Concepts Avancés Pandas

28 novembre 2025

Résumé

Ce document explore les concepts avancés de Pandas : MultiIndex et indexation avancée, méthodes d'agrégation avancées, séries temporelles complexes, et techniques d'optimisation des performances. Chaque concept est expliqué avec sa théorie, ses mathématiques sous-jacentes, sa valeur ajoutée et des exemples pratiques.

Table des matières

1 MultiIndex & Indexation Avancée

1.1 Concept et Définition

Un **MultiIndex** (index hiérarchique) permet de créer des structures de données multidimensionnelles dans Pandas, similaires aux cubes de données en OLAP.

1.2 Implémentation

```python
import pandas as pd
import numpy as np

# Cr ation de MultiIndex
index = pd.MultiIndex.from_product([
    ['2023', '2024'],
    ['Q1', 'Q2', 'Q3', 'Q4']
], names=['Ann e', 'Trimestre'])

df = pd.DataFrame({
    'ventes': np.random.randint(1000, 5000, 8),
    'profits': np.random.randint(100, 1000, 8)
}, index=index)

print("DataFrame avec MultiIndex:")
print(df)
```

Sortie :

```
                ventes  profits
Année Trimestre
2023  Q1          3456      789
      Q2          2345      456
      Q3          4123      234
      Q4          2987      567
2024  Q1          3765      890
      Q2          2543      123
      Q3          3987      678
      Q4          3210      345
```

1.3 Mathématiques Sous-Jacentes

Le MultiIndex utilise une structure d'**arbre B** pour l'indexation :
— **Niveaux** : Chaque niveau représente une dimension
— **Codes** : Références aux labels pour chaque niveau
— **Performance** : Recherche en $O(\log n)$ grâce à l'indexation hiérarchique
La structure peut être représentée comme :

$$\text{MultiIndex} = \{\text{Niveau}_1 \times \text{Niveau}_2 \times \cdots \times \text{Niveau}_k\}$$

1.4 Sélection Avancée

```
# S lection par tuple
ventes_2023_Q1 = df.loc[('2023', 'Q1')]
print("Ventes 2023 Q1:")
print(ventes_2023_Q1)

# S lection par niveau avec xs
ventes_2023 = df.xs('2023', level='Ann e')
print("\nToutes les ventes 2023:")
print(ventes_2023)

# Slicing avanc
premier_trimestre = df.loc[(slice(None), 'Q1'), :]
print("\nTous les Q1:")
print(premier_trimestre)

# S lection conditionnelle multi-niveaux
ventes_elevees = df[df['ventes'] > 3000]
print("\nVentes > 3000:")
print(ventes_elevees)
```

1.5 Valeur Ajoutée

Avantage	Description
Organisation logique	Structure naturelle pour données hiérarchiques
Agrégations flexibles	Groupby sur différents niveaux
Performance	Accès rapide grâce à l'arbre B
Compatibilité	Intègre toutes les opérations Pandas

TABLE 1 – Valeur ajoutée du MultiIndex

2 Méthodes d'Agrégation Avancées

2.1 Agrégations avec Dictionnaires

```python
# Agrégations avancées avec dictionnaires
aggregations = {
    'ventes': ['sum', 'mean', 'std'],
    'profits': ['sum', lambda x: x.quantile(0.8)]  # 80ème percentile
}

resultat_agg = df.groupby('Année').agg(aggregations)
print("Agrégations avancées:")
print(resultat_agg)
```

Sortie :

	ventes			profits	
	sum	mean	std	sum	<lambda_0>
Année					
2023	12911	3227.750000	738.194061	2046	684.6
2024	13505	3376.250000	663.069391	2036	795.4

2.2 Transformation vs Agrégation

```python
# Transformation - maintient la forme originale
df['ventes_moyenne_groupe'] = df.groupby('Année')['ventes'].transform('mean')
df['ecart_moyenne'] = df['ventes'] - df['ventes_moyenne_groupe']

print("DataFrame avec transformations:")
print(df[['ventes', 'ventes_moyenne_groupe', 'ecart_moyenne']])

# Agrégation avec conditions
def taux_croissance(serie):
    return (serie.iloc[-1] - serie.iloc[0]) / serie.iloc[0] * 100

croissance_annuelle = df.groupby('Année')['ventes'].agg(taux_croissance)
print("\nTaux de croissance annuel:")
print(croissance_annuelle)
```

2.3 Mathématiques des Agrégations

- **Aggregation** : Fonction de réduction $f : \mathbb{R}^n \to \mathbb{R}$
- **Transformation** : Fonction de préservation $f : \mathbb{R}^n \to \mathbb{R}^n$
- **Window functions** : Agrégations sur fenêtres glissantes

Formules mathématiques communes :

$$\text{Moyenne} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

$$\text{Écart-type} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

$$\text{Quantile } q = x_{\lceil q \cdot n \rceil}$$

2.4 Valeur Ajoutée

Avantage	Description
Flexibilité	Combinaison libre de métriques
Performance	Opérations vectorisées natives
Expressivité	Syntaxe claire et concise
Réutilisabilité	Schémas d'agrégation réutilisables

TABLE 2 – Valeur ajoutée des agrégations avancées

3 Time Series Avancées

3.1 Resampling Complexe

```
# Cr ation de s ries temporelles
dates = pd.date_range('2023-01-01', periods=1000, freq='D')
ts = pd.Series(np.random.randn(1000).cumsum(), index=dates)

print("S rie temporelle originale:")
print(f"P riode: {ts.index.min()} to {ts.index.max()}")
print(f"Shape: {ts.shape}")

# Resampling mensuel avec agr gations custom
resampled = ts.resample('M').agg(['mean', 'std', lambda x: x.max() - x.
    min()])
resampled.columns = ['moyenne', 'ecart_type', 'range']

print("\nS rie resampl e (mensuelle):")
print(resampled.head())
```

3.2 Rolling Windows Avancées

```
# Fen tres glissantes avec calculs complexes
def calcul_tendance(x):
    """Calcule la pente de la tendance lin aire"""
```

```
    if len(x) < 2:
        return np.nan
    return np.polyfit(range(len(x)), x, 1)[0]  # pente de la r gression

def efficience_march (x):
    """Ratio rendement/volatilit  (Sharpe simplifi )"""
    returns = np.diff(x) / x[:-1]
    return np.mean(returns) / np.std(returns) if len(returns) > 1 else np
        .nan

# Application des rolling windows
rolling_features = pd.DataFrame({
    'mean': ts.rolling(window=30).mean(),
    'volatility': ts.rolling(window=30).std(),
    'trend': ts.rolling(window=30).apply(calcul_tendance, raw=True),
    'efficiency': ts.rolling(window=30).apply(efficience_march , raw=
        True)
})

print("\nFeatures sur fen tres glissantes (30 jours):")
print(rolling_features.tail())
```

3.3 Mathématiques des Séries Temporelles

3.3.1 Resampling

$$\text{Aggrégation temporelle : } f(\{x_t\}_{t\in T}) \to y_{T'}$$
$$\text{Interpolation : } x_t = \sum w_i x_{t_i}$$

3.3.2 Rolling Statistics

$$\text{Moyenne mobile} = \frac{1}{w}\sum_{i=t-w+1}^{t} x_i$$
$$\text{Volatilité} = \sqrt{\frac{1}{w}\sum_{i=t-w+1}^{t}(x_i - \bar{x}_w)^2}$$

3.3.3 Régression Linéaire Mobile

Pour une fenêtre de taille w :

$$\beta = (X^T X)^{-1} X^T y$$

où $X = [1, t]$ et y sont les valeurs dans la fenêtre.

Avantage	Description
Analyse temporelle	Détection de tendances et saisonnalités
Features engineering	Préparation pour machine learning
Performance	Optimisé pour séries temporelles
Visualisation	Support natif pour plots temporels

TABLE 3 – Valeur ajoutée des séries temporelles avancées

3.4 Valeur Ajoutée

4 Performance & Optimisation

4.1 Opérations Vectorisées vs Apply

```python
# Cr ation d'un grand dataset
np.random.seed(42)
grand_df = pd.DataFrame({
    'A': np.random.rand(100000),
    'B': np.random.rand(100000),
    'C': np.random.rand(100000)
})

# MAUVAISE APPROCHE: apply row-wise
def methode_lente(row):
    return row['A'] * row['B'] + row['C']

# BONNE APPROCHE: op rations vectoris es
grand_df['resultat_vectorise'] = grand_df['A'] * grand_df['B'] + grand_df
    ['C']

# APPROCHE INTERM DIAIRE: apply avec axis=1
grand_df['resultat_apply'] = grand_df.apply(methode_lente, axis=1)

print("V rification  quivalence :")
print(f"R sultats identiques: {np.allclose(grand_df['resultat_vectorise
    '], grand_df['resultat_apply'])}")

# Test de performance
import time

def tester_performance():
    # Vectoris
    start = time.time()
    resultat_vec = grand_df['A'] * grand_df['B'] + grand_df['C']
    temps_vec = time.time() - start

    # Apply
    start = time.time()
    resultat_app = grand_df.apply(methode_lente, axis=1)
```

```
    temps_app = time.time() - start

    print(f"\nPerformance:")
    print(f"Vectoris : {temps_vec:.4f}s")
    print(f"Apply: {temps_app:.4f}s")
    print(f"Acc l ration: {temps_app/temps_vec:.1f}x")

tester_performance()
```

4.2 Optimisation avec Numba

```
from numba import jit
import numba

# Fonction optimis e avec Numba
@jit(nopython=True)
def calcul_rapide(a, b, c):
    return a * b + c

# Application sur les arrays numpy sous-jacents
valeurs_a = grand_df['A'].values
valeurs_b = grand_df['B'].values
valeurs_c = grand_df['C'].values

start = time.time()
resultat_numba = calcul_rapide(valeurs_a, valeurs_b, valeurs_c)
temps_numba = time.time() - start

print(f"\nAvec Numba: {temps_numba:.4f}s")

# Utilisation avec pandas
grand_df['resultat_numba'] = calcul_rapide(valeurs_a, valeurs_b,
   valeurs_c)
```

4.3 Techniques d'Optimisation Avancées

```
# 1. Utilisation de cat gories pour les donn es textuelles
   r p titives
df_categories = df.copy()
df_categories['Ann e'] = df_categories['Ann e'].astype('category')
print(f"\n conomie m moire: {df['Ann e'].nbytes} -> {df_categories['
   Ann e'].nbytes} bytes")

# 2. Query optimization
resultat_query = df.query("ventes > 3000 and profits > 500")
print("\nR sultat query optimis :")
print(resultat_query)
```

```python

# 3. M moire usage optimization
def optimiser_memoire(df):
    """Optimise l'usage m moire d'un DataFrame"""
    for col in df.columns:
        if df[col].dtype == 'float64':
            df[col] = df[col].astype('float32')
        elif df[col].dtype == 'int64':
            df[col] = df[col].astype('int32')
    return df

df_opt = optimiser_memoire(df.copy())
print(f"\nM moire originale: {df.memory_usage(deep=True).sum()} bytes")
print(f"M moire optimis e: {df_opt.memory_usage(deep=True).sum()} bytes
    ")
```

4.4 Mathématiques de l'Optimisation

— **Complexité algorithmique** : Choix d'algorithmes $O(n)$ vs $O(n^2)$
— **Vectorisation** : Utilisation d'instructions SIMD
— **Localité mémoire** : Accès contigus vs aléatoires
— **Parallélisation** : Distribution des calculs

4.5 Valeur Ajoutée

Avantage	Description
Performance	Réduction temps calcul 10x-1000x
Échelle	Traitement de datasets massifs
Coût	Réduction infrastructure nécessaire
Productivité	Temps développement réduit

TABLE 4 – Valeur ajoutée de l'optimisation

5 Challenges Pratiques

5.1 Challenge 1 : Analyse Financière Multi-Niveaux

Créez un système d'analyse financière avec MultiIndex :

```python
def challenge_finance():
    # TODO: Cr er un MultiIndex avec: [Entreprise, Ann e , Trimestre]
    # Donn es: CA, b n fice , d penses
    # Analyses requises:
    # 1. Croissance trimestrielle par entreprise
    # 2. Marges par ann e
    # 3. Performance relative entre entreprises
```

```
    # Votre code ici
    pass
```

5.2 Challenge 2 : Système de Monitoring Temporel

Développez un système de monitoring avec séries temporelles :

```
def challenge_monitoring():
    # TODO: Cr er des s ries temporelles de m triques syst me
    # CPU, m moire, r seau, stockage
    # Analyses requises:
    # 1. D tection d'anomalies avec rolling stats
    # 2. Resampling pour rapports horaires/quotidiens
    # 3. Features engineering pour ML

    # Votre code ici
    pass
```

5.3 Challenge 3 : Optimisation de Performance

Optimisez un traitement de données massives :

```
def challenge_optimisation():
    # TODO: Prendre un dataset de 1M+ lignes
    # Optimiser:
    # 1. Temps de traitement (vectorisation, numba)
    # 2. Usage m moire (types, cat gories)
    # 3. Requ tes complexes (query, indexing)

    # Votre code ici
    pass
```

6 Conclusion

Points Clés à Retenir

- **MultiIndex** : Structure hiérarchique pour données complexes
- **Agrégations avancées** : Flexibilité dans le calcul des métriques
- **Séries temporelles** : Analyse sophistiquée des données temporelles
- **Optimisation** : Techniques cruciales pour les grands datasets
- **Vectorisation** : Clé pour les performances avec Pandas

6.1 Performances Typiques

6.2 Recommandations Finales

1. **Préférer MultiIndex** pour les données hiérarchiques

Technique	Temps d'exécution	Usage mémoire
Boucles Python	100%	100%
Apply pandas	50-80%	100-120%
Opérations vectorisées	1-10%	100%
Numba	0.1-5%	100%

TABLE 5 – Comparaison des performances (valeurs relatives)

2. **Utiliser les agrégations dictionnaires** pour la flexibilité
3. **Exploiter les séries temporelles** pour l'analyse temporelle
4. **Toujours vectoriser** les opérations sur les grands datasets
5. **Profiler régulièrement** les performances