GUÍA RÁPIDA PYTHON / ANÁLISIS DE DATOS / PRINCIPIANTE

# Filtrar filas en pandas

Filtra filas de un DataFrame de pandas de forma segura y clara con máscaras booleanas, condiciones múltiples, pruebas de pertenencia, filtros de texto y fechas, comprobaciones de nulos y expresiones query.

Actualizado 4 oct 2026 Versión en línea →



**PROBLEMA**

Necesitas conservar solo las filas que cumplen una condición de negocio:

- pedidos por encima de un importe
- registros de países concretos
- filas entre dos fechas
- clientes con datos faltantes
- estados incluidos en una lista permitida
- texto que contiene un patrón
- filas que cumplen varias condiciones a la vez

El reto no es solo la sintaxis.

Quieres un filtrado que sea:

- correcto
- legible
- seguro ante nulos
- fácil de depurar
- seguro de mantener en código de producción

## RESPUESTA RÁPIDA

Usa máscaras booleanas con `.loc[]`.

PYTHON

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

Para la pertenencia:

PYTHON

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

Para las fechas:

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

Prefiere `.loc[mask]` en código de producción, porque la intención es explícita y fácil de ampliar.

## Código / comandos

### 1. Filtrar con una condición

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

Úsalo para una condición numérica o de igualdad sencilla.

Ejemplos:

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

### 2. Filtrar con varias condiciones Y

Usa `&` y envuelve cada comparación entre paréntesis.

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

No uses el `and` de Python entre Series.

Incorrecto:

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

Correcto:

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

### 3. Filtrar con condiciones O

Usa `|`.

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

Cuando hay muchas opciones de igualdad, `.isin()` es más limpio.

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

### 4. Combinar Y y O con cuidado

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

Los paréntesis importan.

No confíes en la precedencia de operadores cuando la expresión es crítica para el negocio.

### 5. Excluir filas con `~`

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

Esto significa:

> conservar las filas cuyo estado no está en la lista

Para un solo valor:

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

### 6. Filtrar con `.isin()`

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

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

Usa `.isin()` cuando tengas una lista, un conjunto, una Series u otra colección de valores aceptados.

Suele ser más claro que encadenar muchas condiciones O.

### 7. Filtrar filas nulas y no nulas

Filas nulas:

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

Filas no nulas:

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

No compares con `== None` ni con `== np.nan`.

Usa los métodos de pandas para nulos.

### 8. Filtrar por rango numérico

Usa `.between()`.

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

Equivalente:

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

`.between()` suele ser más fácil de leer.

### 9. Filtrar fechas correctamente

Primero asegúrate de que la columna sea de tipo datetime:

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

Después filtra:

```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 las marcas de tiempo incluyen la hora del día, ten cuidado con la lógica de fin de día.

Para un intervalo mensual semiabierto, esta forma suele ser más segura:

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

Así evitas excluir por accidente filas de horas posteriores del 31 de octubre.

### 10. Filtrar texto con `.str.contains()`

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

Opciones importantes:

- `case=False` → no distingue entre mayúsculas y minúsculas
- `na=False` → los valores nulos pasan a ser `False`
- `regex=` → controla si el patrón se trata como expresión regular

Para texto literal:

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

Usa `regex=False` cuando el texto de búsqueda no deba interpretarse como expresión regular.

### 11. Filtrar por prefijo o sufijo de texto

Empieza por:

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

Termina en:

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

### 12. Filtrar texto exacto después de normalizarlo

Cuando los datos de origen pueden tener mayúsculas, minúsculas o espacios incoherentes:

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

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

Es más seguro que modificar permanentemente la columna original durante el análisis inicial.

### 13. Filtrar con una máscara booleana reutilizable

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

filtered = df.loc[mask]
```

Es útil cuando:

- la condición es compleja
- quieres inspeccionar la máscara
- reutilizas la condición
- quieres facilitar la depuración

Comprueba cuántas filas coinciden:

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

### 14. Filtrar y seleccionar columnas a la vez

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

Esto puede reducir el uso de memoria y aclarar la lógica posterior.

### 15. Usar `.query()` para expresiones legibles

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

Las variables externas se usan con `@`:

```python
min_amount = 100

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

`.query()` puede ser muy legible para expresiones sencillas.

Prefiere las máscaras booleanas cuando:

- los nombres de columna son incómodos
- necesitas métodos de cadena complejos
- necesitas un manejo detallado de los nulos
- la expresión es más fácil de depurar de forma explícita

### 16. Filtrar en función de otra columna

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

Útil para comparaciones a nivel de fila.

Ejemplo:

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

### 17. Filtrar grupos según una lógica de agregación

Supón que quieres los clientes cuyos ingresos totales superan 10.000.

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

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

Usa `transform()` cuando necesites lógica a nivel de grupo conservando la granularidad original de las filas.

### 18. Conservar las N primeras filas después de filtrar

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

Para los valores más pequeños:

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

Es más directo que ordenar todo el DataFrame cuando solo necesitas las primeras filas.

### 19. Filtrar duplicados por clave

Conserva solo las filas con claves de negocio duplicadas:

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

Úsalo para investigar antes de decidir cómo deduplicar.

### 20. Filtrar filas que coinciden con valores de otro DataFrame

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

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

Para una correspondencia relacional amplia que necesita columnas de ambos conjuntos de datos, un merge suele ser más adecuado.

Usa `.isin()` para un filtrado sencillo de tipo semi-join.

## Qué significa el resultado

Un filtro genera una decisión booleana para cada fila.

Por ejemplo:

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

produce, conceptualmente, valores como:

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

Después:

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

conserva solo las filas `True`.

Comportamientos importantes:

| Patrón | Significado |
|---|---|
| `&` | Y elemento a elemento |
| ` | ` | ` |
| `~` | NO elemento a elemento |
| `.isin()` | prueba de pertenencia |
| `.isna()` | valor faltante |
| `.notna()` | valor no faltante |
| `.between()` | prueba de rango inclusivo/exclusivo |
| `.str.contains()` | coincidencia de texto |
| `.eq()` | comparación de igualdad |
| `.ne()` | distinto de |
| `.gt()` / `.ge()` | mayor que / mayor o igual que |
| `.lt()` / `.le()` | menor que / menor o igual que |

## Qué revisar

### El número de resultados es cero de forma inesperada

Comprueba:

- discrepancia de dtype
- espacios en blanco
- mayúsculas y minúsculas
- conversión de fechas
- nulos
- agrupación booleana incorrecta
- discrepancia de zona horaria

Ejemplo:

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

antes de dar por hecho que `"active"` existe exactamente como está escrito.

### El número de resultados es inesperadamente alto

Comprueba si las condiciones O son demasiado amplias.

Ejemplo:

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

Esto conserva las filas que cumplen cualquiera de las dos condiciones.

Si querías ambas, usa `&`.

### Los nulos desaparecen de forma inesperada

Las comparaciones con valores nulos suelen dar resultados equivalentes a falso al filtrar.

Si los nulos importan, trátalos de forma explícita.

Ejemplo:

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

### El filtro de texto no encuentra coincidencias evidentes

Comprueba:

- mayúsculas y minúsculas
- espacios al principio o al final
- interpretación como expresión regular
- valores nulos

Usa normalización u opciones explícitas.

### El filtro de fechas pierde las filas del final del día

Ocurre cuando se usa la medianoche como límite superior.

Prefiere los intervalos semiabiertos:

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

para ventanas de un día o un mes completos.

> **NOTAS DE PRODUCCIÓN**
>
> ### Prefiere `.loc[]` para un filtrado explícito
>
> Ambas formas son válidas:
>
> ```python
> df[df["amount"] > 100]
> ```
>
> ```python
> df.loc[df["amount"] > 100]
> ```
>
> `.loc[]` suele ser más claro cuando más adelante agregas selección de columnas o asignaciones.
>
> ### Evita la asignación encadenada después de filtrar
>
> Patrón arriesgado:
>
> ```python
> df[df["amount"] > 100]["flag"] = True
> ```
>
> Usa:
>
> ```python
> df.loc[df["amount"] > 100, "flag"] = True
> ```
>
> Es más claro y evita la ambigüedad de la asignación encadenada.

### Los filtros recorren las columnas implicadas

El filtrado booleano suele ser O(n) sobre las filas implicadas.

En DataFrames muy grandes:

- filtra pronto
- selecciona solo las columnas necesarias
- evita repetir máscaras equivalentes
- considera el predicate pushdown en la lectura cuando esté disponible
- considera un motor distribuido si el conjunto de datos ya no cabe cómodamente en memoria

### Las expresiones regulares pueden ser costosas

`str.contains()` trata el patrón como expresión regular de forma predeterminada.

Para una coincidencia literal, usa:

```python
regex=False
```

Esto también evita una semántica de expresión regular accidental.

### Las zonas horarias importan

Comparar marcas de tiempo con y sin zona horaria puede fallar o producir una lógica incorrecta si se tratan de forma incoherente.

Normaliza el modelo temporal antes de filtrar.

### La semántica de los nulos debe ser deliberada

No des por hecho que los valores faltantes deben excluirse.

Las reglas de negocio pueden exigir:

- conservar los nulos
- excluir los nulos
- tratar el nulo como desconocido
- derivar los nulos para su corrección

Expresa esa intención de forma explícita.

## ERRORES COMUNES

### 1. Usar `and` / `or` en lugar de `&` / `|`

Incorrecto:

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

Correcto:

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

### 2. Olvidar los paréntesis

Incorrecto o ambiguo:

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

Correcto:

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

### 3. Comparar nulos con `== np.nan`

Usa:

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

no:

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

### 4. Usar expresiones regulares sin querer

Esto:

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

trata `+` como sintaxis de expresión regular.

Usa:

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

para texto literal.

### 5. Filtrar fechas mientras siguen siendo cadenas

El orden lexicográfico de las cadenas no sustituye de forma fiable a la semántica de fechas ya convertidas.

Conviértelas primero.

### 6. Sobrescribir columnas de origen durante la exploración

Construye primero Series derivadas cuando valides transformaciones.

Después, modifica los datos de forma deliberada.

### 7. Filtrar después de operaciones costosas

Si puedes reducir el conjunto de datos antes, hazlo.

Filtrar antes de joins y agregaciones puede reducir mucho la memoria y el trabajo.

**SIGUIENTE PASO**

Usa el resultado del filtrado para decidir qué hacer después.

- **Necesitas totales por grupo** → pasa a GroupBy y agregaciones en pandas.
- **Necesitas datos de otra tabla o de otro DataFrame** → usa Combinar DataFrames con merge y join.
- **Comportamiento inesperado de los nulos** → revisa la semántica de los valores faltantes.
- **La lógica de fechas es compleja** → usa la guía rápida Dates and Timestamps.
- **El resultado filtrado tiene claves duplicadas** → revisa la granularidad de negocio y la deduplicación.
- **El filtrado es lento con datos muy grandes** → reduce columnas o filas antes, o acerca la operación a la fuente de datos.
- **El mismo filtro se reutiliza una y otra vez** → construye una máscara reutilizable y ponle nombre.

## Para profundizar

- Guía rápida relacionada: Inspeccionar un DataFrame de pandas
- Guía rápida relacionada: GroupBy y agregaciones en pandas
- Guía rápida relacionada: Combinar DataFrames con merge y join
- Guía rápida relacionada: Gestionar valores faltantes en pandas
- Guía rápida relacionada: Trabajar con fechas y marcas de tiempo
- Guía rápida relacionada: Ordenar, clasificar y deduplicar datos
- Guía rápida relacionada: Depurar problemas comunes de pandas