec `label = 'mesures'`.
   (b) La courbe des estimations de Lagrange par la méthode implémentée en bleu avec `label = 'lagrange_impl'`.
   (c) La courbe des estimations de Lagrange par la fonction prédéfinie en vert avec `label = 'lagrange_import'`.

Fin du Manuel

Informatique — Niveau : $2^{ème}$ année

Félicitations ! *Vous avez parcouru l'ensemble des chapitres dédiés aux structures de données, à la programmation orientée objet, aux bases de données et au traitement numérique de l'image.*
N'oubliez jamais que ***l'algorithmique est le fruit de la pensée humaine*** *; l'ordinateur n'est que l'outil qui donne vie à votre logique.*

"Le savoir est la seule matière qui s'accroît quand on la partage : n'hésitez jamais à transmettre ce que vous avez appris. On retient ce que l'on partage, mais on finit par oublier ce que l'on cache."

Bonne réussite à vos concours !

Puisse ce parcours vous ouvrir les portes des plus grandes écoles d'ingénieurs.

Anis SAIED — 2024-2025
IPEIN – Institut Préparatoire aux Études d'Ingénieur de Nabeul