



**TORNADE.IO**  
WEB - CYBERSÉCURITÉ



EMILIE

---

BOUT

Co-Fondateur de [tornade.io](#) et de [learning.tornade.io](#)

Docteur en cybersécurité à Tornade.io

[emilie@tornade.io](mailto:emilie@tornade.io)

# Traitement de données avec Python

13/10/2025 - Next-U

# Python avancé pour la donnée.

Partie 1- 120 minutes



- Concevoir des scripts python pour manipuler la donnée.
- Lire, écrire des fichiers de formats variés (CSV,JSON)
- Nettoyer et filtrer des données avec Pandas
- Identifier les limites de Pandas

# Pourquoi le traitement de donnée ?

1. Obtenir des informations exploitables

3. Automatiser des tâches

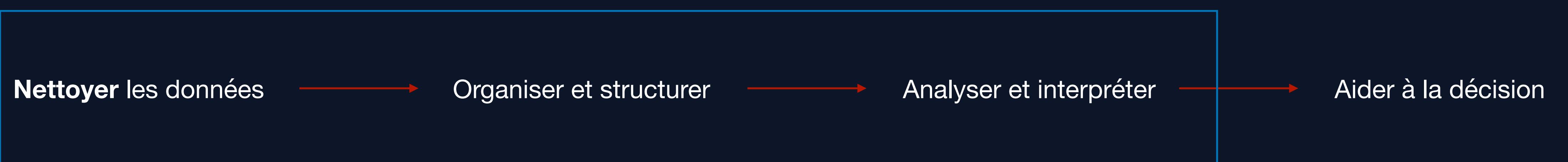
2. Aider à la prise de décision

4. Préparer les données pour l'IA / le machine learning

# Pourquoi le traitement de donnée ?

“Les données brutes nourrissent le traitement,  
le traitement nourrit l’analyse,  
et l’analyse nourrit l’intelligence.”

# Pourquoi le traitement de données ?



# Structures de données avancées

1. Les listes

2. Les tuiles

3. Les Sets

4. Les dictionnaires

# Structures de données avancées

## Les listes:

Une liste permet de stocker plusieurs valeurs (possible différents types) dans une variable.

```
# Création d'une liste simple
fruits = ["pomme", "banane", "cerise"]

# Accès par index
print(fruits[0]) # pomme

# Modification
fruits[1] = "kiwi"

# Ajout d'un élément
fruits.append("mangue")

# Suppression
fruits.remove("cerise")

print(fruits)
```

# Structures de données avancées

## Les tuples:

C'est une liste ordonnée et immuable d'éléments.  
Contrairement à une liste on ne peut modifier son contenu une fois le tuple créé.

## Avantages:

- Optimisation ( moins d'espace mémoire car taille fixe)
- Hashable ( accès aux éléments par clés)

```
# Création d'un tuple
coordonnees = (48.8566, 2.3522)

# Accès par index
print(coordonnees[0]) # 48.8566

# Tuple hétérogène
personne = ("Alice", 25, "Paris")

# Impossibilité de modifier un tuple
personne[1] = 30 ✘ provoque une erreur : TypeError

def min_et_max(liste):
    return (min(liste), max(liste))

bornes = min_et_max([10, 3, 25, 7])

print(bornes) # (3, 25)

# Déballage (unpacking)

mini, maxi = bornes

print(f"Min : {mini}, Max : {maxi} »")
```

# Structures de données avancées

## Les tuples - hashables:

|  
  |  
  | Hashable -> clé de dictionnaire, objet obligatoirement immuable

```
coord = (48.8566, 2.3522)

villes = {coord: "Paris"} # OK

print(villes[(48.8566, 2.3522)]) # Paris
```

# Structures de données avancées

## Les ensembles - set

Un ensemble est une collection non ordonnée,  
mutable et **sans doublons**

```
fruits = {"pomme", "banane", "cerise"}  
  
print(fruits)  
  
fruits.add("kiwi")      # ajout  
  
fruits.remove("banane") # supprime, sinon KeyError  
  
print(fruits)  
  
fruits.add("cerise")  
  
fruits.clear()  
  
print(fruits)
```

# Structures de données avancées

## Les ensembles - set - Opérations ensembliste

Union ( | ) : Combine tous les éléments sans doublons

```
A = {1, 2, 3, 4}  
B = {3, 4, 5, 6}  
  
C = A | B  
  
print(C) # {1, 2, 3, 4, 5, 6}
```

# Structures de données avancées

## Les ensembles - set - Opérations ensembliste

**Intersection ( & )** : Eléments communs aux deux ensembles

```
A = {1, 2, 3, 4}  
B = {3, 4, 5, 6}  
  
C = A & B  
  
print(C) # {3, 4}
```

# Structures de données avancées

## Les ensembles - set - Opérations ensembliste

**Différence ( - ) :** Eléments présents dans la listeA mais pas dans la listeB

```
A = {1, 2, 3, 4}  
B = {3, 4, 5, 6}  
  
C = A - B  
  
print(C) # {1, 2}
```

# Structures de données avancées

## Les ensembles - set - Opérations ensembliste

**Différence symétrique ( ^ ) :** Eléments présents dans la listeA ou la listeB mais pas dans les deux

```
A = {1, 2, 3, 4}  
B = {3, 4, 5, 6}  
  
C = A ^ B  
  
print(C) # {1, 2}
```

# Structures de données avancées

## Les Dictionnaires- dict

Un dictionnaire est une collection non ordonnée, mutable et indexé par clés uniques.

Chaque élément est stocké sous forme *clé: valeur*

```
personne = {"nom": "Alice", "âge": 25, "ville": "Paris"}
```

```
# Accès direct
```

```
print(personne["nom"]) # Alice
```

```
# Accès sécurisé avec get()
```

```
print(personne.get("pays", "France")) # France si la clé  
n'existe pas
```

# Structures de données avancées

## Les Dictionnaires- dict - Ajout et modification

```
# Modification
personne["âge"] = 26

# Ajout d'une nouvelle clé
personne["pays"] = "France"

print(personne)
# {'nom': 'Alice', 'âge': 26, 'ville': 'Paris', 'pays':
'France'}
```

# Structures de données avancées

## Les Dictionnaires- dict - Suppression

```
personne.pop("ville")      # supprime et retourne la valeur  
  
personne.popitem()        # supprime le dernier élément ajouté (Python ≥3.7)  
  
del personne["nom"]       # supprime une clé spécifique  
  
personne.clear()          # vide tout le dictionnaire
```

# Structures de données avancées

## Les Dictionnaires- dict - Itération

```
# itération des clés
for cle in personne:
    print(cle)

#itération des valeurs
for valeur in personne.values():
    print(valeur)

#iteration clé et valeurs:
for cle, valeur in personne.items()
    printtf"{cle} : {valeur}")
```

# Structures de données avancées

## Récapitatif

Tableau 1

Structure	Mutable	Ordonnée	Doublons	Hashable	Usage principal
list	✓	✓	✓	✗	Séquences modifiables
tuple	✗	✓	✓	✓	Données fixes, retour multi valeurs
set	✓	✗	✗	✗	Élimination doublons, opérations ensemblistes
dict	✓	✓	Clés uniques	✗	Stockage clé → valeur, Itération rapide

# Structures de données avancées

## Exercices:

### 1. Liste de toutes les actions

- Stocker le nom de l'action pour **chaque transaction** dans une liste actions\_liste.

### 2. Ensemble d'actions uniques

- Créer un set actions\_uniques à partir de la liste précédente.

### 3. Transactions extrêmes

- Trouver :
  - La transaction avec le **prix le plus élevé**.
  - La transaction avec le **prix le plus bas**.
- Stocker ces transactions dans un **tuple** prix\_extremes = (min\_transaction, max\_transaction)

### 4. Dictionnaire de totaux

- Calculer pour chaque action le **nombre total de titres achetés**.
- Stocker dans un dictionnaire totaux sous forme {"AAPL": 15, "GOOGL": 5, ...}.

```
transactions =  
[ ("AAPL", 10, 145.0),  
 ("GOOGL", 5, 2800.0),  
 ("TSLA", 8, 700.0),  
 (« AAPL», 5, 150.0),  
 ("MSFT", 12, 300.0),  
 ("TSLA", 2, 710.0)]
```

# Structures de données avancées

## Exercices:

Correction:

Transaction est une **liste** [...] de **tuples** ("AAPL", 10, 145.0).

Chaque **tuple** représente **une transaction** (ou une ligne de données).

Chaque élément du tuple contient une valeur différente :

- t[0] → le nom de l'action (chaîne)
- t[1] → la quantité (entier)
- t[2] → le prix (float)

Une transaction est donc une donnée qui ne change pas - **immutable**

```
transactions =  
[ ("AAPL", 10, 145.0),  
  ("GOOGL", 5, 2800.0),  
  ("TSLA", 8, 700.0),  
  ("AAPL", 5, 150.0),  
  ("MSFT", 12, 300.0),  
  ("TSLA", 2, 710.0)]
```

# Structures de données avancées

## Exercices:

### 1. Correction

- Stocker le nom de l'action pour **chaque transaction** dans une liste actions\_liste.

Solution 1:

```
actions_liste = []

for t in transactions:
    actions_liste.append(t[0])

print(actions_liste)
```

Solution 2:

```
actions_liste = [t[0] for t in
transactions]

print(actions_liste)
```

# Structures de données avancées

## Exercices:

### Correction

- Créer un set actions\_uniques à partir de la liste actions\_liste.

## Solution:

```
actions_uniques = set(actions_liste)

print("Actions uniques :", actions_uniques)
```

# Structures de données avancées

## Exercices:

### Correction

#### Transactions extrêmes

- Trouver :
  - La transaction avec le **prix le plus élevé**. -> `max(..., key = )`
  - La transaction avec le **prix le plus bas**. -> `min(... , key = )`

Ici la clé correspond à l'élément 2 d'un tuples ( le prix).

```
max_transaction = max(transactions, key=lambda t: t[2])  
min_transaction = min(transactions, key=lambda t: t[2])
```

Tableau 1

Élément	Rôle
<b>key=</b>	Définit la fonction utilisée pour comparer les éléments
<b>lambda t: t[2]</b>	Prend un tuple et retourne le **3 <sup>e</sup> élément**
<b>max(..., key=...)</b>	Renvoie **l'objet complet** (pas juste la valeur max)

# Structures de données avancées

## Exercices:

### Correction

- Stocker ces transactions dans un **tuple** `prix_extremes = (min_transaction, max_transaction)`

```
prix_extremes = (min_transaction, max_transaction)

print("Transactions extrêmes :", prix_extremes)
```

# Structures de données avancées

## Exercices:

### Correction

#### . Dictionnaire de totaux

- Calculer pour chaque action le **nombre total de titres achetés**.
- Stocker dans un dictionnaire totaux sous forme {"AAPL": 15, "GOOGL": 5, ...}.

.get(key,0): récupère la valeur actuelle associée à la clé si elle existe, sinon retourne 0 .

```
totaux = {}

for t in transactions:
    action, quantite, prix = t
    totaux[action] = totaux.get(action, 0) + quantite

print("Totaux par action :", totaux)
```

# Pandas et NumPy : les fondations de l'analyse de données en Python

**1. Introduction à Pandas**

**2. Manipuler des données avec Pandas**

**3. Manipuler des fichiers volumineux avec Pandas**

# Pandas

- **Pandas ( Panel Data)** est une bibliothèque Python pour **analyser, nettoyer et manipuler des données**, surtout quand elles viennent de fichiers comme CSV, Excel, JSON, SQL, etc.
- Elle est très utilisée en **data science, finance, analyse métier**, etc.
- Open-Source
- Basée sur **Numpy** qui est aussi une bibliothèque mathématique permettant d'implémenter de façon efficace de l'algèbre linéaire et des calculs standards.

Cette bibliothèque permet de :

- Lire / écrire des fichiers (CSV, Excel, JSON...)
- Nettoyer les données (supprimer des lignes, renommer des colonnes...)
- Faire des statistiques (moyennes, sommes, regroupements...)
- Préparer la visualisation des données avec Matplotlib / Seaborn



# Structure de donnée: DataFrame

- **Pandas** utilise comme structure de donnée des Dataframe.
- Grosso-modo un **DataFrame**, c'est une **table en 2 dimensions** (comme une feuille Excel) de ce style où chaque **colonne** est une **Série Pandas** (type pd.Series), comme une liste de valeurs du même type.

Ici une ligne correspond à un enregistrement (ex. une transaction) et une colonne à un attribut / variable (ex. action,prix, quantité).

Tableau 1

action	quantite	prix
AAPL	10	145.0
TSLA	8	700.0
GOOGL	5	2800.0

# Structure de donnée: DataFrame

- Création d'un DataFrame a partir d'une liste de dictionnaire.

Code

```
import pandas as pd

transactions = [
    {"action": "AAPL", "quantite": 10, "prix": 145.0},
    {"action": "GOOGL", "quantite": 5, "prix": 2800.0},
    {"action": "TSLA", "quantite": 8, "prix": 700.0} ]

df = pd.DataFrame(transactions)

print(df)
```

Résultat

	action	quantite	prix
0	AAPL	10	145.0
1	GOOGL	5	2800.0
2	TSLA	8	700.0

# Structure de donnée: DataFrame

- Création d'un DataFrame a partir d'un dictionnaire de listes

Code

```
import pandas as pd

data = {
    "action": ["AAPL", "GOOGL", "TSLA"],
    "quantite": [10, 5, 8],
    "prix": [145.0, 2800.0, 700.0]}

df = pd.DataFrame(data)

print(df)
```

Resultat

	action	quantite	prix
0	AAPL	10	145.0
1	GOOGL	5	2800.0
2	TSLA	8	700.0

# Structure de donnée: DataFrame

- **Création d'un DataFrame a partir d'un fichier CSV et son exportation**

```
import pandas as pd  
  
df = pd.read_csv("transactions.csv")  
  
print(df)  
  
df.to_csv("transactions_clean.csv", index=False)
```

# Structure de donnée: DataFrame

- Les fonctions de bases pour explorer un DataFrame

Tableau 1

Fonction	Description
<code>df.head(n)</code>	Affiche les n premières lignes (par défaut n=5)
<code>df.tail(n)</code>	Affiche les n dernières lignes
<code>df.shape</code>	Dimensions (lignes, colonnes)
<code><u>df.info()</u></code>	Infos sur colonnes, type de données, valeurs non-null
<code>df.describe()</code>	Statistiques descriptives pour colonnes numériques
<code>df.columns</code>	Liste des noms de colonnes
<code>df.index</code>	Indices des lignes

# Structure de donnée: DataFrame

- Les fonctions de bases pour accéder au données

## A) Sélection d'une colonne.

```
df["action"] # retourne une Series  
df.action    # même chose si nom de colonne simple
```

## B) Sélection de plusieurs colonnes

```
df[["action", "prix"]]
```

## C) Sélection par lignes

```
df.iloc[0]      # 1ère ligne par index  
df.iloc[0:2]    # 1ère et 2ème lignes
```

## D) Filtrer avec conditions

```
df[df["prix"] > 500]  
df[(df["prix"] > 500) & (df["quantite"] > 5)]
```

# Structure de donnée: DataFrame

- Les fonctions de bases pour modifier les données

A) Ajouter une colonne

```
df["total"] = df["quantite"] * df["prix"]
```

B) Supprimer une colonne

```
df = df.drop(columns="total")
```

C) Renommer une colonne

```
df = df.rename(columns={"quantite": "volume"})
```

D) Supprimer des lignes

```
df = df.drop(index=1) # supprime ligne d'index 1
```

# Structure de donnée: DataFrame

- Les fonctions essentielles pour le traitement de données

Tableau 1

Fonction	Description
<code>df.dropna()</code>	Supprimer les lignes avec valeurs manquantes
<code>df.fillna(valeur)</code>	Remplacer les valeurs manquantes
<code>df.drop_duplicates()</code>	Supprimer les doublons
<code>df.astype(type)</code>	Changer le type d'une colonne
<code>df.replace(old, new)</code>	Remplacer certaines valeurs
<code>df.str.strip()</code>	Nettoyer les espaces en début/fin de texte (pour les colonnes string)
<code>df.str.lower()/str.upper()</code>	Normaliser les textes
<code>df.duplicated()</code>	Identifier les doublons
<code>df.sort_values(by="colonne")</code>	Trier par une colonne
<code>df.reset_index(drop=True)</code>	Réinitialiser les index
<code>df.astype(str).str.replace(...)</code>	Nettoyage avancé de chaînes

# Pandas et Numpy

- **Sous le capot de Pandas c'est Numpy (Numerical Python) qui permet de stocker les données efficacement. Chaque colonne d'un DataFrame est en réalité un tableau NumPy (pd.Series) sous le capot. De ce fait tu peux appliquer toutes les fonctions de Numpy sur chaque pd.Series (colones) d'un DataFrame.**

```
print(np.sum(df["quantite"]))    # Affiche la somme des quantité  
  
print(np.mean(df["prix"]))       # Affiche la moyenne des prix  
  
print(np.max(df["prix"]))        # Affiche le prix maximum  
  
print(np.min(df["prix"]))        # Affiche le prix minimum
```

# Exercice:

A partir d'un fichier CSV fictif contenant des transactions boursières réaliser les opérations suivantes:

- Charger le CSV dans un DataFrame.
- Afficher les 5 premières lignes et les infos générales (info()).
- Supprimer les doublons.
- Supprimer les lignes avec valeurs manquantes.
- Créer une colonne total = quantite \* prix.
- Ajouter une colonne prix\_log = log(prix) avec NumPy.
- Filtrer pour ne garder que les transactions où prix > 100.
- Trier le DataFrame par prix décroissant.
- Calculer le total des quantités par action.
- Exporter le DataFrame final nettoyé en CSV transactions\_clean.csv

# Exercice:

A partir d'un fichier CSV fictif contenant des transactions boursières réaliser les opérations suivantes:

- Charger le CSV dans un DataFrame.
- Afficher les 5 premières lignes et les infos générales (info()).

```
import pandas as pd
import numpy as np

# Charger le CSV

df = pd.read_csv("transactions.csv")

# Afficher les 5 premières lignes et les infos générales

print("5 premières lignes :")
print(df.head())
print("\nInfos générales :")
print(df.info())
```

# Exercice:

A partir d'un fichier CSV fictif contenant des transactions boursières réaliser les opérations suivantes:

- Supprimer les doublons.
- Supprimer les lignes avec valeurs manquantes et les colonnes entièrement vides si il y a

```
# Supprimer les doublons

df = df.drop_duplicates()
print("\nAprès suppression des doublons :")
print(df)

# Supprimer les colonnes vides

df = df.dropna(axis=1, how="all")
print("\nAprès suppression des colonnes vides :")
print(df)

# Supprimer les lignes avec valeurs manquantes

df = df.dropna()
print("\nAprès suppression des valeurs manquantes :")
print(df)
```

# Exercice:

A partir d'un fichier CSV fictif contenant des transactions boursières réaliser les opérations suivantes:

- **Créer une colonne total = quantite \* prix.**
- **Ajouter une colonne prix\_log = log(prix) avec NumPy.**

```
# Créer une colonne total = quantite * prix

df["total"] = df["quantite"] * df["prix"]
print("\nAprès ajout de la colonne 'total' :")
print(df)

# Ajouter colonne prix_log = log(prix) avec NumPy

df["prix_log"] = np.log(df["prix"])
print("\nAprès ajout de la colonne 'prix_log' :")
print(df)
```

# Exercice:

A partir d'un fichier CSV fictif contenant des transactions boursières réaliser les opérations suivantes:

- **Filtrer pour ne garder que les transactions où prix > 100.**
- **Trier le DataFrame par prix décroissant**

```
# Filtrer pour prix > 100

df = df[df["prix"] > 100]
print("\nAprès filtrage prix > 100 :")
print(df)

# Trier par prix décroissant

df = df.sort_values(by="prix", ascending=False)
print("\nAprès tri par prix décroissant :")
print(df)
```

# Exercice:

A partir d'un fichier CSV fictif contenant des transactions boursières réaliser les opérations suivantes:

- **Calculer le total des quantités par action.**
- **Exporter le DataFrame final nettoyé en CSV transactions\_clean.csv**

```
# Calculer le total des quantités par action

totaux = df.groupby("action")["quantite"].sum()
print("\nTotal des quantités par action :")
print(totaux)

# Exporter le DataFrame final

df.to_csv("transactions_clean.csv", index=False)
print("\nDataFrame final exporté dans 'transactions_clean.csv'")
```

# Manipulations des fichiers volumineux

1. Rappel les formats de fichiers

2. Pandas et les fichiers volumineux

# Rappel les formats de fichiers:

CSV:

- **Comma-Separated Values** (valeurs séparées par des virgules).
- Fichier texte simple où chaque ligne représente un enregistrement et chaque valeur est séparée par un délimiteur (virgule, point-virgule, tabulation...).
- Facile à lire, compatible avec de nombreux logiciels, idéal pour les fichiers pas trop volumineux
- Pas adapté pour des structures complexes.
- Extension: .csv
- 

nom,age,ville
Alice,30,Paris
Bob,25,Lyon

# Rappel les formats de fichiers:

## JSON:

- **JavaScript Object Notation (valeurs séparées par des virgules).**
- **Fichier texte qui stocke des données sous forme d'objets et de listes, idéal pour des données structurées ou hiérarchiques**
- **Gère facilement les données imbriquées (ex: commandes d'un client)**
- **Très utilisé pour les API et lisibles pour les machines et humains facilement**
- **Moins efficace que Parquet pour les fichiers volumineux mais mieux que CVS**
- **Peut consommer beaucoup de mémoire si mal géré**
- **Extension: .json**

```
[  
  
  {“nom”: “Alice”, “age”: 30, “ville”: “Paris”},  
  
  {“nom”: “Bob”, “age”: 25, “ville”: “Lyon”}  
  
]
```

# Rappel les formats de fichiers:

## Parquet:

- Parquet est un format binaire colonne-orienté conçu pour le stockage de données volumineux
- Optimisé pour l'écriture et la lecture rapide
- Très rapide sur les calculs de gros volumes
- Conserve les types de données.
- Moins lisible pour l'humain
- Gère facilement les données imbriquées (ex: commandes d'un client)
- Très utilisé pour les API et lisibles pour les machines et humains facilement
- Moins efficace que Parquet pour les fichiers volumineux mais mieux que CVS
- Peut consommer beaucoup de mémoire si mal géré

# Pandas et fichiers volumineux:

**Un fichier volumineux :**

- Quand on parle de gros fichiers, il s'agit de fichiers dont la taille dépasse la RAM disponible (ex: plusieurs Go).
- Pandas ne peut pas charger tout le fichier en mémoire d'un coup → risque de plantage ou ralentissement

**Objectif : apprendre à traiter ces fichiers avec Pandas par morceaux et à optimiser la mémoire**

# Pandas et fichiers volumineux:

## 1- Les chunks :

- ***chunksize* permet de lire un fichier en morceaux (chunks) de taille définie.**
- **Chaque chunk est un DataFrame Pandas classique.**
- **On peut traiter chaque chunk indépendamment et agréger les résultats.**

```
import pandas as pd

chunksize = 100000 # 100k lignes par chunk

total_rows = 0

for chunk in pd.read_csv("gros_fichier.csv",
chunksize=chunksize):

    # Traitement du chunk
    total_rows += len(chunk)

print(f"Nombre total de lignes : {total_rows}")
```

# Pandas et fichiers volumineux:

## 1- Les chunks :

- Exemple : somme d'une colonne

```
import pandas as pd
total = 0

for chunk in pd.read_csv("gros_fichier.csv", chunksize=50000, usecols=['quantite']):

    total += chunk['quantite'].sum()

print(f"Total : {total}")
```

# Pandas et fichiers volumineux:

## 1- Les chunks :

- Exemple : filtrage

```
chunkszie = 50000

filtered_list = []

for chunk in pd.read_csv("gros_fichier.csv", chunkszie=chunkszie):

    filtered_chunk = chunk[chunk['ville'] == 'Paris']

    filtered_list.append(filtered_chunk)

df_filtered = pd.concat(filtered_list, ignore_index=True)
```

# Pandas et fichiers volumineux:

## 2- Optimisation mémoire - Choisir les bon types :

- Les colonnes int64 → int32 ou int16
- Les colonnes float64 → float32
- Les colonnes textes avec peu de valeurs uniques → category

```
df = pd.read_csv("data.csv",
                 dtype={'col1':'int32','col2':'float32','col3':'category'})
```

# Pandas et fichiers volumineux:

## 2- Optimisation mémoire - Charger seulement les colonnes utiles

```
df = pd.read_csv("data.csv", usecols=['col1','col2'])
```

# Pandas et fichiers volumineux:

## 2- Optimisation mémoire - Libérer la mémoire

- `del df` supprime la **référence** à l'objet dans le namespace actuel ( libérer un DataFrame qui n'est plus utile).

- Forcer le garbage collector

Python utilise un **garbage collector (GC)** pour gérer la mémoire automatiquement.

- Il supprime les objets **qui ne sont plus référencés**.
- Mais parfois, notamment avec des objets volumineux ou des cycles de références, il peut être utile de le forcer. Il parcourt les objets orphelins et les libère physiquement de la RAM.

```
import gc

del df      # supprime la référence

gc.collect() # force le garbage collector à libérer la
            # mémoire
```

# Pandas et fichiers volumineux:

## Bonnes pratiques:

- Toujours limiter les colonnes et lignes si possible.
- Utiliser chunksize pour les fichiers volumineux.
- Convertir les colonnes en types optimisés (int32, float32, category).
- Préférer les formats Parquet ou Feather pour la performance.
- Profiler la mémoire régulièrement :

```
print(df.memory_usage(deep=True))
```

# Pandas et fichiers volumineux:

## Exercices:

Calculer des statistiques et filtrer les données sans charger tout le fichier en mémoire.

- Lecture par chunks ( 100000) et afficher les 5 premières lignes de chaque chunk ainsi que leur taille
- Calculer la quantité totale vendue pour le produit « ProduitA » ( Faire le calcul chunk par chunk et sommer le résultat)  
Résultat attendu: 1673156
- Convertir les colonnes numériques en types optimisés (`int32, float32`)
- Supprimer les chunks après traitement pour éviter la saturation RAM
- Forcer le garbage collector

# Pandas et fichiers volumineux:

## Exercices:

**Calculer des statistiques et filtrer les données sans charger tout le fichier en mémoire.**

- **Lecture par chunks ( 100000) et afficher les 5 premières lignes de chaque chunk ainsi que leur taille**

```
import pandas as pd  
  
chunksize = 100_000  
  
for chunk in pd.read_csv("ventes_gros.csv", chunksize=chunksize):  
  
    print(len(chunk))  
  
    print(chunk.head())
```

# Pandas et fichiers volumineux:

## Exercices:

**Calculer des statistiques et filtrer les données sans charger tout le fichier en mémoire.**

- Calculer la quantité totale vendue pour le produit « ProduitA » ( Faire le calcul chunk par chunk et sommer le résultat)

```
total_quantite = 0

for chunk in pd.read_csv("fichier_volumineux.csv", chunksize=chunksize,
usecols=['Produit','Quantité']):

    filtered = chunk[chunk['Produit'] == 'ProduitA']

    total_quantite += filtered['Quantité'].sum()

print(f"Quantité totale vendue de ProduitA : {total_quantite}")
```

# Pandas et fichiers volumineux:

## Exercices:

Calculer des statistiques et filtrer les données sans charger tout le fichier en mémoire.

- Convertir les colonnes numériques en types optimisés (`int32`, `float32`)

```
for chunk in pd.read_csv("fichier_volumineux.csv", chunksize=chunksizes):  
  
    print(chunk.memory_usage(deep=True))  
  
    chunk['Quantité'] = chunk['Quantité'].astype('int32')  
  
    chunk['Prix_Unitaire'] = chunk['Prix_Unitaire'].astype('float32')  
  
    print(chunk.memory_usage(deep=True))
```

# Pandas et fichiers volumineux:

## Exercices:

**Calculer des statistiques et filtrer les données sans charger tout le fichier en mémoire.**

- Supprimer les chunks après traitement pour éviter la saturation RAM
- Forcer le garbage collector

```
import gc

for chunk in pd.read_csv("fichier_volumineux.csv", chunksize=chunksize):

    # traitement
    del chunk
    gc.collect()
```

# Parquet et PySpark

**1. Introduction à Parquet**

**2. Manipuler des fichiers Parquet  
avec Pandas**

**3. Manipuler des Parquet avec  
PyShark**

**4. Panda Vs PySpark**

# Introduction à Parquet

- Format de fichier
- Parquet est un format de stockage de données colonne-orienté (contrairement à CSV qui est ligne-orienté).
- Il a été développé par Apache pour les systèmes de traitement massif de données (Hadoop, Spark...).
- Il stocke les données de manière binaire et compressée, ce qui le rend rapide et économique en espace.

## Avantages

Aspect	Avantages
Performance	Lecture/écriture <b>beaucoup plus rapides</b> que CSV pour les gros
Compression	Fichiers souvent <b>5 à 10 fois plus petits</b> que les CSV
Types de données	<b>Préserve les types</b> (int, float, bool, datetime) sans conversion
Lecture partielle	Permet de <b>charger seulement certaines colonnes</b> (columns=[...])
Interopérabilité	Compatible avec <b>Pandas, PySpark, Dask, Polars</b> , etc

## Inconvénients

Aspect	Avantages
Lisibilité	<b>Non lisible directement</b> (format binaire)
Dépendances	Nécessite une <b>librairie</b> comme pyarrow ou fastparquet
Débogage	<b>Un peu plus complexe à inspecter</b> qu'un CSV texte

# Manipuler des fichiers Parquet

- Prérequis:
  - Install visual studio code et python  
[Voir fichier markdown « Installation »](#)
  - La manipulation de Parquet nécessite d'avoir pyarrow d'installé -> pip install pyarrow

# Manipuler des fichiers Parquet

- Lecture d'un fichier Parquet

```
import pandas as pd

df = pd.read_parquet("data.parquet", engine="pyarrow")

print(df.head())
```

# Manipuler des fichiers Parquet

- Ecriture d'un fichier Parquet

```
df.to_parquet("data_export.parquet", engine="pyarrow", compression="snappy")
```

- Il existe plusieurs type de compression

Tableau 1

◆ Méthode	⚡ Vitesse de compression	taille de compression	Utilisation CPU	🧠 Avantage principal	⌚ Cas d'usage recommandé	
<b>Snappy</b>		Très rapide	Moyenne	Faible	Excellent compromis entre vitesse et taille	Traitement interactif sur gros volumes
<b>GZIP (zlib)</b>		Moyenne à lente	Élevé	Moyen	Taille réduite, format universel	Archivage, stockage longue durée
<b>Brotli</b>		Lente	Tres eleve	Élevé	Compression maximale	Sauvegarde ou échange sur web
<b>ZSTD (Zstandard)</b>		Rapide	Élevé	Modéré	Très bon équilibre performance/taille	Pipelines de production, Big Data

# Manipuler des fichiers Parquet

- Ecriture d'un fichier Parquet

```
df.to_parquet("data_export.parquet", engine="pyarrow", compression="snappy")
```

- Il existe plusieurs type de compression

Tableau 1

◆ Méthode	⚡	Vitesse de compression	Taux de compression	Utilisation CPU	🧠 Avantage principal	⌚ Cas d'usage recommandé
<b>Snappy</b>		Très rapide	Moyenne	Faible	Excellent compromis entre vitesse et taille	Traitement interactif sur gros volumes
<b>GZIP (zlib)</b>		Moyenne à lente	Élevé	Moyen	Taille réduite, format universel	Archivage, stockage longue durée
<b>Brotli</b>		Lente	Tres eleve	Élevé	Compression maximale	Sauvegarde ou échange sur web
<b>ZSTD (Zstandard)</b>		Rapide	Élevé	Modéré	Très bon équilibre performance/taille	Pipelines de production, Big Data

# Manipuler des fichiers Parquet

- Lecture selective d'un fichier Parquet

```
df_partial = pd.read_parquet("data_export.parquet", columns=["Produit", "Quantité"])
```

- Toujours dans le but de réduire la consommation de mémoire lors de l'ouverture d'un gros fichier. Pandas ne lit que les colonnes demandées, ce qui économise énormément de mémoire.

# Manipuler des fichiers Parquet

- Exercices:

A partir du fichier data.parquet réaliser un script permettant de calculer le montant de chaque vente et de faire une agrégation par ville en veillant à optimiser les ressources au maximum.

Voici la forme du fichier data.parquet

```
● emiliebout@MacBook-Air-de-Emilie Analyse de donnees % python3 €
   Produit  Quantite    Prix      Ville        Date
  0       B          6  21.70      Lyon  2025-01-01 00:00:00
  1       C          3  16.59  Marseille 2025-01-01 00:01:00
  2       A          2  37.52  Marseille 2025-01-01 00:02:00
  3       C          8  39.48  Marseille 2025-01-01 00:03:00
  4       B          7  42.58  Marseille 2025-01-01 00:04:00
```

# Manipuler des fichiers Parquet

- Exercices:

- 1 ère étape: Lire le fichier `data.parquet` en ne sélectionnant uniquement les colonnes utiles

```
import pandas as pd

# 1 Étape 1 : Lecture du fichier Parquet

# Lecture uniquement des colonnes nécessaires pour économiser la mémoire

colonnes_utiles = ["Produit", "Quantite", "Prix", "Ville"]

df = pd.read_parquet("data.parquet", columns=colonnes_utiles)
```

# Manipuler des fichiers Parquet

- Exercices:
  - 2 eme étape: optimiser les données lues tout en conservant le type de donnée.

```
# ❷ Étape 2 : Optimisation des types de données
```

```
df["Quantite"] = df["Quantite"].astype("int32")
```

```
df["Prix"] = df["Prix"].astype("float32")
```

# Manipuler des fichiers Parquet

- Exercices:

- 3 ème étape: Calcul du montant de chaque vente

```
# 3 Étape 3 : Calcul du montant de chaque vente  
  
df["montant_total"] = df["quantite"] * df["prix_unitaire"]
```

# Manipuler des fichiers Parquet

- Exercices:

- 4 ème étape: Agrégation par ville

```
# 4 Étape 4 : Agrégation par ville

ca_par_ville = df.groupby("Ville")["montant_total"].sum().reset_index()

print(ca_par_ville)
```

# Manipuler des fichiers Parquet

- Exercices:
  - 5 ème étape: Sauvegarder les résultats dans un fichier parquet

```
# 5 Étape 5 : Sauvegarde du résultat optimisée en Parquet compressé
```

```
ca_par_ville.to_parquet("ca_par_ville.parquet", compression="snappy")
```

# Manipuler des fichiers Parquet

- Comparaison CVS vs Parquet ( données identiques)

```
import time
import pandas as pd

start = time.time()

pd.read_parquet("data.parquet")
print("Lecture Parquet :", round(time.time() - start, 3), "secondes")
```

```
start=0
start = time.time()
pd.read_csv("data.csv")
print("Lecture CSV :", round(time.time() - start, 3), "secondes")
```

```
● emiliebout@MacBook-Air-de-Emilie Analyse de donnees % python3
Lecture Parquet : 0.017 secondes
Lecture CSV : 0.042 secondes
```

# Manipuler des fichiers Parquet

- Comparaison CSV vs Parquet

En résumé

Tableau 1

Format	taille ( 1M de ligne)	Temps de lecture	Types conservés
CSV	~70 Mo	~3-5 secondes	Non
Parquet	~10-15 Mo	~0.5 secondes	Oui

# PySpark

**Spark** : créé en 2009 il va permettre d'accélérer le traitement des données massives, tout en gardant la flexibilité et l'interopérabilité.

Spark utilise le **calcul distribué en mémoire** :

- Les données sont **chargées en RAM** sur plusieurs machines (clusters)
  - Les calculs se font directement en mémoire plutôt que d'écrire chaque étape sur le disque
- 
- Résultat : **exécution beaucoup plus rapide** (10 à 100 fois plus rapide que MapReduce - l'ancêtre de Spark - sur certaines tâches).

Spark est écrit en Scala mais Python étant le langage devenu par défaut le plus populaire pour le traitement de données, la librairie PySpark a été créée.

**PySpark** est donc une API Python permettant d'utiliser **Spark en Python**.

Elle permet donc de **traiter de très gros volumes de données** en parallèle sur plusieurs machines ou cœurs.

- Idéal pour :
  - ETL (Extract, Transform, Load)
  - Analyse de données massives
  - Machine Learning avec **MLlib**
  - Graph Processing et Streaming

# PySpark

Pre-requis:

- Installer pyspark : pip install pyspark

Pour vérifier l'installation:

```
import pyspark

from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("Test").getOrCreate()

print(spark.version)
```

# PySpark

## Les SparkSession

**SparkSession** est l'**objet principal** pour interagir avec Spark à partir de PySpark.

- C'est le **point d'entrée unique** pour toutes les opérations sur **DataFrame, SQL, RDD et Streaming**.
- Il a été introduit à partir de **Spark 2.0**, pour **simplifier l'ancienne API** (avant, il fallait créer SparkContext, SQLContext et HiveContext séparément).
- Ne créer qu'une seule SparkSession par application PySpark.
- Fermer la session à la fin si nécessaire.
- Utiliser la SparkSession pour **toutes les lectures/écritures et transformations** pour garantir la cohérence et optimiser les ressources.

**SparkSession = porte d'entrée unique de PySpark** pour manipuler DataFrame, RDD, SQL et Streaming, tout en configurant l'environnement Spark.

```
import pyspark
from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("Test").getOrCreate()

spark.stop()
```

# PySpark

## Les DataFrame avec PySpark

Un **DataFrame PySpark** est une **structure de données distribuée** (comme un DataFrame Pandas, mais réparti sur plusieurs machines)

- Chaque **partition** d'un DataFrame est traitée par un nœud du cluster Spark.
- Les opérations (filtres, agrégations, jointures...) sont **parallélisées** automatiquement.

Création d'un DataFrame a partir d'un tableau

```
data = [(
    ("Alice", 34, "Paris"),
    ("Bob", 45, "Lyon"),
    ("Chloé", 29, "Marseille")]

colonnes = ["Nom", "Age", "Ville"]

df = spark.createDataFrame(data, colonnes)

df.show()
```

Résultat

```
+-----+-----+
| Nom | Age | Ville |
+-----+-----+
| Alice | 34 | Paris |
| Bob | 45 | Lyon |
| Chloé | 29 | Marseille |
+-----+-----+
```

# PySpark

## Les DataFrame avec PySpark

Création d'un DataFrame a partir d'un fichier CSV

```
df_csv = spark.read.csv("data.csv", header=True, inferSchema=True)
```

Création d'un DataFrame a partir d'un fichier parquet

```
df_parquet = spark.read.parquet("data.parquet")
```

# PySpark vs Pandas

Tableau 1

Critère	Pandas	PySpark
<b>But principal</b>	Traitement de données sur un seul ordinateur	Traitement de données massives distribuées sur plusieurs machines (Big Data)
<b>Taille des données</b>	Quelques Mo à Quelques Go	Plusieurs Go à plusieurs To
<b>Stockage / calcul</b>	En mémoire (RAM)	Distribué, Spark gère le parallélisme automatiquement
<b>Performance</b>	Rapide pour dataset petit/moyen	Optimisé pour datasets très volumineux
<b>API</b>	pandas DataFrame, numpy array	pyspark.sql.DataFrame, RDD
<b>Syntaxe</b>	Similaire à SQL mais orienté objet	Très proche de pandas pour les DataFrames, mais certaines opérations diffèrent (ex: `select`, `withColumn`, `groupBy`)
<b>Parallelisme</b>	Manuel ou via processing	Automatique, gestion du cluster incluse
<b>Visualisation</b>	Matplotlib, seaborn, plotly	Limité à de petits échantillons (puis matplotlib/seaborn)
<b>ML / IA</b>	Scikit-learn, TensorFlow	Mlib (Spark ML), pipelines ML distribués
<b>Cas typique</b>	Analyse locale, prototype	Big Data, logs massifs, analytics distribués, préparation pour IA sur gros volumes
<b>Installation</b>	Simple	Plus complexe (`pyspark`, configuration du cluster possible)

# MERCI !

FIN

[emilie@TORNADE.IO](mailto:emilie@TORNADE.IO)



**TORNADE.IO**

WEB - CYBERSÉCURITÉ