Python pour le Data Engineering

Les essentiels pour des pipelines robustes

Maîtrisez l'environnement Python, l'I/O efficace et le code testable pour industrialiser vos projets data. Formation pratique destinée aux ingénieurs données et data scientists.

Environnements & gestion des dépendances

Versions Python

Utilisez **venv** ou **pyenv** pour gérer plusieurs versions Python et isoler vos projets efficacement.

Gestion des packages

Adoptez **uv** ou **pip** avec requirements.txt et lockfile pour verrouiller les versions et garantir la reproductibilité.

Isolation

Créez des environnements virtuels dédiés pour chaque projet afin d'éviter les conflits de dépendances.

Organisation du code & architecture projet

Structure claire

- src/ — code source
- tests/ — tests unitaires
- notebooks/ — explorations
- scripts/ — exécutables
- configs/ — configurations

Modularisation intelligente

Séparez clairement les responsabilités : I/O, transformations, utilitaires et CLI dans des modules distincts.

Privilégiez les **fonctions pures** avec une petite surface d'effets de bord pour faciliter les tests et la maintenance.

Typage, docs et qualité de code

1

Type hints

Utilisez **typing** et **mypy** pour des vérifications statiques qui préviennent les erreurs avant l'exécution.

2

Documentation

Rédigez des docstrings (style Google/NumPy) et maintenez un README minimal pour faciliter la collaboration.

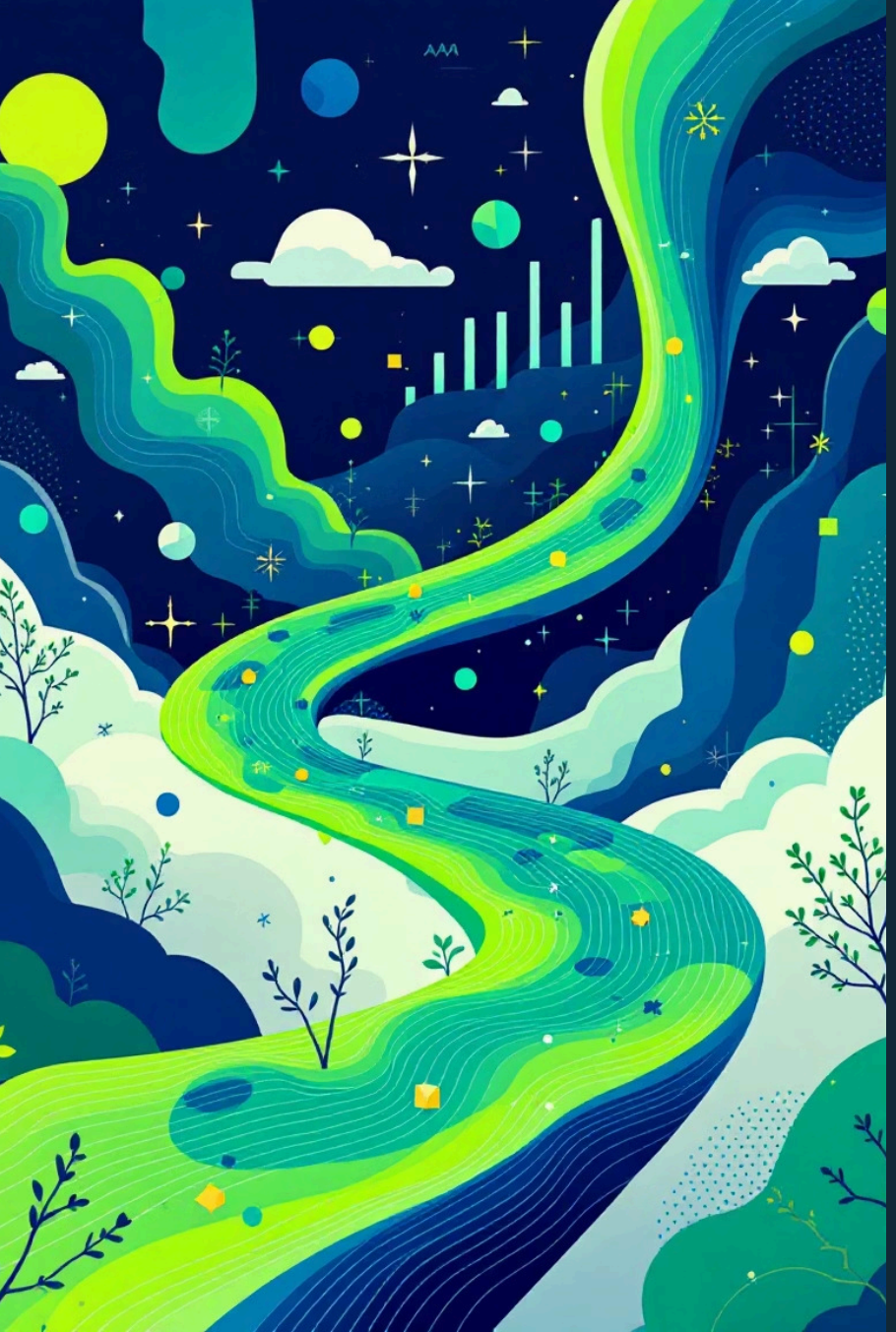
3

Formatage & lint

Adoptez **black**, **isort** et **flake8** pour garantir la cohérence du code dans toute l'équipe.

```
def transform(df: pd.DataFrame) -> pd.DataFrame:
    """Transforme le DataFrame selon les règles métier."""
    return df.dropna().reset_index(drop=True)
```

Typez systématiquement les interfaces publiques (entrée/sortie) pour améliorer la maintenabilité.



I/O fichiers & stockage

pathlib

Manipulez les chemins de fichiers de manière élégante et portable avec la bibliothèque pathlib.

Context managers

Utilisez `with open(...) as f:` pour garantir la fermeture automatique des fichiers.

Lecture chunkée

Traitez les gros fichiers par morceaux avec `iter_chunks` ou `pd.read_csv(..., chunksize=...)`.

Librairies essentielles & quand les utiliser

pandas

Idéal pour le développement rapide et l'exploration interactive des données. Interface intuitive et riche écosystème.

polars

Privilégiez polars quand le dataset dépasse la RAM ou pour des besoins de performance critiques. Faible empreinte mémoire.

numpy

La fondation pour le calcul numérique en Python. Opérations vectorisées ultra-rapides sur des tableaux multidimensionnels.

Règle simple : pandas pour le dev rapide ; polars quand dataset > RAM ou besoins de performance élevés.

Formats & sérialisation

Parquet via pyarrow

Format colonnaire optimisé pour l'analytics. Compression efficace et lecture rapide des colonnes sélectionnées.

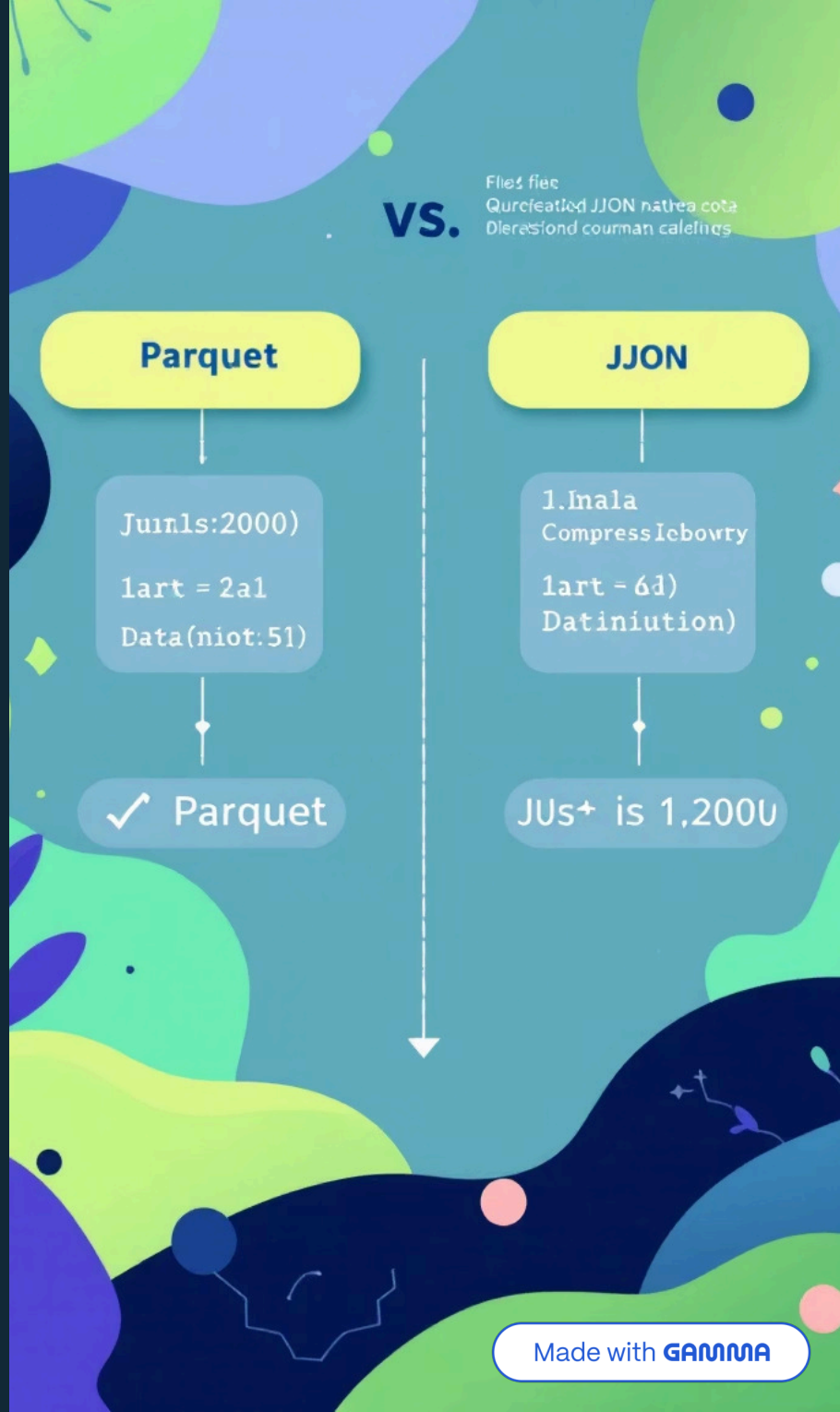
```
df.to_parquet(
    'data.parquet',
    compression='snappy'
)
```

JSONLines pour logs

Format ligne par ligne idéal pour les logs et le streaming. Facile à parser et à traiter incrémentalement.

Incluez des **métadonnées** (schema, version) dans vos fichiers quand possible.

Bonne pratique : Évitez les nombreux petits fichiers — groupez par date ou shard cohérent pour optimiser les performances.



Accès aux bases depuis Python

01

Abstraction avec SQLAlchemy

Utilisez **sqlalchemy** pour une abstraction propre et des drivers comme **psycopg2** ou **asyncpg** pour PostgreSQL.

02

Requêtes paramétrées

Évitez le string formatting — utilisez toujours des requêtes paramétrées pour la sécurité et les performances.

03

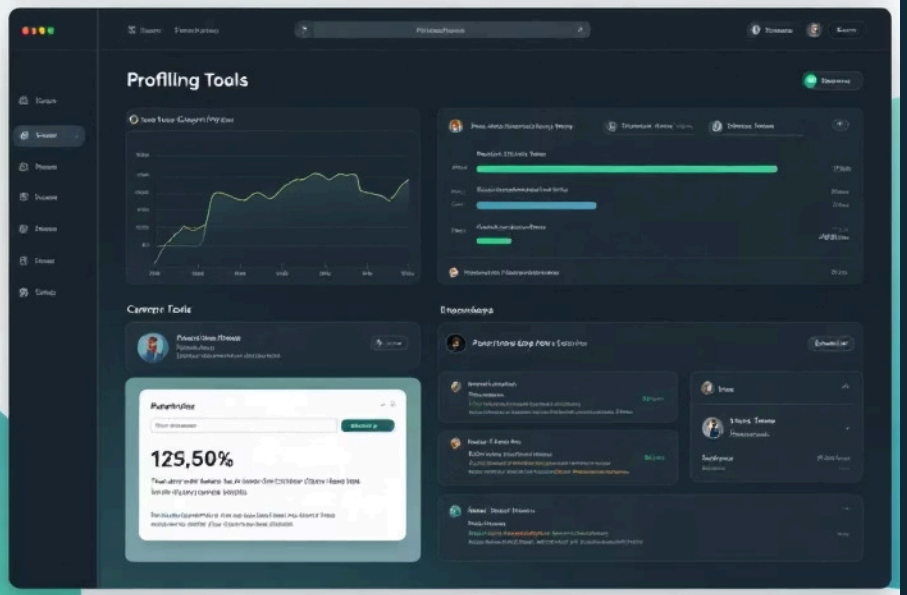
Opérations en batch

Pour les gros volumes, privilégiez **COPY** ou les batch inserts plutôt que les inserts ligne-à-ligne.

04

Gestion des connexions

Utilisez un pool de connexions et fermez/committez explicitement les transactions.



Performance & debugging

Vectorisation

Évitez les boucles Python sur les lignes. Privilégiez les opérations vectorisées et les fonctions C-backed pour des gains de performance massifs.

Techniques d'optimisation

- Chunking pour gros datasets
- Lazy evaluation
- Memory-mapping
- Éviter les copies inutiles

Outils de profiling

cProfile, **memory_profiler**, **pyinstrument** — mesurez avant d'optimiser. Ciblez les hotspots identifiés.

Principe clé : Pas d'optimisation prématurée — identifiez d'abord les goulots d'étranglement réels.

Validation, tests et checklist production

1

Validation des données

Utilisez **pydantic** ou **pandera** pour définir et valider les contrats de données (schema, types).

2

Tests unitaires

Testez vos fonctions de transformation avec **pytest**. Créez des fixtures pour générer des données synthétiques reproductibles.

3

Checklist pré-déploiement

- Environnement verrouillé
- Tests unitaires validés
- Lectures chunk-aware
- Gestion erreurs + retries
- Documentation minimale

```
import pandera as pa

schema = pa.DataFrameSchema({
    "id": pa.Column(int),
    "value": pa.Column(float, nullable=True)
})
schema.validate(df)
```

Automatisez vos tests locaux et maintenez des cellules reproductibles pour garantir la qualité en production.