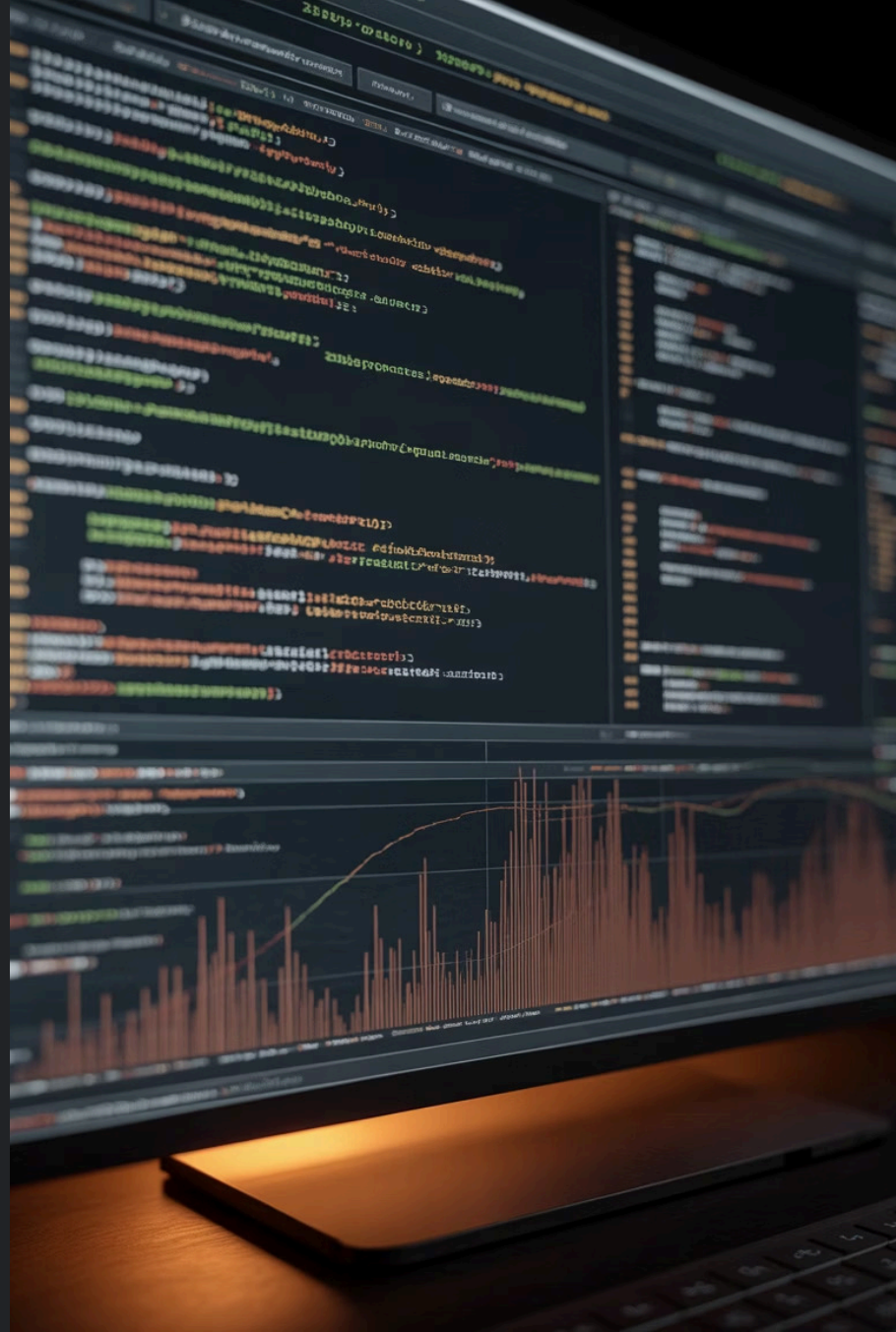
Manipulation de fichiers CSV avec Python

Un cours pratique pour le BTS SIO - Options SISR et SLAM

Cours de David DONISA , Enseignant en BTS SIO



Objectifs pédagogiques du cours

À la fin de ce module, vous serez capable de maîtriser l'ensemble des opérations essentielles sur les fichiers CSV en Python. Ces compétences sont fondamentales dans le développement d'applications et la gestion de systèmes d'information.

Comprendre le format CSV

Identifier la structure et les caractéristiques des fichiers CSV utilisés en entreprise

Lire et écrire des fichiers

Maîtriser les techniques d'import et d'export de données structurées

Manipuler les données

Filtrer, trier et transformer les informations selon les besoins métier

Utiliser le module csv

Exploiter les fonctionnalités natives de Python pour un code efficace

Qu'est-ce qu'un fichier CSV ?

Un fichier CSV (Comma-Separated Values) est un fichier texte simple contenant des données organisées en tableau. Chaque ligne représente un enregistrement et chaque champ est séparé par un délimiteur, généralement une virgule ou un point-virgule.

Format standard en France : Le point-virgule (;) est le séparateur le plus courant, notamment pour la compatibilité avec Excel et les logiciels européens.

Caractéristiques principales

- Format texte brut, lisible et portable
- Compatible avec tous les systèmes d'exploitation
- Facilement exploitable par les tableurs et bases de données
- Léger et rapide à traiter

Exemple de structure

```
id;nom;email;ville
1;Martin;martin@example.com;Paris
2;Nicole;nicole@example.com;Lyon
3;Dupont;dupont@example.com;Paris
```

Chaque ligne après l'en-tête correspond à un enregistrement avec ses différents champs séparés par le délimiteur.

Pourquoi le CSV est essentiel en BTS SIO ?

Pour les SISR - Solutions d'Infrastructure

Import/export de configurations réseau : Les administrateurs utilisent les CSV pour gérer les configurations DHCP, DNS et les inventaires de matériel.

Gestion des utilisateurs Active Directory : Création massive de comptes utilisateurs et attribution de droits via scripts PowerShell ou Python.

Monitoring et logs : Analyse des fichiers de journaux système exportés en CSV pour détecter les anomalies et incidents de sécurité.

Pour les SLAM - Solutions Logicielles

Échange de données entre applications : Format universel pour l'import/export de données dans les systèmes de gestion.

Migration de bases de données : Transfert de données entre différents SGBD lors de changements de plateforme.

Tests et jeux de données : Préparation de données de test pour valider les fonctionnalités développées.

Le module csv de Python

Python propose un module natif dédié à la manipulation des fichiers CSV, offrant des fonctionnalités puissantes et une syntaxe intuitive. Ce module fait partie de la bibliothèque standard, aucune installation supplémentaire n'est nécessaire.

01

Import du module

Commencez toujours par importer le module dans votre script

02

Ouverture du fichier

Utilisez la fonction open() avec le bon encodage (utf-8)

03

Lecture ou écriture

Choisissez reader/writer ou DictReader/DictWriter selon vos besoins

04

Traitement des données

Parcourez, filtrez ou modifiez les données ligne par ligne

05

Fermeture automatique

Le contexte with garantit la fermeture propre du fichier

```
import csv

# Le module est maintenant disponible
# pour toutes vos opérations CSV
```

Lire un fichier CSV : Deux méthodes

Méthode 1 : csv.reader

Cette approche retourne chaque ligne sous forme de liste. C'est la méthode la plus simple pour un accès par index.

```
import csv

with open("clients.csv", "r",
          encoding="utf-8") as f:
    lecteur = csv.reader(f, delimiter=";")

    for ligne in lecteur:
        print(ligne)
        # Affiche : ['1', 'Martin', 'martin@example.com']
```

Résultat attendu:

```
['id', 'nom', 'email', 'ville']
['1', 'Martin', 'martin@example.com', 'Paris']
['2', 'Nicole', 'nicole@example.com', 'Lyon']
```

Avantages : Simple, rapide, accès par position

Inconvénients : Nécessite de connaître l'ordre des colonnes

Méthode 2 : csv.DictReader

Retourne chaque ligne comme un dictionnaire avec les en-têtes comme clés. Plus lisible et maintenable pour du code complexe.

```
import csv

with open("clients.csv", "r",
          encoding="utf-8") as f:
    lecteur = csv.DictReader(f, delimiter=";")

    for ligne in lecteur:
        print(ligne["nom"], ligne["email"])
        # Affiche : Martin martin@example.com
```

Résultat attendu:

```
Martin martin@example.com
Nicole nicole@example.com
Dupont dupont@example.com
```

Avantages : Code plus clair, accès par nom de colonne

Inconvénients : Légèrement plus lent sur gros volumes

Conseil pratique : Utilisez DictReader pour la plupart de vos projets. La lisibilité du code prime sur la micro-optimisation.

Vérification de l'existence d'un fichier avec `os.path.exists()`

Avant toute opération de lecture ou d'écriture sur un fichier, il est crucial de vérifier son existence pour éviter les erreurs. Le module standard `os` de Python offre la fonction `os.path.exists()` qui retourne `True` si le fichier ou le répertoire existe, et `False` dans le cas contraire.

Cette approche permet de construire des applications robustes, capables de gérer automatiquement la présence ou l'absence des fichiers requis, prévenant ainsi les exceptions `FileNotFoundError`.

```
import os
import csv

file_path = './clients.csv'

if os.path.exists(file_path):
    print("Le fichier existe, on peut le lire")
    with open(file_path, 'r', encoding='utf-8') as f:
        lecteur = csv.reader(f, delimiter=';')
        for ligne in lecteur:
            print(ligne)
else:
    print("Le fichier n'existe pas, création nécessaire")
    with open(file_path, 'w', encoding='utf-8', newline='') as f:
        writer = csv.writer(f, delimiter=';')
        writer.writerow(['id', 'nom', 'email'])
```

Dans cet exemple, si `clients.csv` est présent, le script le lira et affichera son contenu. S'il est absent, il affichera un message indiquant sa non-existence et le créera avec une ligne d'en-tête.

Cette technique est essentielle en production pour éviter les erreurs et gérer automatiquement la création de fichiers manquants.

Focus sur csv.reader : Utiliser `next()` pour gérer l'en-tête CSV

La fonction Python intégrée `next()` est un outil puissant pour interagir avec les itérateurs. Dans le contexte du module `csv`, elle est particulièrement utile pour gérer la ligne d'en-tête de vos fichiers de données.

Lecture et avancement

`next(iterateur)` lit la ligne courante d'un itérateur et avance automatiquement à la ligne suivante.

Sauter l'en-tête

Son utilisation la plus courante est de sauter la première ligne (l'en-tête) d'un fichier CSV avant de commencer le traitement des données réelles.

Prévention des erreurs

Cela garantit que l'en-tête n'est jamais traité par erreur comme un enregistrement de données, simplifiant ainsi la logique de votre script.

Stockage de l'en-tête

La ligne d'en-tête lue par `next()` peut être stockée dans une variable pour être utilisée ultérieurement, par exemple pour des affichages ou des validations.

Exemple d'utilisation avec `next()`

```
import csv

with open('clients.csv', 'r', encoding='utf-8') as f:
    lecteur = csv.reader(f, delimiter=';')

    # Lire et stocker l'en-tête, puis avancer l'itérateur
    en_tete = next(lecteur)
    print(f"En-tête : {en_tete}")
    # Résultat attendu : ['id', 'nom', 'email', 'ville']

    # Le reste de la boucle for traite uniquement les données
    for ligne in lecteur:
        # L'en-tête est déjà passé, on accède directement aux données
        print(f"Client : {ligne[1]}, Email : {ligne[2]}")

# Résultat attendu :
# En-tête : ['id', 'nom', 'email', 'ville']
# Client : Martin, Email : martin@example.com
# Client : Nicole, Email : nicole@example.com
# Client : Dupont, Email : dupont@example.com
```

Sans `next()` : L'en-tête traité comme une donnée

```
# Sans next(), la première itération de la boucle
# traitera la ligne d'en-tête comme une donnée ordinaire.
for ligne in lecteur: # Ici 'lecteur' serait un nouvel itérateur
    print(ligne)
# Affichera : ['id', 'nom', 'email', 'ville']
# Suivi des lignes de données réelles.
```

Avec `csv.DictReader`, l'utilisation explicite de `next()` n'est généralement pas nécessaire car il utilise automatiquement la première ligne comme clés de dictionnaire.

Stocker les données CSV dans une liste Python

Après avoir lu un fichier CSV, il est souvent utile de conserver les données en mémoire pour des traitements ultérieurs. Python permet de stocker facilement ces informations dans une liste, où chaque ligne du CSV devient une sous-liste.

En stockant les données dans une structure comme `mes_donnees = []` et en utilisant la méthode `.append()` pour ajouter chaque ligne lue, vous pouvez manipuler l'ensemble du jeu de données sans avoir à relire le fichier, ce qui est idéal pour des traitements multiples, du tri ou du filtrage.

```
import csv

mes_donnees = []  # Liste vide pour stocker les données

with open('clients.csv', 'r', encoding='utf-8') as f:
    lecteur = csv.reader(f, delimiter=';')

    # Sauter l'en-tête (première ligne)
    en_tete = next(lecteur)
    print(f"Colonnes : {en_tete}")

    # Stocker chaque ligne dans la liste
    for ligne in lecteur:
        mes_donnees.append(ligne)
        print(f"Ligne ajoutée : {ligne}")

# Maintenant on peut réutiliser les données après la lecture complète du fichier
print(f"\nNombre total de lignes (hors en-tête) : {len(mes_donnees)}")
print(f"Première ligne stockée : {mes_donnees[0]}")
print(f"Dernière ligne stockée : {mes_donnees[-1]}")
```

Résultat attendu avec des données CSV de l'exemple précédent :

```
Colonnes : ['id', 'nom', 'email', 'ville']
Ligne ajoutée : ['1', 'Martin', 'martin@example.com', 'Paris']
Ligne ajoutée : ['2', 'Nicole', 'nicole@example.com', 'Lyon']
Ligne ajoutée : ['3', 'Dupont', 'dupont@example.com', 'Paris']

Nombre total de lignes (hors en-tête) : 3
Première ligne stockée : ['1', 'Martin', 'martin@example.com', 'Paris']
Dernière ligne stockée : ['3', 'Dupont', 'dupont@example.com', 'Paris']
```

Manipulation Flexible

Triez, filtrez ou modifiez les données facilement, comme avec n'importe quelle liste Python.

Accès Répété

Réutilisez le jeu de données complet plusieurs fois sans relire le fichier, optimisant les performances.

Export Multi-formats

Préparez les données pour l'exportation vers d'autres formats comme JSON, XML ou une base de données.

Écrire dans un fichier CSV

L'écriture de fichiers CSV permet d'exporter les résultats de vos traitements, de créer des rapports ou de préparer des données pour d'autres applications. Python offre deux approches symétriques à la lecture.

csv.writer - Écriture par liste

```
with open("export.csv", "w",
          encoding="utf-8",
          newline="") as f:
    writer = csv.writer(f, delimiter=";")

    # Écriture de l'en-tête
    writer.writerow(["id", "mot_cle", "volume"])

    # Écriture des données
    writer.writerow([1, "python cours", 1200])
    writer.writerow([2, "csv tutorial", 850])
```

Le paramètre **newline=""** est crucial sous Windows pour éviter les lignes vides supplémentaires.

Résultat dans le fichier export.csv:

```
id;mot_cle;volume
1;python cours;1200
2;csv tutorial;850
```

csv.DictWriter - Écriture par dictionnaire

```
import csv

with open("export.csv", "w",
          encoding="utf-8",
          newline="") as f:
    champs = ["id", "mot_cle", "volume"]
    writer = csv.DictWriter(f,
                            fieldnames=champs,
                            delimiter=";")

    writer.writeheader()
    writer.writerow({"id": 1,
                     "mot_cle": "python cours",
                     "volume": 1200,
                     "auteur": "Jean"}) # Ajout d'un champ "auteur" non
défini dans fieldnames
```

Cette méthode garantit l'ordre des colonnes grâce au paramètre **fieldnames**.

- Le paramètre `fieldnames` est une liste qui définit les noms des colonnes et leur ordre exact dans le fichier CSV.
- Il doit contenir toutes les clés que vous prévoyez d'utiliser dans les dictionnaires que vous écrirez.
- L'ordre des éléments dans la liste `fieldnames` détermine l'ordre des colonnes dans le fichier CSV de sortie.
- Si vous essayez d'écrire un dictionnaire avec des clés qui ne sont pas dans `fieldnames`, une erreur sera générée par défaut.
- Vous pouvez aussi utiliser `fieldnames` pour contrôler précisément quels champs sont exportés, en omettant délibérément des champs non désirés du dictionnaire.

Dans l'exemple ci-dessus, même si le dictionnaire contient une clé "auteur", elle ne sera pas écrite dans le fichier car elle n'est pas présente dans la liste `champs`.

Pour gérer les clés supplémentaires non définies dans `fieldnames`, vous pouvez utiliser le paramètre `extrasaction` lors de l'initialisation de `csv.DictWriter`. Par exemple, `extrasaction='ignore'` ignorera silencieusement les clés non présentes dans `fieldnames` au lieu de générer une erreur.

Exemple avec extrasaction='ignore'

```
import csv

with open("export.csv", "w",
          encoding="utf-8",
          newline="") as f:
    champs = ["id", "mot_cle", "volume"]
    writer = csv.DictWriter(f,
                            fieldnames=champs,
                            delimiter=";",
                            extrasaction='ignore')

    writer.writeheader()

    # Ce dictionnaire contient des champs supplémentaires
    donnees = {
        "id": 1,
        "mot_cle": "python cours",
        "volume": 1200,
        "auteur": "Jean",      # Sera ignoré
        "date": "2024-01-15"   # Sera ignoré
    }

    writer.writerow(donnees)
    # Aucune erreur ! Les champs "auteur" et "date" sont
ignorés
```

Résultat dans le fichier export.csv:

```
id;mot_cle;volume
1;python cours;1200
```

Avec extrasaction='ignore', les clés supplémentaires (auteur, date) sont silencieusement ignorées. Sans ce paramètre, Python générerait une ValueError.

Résultat dans le fichier export.csv:

```
id;mot_cle;volume
1;python cours;1200
```

Écrire plusieurs lignes avec writerows()

La méthode `writerows()` est une alternative puissante et plus efficace à l'appel répété de `writerow()` lorsque vous avez plusieurs lignes de données à écrire dans un fichier CSV. Elle accepte une liste de listes (ou tout itérable d'itérables) où chaque sous-liste représente une ligne à écrire.

```
import csv

# Méthode 1 : Écriture ligne par ligne (à des fins de comparaison)
print("Écriture de 'export1.csv' ligne par ligne...")
with open('export1.csv', 'w', encoding='utf-8', newline='') as f:
    writer = csv.writer(f, delimiter=';')
    writer.writerow(['id', 'produit', 'prix'])
    writer.writerow([1, 'Clavier', 45.99])
    writer.writerow([2, 'Souris', 25.50])
    writer.writerow([3, 'Écran', 299.00])
print("'export1.csv' créé.")

# Méthode 2 : Écriture multiple avec writerows()
donnees = [
    ['id', 'produit', 'prix'],
    [1, 'Clavier', 45.99],
    [2, 'Souris', 25.50],
    [3, 'Écran', 299.00]
]

print("\nÉcriture de 'export2.csv' avec writerows()...")
with open('export2.csv', 'w', encoding='utf-8', newline='') as f:
    writer = csv.writer(f, delimiter=';')
    writer.writerows(donnees)  # Une seule instruction pour toutes les données !
print("'export2.csv' créé.")
```

Résultat dans les fichiers `export1.csv` et `export2.csv`:

```
id;produit;prix
1;Clavier;45.99
2;Souris;25.50
3;Ecran;299.00
```

Avantage en termes de performance et de lisibilité

L'utilisation de `writerows()` est plus efficace pour les opérations de lot, car elle réduit le nombre d'appels à la méthode d'écriture, ce qui peut améliorer les performances, surtout avec de grands volumes de données. De plus, le code est plus concis et plus lisible.

Cas pratique : Exporter les résultats d'une requête SQL ou d'une API

Imaginez que vous récupérez des milliers d'enregistrements d'une base de données ou d'une API sous forme de liste de listes (ou de dictionnaires, que vous pouvez convertir). La méthode `writerows()` est parfaitement adaptée pour exporter ces résultats en une seule opération vers un fichier CSV, simplifiant ainsi le processus d'extraction et de rapport de données volumineuses.

Manipulations avancées des données

Filtrer les données

Sélectionnez uniquement les enregistrements qui répondent à vos critères métier.

```
with open("clients.csv", "r",
          encoding="utf-8") as f:
    lecteur = csv.DictReader(f, delimiter=";")

    for ligne in lecteur:
        if ligne["ville"] == "Paris":
            print(ligne["nom"])
```

Résultat attendu:

```
Martin
Dupont
```

Cas d'usage SISR : Filtrer les serveurs nécessitant une mise à jour de sécurité

Cas d'usage SLAM : Extraire les utilisateurs actifs pour statistiques

Trier les données

Organisez vos enregistrements selon un ou plusieurs critères pour faciliter l'analyse.

```
with open("clients.csv",
          encoding="utf-8") as f:
    lecteur = csv.DictReader(f, delimiter=";")
    clients = list(lecteur)

    # Tri alphabétique par nom
    clients_tries = sorted(clients,
                           key=lambda c: c["nom"])

    # Tri numérique par âge
    par_age = sorted(clients,
                     key=lambda c: int(c["age"]))
```

Résultat du tri alphabétique:

```
Dupont, Martin, Nicole
```

Résultat du tri par âge (croissant) :

```
Nicole (22 ans), Martin (35 ans), Dupont (45 ans)
```

Astuce : Utilisez `reverse=True` pour un tri décroissant

Conversion de types

Transformez les chaînes de caractères en types appropriés pour vos calculs.

```
# Par défaut, tout est lu en str
age_str = ligne["age"]  # "25"

# Conversion en entier
age = int(ligne["age"])  # 25

# Conversion en décimal
prix = float(ligne["prix"])  # 19.99

# Conversion en booléen
actif = ligne["actif"] == "true"
```

Exemples concrets:

Avant conversion:

```
"25" (type: str)
```

Après conversion:

```
25 (type: int)
```

Attention : Gérez les valeurs vides avec des tests conditionnels pour éviter les erreurs

Nettoyage et validation des données

Les données CSV provenant de sources externes contiennent souvent des espaces superflus, des variations de casse ou des caractères indésirables. Le nettoyage est une étape cruciale avant tout traitement.

Techniques de nettoyage essentielles

```
# Supprimer les espaces en début/fin
texte = ligne["mot_cle"].strip()
# "  python  " → "python"
```

Résultat: " python " → "python"

```
# Normaliser la casse
texte_lower = ligne["email"].lower()
# "Martin@Example.COM" → "martin@example.com"
```

Résultat: "Martin@Example.COM" → "martin@example.com"

```
# Combiner plusieurs opérations
mot_cle_propre = ligne["mot_cle"].strip().lower()
```

Résultat: " Python Course " → "python course"

```
# Remplacer des caractères
telephone = ligne["tel"].replace(" ", "").replace("-", "")
# "01 23 45-67-89" → "0123456789"
```

Résultat: "01 23 45-67-89" → "0123456789"

Validation des données

- Vérifier la présence de champs obligatoires
- Contrôler le format des emails avec des expressions régulières
- Valider les plages de valeurs numériques
- Détecter et gérer les doublons

30%

Données sales

Proportion moyenne de données nécessitant un nettoyage dans les fichiers CSV d'entreprise

2x

Gain de temps

Facteur d'accélération des traitements après nettoyage des données

Projet pratique : Créez une fonction `nettoyer_ligne()` réutilisable dans tous vos scripts

Synthèse et mise en pratique

Points clés à retenir

Le format CSV est universel

Standard de facto pour l'échange de données entre systèmes hétérogènes, essentiel en infrastructure (SISR) comme en développement (SLAM)

Python simplifie la manipulation

Le module csv offre des outils puissants : reader/writer pour la simplicité, DictReader/DictWriter pour la clarté du code

La qualité des données est primordiale

Nettoyage, validation et conversion de types garantissent la fiabilité de vos traitements automatisés