Sujet jour 2 : Lire et écrire des fichier, découverte de *pandas*

Sommaire

Partie 0 : Prérequis

Lors de cette journée, nous approfondirons la manipulation de divers types de données en Python en utilisant la bibliothèque pandas. Avant de commencer, vous aurez besoin de configurer un environnement virtuel Python, de créer un fichier « requirements.txt » et d'installer pandas. Cette configuration garantit que toutes les dépendances nécessaires pour la journée soient installées selon les standards Python.

1. Configuration d'un environnement virtuel

Le module venv permet de créer des « environnements virtuels » légers, chacun avec son propre ensemble indépendant de paquets Python installés. Un environnement virtuel est créé sur la base d'une installation Python existante, et peut éventuellement être isolé des paquets de l'environnement de base, de sorte que seuls ceux explicitement installés dans l'environnement virtuel soient disponibles.

Lorsqu'il est utilisé à partir d'un environnement virtuel, les outils d'installation tels que pip installeront des paquets Python dans un environnement virtuel sans qu'il soit nécessaire de leur indiquer explicitement de le faire.

a. Créer un environnement virtuel

Peu importe votre OS, ouvrez un terminal et naviguez jusqu'à votre répertoire de projet et exécutez : « python3 -m venv .venv ».
Cette commande crée un nouveau dossier appelé « .venv » contenant l'environnement virtuel.

b. Activer l'environnement virtuel

Dans le même terminal, exécutez :

- **Windows** : « .venv\Scripts\activate »
- **macOS/Linux** : « source .venv/bin/activate »

Après activation, votre terminal affichera généralement le nom de l'environnement virtuel (dans ce cas, « .venv »), et vous pourrez commencer à l'utiliser pour installer des paquets.

Note : Si vous utilisez PyCharm, lors de la création d'un nouveau projet, vous avez le choix de créer automatiquement un nouvel environnement virtuel pour celui-ci. L'environnement virtuel sera automatiquement activé, que ce soit lorsque vous installez des paquets, ouvrez un terminal ou exécutez votre code.

2. Création et utilisation d'un fichier « requirements.txt »

a. requirements.txt

Le fichier « requirements.txt » est utilisé pour conserver les détails des dépendances nécessaires pour exécuter le projet. Il facilite l'installation de ces dépendances sur d'autres machines ou environnements.

Créez un fichier « requirements.txt » dans le répertoire de votre projet. Ouvrez-le dans un éditeur de texte et ajoutez la ligne suivante : « pandas ».
Il est possible de spécifier des versions précises si nécessaire, par exemple « pandas==2.2.0 ». Vous pouvez trouver le format complet dans la documentation.

Pandas est une librairie Python « open source » fournissant des structures de données performantes et faciles à utiliser ainsi que des outils d'analyse de données.

b. Installer les dépendances

Avec votre environnement virtuel activé, installez les paquets requis en exécutant « python3 -m pip install -r requirements.txt »

c. Vérification de l'installation

Assurez-vous que pandas est correctement installé en essayant de l'importer dans une console Python :

- Lancez l'interpréteur Python en exécutant « python3 » dans un terminal
- Entrez les lignes suivantes: « import pandas as pd », « print(pd.__version__) »

Cela devrait afficher le numéro de version de pandas sans erreur, confirmant que tout est installé et fonctionne correctement.

En suivant ces étapes, vous avez configuré un environnement virtuel Python avec pandas installé. Vous êtes donc prêt pour les exercices de la journée. Cette façon de faire aide non seulement à maintenir proprement les dépendances de votre projet mais garantit également que l'environnement de développement est facilement répliqué sur tout autre système.

Partie 1 : Manipulation de fichier native

Exercice 1 : Lecture d'un fichier

Prototype : *read_one_line*(filename: str) -> str
Rendu : partie1.py

Écrire une fonction *read_one_line* qui prend en paramètre un nom de fichier. La fonction doit ouvrir le fichier, lire la première ligne et retourner le résultat dans une chaîne de caractères.

Indice : open / read

Exercice 2 : Écriture dans un fichier

Prototype : *write_text*(filename: str, text: str)
Rendu : partie1.py

Écrire une fonction qui prend en paramètre un nom de fichier et un texte. La fonction crée un nouveau fichier avec le nom donné et y écrit le texte passé en paramètre.

Indice : Un fichier doit être explicitement fermé avec la méthode `close` après les manipulations

Exercice 3 : Lire et écrire dans un fichier

Prototype : *copy_characters*(input_file: str, output_file: str, nb: int)
Rendu : partie1.py

Écrire une fonction *copy_characters* qui prend 3 paramètres : un fichier source (*input_file*), un fichier destination (*ouput_*file) et un nombre de caractère à copier (*nb*).
La fonction doit ouvrir le fichier source et copier les *nb* premiers caractères du fichier dans le fichier destination.

Attention : Les caractères doivent être ajoutés à la fin du fichier destination sur une nouvelle ligne. Le contenu déjà existant doit rester intact.

Exercice 4 : Lecture d'un fichier entier par lignes

Prototype : *read_all_lines*(filename: str) -> (list[str], list[str])
Rendu : partie1.py

Écrire une fonction *read_all_lines* qui prend en paramètre un nom de fichier. La fonction doit lire le fichier ligne par ligne et retourner un tuple contenant une liste avec toutes les lignes et une autre liste avec une ligne sur deux. Vous devez utiliser le slicing ou la compréhension de liste pour créer la deuxième liste.

Attention : Vous n'avez pas le droit d'utiliser d'instructions de conditions (if-elif-else) pour cet exercice

Indice : Vous pouvez dupliquer une liste avec copy ou deepcopy

Exercice 5 : Bonne pratique de l'ouverture de fichiers

Prototype :

- *write_text_better*(filename: str, text: str)
- *copy_characters_better*(input_file: str, output_file: str, nb: int)

Rendu : partie1.py

Section 1 :
Réécrire la fonction de l'exercice 2 en utilisant le mot-clé *with* pour ouvrir le fichier passé un paramètre.
« C'est une bonne pratique d'utiliser le mot-clé with lorsque vous traitez des fichiers. Vous fermez ainsi toujours correctement le fichier, même si une exception est levée. »
Documentation Python.org

Avertissement vu dans la documentation : « Appeler f.write() sans utiliser le mot clé with ni appeler f.close() pourrait mener à une situation où les arguments de f.write() ne seraient pas complètement écrits sur le disque, même si le programme se termine avec succès. »

Section 2 :
Réécrire la fonction de l'exercice 3 en utilisant le mot-clé *with* pour ouvrir les fichiers *input_file* et *output_file*.

Indice : Vous pouvez ouvrir deux fichiers simultanément dans une seule instruction `with`.

À partir de cet exercice, on s'attend à ce que vous utilisiez le mot-clé *with* pour chaque ouverture de fichier effectuée avec *open.*

Partie 2 : Manipulation native de fichier CSV

Exercice 1 : Lire un fichier CSV

Prototype : `native_csv_read(file: str) -> list[tuple]`
Rendu : partie2.py

Écrire une fonction *native_csv_read* qui prend en paramètre un fichier CSV. La fonction doit lire le fichier et créer une liste de tuple contenant chaque ligne du fichier CSV à l'exception de la ligne contenant le nom des colonnes.

Ajouter à chaque ligne l'index de celle-ci au début du tuple. Le fichier peut contenir un nombre indéfini de colonnes.
Le caractère de séparation du fichier sera forcément un point-virgule ';'.

Exemple :

```
input_file = 'resources/orders_semicolon2.csv'
t = native_csv_read(input_file)
print(t)
```

orders_semicolon2.csv

```
1  product;quantity;total_price
2  Smartphone;7;4911.924342502764
3  Headphones;4;753.3786756443856
4  Printer;8;4701.49172549989
5
```

```
(.venv) → Jour2 git:(main) × python3 partie2.py | cat -e
[(0, 'Smartphone', '7', '4911.924342502764'), (1, 'Headphones', '4', '753.3786756443856'), (2, 'Printer', '8', '4701.49172549989')]$
(.venv) → Jour2 git:(main) ×
```

Indice : Un élément d'un objet de type *iterable* peut être ignoré pour passer à l'élément suivant avec la fonction next.

Exercice 2 : Écrire dans un fichier CSV

Prototype : native_csv_*write*(file: str, headers: list, data: list[tuple])
Rendu : partie2.py

Écrire une fonction native_csv_*write* qui prend en paramètre un nom de fichier CSV et une liste de liste résultant de l'exercice précédent. La fonction doit créer le fichier CSV, parcourir la liste et ajouter chaque ligne dans le fichier CSV. Attention, vous ne devez pas intégrer l'index de chaque ligne lorsque vous écrivez dans le fichier CSV.
La première ligne du fichier doit contenir les headers pris en paramètres.

Le caractère de séparation du fichier doit être une virgule ','.

Partie 3 : Découverte de **pandas** et ses structures de données

Dans cette partie et les suivantes vous devrez utiliser la librairie Python ***pandas***. C'est une libraire très utilisée dans la manipulation et le traitement de donnée. Vois pourrez trouver toutes les informations dont vous aurez besoin dans la documentation officielle et plus spécifiquement la section « input/output » pour les parties de manipulation de fichiers.

Note importante : Pour tous les exercices suivant, il pourra y avoir mention de DataFrame ayant une structure de type « orders ». Cela signifie qu'on attend du DataFrame qu'il possède 3 colonnes décrites ci-dessous. Le premier exemple sera dans l'exercice 3 de cette partie. Voici les 3 colonnes :

- `product`, une chaîne de caractères contenant le nom du produit acheté
- `quantity`, un nombre entier contenant la quantité de produit acheté
- `total_price, un nombre décimal contenant le prix total de la commande

Exercice 1 : Création d'une Series

Prototype : *create_series*() -> pd.Series
Rendu : partie3.py

Écrire une fonction *create_series* qui utilise tous les arguments du programme pour créer une « Series », structure de donnée propre à *pandas* et retourne celle-ci. Tous les arguments insérés doivent être de type *int.*

Exercice 2 : Opérations de base sur une Series

Prototype : *series_operations*(series: pd.Series) -> (int, float, float)
Rendu : partie3.py

Écrire une fonction *series_operations* prenant en paramètre une Series issue de l'exercice précédent. Réaliser les opérations arithmétiques suivantes : somme, moyenne et écart-type, et les retourner dans un tuple.

Exemple :

```
t = create_series()
print(series_operations(t))
```

```
(.venv) →  Jour2 git:(main) ✗ python3 partie3.py 1 2 3 4 5 6 7 8 9 | cat -e
(np.int64(45), np.float64(5.0), np.float64(2.7386127875258306))$
(.venv) →  Jour2 git:(main) ✗
```

Exercice 3 : Création d'un DataFrame

Prototype : *create_dataframe*(products: list[str], quantities: list[int], prices: list[float]) -> pd.DataFrame
Rendu : partie3.py

Écrire une fonction *create_dataframe* prenant en paramètre trois listes qui contiennent des informations sur des commandes de produits. Créer un « DataFrame » de type « orders », et le remplir avec les listes prises en paramètre. Attention à conserver l'ordre des données lors de la création du DataFrame. Retourner le DataFrame résultant.

Expérience : Essayez d'afficher et de comprendre la structure de donnée créée.

Indice : Vous pouvez utiliser un dictionnaire pour créer le DataFrame, avec des clés comme noms de colonnes et des listes comme valeurs de colonnes.

Indice 2 : Pour tester votre fonction, il est possible de lire le fichier « orders.csv » fourni avec ce sujet grâce à la fonction *native_csv_read* créée lors de la partie précédente. En python, il est possible de transposer une liste de tuple/liste en faisant "list(zip(*data))". Sinon, un fichier « data_partie3.py » vous est fourni avec le sujet contenant les 3 listes nécessaires.

Exercice 4 : Accès aux éléments d'un DataFrame

Prototype : *dataframe_accession*(data: pd.DataFrame) -> tuple
Rendu : partie3.py

Écrire une fonction *dataframe_accession* prenant en paramètre un DataFrame de type « orders ». Votre fonction doit retourner dans un tuple : les valeurs de la colonne `Product` sous forme de liste, la deuxième ligne du DataFrame sous forme de dictionnaire et la quantité du troisième produit.

Indice : Les valeurs d'un DataFrame peuvent être récupérées avec les noms de colonnes ou l'accesseur `.loc` pour sélectionner des parties spécifiques du DataFrame.

Exemple :

```
products = ['Smartphone', 'Headphones', 'Printer', 'Keyboard']
quantities = [7, 4, 8, 5]
prices = [4911.924342502764, 753.3786756443856, 4701.49172549989,
5099.099081811153]
df = create_dataframe(products, quantities, prices)
res = dataframe_accession(df)
for elem in res:
    print(elem)
```

```
(.venv) →  Jour2 git:(main) × python3 partie3.py | cat -e
['Smartphone', 'Headphones', 'Printer', 'Keyboard']$
{'product': 'Headphones', 'quantity': 4, 'total_price': 753.3786756443856}$
8$
(.venv) →  Jour2 git:(main) ×
```

Partie 4 : Manipulation de fichier CSV avec ***pandas***

Exercice 1 : Lire un fichier CSV

Prototype : *pandas*_csv_*read*(file: str) -> pd.DataFrame
Rendu : partie4.py

Écrire une fonction *pandas_csv_read* qui prend en paramètre un fichier, l'ouvre avec *pandas* et retourne la structure de donnée crée.

Exercice 2 : Écrire dans un fichier CSV

Prototype : *pandas*_csv_*write*(file: str, headers: list, data: list[tuple])
Rendu : partie4.py

Réaliser la même tâche que dans l'exercice 2.2, mais cette fois, utiliser *pandas* pour créer un DataFrame à partir de la liste de liste et écrire ce DataFrame dans le fichier CSV donné.

Indice : Il est possible de faire une transposition pour créer un dictionnaire comme pour l'exercice 3.3.

Note : Comme vous l'avez surement observé, utiliser pandas pour lire et écrire des fichiers de données, tels que fichiers CSV, simplifie considérablement le processus par rapport aux méthodes Python natives. Avec pandas, des opérations qui nécessitent généralement plusieurs lignes de code et la gestion de cas particuliers – comme les problèmes d'encodage ou les conversions de types de données – sont réduites à une seule ligne de code. Cela accélère votre vitesse de développement et réduit le potentiel d'erreurs.

De plus, pandas prend en charge plusieurs formats de fichiers, y compris ceux qui n'ont pas de support natif dans la bibliothèque standard de Python, tels que les fichiers Excel et Parquet. Cette polyvalence rend pandas indispensable pour l'analyse de données, vous permettant de gérer facilement et efficacement des sources de données diverses.

Partie 5 : Manipulation de fichier Excel avec *pandas*

Exercice 1 : Lire un fichier Excel

Prototype : *pandas*_excel_*read*(file: str, sheet: str) -> pd.DataFrame
Rendu : partie5.py

Écrire une fonction *pandas*_excel_*read* prenant en paramètre un fichier Excel et un nom de feuille. La fonction doit charger la feuille donnée du fichier Excel dans un DataFrame et le retourner.

Exercice 2 : Écrire dans un fichier Excel

Prototype : *pandas*_excel_*write*(data: pd.DataFrame, filename: str)
Rendu : partie5.py

Écrire une fonction *pandas*_excel_*write* prenant en paramètre un DataFrame de type « orders » et un nom de fichier. Vous devez stocker le DataFrame dans le fichier donné et dans une feuille nommée `orders`.
Si la feuille `orders` existe déjà, vous devez remplacer le contenu de la feuille par celui du DataFrame. Les autres feuilles du fichier Excel doivent rester intactes.

Exercice 3 : Lecture de colonnes et lignes spécifiques

Prototype : *pandas*_excel_*selective_read*(filename: str) -> pd.DataFrame
Rendu : partie5.py

Écrire une fonction *pandas*_excel_*selective_read* qui ouvre la feuille `orders` du fichier Excel pris en paramètre et charge les colonnes `product` et `total_price` uniquement. Sauter les 10 premières lignes (hors headers) du fichier **pendant** la lecture.
La fonction doit retourner un DataFrame contenant la liste des produits et montant total des ventes pour chacun. Chaque produit ne doit apparaitre qu'une fois dans le DataFrame.

Indice : Observer les paramètres de `read_excel`. Attention à ne pas skip la première ligne contenant les headers du fichier.

Indice 2 : Il est possible de regrouper les données d'un DataFrame avec la méthode `groupby`.

Exercice 4 : Manipulation et ajout de données

Prototype : *pandas_excel_manipulation*(filename: str)
Rendu : partie5.py

Écrire une fonction *pandas_excel_manipulation* qui ouvre la feuille `orders` du fichier Excel pris en paramètre et charge les données dans un DataFrame. Pour chaque produit ayant exactement le même nom, effectuer les calculs suivants : le nombre de commandes, la quantité totale de produit vendu, la quantité moyenne de produit acheté par commande arrondie à la deuxième décimale.

Stocker ces données calculées dans une nouvelle feuille `summary`, cette feuille doit contenir 4 colonnes : `product`, `total orders`, `total quantity`, `mean quantity per order `.
Si la feuille existe déjà, vous devez remplacer le contenu de la feuille par celui du DataFrame. Les autres feuilles du fichier Excel doivent rester intactes.

Indice : Vous pouvez combiner la fonction `groupby` avec `aggregate` pour réaliser des opérations différentes sur chaque colonne.

Partie 6 : Manipulation de fichier JSON avec ***pandas***

Exercice 1 : Lecture d'un fichier JSON

Prototype : *pandas_json_read*(file: str) -> pd.DataFrame
Rendu : partie6.py

Écrire une fonction *pandas_json_read* qui prend en paramètre un fichier JSON, le charge dans un DataFrame et retourne le DataFrame créé.

Exercice 2 : Écriture dans un fichier JSON

Prototype : *pandas_json_write*(file: str, data: pd.DataFrame)
Rendu : partie6.py

Écrire une fonction *pandas_json_write* prenant en paramètre un nom de fichier JSON et un DataFrame de type « orders ». Écrire le DataFrame dans le fichier JSON pris en paramètre. L'indentation du fichier doit être fixée à 4 espaces, chaque ligne doit être représentée dans un objet JSON avec le nom des colonnes comme clés.

Format du fichier résultant :

```
[
    {
        "product":"Smartphone",
        "quantity":7,
        "total_price":4911.9243425028
    },
    {
        "product":"Headphones",
        "quantity":4,
        "total_price":753.3786756444
    },
    ...
]
```

Exercice 3 : Structures JSON Complexes

Prototype : *pandas_complex_json(*file: str, product: dict)
Rendu : partie6.py

Les fichier Excel et CSV contiennent des données « plates », c'est un type de structure relativement simple et qui ne convient pas à tous les usages. De nombreuses applications nécessitent des structures de données bien plus complexes, capables de représenter des hiérarchies ou des objets imbriqués sur plusieurs niveaux. Il existe donc des formats de fichier qui permettent de stocker ces structures complexes comme les formats XML et JSON. Dans l'exercice suivant, nous allons explorer comment manipuler une structure JSON complexe en utilisant *pandas*.

Continuons dans l'exemple de commandes, cependant nous allons complexifier celui-ci en ajoutant la liste des produits achetés pour chaque commande ainsi que les informations du client.

```
[
  {
    "order_id": "order123",
    "customer": {"firstname ": " Alice", "name": "DUPONT"},
    "products": [
      {"product_id": "prod1", "name": "Laptop", "quantity": 1, "price":
1999},
      {"product_id": "prod2", "name": "Mouse", "quantity": 2, "price": 20.99}
    ]
  }
]
```

Écrire une fonction *pandas_complex_json* qui prend un nom de fichier JSON et un dictionnaire contenant les informations d'un produit. Ouvrir le fichier et charger son contenu dans un DataFrame. La structure du fichier JSON sera la même que celle présentée ci-dessus.

Modifier ensuite la première transaction pour y ajouter le produit pris en paramètre, puis supprimer la deuxième transaction et finalement, supprimer le deuxième produit de la troisième transaction.

Stocker le résultat dans le même fichier JSON en conservant la structure initiale. Le fichier final doit être indenté avec 2 espaces.

Expérience : Afin de traiter des données imbriquées, il est possible d'utiliser `json_normalize` pour aplatir la structure de données. Dans le cas de cet exercice, ce n'est pas nécessaire, mais connaitre son existence est important.

Partie 7 : Manipulations basiques de données avec *pandas*

Exercice 1 : Trier des données

Prototypes :

- *sort_dataframe_simple(*data: pd.DataFrame) -> pd.DataFrame
- *sort_dataframe_advanced(*data: pd.DataFrame) -> pd.DataFrame

Rendu : partie7.py

Écrire une fonction *sort_dataframe_simple* prenant en paramètre un DataFrame de type « orders ». Trier le DataFrame par produit dans l'ordre alphabétique descendant et le retourner.

Écrire une fonction *sort_dataframe_advanced* prenant en paramètre un DataFrame de type « orders ». Trier le DataFrame par quantité croissante, puis par prix décroissant, et enfin par produit dans l'ordre alphabétique et le retourner.

Expérience : Après avoir trié un DataFrame, il est possible de réattribuer les index (0, 1, 2, …) du DataFrame en fonction du tri qui a été fait grâce à la fonction reset_index. Observer le DataFrame résultant et essayer de faire en sorte que les anciens index ne soient pas conservés.

Exercice 2 : Ajout de colonnes à un DataFrame

Prototype : *add_columns(*data: pd.DataFrame) -> pd.DataFrame
Rendu : partie7.py

Écrire une fonction *add_columns* prenant en paramètre un DataFrame de type « orders ». Vous devez ajouter deux nouvelles colonnes au DataFrame et le retourner :

- `order_number` qui doit contenir le numéro de la transaction. Les numéros de transactions sont l'index (1 à x) de la transaction répété 5 fois. Si le numéro dépasse 5 chiffres, celui-ci doit être compressé via un modulo pour faire exactement 5 chiffres de long.
- `unit_price` qui doit contenir le prix de vente d'une unité arrondi à la deuxième décimale. Vous devez trouver ce prix grâce aux colonnes `montant` et `quantité`.

Avertissement : vous n'avez pas le droit d'utiliser de boucles (for, while, …) pour cet exercice

Indice : Il existe plusieurs façon d'ajouter une colonne à un DataFrame. En modifiant le DataFrame existant ou en créant un nouveau DataFrame avec la nouvelle colonne. Utilisez la méthode que vous trouvez la plus approprié dans le cas de cet exercice.

Exercice 3 : Filtrer des données

Prototypes :

- *filter_dataframe_simple(*data: pd.DataFrame, product: str) -> pd.DataFrame
- *filter_dataframe_advanced(*data: pd.DataFrame) -> pd.DataFrame

Rendu : partie7.py

Écrire une fonction *filter_dataframe_simple* prenant en paramètre un nom de produit et un DataFrame de type « orders ». Trier le DataFrame pour conserver uniquement les ventes de produits avec une quantité supérieure ou égale à 5 et ayant pour produit celui pris en paramètre. Retourner le DataFrame résultant.

Écrire une fonction *filter_dataframe_advanced* prenant en paramètre un DataFrame résultant de la fonction *add_columns* de l'exercice 7.2. Filtrer les données pour conserver uniquement les transactions ayant un numéro de transaction commençant par `1` et ayant un prix unitaire entre 0 et 2 inclus. Trier le DataFrame par produit par ordre alphabétique et retourner le DataFrame résultant.

Indice : Un DataFrame peut être filtré grâce à l'indexation booléenne ou grâce à la méthode Query.
Indice 2 : Il est possible de convertir la colonne `date` en datetime avec `pd.to_datetime` pour simplifier les comparaisons.

Exercice 4 : Opérations de base sur un DataFrame

Prototype : *dataframe_operations*(data: pd.DataFrame) -> (float, int, float, float, float)
Rendu : partie7.py

Écrire une fonction *dataframe_operations* prenant en paramètre un DataFrame de type « orders ». Retourner un tuple contenant : le montant total de toutes les commandes réalisées, la quantité totale de produit vendus, le prix moyen par commande, le montant maximum et le montant minimum de toutes les commandes. Tous les calculs flottants doivent être arrondis à la deuxième décimale

Partie 8 : Nettoyage basique de données avec *pandas*

Exercice 1 : Supprimer des éléments

Prototypes :

- *static_suppression*(df: pd.DataFrame) -> pd.DataFrame
- *missing_data_suppression*(df: pd.DataFrame) -> pd.DataFrame
- *duplicate_data_suppression*(df: pd.DataFrame) -> pd.DataFrame

Rendu : partie8.py

Avertissement : Pour chaque fonction de cet exercice, vous devez réinitialiser l'index et supprimer l'ancien du DataFrame avant de le retourner.

Écrire une fonction *static_suppresion* prenant en paramètre un DataFrame résultant de la fonction *add_columns* de l'exercice 7.2. Supprimer la colonne `total_price` du DataFrame puis supprimer les lignes 5 à 12 inclus. Retourner le DataFrame résultant.

Écrire une fonction *missing_data_suppresion* prenant en paramètre un DataFrame de type « orders ». Supprimer les lignes dont le prix est vide et retourner le DataFrame résultant.

Écrire une fonction *duplicate_data_suppression* prenant en paramètre un DataFrame de type « orders ». Supprimer les doublons – Un doublon consiste en deux ou plus lignes qui sont exactement les mêmes – et retourner le DataFrame résultant.

Exercice 2 : Uniformisation des données

Prototypes :

- *number_uniformisation*(df: pd.DataFrame) -> pd.DataFrame
- *string_uniformisation*(df: pd.DataFrame) -> pd.DataFrame

Rendu : partie8.py

Écrire une fonction *number_uniformisation* prenant en paramètre un DataFrame de type « orders ». Changer toutes les valeurs de la colonne `quantity` pour qu'elles soient des nombres entiers (int) et que les valeurs de la colonne `total_price` soient des nombres décimaux (float). Faire en sorte que la colonne `total_price` soit arrondie à la deuxième décimale. Retourner le DataFrame résultant.

Écrire une fonction *string_uniformisation* prenant en paramètre un DataFrame de type « orders ». Modifier la colonne `product` afin que tous les noms de produits soient en minuscules.

Avertissement : Vous n'avez pas le droit d'utiliser de boucles (for, while, …) pour cet exercice.

Exercice 3 : Validation de plages numériques

Prototype : *number_validation*(df: pd.DataFrame) -> bool
Rendu : partie8.py

Écrire une fonction *number_validation* prenant en paramètre un DataFrame de type « orders ». Valider que toutes les valeurs des colonnes `total_price` sont strictement supérieures à 0 et celles de `quantity` sont comprises entre 1 et 10 inclus. Retourner True si toutes les valeurs sont correctes, sinon False.

Exercice 4 : Validation d'enum

Prototype : *enum_validation*(df: pd.DataFrame, products: list) -> pd.DataFrame
Rendu : partie8.py

Écrire une fonction *enum_validation* prenant en paramètre un DataFrame de type « orders » et une liste de produits. Valider que toutes les entrées de la colonne `product` appartiennent à la liste de produits passés en paramètre. Retourner un DataFrame avec toutes les lignes qui ont des produits invalides. Si toutes les lignes sont valides, retourner un DataFrame vide.

Indice : La méthode `.isin` permet de filtrer les lignes correspondant à un ensemble de valeur autorisées.

Exercice 5 : Validation croisée

Prototype : *cross_column_validation*(df: pd.DataFrame) -> pd.DataFrame
Rendu : partie8.py

Écrire une fonction *cross_column_validation* prenant en paramètre un DataFrame dont la structure est la même qu'un DataFrame résultant de la fonction *add_columns* de l'exercice 7.2. Valider que le prix unitaire est inférieur ou égal au montant de la vente. Retourner un DataFrame contenant les lignes ne respectant pas cette condition.