AIDE-MÉMOIRE PYTHON / ANALYSE DE DONNÉES / DÉBUTANT

Filtrer des lignes avec pandas

Filtrez les lignes d'un DataFrame pandas de façon sûre et lisible grâce aux masques booléens, aux conditions multiples, aux tests d'appartenance, aux filtres sur le texte et les dates, aux contrôles des valeurs nulles et aux expressions query.

Mis à jour le 4 oct. 2026 Version en ligne →



PROBLÈME

Vous devez conserver uniquement les lignes qui répondent à une condition métier :

- les commandes au-dessus d'un montant
- les enregistrements de certains pays
- les lignes comprises entre deux dates
- les clients dont des données sont manquantes
- les statuts figurant dans une liste autorisée
- le texte qui contient un motif
- les lignes qui répondent à plusieurs conditions à la fois

La difficulté ne tient pas seulement à la syntaxe.

Vous voulez un filtrage qui soit :

- correct
- lisible
- sûr face aux valeurs nulles
- facile à déboguer
- sûr à maintenir dans du code de production

RÉPONSE RAPIDE

Utilisez des masques booléens avec `.loc[]`.

PYTHON
```python
filtered = df.loc[
    (df["status"] == "active")
    & (df["amount"] >= 100)
]
```

Pour l'appartenance :

PYTHON
```python
filtered = df.loc[
    df["country"].isin(["US", "CA", "MX"])
]
```

Pour les dates :

```python
filtered = df.loc[
    df["event_time"].between(
        pd.Timestamp("2026-10-01"),
        pd.Timestamp("2026-10-31"),
        inclusive="both",
    )
]
```

Préférez `.loc[mask]` dans le code de production, car l'intention est explicite et facile à étendre.

Code / commandes

1. Filtrer avec une seule condition

```python
filtered = df.loc[df["amount"] > 100]
```

Utilisez cette forme pour une condition numérique ou d'égalité simple.

Exemples :

```python
active = df.loc[df["status"] == "active"]
failed = df.loc[df["status"] == "failed"]
large_orders = df.loc[df["amount"] >= 1000]
```

2. Filtrer avec plusieurs conditions ET

Utilisez `&` et placez chaque comparaison entre parenthèses.

```python
filtered = df.loc[
    (df["status"] == "active")
    & (df["amount"] >= 100)
    & (df["country"] == "US")
]
```

N'utilisez pas le `and` de Python entre des Series.

Incorrect :

```python
df["status"] == "active" and df["amount"] >= 100
```

Correct :

```python
(df["status"] == "active") & (df["amount"] >= 100)
```

3. Filtrer avec des conditions OU

Utilisez `|`.

```python
filtered = df.loc[
    (df["status"] == "failed")
    | (df["status"] == "cancelled")
]
```

Lorsque les valeurs d'égalité possibles sont nombreuses, `.isin()` est plus propre.

```python
filtered = df.loc[
    df["status"].isin(["failed", "cancelled"])
]
```

4. Combiner ET et OU avec prudence

```python
filtered = df.loc[
    (df["country"] == "US")
    & (
        (df["status"] == "active")
        | (df["priority"] == "high")
    )
]
```

Les parenthèses comptent.

Ne vous reposez pas sur la priorité des opérateurs lorsque l'expression est critique pour le métier.

5. Exclure des lignes avec `~`

```python
filtered = df.loc[
    ~df["status"].isin(["cancelled", "deleted"])
]
```

Cela signifie :

> conserver les lignes dont le statut ne figure pas dans la liste

Pour une seule valeur :

```python
filtered = df.loc[df["status"] != "cancelled"]
```

6. Filtrer avec `.isin()`

```python
allowed_countries = ["US", "CA", "MX"]

filtered = df.loc[
    df["country"].isin(allowed_countries)
]
```

Utilisez `.isin()` lorsque vous disposez d'une liste, d'un ensemble, d'une Series ou d'une autre collection de valeurs acceptées.

C'est généralement plus clair que d'enchaîner de nombreuses conditions OU.

7. Filtrer les lignes nulles et non nulles

Lignes nulles :

```python
missing_email = df.loc[
    df["email"].isna()
]
```

Lignes non nulles :

```python
has_email = df.loc[
    df["email"].notna()
]
```

Ne comparez pas avec `== None` ni avec `== np.nan`.

Utilisez les méthodes de pandas dédiées aux valeurs nulles.

8. Filtrer sur une plage numérique

Utilisez `.between()`.

```python
filtered = df.loc[
    df["amount"].between(100, 500, inclusive="both")
]
```

Équivalent :

```python
filtered = df.loc[
    (df["amount"] >= 100)
    & (df["amount"] <= 500)
]
```

`.between()` est souvent plus facile à lire.

9. Filtrer correctement les dates

Assurez-vous d'abord que la colonne est de type datetime :

```python
df["event_time"] = pd.to_datetime(
    df["event_time"],
    errors="coerce",
)
```

Puis filtrez :

```python
start = pd.Timestamp("2026-10-01")
end = pd.Timestamp("2026-10-31")

filtered = df.loc[
    df["event_time"].between(
        start,
        end,
        inclusive="both",
    )
]
```

Si les horodatages comportent une heure, soyez attentif à la logique de fin de journée.

Pour un intervalle mensuel semi-ouvert, cette forme est souvent plus sûre :

```python
filtered = df.loc[
    (df["event_time"] >= pd.Timestamp("2026-10-01"))
    & (df["event_time"] < pd.Timestamp("2026-11-01"))
]
```

Elle évite d'exclure par accident des lignes plus tardives du 31 octobre.

10. Filtrer du texte avec `.str.contains()`

```python
filtered = df.loc[
    df["description"]
    .str.contains("error", case=False, na=False)
]
```

Options importantes :

- `case=False` → insensible à la casse
- `na=False` → les valeurs nulles deviennent `False`
- `regex=` → détermine si le motif est traité comme une expression régulière

Pour du texte littéral :

```python
filtered = df.loc[
    df["description"]
    .str.contains(
        "a+b",
        case=False,
        na=False,
        regex=False,
    )
]
```

Utilisez `regex=False` lorsque le texte recherché ne doit pas être interprété comme une expression régulière.

11. Filtrer sur un préfixe ou un suffixe de texte

Commence par :

```python
filtered = df.loc[
    df["code"].str.startswith("ABC", na=False)
]
```

Se termine par :

```python
filtered = df.loc[
    df["filename"].str.endswith(".csv", na=False)
]
```

12. Filtrer sur un texte exact après normalisation

Lorsque les données sources peuvent présenter une casse ou des espaces incohérents :

```python
normalized_status = (
    df["status"]
    .astype("string")
    .str.strip()
    .str.lower()
)

filtered = df.loc[
    normalized_status.eq("active")
]
```

C'est plus sûr que de modifier définitivement la colonne d'origine pendant l'analyse initiale.

13. Filtrer avec un masque booléen réutilisable

```python
mask = (
    (df["status"] == "active")
    & (df["amount"] > 100)
    & df["country"].isin(["US", "CA"])
)

filtered = df.loc[mask]
```

C'est utile lorsque :

- la condition est complexe
- vous voulez inspecter le masque
- vous réutilisez la condition
- vous voulez faciliter le débogage

Vérifiez combien de lignes correspondent :

```python
print(mask.sum())
```

14. Filtrer et sélectionner des colonnes en même temps

```python
filtered = df.loc[
    df["status"] == "active",
    ["customer_id", "amount", "event_time"],
]
```

Cela peut réduire la mémoire utilisée et clarifier la logique en aval.

15. Utiliser `.query()` pour des expressions lisibles

```python
filtered = df.query(
    "status == 'active' and amount >= 100"
)
```

Les variables externes s'utilisent avec `@` :

```python
min_amount = 100

filtered = df.query(
    "status == 'active' and amount >= @min_amount"
)
```

`.query()` peut être très lisible pour des expressions simples.

Préférez les masques booléens lorsque :

- les noms de colonnes sont peu pratiques
- vous avez besoin de méthodes de chaîne complexes
- vous avez besoin d'une gestion fine des valeurs nulles
- l'expression est plus facile à déboguer sous forme explicite

16. Filtrer en fonction d'une autre colonne

```python
filtered = df.loc[
    df["actual_amount"] > df["expected_amount"]
]
```

Utile pour les comparaisons au niveau de la ligne.

Exemple :

```python
late = df.loc[
    df["actual_date"] > df["expected_date"]
]
```

17. Filtrer des groupes selon une logique d'agrégation

Supposons que vous vouliez les clients dont le chiffre d'affaires total dépasse 10 000.

```python
customer_totals = (
    df.groupby("customer_id")["revenue"]
    .transform("sum")
)

filtered = df.loc[
    customer_totals > 10_000
]
```

Utilisez `transform()` lorsque vous avez besoin d'une logique au niveau du groupe tout en conservant la granularité d'origine des lignes.

18. Conserver les N premières lignes après le filtrage

```python
filtered = (
    df.loc[df["status"] == "active"]
    .nlargest(10, "amount")
)
```

Pour les plus petites valeurs :

```python
filtered = (
    df.loc[df["status"] == "active"]
    .nsmallest(10, "amount")
)
```

C'est plus direct que de trier tout le DataFrame lorsque seules les premières lignes vous intéressent.

19. Filtrer les doublons par clé

Ne conserver que les lignes dont les clés métier sont en double :

```python
duplicates = df.loc[
    df.duplicated(
        subset=["customer_id", "order_id"],
        keep=False,
    )
]
```

Utilisez cette forme pour enquêter avant de décider comment dédoublonner.

20. Filtrer les lignes qui correspondent aux valeurs d'un autre DataFrame

```python
valid_ids = lookup_df["customer_id"]

filtered = df.loc[
    df["customer_id"].isin(valid_ids)
]
```

Pour une mise en correspondance relationnelle importante qui nécessite des colonnes des deux jeux de données, un merge est souvent plus approprié.

Utilisez `.isin()` pour un filtrage simple de type semi-jointure.

Interpréter le résultat

Un filtre produit une décision booléenne pour chaque ligne.

Par exemple :

```python
mask = df["amount"] > 100
```

produit, conceptuellement, des valeurs comme :

```
True
False
True
False
...
```

Ensuite :

```python
df.loc[mask]
```

ne conserve que les lignes `True`.

Comportements importants :

Motif	Signification
`&`	ET élément par élément
`	`	`
`~`	NON élément par élément
`.isin()`	test d'appartenance
`.isna()`	valeur manquante
`.notna()`	valeur non manquante
`.between()`	test de plage inclusive/exclusive
`.str.contains()`	correspondance de texte
`.eq()`	comparaison d'égalité
`.ne()`	différent de
`.gt()` / `.ge()`	supérieur à / supérieur ou égal
`.lt()` / `.le()`	inférieur à / inférieur ou égal

Points à vérifier

Le nombre de résultats est nul de manière inattendue

Vérifiez :

- une incohérence de dtype
- les espaces
- la casse
- l'analyse des dates
- les valeurs nulles
- un mauvais regroupement des conditions booléennes
- une incohérence de fuseau horaire

Exemple :

```python
print(df["status"].value_counts(dropna=False))
```

avant de supposer que `"active"` existe exactement tel quel.

Le nombre de résultats est anormalement élevé

Vérifiez si les conditions OU sont trop larges.

Exemple :

```python
mask = (
    (df["country"] == "US")
    | (df["status"] == "active")
)
```

Ce masque conserve les lignes qui répondent à l'une ou l'autre condition.

Si vous vouliez les deux, utilisez `&`.

Les valeurs nulles disparaissent de manière inattendue

Les comparaisons avec des valeurs nulles donnent généralement des résultats assimilés à faux lors du filtrage.

Si les valeurs nulles comptent, traitez-les explicitement.

Exemple :

```python
mask = (
    df["amount"].isna()
    | (df["amount"] > 100)
)
```

Le filtre sur le texte manque des correspondances évidentes

Vérifiez :

- la casse
- les espaces en début et en fin
- l'interprétation comme expression régulière
- les valeurs nulles

Utilisez une normalisation ou des options explicites.

Le filtre sur les dates manque les lignes de fin de journée

Cela se produit lorsque minuit sert de borne supérieure.

Préférez les intervalles semi-ouverts :

```python
start <= timestamp < next_period_start
```

pour les fenêtres d'une journée ou d'un mois complets.

> **NOTES DE PRODUCTION**
>
> ### Préférez `.loc[]` pour un filtrage explicite
>
> Ces deux formes sont valides :
>
> ```python
> df[df["amount"] > 100]
> ```
>
> ```python
> df.loc[df["amount"] > 100]
> ```
>
> `.loc[]` est souvent plus clair lorsque vous ajoutez ensuite une sélection de colonnes ou des affectations.
>
> ### Évitez l'affectation chaînée après un filtrage
>
> Forme risquée :
>
> ```python
> df[df["amount"] > 100]["flag"] = True
> ```
>
> Utilisez plutôt :
>
> ```python
> df.loc[df["amount"] > 100, "flag"] = True
> ```
>
> C'est plus clair et cela évite l'ambiguïté de l'affectation chaînée.

Les filtres parcourent les colonnes concernées

Le filtrage booléen est généralement en O(n) sur les lignes concernées.

Sur de très grands DataFrames :

- filtrez tôt
- ne sélectionnez que les colonnes nécessaires
- évitez de recalculer des masques équivalents
- envisagez le predicate pushdown à la lecture lorsqu'il est pris en charge
- envisagez un moteur distribué si le jeu de données ne tient plus confortablement en mémoire

Les expressions régulières peuvent coûter cher

`str.contains()` interprète le motif comme une expression régulière par défaut.

Pour une correspondance littérale, utilisez :

```python
regex=False
```

Cela évite aussi une sémantique d'expression régulière involontaire.

Les fuseaux horaires comptent

Comparer des horodatages avec et sans fuseau horaire peut échouer ou produire une logique incorrecte s'ils sont traités de façon incohérente.

Normalisez le modèle temporel avant de filtrer.

La sémantique des valeurs nulles doit être délibérée

Ne supposez pas que les valeurs manquantes doivent être exclues.

Les règles métier peuvent exiger de :

- conserver les valeurs nulles
- exclure les valeurs nulles
- traiter la valeur nulle comme inconnue
- orienter les valeurs nulles vers une correction

Exprimez cette intention explicitement.

ERREURS FRÉQUENTES

1. Utiliser `and` / `or` au lieu de `&` / `|`

Incorrect :

```python
(df["a"] > 1) and (df["b"] < 5)
```

Correct :

```python
(df["a"] > 1) & (df["b"] < 5)
```

2. Oublier les parenthèses

Incorrect ou ambigu :

```python
df["a"] > 1 & df["b"] < 5
```

Correct :

```python
(df["a"] > 1) & (df["b"] < 5)
```

3. Comparer les valeurs nulles avec `== np.nan`

Utilisez :

```python
df["col"].isna()
```

et non :

```python
df["col"] == np.nan
```

4. Utiliser une expression régulière par accident

Ceci :

```python
str.contains("a+b")
```

traite `+` comme de la syntaxe d'expression régulière.

Utilisez :

```python
str.contains("a+b", regex=False)
```

pour du texte littéral.

5. Filtrer des dates encore stockées sous forme de chaînes

L'ordre lexical des chaînes ne remplace pas de façon fiable la sémantique de dates analysées.

Convertissez d'abord.

6. Écraser les colonnes sources pendant l'exploration

Construisez d'abord des Series dérivées lorsque vous validez des transformations.

Modifiez ensuite les données délibérément.

7. Filtrer après des opérations coûteuses

Si vous pouvez réduire le jeu de données plus tôt, faites-le.

Filtrer avant les jointures ou les agrégations peut réduire nettement la mémoire et le travail nécessaires.

ÉTAPE SUIVANTE

Utilisez le résultat du filtrage pour décider de la suite.

- **Besoin de totaux par groupe** → passez à GroupBy et agrégations avec pandas.
- **Besoin de données d'une autre table ou d'un autre DataFrame** → utilisez Joindre et fusionner des DataFrames.
- **Comportement inattendu des valeurs nulles** → examinez la sémantique des valeurs manquantes.
- **Logique de dates complexe** → utilisez l'aide-mémoire Dates and Timestamps.
- **Le résultat filtré contient des clés en double** → examinez la granularité métier et le dédoublonnage.
- **Le filtrage est lent sur de très gros volumes** → réduisez les colonnes ou les lignes plus tôt, ou rapprochez l'opération de la source de données.
- **Le même filtre est réutilisé à plusieurs reprises** → construisez et nommez un masque réutilisable.

Pour aller plus loin

- Aide-mémoire associé : Inspecter un DataFrame pandas
- Aide-mémoire associé : GroupBy et agrégations avec pandas
- Aide-mémoire associé : Joindre et fusionner des DataFrames
- Aide-mémoire associé : Gérer les valeurs manquantes avec pandas
- Aide-mémoire associé : Manipuler des dates et des horodatages
- Aide-mémoire associé : Trier, classer et dédoublonner des données
- Aide-mémoire associé : Déboguer les problèmes courants de pandas