MODULE 1 : Bootcamp Python Data & Setup

Partie 1 : L'environnement de travail professionnel

1.1 Pourquoi ces outils ?

Outil	À quoi ça sert ?	Pourquoi c'est indispensable ?
VS Code	Éditeur de code	Gratuit, extensions Python puissantes, debugger intégré, standard industriel
Git	Versioning du code	Sauvegarde l'historique, travail en équipe, déploiement automatisé
Docker Desktop	Conteneurisation	"Ça marche sur mon PC" → "Ça marche partout". Reproductibilité totale
Conda	Gestion environnements	Isole les projets (évite conflits de versions), gère Python + librairies système

Analogie : Imaginez que vous cuisinez.

> Conda = cuisine séparée par projet (pas de mélange sucre/sel).
> Git = carnet de recettes qui mémorise chaque version.
> Docker = lunchbox qui garde le repas identique partout où vous allez.

1.2 Installation et premier pas

Étape 1 : Vérifier l'installation Python

```
# Dans VS Code, terminal (Ctrl+J), tapez : python --version
# Résultat attendu : Python 3.11.x ou 3.12.x
```

Étape 2 : Créer un environnement Conda (isolation totale)

```
# TERMINAL - Ligne 1
conda create -n cours_data_ia python=3.11
```

Explication : Crée un environnement nommé `cours_data_ia` avec Python 3.11. C'est comme créer une pièce séparée dans votre maison où vous installerez UNIQUEMENT les outils de ce cours. Si vous cassez quelque chose ici, le reste de votre ordinateur est protégé.

```
# TERMINAL - Ligne 2
conda activate cours_data_ia
```

Explication : Entre dans la pièce. Votre terminal indique maintenant `(cours_data_ia)` au début. Tout ce que vous installerez maintenant restera dans cette pièce.

Dans VS Code → Sélectionner le bon interpréteur

Dans VS Code :

- Ctrl + Shift + P
- Tapez : Python: Select Interpreter
- Choisissez : Python 3.11 (cours_data_ia)

```
# TERMINAL - Ligne 3
conda install pandas polars pyarrow jupyter

pip install pandas polars pyarrow jupyter
```

Explication : Installe 4 outils essentiels :

- **`pandas`** : manipulation de données (Excel en Python)
- **`polars`** : pandas ultra-rapide (même usage, 10x plus vite)
- **`pyarrow`** : format de données moderne (utilisé par polars et Spark)
- **`jupyter`** : notebooks interactifs (pour expérimenter)

1.3 Premier contact avec Python

Créez un fichier `test_environment.py` dans VS Code :

```
# LIGNE 1 : Importation des librairies
import pandas as pd       # 'pd' = alias conventionnel pour pandas
import polars as pl       # 'pl' = alias pour polars
import sys                # Module système de Python (infos sur l'environnement)

# LIGNE 5 : Vérification des versions (évite les bugs de compatibilité)
print("=" * 50)  # Affiche 50 fois le caractère "=" (séparateur visuel)
print("VÉRIFICATION DE L'ENVIRONNEMENT")
print("=" * 50)

# LIGNE 10 : f-string = formatage de texte moderne en Python
# {sys.version} = insère la version de Python automatiquement
print(f"Python version : {sys.version}")

# LIGNE 13 : Vérification que pandas est bien installé
# pd.__version__ = attribut interne de pandas contenant sa version
print(f"Pandas version : {pd.__version__}")

# LIGNE 16 : Vérification polars
print(f"Polars version : {pl.__version__}")

# LIGNE 19 : Message de succès
print("\n Environnement prêt ! Vous pouvez commencer le cours.")
```

Explication:

`import` charge des fonctionnalités externes. **`as`** crée un raccourci. Au lieu de taper **`pandas.DataFrame`**, vous taperez **`pd.DataFrame`**

`"=" * 50` multiplie la chaîne "=" 50 fois. C'est du Python pur, pas spécifique aux données
`f"..."` = f-string (format string). Permet d'insérer des variables directement dans le texte avec `{}`
`__version__` = convention Python pour les métadonnées internes d'un package

Pour exécuter : `python test_environment.py`

Partie 2 : Les structures de données fondamentales

2.1 Rappel : Les types Python de base

Avant de manipuler des données massives, maîtrisez ces 4 types :

```
# Type 1 - NOMBRE (integer)
age = 29
# 'age' est le nom de la boîte, 29 est la valeur entière (int)

# Type 2 - TEXTE (string, toujours entre guillemets)
nom = "Abdelali El Gourari"
# Les guillemets délimitent le texte. "29" (texte) ≠ 29 (nombre)

# Type 3 - LISTE (collection ordonnée, modifiable)
notes = [15, 18, 12, 14]
# Crochets [] = liste. Index commence à 0 : notes[0] = 15, notes[1] = 18

# Type 4 - DICTIONNAIRE (clé-valeur, comme un formulaire)
etudiant = {
    "nom": "Ali Mouhou",        # Clé "nom", valeur "Ali Mouhou"
    "age": 25,                  # Clé "age", valeur 25
    "notes": [15, 18, 12, 14]   # Clé "notes", valeur = une liste
}
# Accès : etudiant["nom"] retourne "Ali Mouhou"

# Vérification
print(f"Type de 'age' : {type(age)}")           # <class 'int'>
print(f"Type de 'nom' : {type(nom)}")           # <class 'str'>
print(f"Type de 'notes' : {type(notes)}")       # <class 'list'>
print(f"Type de 'etudiant' : {type(etudiant)}") # <class 'dict'>

# Accès aux données
print(f"\nPremière note : {notes[0]}")          # Index 0 = premier élément
print(f"Nom de l'étudiant : {etudiant["nom"]}") # Accès par clé
```

Pourquoi c'est crucial ? : Pandas et Polars utilisent ces structures en interne. Un DataFrame n'est qu'une **liste de dictionnaires optimisée**.

2.2 Premier vrai traitement : Pandas (le classique)

Créez `premier_pipeline.py` :

```
# Importation
import pandas as pd  # On importe la librairie de manipulation de données

# CRÉATION DE DONNÉES BRUTES (simulation d'un fichier CSV)
```

```python
# Liste de dictionnaires = structure typique des données tabulaires
donnees_brutes = [
    {"client_id": 1, "nom": "Alice", "achat": 150.0, "ville": "Paris"},
    {"client_id": 2, "nom": "Bob", "achat": 0.0, "ville": "Lyon"},
                # 0 = problème
    {"client_id": 3, "nom": "Charlie", "achat": 230.5, "ville": None},
                # None = donnée manquante
    {"client_id": 4, "nom": "Diana", "achat": -50.0, "ville": "Paris"},
                # Négatif = erreur
    {"client_id": 5, "nom": "Eve", "achat": 89.0, "ville": "Marseille"},
]

# CRÉATION DU DATAFRAME (tableau de données)
# pd.DataFrame() convertit la liste de dict en tableau structuré
df = pd.DataFrame(donnees_brutes)

# EXPLORATION RAPIDE (indispensable en data)
print("=" * 50)
print("DONNÉES BRUTES")
print("=" * 50)
print(df)   # Affiche le tableau complet
print(f"\nDimensions : {df.shape}")   # shape = (nb_lignes, nb_colonnes)
# Résultat : (5, 4) = 5 clients, 4 colonnes

# DÉTECTION DES PROBLÈMES
print("\n" + "=" * 50)
print("ANALYSE DE QUALITÉ")
print("=" * 50)

# df.isnull() = tableau booléen (True si valeur manquante)
# .sum() compte les True par colonne
print(f"Valeurs manquantes :\n{df.isnull().sum()}")

# df.describe() = statistiques descriptives automatiques
print(f"\nStatistiques :\n{df.describe()}")

# NETTOYAGE DES DONNÉES (Data Cleaning)

# Étape 1 - Supprimer les achats négatifs (incohérents)
# df['achat'] sélectionne la colonne
# >= 0 crée un masque booléen (lignes à garder)
df_clean = df[df['achat'] >= 0].copy()   # .copy() évite les warnings
# Explication : On garde uniquement les lignes où achat >= 0

# Étape 2 - Remplacer les valeurs manquantes
# .fillna() = "fill NA (Not Available)"
df_clean['ville'] = df_clean['ville'].fillna("Inconnue")
# Charlie avait ville=None, maintenant ville="Inconnue"

# Étape 3 - Créer une nouvelle colonne (feature engineering)
# Calcul du montant TTC (TVA 20%)
df_clean['achat_ttc'] = df_clean['achat'] * 1.20
# '*' multiplie chaque valeur de la colonne par 1.20

# RÉSULTAT FINAL
print("\n" + "=" * 50)
print("DONNÉES NETTOYÉES")
print("=" * 50)
print(df_clean)
```

```python
# EXPORT (pour le module suivant)
# Parquet = format moderne, compressé, rapide (remplace CSV)
df_clean.to_parquet("clients_nettoyes.parquet", index=False)
# index=False = ne pas sauvegarder la colonne d'index (0,1,2,3)
print("\n Fichier sauvegardé : clients_nettoyes.parquet")
```

Explications détaillées des opérations clés :

Concept	Explication
DataFrame	Structure tabulaire 2D (lignes = observations, colonnes = variables). C'est Excel en Python.
isnull()	Détecte les "trous" dans les données. Critique : les modèles IA détestent les valeurs manquantes.
Filtrage	`df[condition]` garde uniquement les lignes où condition = True. Ici, on élimine les erreurs de saisie (négatifs).
fillna()	Stratégie de "imputation" (remplissage). Alternative : supprimer la ligne, ou remplacer par la moyenne.
Feature Engineering	Création de nouvelles informations à partir des existantes. Ici, on ajoute la TVA. Essentiel pour l'IA.
Parquet	Format colonne-orienté (vs ligne pour CSV). 10x plus rapide, 5x moins volumineux. Standard Big Data.

2.3 Passer à la vitesse supérieure : Polars (le moderne)

Même traitement, **10x** plus rapide (utile pour gros volumes) :

```python
# Importation
import polars as pl  # Polars = concurrent moderne de pandas, écrit en Rust

# Mêmes données brutes
donnees_brutes = [
    {"client_id": 1, "nom": "Alice", "achat": 150.0, "ville": "Paris"},
    {"client_id": 2, "nom": "Bob", "achat": 0.0, "ville": "Lyon"},
    {"client_id": 3, "nom": "Charlie", "achat": 230.5, "ville": None},
    {"client_id": 4, "nom": "Diana", "achat": -50.0, "ville": "Paris"},
    {"client_id": 5, "nom": "Eve", "achat": 89.0, "ville": "Marseille"},
]

# CRÉATION DU DATAFRAME POLARS
# pl.DataFrame = même concept, implémentation différente
df = pl.DataFrame(donnees_brutes)

# SYNTAXE POLARS - Méthodes enchaînées (fluent API)
```

```
# L'avantage : Polars optimise l'exécution automatiquement
df_clean = (
    df
    # Filtrage (même logique que pandas)
    .filter(pl.col("achat") >= 0)   # pl.col() sélectionne une colonne

    # Remplacement des nulls
    .with_columns(
        pl.col("ville").fill_null("Inconnue")
        # with_columns = ajoute/modifie colonnes
    )

        # Création de la colonne TTC
    .with_columns(
        (pl.col("achat") * 1.20).alias("achat_ttc")
        # .alias() nomme la nouvelle colonne
    )
)

# AFFICHAGE
print("Résultat avec Polars :")
print(df_clean)

# Export identique
df_clean.write_parquet("clients_nettoyes_polars.parquet")
print("\n Export Polars réussi")
```

Différences clés Pandas vs Polars :

Aspect	Pandas	Polars
Syntaxe	`df[df['col'] > 0]`	`df.filter(pl.col("col") > 0)`
Performance	1x (référence)	10-50x plus rapide
Mémoire	Copie les données souvent	"Lazy evaluation" (optimisé)
Quand l'utiliser	< 100k lignes, prototypage	> 1M lignes, production

Partie 3 : Git — Sauvegarder son travail comme un pro

3.1 Pourquoi Git est non-négociable ?

Imaginez : vous travaillez 3h sur un pipeline, vous supprimez une ligne par erreur, tout casse. **Sans Git** : vous pleurez. **Avec Git** : vous revenez en arrière en 10 secondes.

3.2 Les commandes essentielles (à taper dans le terminal)

```
# ÉTAPE 1 : Initialiser un repository (une fois par projet)
git init
```

```
# Crée un dossier caché .git qui mémorise TOUT l'historique

# ÉTAPE 2 : Voir l'état actuel
git status
# Affiche : fichiers modifiés, fichiers nouveaux, fichiers prêts à être sauvegardés

# ÉTAPE 3 : Ajouter des fichiers à la "zone de staging" (préparation)
git add test_environment.py
git add premier_pipeline.py
# Ou pour tout ajouter : git add .

# ÉTAPE 4 : Créer un "commit" (point de sauvegarde nommé)
git commit -m "Module 1 : Setup environnement et premier pipeline"
# -m = message décrivant ce que vous avez fait
# Règle d'or : message clair, au présent, impératif

# ÉTAPE 5 : Voir l'historique
git log --oneline
# Affiche : a1b2c3d Module 1 : Setup environnement et premier pipeline

# ÉTAPE 6 (si besoin) : Revenir en arrière
git checkout a1b2c3d   # Retourne au commit a1b2c3d

# Ou annuler les modifications non commitées :
git restore premier_pipeline.py
```

ÉTAPE 7 : Vérifier la branche

```
git branch
```

Normalement :

```
* main
```

ou

```
* master
```

ÉTAPE 8 : Vérifier le remote

```
git remote -v
```

Si rien ne s'affiche → vous devez connecter le dépôt distant.

Si le remote n'est pas encore configuré

1. Créez un repository vide sur GitHub (ex : architectures-donnees)
2. Puis ajoutez le remote :

```
git remote add origin https://github.com/VOTRE_USER/architectures-donnees.git
```

ÉTAPE 9 : Push vers GitHub

Si c'est la première fois :

```
git push -u origin main
```

ou si votre branche est master :

```
git push -u origin master
```

➔ -u permet de lier la branche locale au remote (une seule fois).

Si Git demande authentification

GitHub n'accepte plus mot de passe simple.
Il faut :

- Soit utiliser GitHub Desktop
- Soit utiliser un **Personal Access Token**
- Soit configurer SSH

Vérification finale
Allez sur GitHub → vous devez voir vos fichiers :

- test_environment.py
- premier_pipeline.py

Workflow normal ensuite
Après modification :

```
git add .
git commit -m "Ajout preprocessing"
git push
```

➔ (plus besoin de -u)

Analogie : Git est comme un **système de sauvegarde dans un jeu vidéo**. Vous créez des points de sauvegarde (commits) avant les boss difficiles (changements risqués). Vous pouvez recharger n'importe quelle sauvegarde.

Partie 4 : Docker — L'environnement reproductible

4.1 Le problème résolu par Docker

"Mais ça marchait sur mon ordinateur !" — Phrase la plus entendue en Data.

*Docker encapsule votre application + ses dépendances dans une **boîte isolée** qui fonctionne identiquement sur tous les ordinateurs.*

4.2 Premier Dockerfile (fichier de configuration)

Créez un fichier nommé `Dockerfile` (sans extension) :

```
# Image de base = système d'exploitation minimal avec Python
FROM python:3.11-slim
# 'FROM' = on part de l'image officielle Python 3.11 (version légère 'slim')

# Définition du répertoire de travail dans le conteneur
WORKDIR /app
# Toutes les commandes suivantes s'exécuteront dans /app

# Copie du fichier de dépendances (on le crée juste après)
COPY requirements.txt .
# 'COPY' copie depuis votre PC vers le conteneur
# Le point '.' = répertoire actuel (/app)

# Installation des librairies Python
RUN pip install --no-cache-dir -r requirements.txt
# 'RUN' exécute une commande lors du BUILD de l'image
# --no-cache-dir = évite de stocker les fichiers temporaires (allège l'image)
```

```
# Copie de tout votre code source
COPY . .
# Copie tous les fichiers du dossier courant vers /app

# Commande par défaut lors du démarrage du conteneur
CMD ["python", "premier_pipeline.py"]
# 'CMD' = ce qui s'exécute quand vous lancez le conteneur
```

Créez `requirements.txt` (liste des dépendances) :

```
pandas==2.1.4
polars==0.20.3
pyarrow==14.0.1
```

4.3 Construction et exécution

```
# ÉTAPE 1 : Construire l'image (créer la boîte)
docker build -t mon-premier-pipeline:v1 .
# -t = tag (nom de l'image)
# . = utilise le Dockerfile dans le répertoire courant

# ÉTAPE 2 : Vérifier que l'image existe
docker images
# Affiche : mon-premier-pipeline | v1 | 850MB | il y a 2 minutes

# ÉTAPE 3 : Exécuter le conteneur (lancer la boîte)
docker run mon-premier-pipeline:v1
# Vous voyez le résultat de votre script Python s'afficher !

# ÉTAPE 4 : Exécuter avec un volume partagé (pour récupérer les fichiers générés)
docker run -v $(pwd)/output:/app/output mon-premier-pipeline:v1
# -v = monte un dossier local (output) dans le conteneur (/app/output)
# Les fichiers créés dans /app/output seront visibles dans ./output sur votre PC
```

Exercice d'évaluation du Module 1

Mission : Créer un pipeline complet qui :

1) **Lit** un fichier CSV `ventes.csv`
2) **Nettoie** : supprime les prix négatifs, remplit les catégories manquantes par "Non classé"
3) **Transforme** : calcule le prix total (quantité × prix unitaire) et applique une remise de 10% si quantité > 10
4) **Exporte** en Parquet compressé
5) **Versionne** avec Git (minimum 3 commits clairs)
6) **Containerise** avec Docker (l'image doit s'exécuter sans erreur)

Critères de réussite :

- Code Python exécutable sans erreur
- Commentaires expliquant chaque étape
- Historique Git propre (`git log` montre la progression)
- `docker run` produit le fichier Parquet attendu

Récapitulatif des concepts acquis

Concept	Définition simple	Pourquoi c'est important pour l'IA
Environnement Conda	Pièce isolée pour chaque projet	Évite les conflits entre versions de librairies ML
DataFrame	Tableau Excel programmable	Structure universelle pour préparer les données d'entraînement
Parquet	Format de fichier compressé, rapide	Réduit les coûts de stockage cloud, accélère le chargement des datasets
Git	Historique des modifications	Collaboration en équipe, reproductibilité des expériences ML
Docker	Boîte isolée reproductible	Déployer un modèle IA sur un serveur distant sans surprise