pandas / polars

Lecture 09

Dr. Colin Rundel

Filtering rows

The `query()` method can be used for filtering rows, it evaluates a string expression in the context of the data frame.

```
df.query('date == "2022-02-01"')
```

```
Empty DataFrame
Columns: [id, weight, height, date]
Index: []
```

```
df.query('weight > 50')
```

```
        id     weight      height        date
anna   202  79.477217  162.607949 2025-02-01
bob    535  97.369002  175.888696 2025-02-02
carol  960  51.663463  156.062230 2025-02-03
dave   370  67.517056  171.197477 2025-02-04
```

```
df.query('weight > 50 & height < 165')
```

```
        id     weight      height        date
anna   202  79.477217  162.607949 2025-02-01
carol  960  51.663463  156.062230 2025-02-03
```

```
qid = 202
df.query('id == @qid')
```

```
       id     weight      height        date
anna  202  79.477217  162.607949 2025-02-01
```

Selecting Columns

Beyond the use of `loc()` and `iloc()` there is also the `filter()` method which can be used to select columns (or indices) by name with pattern matching

```python
df.filter(items=["id","weight"])
```

```
        id     weight
anna   202  79.477217
bob    535  97.369002
carol  960  51.663463
dave   370  67.517056
erin   206  29.780742
```

```python
df.filter(regex="ght$")
```

```
          weight      height
anna   79.477217  162.607949
bob    97.369002  175.888696
carol  51.663463  156.062230
dave   67.517056  171.197477
erin   29.780742  167.607252
```

```python
df.filter(like = "i")
```

```
        id     weight      height
anna   202  79.477217  162.607949
bob    535  97.369002  175.888696
carol  960  51.663463  156.062230
dave   370  67.517056  171.197477
erin   206  29.780742  167.607252
```

```python
df.filter(like="a", axis=0)
```

```
        id     weight      height       date
anna   202  79.477217  162.607949 2025-02-01
carol  960  51.663463  156.062230 2025-02-03
dave   370  67.517056  171.197477 2025-02-04
```

Adding columns

Indexing with assignment allows for inplace modification of a DataFrame, while `assign()` creates a new object (but is chainable)

```
df['student'] = [True, True, True, False, None]
df['age'] = [19, 22, 25, None, None]
df
```

```
        id     weight      height       date student   age
anna   202  79.477217  162.607949 2025-02-01    True  19.0
bob    535  97.369002  175.888696 2025-02-02    True  22.0
carol  960  51.663463  156.062230 2025-02-03    True  25.0
dave   370  67.517056  171.197477 2025-02-04   False   NaN
erin   206  29.780742  167.607252 2025-02-05    None   NaN
```

```
df.assign(
  student = lambda x: np.where(x.student, "yes", "no"),
  rand = np.random.rand(5)
)
```

```
        id     weight      height       date student   age      rand
anna   202  79.477217  162.607949 2025-02-01     yes  19.0  0.938553
bob    535  97.369002  175.888696 2025-02-02     yes  22.0  0.000779
carol  960  51.663463  156.062230 2025-02-03     yes  25.0  0.992212
dave   370  67.517056  171.197477 2025-02-04      no   NaN  0.617482
erin   206  29.780742  167.607252 2025-02-05      no   NaN  0.611653
```

Removing columns (and rows)

Columns or rows can be removed via the `drop()` method,

```python
df.drop(['student'])
```

```
KeyError: "['student'] not found in axis"
```

```python
df.drop(['student'], axis=1)
```

```
        id     weight      height       date   age
anna   202  79.477217  162.607949 2025-02-01  19.0
bob    535  97.369002  175.888696 2025-02-02  22.0
carol  960  51.663463  156.062230 2025-02-03  25.0
dave   370  67.517056  171.197477 2025-02-04   NaN
erin   206  29.780742  167.607252 2025-02-05   NaN
```

```python
df.drop(['anna','dave'])
```

```
        id     weight      height       date student   age
bob    535  97.369002  175.888696 2025-02-02    True  22.0
carol  960  51.663463  156.062230 2025-02-03    True  25.0
erin   206  29.780742  167.607252 2025-02-05    None   NaN
```

```python
df.drop(columns = df.columns == "age")
```

```
KeyError: '[False, False, False, False, False, True] not found in axis'
```

```python
df.drop(columns = df.columns[df.columns == "age"])
```

```
        id     weight      height       date student
anna   202  79.477217  162.607949 2025-02-01    True
bob    535  97.369002  175.888696 2025-02-02    True
carol  960  51.663463  156.062230 2025-02-03    True
dave   370  67.517056  171.197477 2025-02-04   False
erin   206  29.780742  167.607252 2025-02-05    None
```

```python
df.drop(columns = df.columns[df.columns.str.contains("ght")])
```

```
        id       date student   age
anna   202 2025-02-01    True  19.0
bob    535 2025-02-02    True  22.0
carol  960 2025-02-03    True  25.0
dave   370 2025-02-04   False   NaN
erin   206 2025-02-05    None   NaN
```

Sorting

DataFrames can be sorted on one or more columns via `sort_values()`,

```
df
```

```
        id     weight      height       date student   age
anna   202  79.477217  162.607949 2025-02-01    True  19.0
bob    535  97.369002  175.888696 2025-02-02    True  22.0
carol  960  51.663463  156.062230 2025-02-03    True  25.0
dave   370  67.517056  171.197477 2025-02-04   False   NaN
erin   206  29.780742  167.607252 2025-02-05    None   NaN
```

```
df.sort_values(by=["student","id"], ascending=[True,False])
```

```
        id     weight      height       date student   age
dave   370  67.517056  171.197477 2025-02-04   False   NaN
carol  960  51.663463  156.062230 2025-02-03    True  25.0
bob    535  97.369002  175.888696 2025-02-02    True  22.0
anna   202  79.477217  162.607949 2025-02-01    True  19.0
erin   206  29.780742  167.607252 2025-02-05    None   NaN
```

join vs merge vs concat

All three can be used to combine data frames,

- `concat()` stacks DataFrames on either axis, with basic alignment based on (row) indexes. `join` argument only supports "inner" and "outer".
- `merge()` aligns based on one or more shared columns. `how` supports "inner", "outer", "left", "right", and "cross".
- `join()` uses `merge()` behind the scenes, but prefers to join based on (row) indexes. Also has different default `how` compared to `merge()`, "left" vs "inner".

Pivoting - long to wide

```
df
```

```
   country  year   type count
0        A  1999  cases  0.7K
1        A  1999    pop   19M
2        A  2000  cases    2K
3        A  2000    pop   20M
4        B  1999  cases   37K
5        B  1999    pop  172M
6        B  2000  cases   80K
7        B  2000    pop  174M
8        C  1999  cases  212K
9        C  1999    pop    1T
10       C  2000  cases  213K
11       C  2000    pop    1T
```

```
df_wide = df.pivot(
  index=["country","year"],
  columns="type",
  values="count"
)
df_wide
```

```
type          cases   pop
country year
A       1999   0.7K   19M
        2000     2K   20M
B       1999    37K  172M
        2000    80K  174M
C       1999   212K    1T
        2000   213K    1T
```

pivot indexes

```python
df_wide.index
```

```
MultiIndex([('A', 1999),
            ('A', 2000),
            ('B', 1999),
            ('B', 2000),
            ('C', 1999),
            ('C', 2000)],
           names=['country', 'year'])
```

```python
df_wide.columns
```

```
Index(['cases', 'pop'], dtype='object',
name='type')
```

```python
( df_wide
  .reset_index()
  .rename_axis(
    columns=None
  )
)
```

```
  country  year cases   pop
0       A  1999  0.7K   19M
1       A  2000    2K   20M
2       B  1999   37K  172M
3       B  2000   80K  174M
4       C  1999  212K    1T
5       C  2000  213K    1T
```

Wide to long (melt)

```
df
```

```
  country  1999  2000
0       A  0.7K    2K
1       B   37K   80K
2       C  212K  213K
```

```
df_long = df.melt(
  id_vars="country",
  var_name="year",
  value_name="value"
)
df_long
```

```
  country  year value
0       A  1999  0.7K
1       B  1999   37K
2       C  1999  212K
3       A  2000    2K
4       B  2000   80K
5       C  2000  213K
```

Exercise 1 - Tidying

How would you tidy the following data frame so that the rate column is split into cases and population columns?

```
df = pd.DataFrame({
  "country": ["A","A","B","B","C","C"],
  "year":    [1999, 2000, 1999, 2000, 1999, 2000],
  "rate":    ["0.7K/19M", "2K/20M", "37K/172M", "80K/174M", "212K/1T", "213K/1T"]
})
df
```

```
  country  year      rate
0       A  1999  0.7K/19M
1       A  2000    2K/20M
2       B  1999  37K/172M
3       B  2000  80K/174M
4       C  1999   212K/1T
5       C  2000   213K/1T
```

Split-Apply-Combine

cereal data

```python
cereal = pd.read_csv("https://sta663-sp25.github.io/slides/data/cereal.csv")
cereal
```

```
                         name             mfr  ...  sugars     rating
0                   100% Bran         Nabisco  ...       6  68.402973
1           100% Natural Bran     Quaker Oats  ...       8  33.983679
2                    All-Bran       Kellogg's  ...       5  59.425505
3   All-Bran with Extra Fiber       Kellogg's  ...       0  93.704912
4              Almond Delight  Ralston Purina  ...       8  34.384843
..                        ...             ...  ...     ...        ...
72                    Triples   General Mills  ...       3  39.106174
73                       Trix   General Mills  ...      12  27.753301
74                 Wheat Chex  Ralston Purina  ...       3  49.787445
75                   Wheaties   General Mills  ...       3  51.592193
76       Wheaties Honey Gold   General Mills  ...       8  36.187559

[77 rows x 6 columns]
```

groupby

Groups can be created within a DataFrame via `groupby()` - these groups are then used by the standard summary methods (e.g. `sum()`, `mean()`, `std()`, etc.).

```
cereal.groupby("type")
```

```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x17bae3860>
```

```
cereal.groupby("type").groups
```

```
{'Cold': [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 21,
22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42,
44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 58, 59, 60, 61, 62, 63, 64, 65,
66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76], 'Hot': [20, 43, 57]}
```

```
cereal.groupby("mfr").groups
```

```
{'General Mills': [5, 7, 11, 12, 13, 14, 18, 22, 31, 36, 40, 42, 47, 51, 59, 69, 70,
71, 72, 73, 75, 76], 'Kellogg's': [2, 3, 6, 16, 17, 19, 21, 24, 25, 26, 28, 38, 39,
46, 48, 49, 50, 53, 58, 60, 62, 66, 67], 'Maltex': [43], 'Nabisco': [0, 20, 63, 64,
65, 68], 'Post': [9, 27, 29, 30, 32, 33, 34, 37, 52], 'Quaker Oats': [1, 10, 35, 41,
54, 55, 56, 57], 'Ralston Purina': [4, 8, 15, 23, 44, 45, 61, 74]}
```

groupby and arregation methods

```
cereal.groupby("type").mean()
```

```
TypeError: agg function failed [how->mean,dtype->object]
```

```
( cereal
  .groupby("type")
  .mean(numeric_only=True)
)
```

```
        calories    sugars     rating
type
Cold  107.162162  7.175676  42.095218
Hot   100.000000  1.333333  56.737708
```

```
cereal.groupby("mfr").size()
```

```
mfr
General Mills      22
Kellogg's          23
Maltex              1
Nabisco             6
Post                9
Quaker Oats         8
Ralston Purina      8
dtype: int64
```

Selecting groups

Groups can be accessed via `get_group()`

```
cereal.groupby("type").get_group("Hot")
```

```
                     name           mfr type  calories  sugars     rating
20  Cream of Wheat (Quick)       Nabisco  Hot       100       0  64.533816
43                   Maypo        Maltex  Hot       100       3  54.850917
57          Quaker Oatmeal   Quaker Oats  Hot       100       1  50.828392
```

```
cereal.groupby("mfr").get_group("Post")
```

```
                                     name   mfr  ... sugars     rating
9                             Bran Flakes  Post  ...      5  53.313813
27  Fruit & Fibre Dates; Walnuts; and Oats  Post  ...     10  40.917047
29                         Fruity Pebbles  Post  ...     12  28.025765
30                           Golden Crisp  Post  ...     15  35.252444
32                      Grape Nuts Flakes  Post  ...      5  52.076897
33                             Grape-Nuts  Post  ...      3  53.371007
34                     Great Grains Pecan  Post  ...      4  45.811716
37                             Honey-comb  Post  ...     11  28.742414
52                  Post Nat. Raisin Bran  Post  ...     14  37.840594

[9 rows x 6 columns]
```

Iterating groups

`DataFrameGroupBy`'s can also be iterated over,

```python
for name, group in cereal.groupby("type"):
  print(f"# {name}\n{group}\n\n")
```

```
# Cold
                         name             mfr  ...  sugars     rating
0                   100% Bran         Nabisco  ...       6  68.402973
1           100% Natural Bran     Quaker Oats  ...       8  33.983679
2                    All-Bran       Kellogg's  ...       5  59.425505
3   All-Bran with Extra Fiber       Kellogg's  ...       0  93.704912
4              Almond Delight  Ralston Purina  ...       8  34.384843
..                        ...             ...  ...     ...        ...
72                    Triples   General Mills  ...       3  39.106174
73                       Trix   General Mills  ...      12  27.753301
74                Wheat Chex  Ralston Purina  ...       3  49.787445
75                   Wheaties   General Mills  ...       3  51.592193
76        Wheaties Honey Gold   General Mills  ...       8  36.187559

[74 rows x 6 columns]


# Hot
```

Aggregation

The `aggregate()` function or `agg()` method can be used to compute summary statistics for each group,

```
cereal.groupby("mfr").agg("mean")
```

```
TypeError: agg function failed [how->mean,dtype->object]
```

```
cereal.groupby("mfr").agg("mean", numeric_only = True)
```

```
                 calories    sugars     rating
mfr
General Mills   111.363636  7.954545  34.485852
Kellogg's       108.695652  7.565217  44.038462
Maltex          100.000000  3.000000  54.850917
Nabisco          86.666667  1.833333  67.968567
Post            108.888889  8.777778  41.705744
Quaker Oats      95.000000  5.500000  42.915990
Ralston Purina  115.000000  6.125000  41.542997
```

Aggregation by column

```
cereal.groupby("mfr").agg({
  "calories": ['min', 'max'],
  "sugars":   ['median'],
  "rating":   ['sum', 'count']
})
```

```
               calories      sugars       rating
                    min  max median          sum count
mfr
General Mills       100  140    8.5   758.688737    22
Kellogg's            50  160    7.0  1012.884634    23
Maltex              100  100    3.0    54.850917     1
Nabisco              70  100    0.0   407.811403     6
Post                 90  120   10.0   375.351697     9
Quaker Oats          50  120    6.0   343.327919     8
Ralston Purina       90  150    5.5   332.343977     8
```

Named aggregation

It is also possible to use special syntax to aggregate specific columns into a named output column,

```
cereal.groupby("mfr", as_index=False).agg(
  min_cal = ("calories", "min"),
  max_cal = ("calories", max),
  med_sugar = ("sugars", "median"),
  avg_rating = ("rating", np.mean)
)
```

```
               mfr  min_cal  max_cal  med_sugar  avg_rating
0    General Mills      100      140        8.5   34.485852
1        Kellogg's       50      160        7.0   44.038462
2           Maltex      100      100        3.0   54.850917
3          Nabisco       70      100        0.0   67.968567
4             Post       90      120       10.0   41.705744
5      Quaker Oats       50      120        6.0   42.915990
6   Ralston Purina       90      150        5.5   41.542997
```

Transformation

The `transform()` method returns a DataFrame with the aggregated result matching the size (or length 1) of the input group(s),

```
( cereal
  .groupby("mfr")
  .transform(
    np.mean, numeric_only=True
  )
)
```

```
      calories    sugars    rating
0    86.666667  1.833333  67.968567
1    95.000000  5.500000  42.915990
2   108.695652  7.565217  44.038462
3   108.695652  7.565217  44.038462
4   115.000000  6.125000  41.542997
..         ...       ...        ...
72  111.363636  7.954545  34.485852
73  111.363636  7.954545  34.485852
74  115.000000  6.125000  41.542997
75  111.363636  7.954545  34.485852
76  111.363636  7.954545  34.485852

[77 rows x 3 columns]
```

```
( cereal
  .groupby("type")
  .transform(
    "mean", numeric_only=True
  )
)
```

```
      calories    sugars     rating
0   107.162162  7.175676  42.095218
1   107.162162  7.175676  42.095218
2   107.162162  7.175676  42.095218
3   107.162162  7.175676  42.095218
4   107.162162  7.175676  42.095218
..         ...       ...        ...
72  107.162162  7.175676  42.095218
73  107.162162  7.175676  42.095218
74  107.162162  7.175676  42.095218
75  107.162162  7.175676  42.095218
76  107.162162  7.175676  42.095218

[77 rows x 3 columns]
```

Practical transformation

`transform()` will generally be most useful via a user defined function, the lambda is applied to each column of each group.

```python
( cereal
  .drop(["name","type"], axis=1)
  .groupby("mfr")
  .transform( lambda x: (x - np.mean(x))/np.std(x, axis=0) )
)
```

```
    calories    sugars    rating
0  -1.767767  1.597191  0.086375
1   0.912871  0.559017 -0.568474
2  -1.780712 -0.582760  1.088220
3  -2.701081 -1.718649  3.512566
4  -0.235702  0.562544 -1.258442
..       ...       ...       ...
72 -0.134568 -1.309457  0.528580
73 -0.134568  1.069190 -0.770226
74 -0.707107 -0.937573  1.449419
75 -1.121403 -1.309457  1.957022
76 -0.134568  0.012013  0.194681

[77 rows x 3 columns]
```

Filtering groups

`filter()` also respects groups and allows for the inclusion / exclusion of groups based on user specified criteria,

filter | Group sizes

```
( cereal
  .groupby("mfr")
  .filter(lambda x: len(x) > 10)
)
```

```
                        name            mfr  ... sugars     rating
2                   All-Bran      Kellogg's  ...      5  59.425505
3  All-Bran with Extra Fiber      Kellogg's  ...      0  93.704912
5    Apple Cinnamon Cheerios  General Mills  ...     10  29.509541
6                Apple Jacks      Kellogg's  ...     14  33.174094
7                    Basic 4  General Mills  ...      8  37.038562
11                  Cheerios  General Mills  ...      1  50.764999
12     Cinnamon Toast Crunch  General Mills  ...      9  19.823573
13                  Clusters  General Mills  ...      7  40.400208
14               Cocoa Puffs  General Mills  ...     13  22.736446
16               Corn Flakes      Kellogg's  ...      2  45.863324
17                 Corn Pops      Kellogg's  ...     12  35.782791
18            Count Chocula  General Mills  ...     13  22.396513
19        Cracklin' Oat Bran      Kellogg's  ...      7  40.448772
21                   Crispix      Kellogg's  ...      3  46.895644
22    Crispy Wheat & Raisins  General Mills  ...     10  36.176196
24               Froot Loops      Kellogg's  ...     13  32.207582
25            Frosted Flakes      Kellogg's  ...     11  31.435973
```

Polars

polars

> Polars is a blazingly fast DataFrame library for manipulating structured data. The core is written in Rust, and available for Python, R and NodeJS.
>
> The goal of Polars is to provide a lightning fast DataFrame library that:
>
> - Utilizes all available cores on your machine.
> - Optimizes queries to reduce unneeded work/memory allocations.
> - Handles datasets much larger than your available RAM.
> - A consistent and predictable API.
> - Adheres to a strict schema (data-types should be known before running the query).

```python
import polars as pl
pl.__version__
```

```
'1.21.0'
```

Series

Just like Pandas, Polars also has a `Series` type used for columns. For a complete list of polars dtypes see here.

```
pl.Series("ints", [1, 2, 3, 4, 5])
```

shape: (5,)

ints
i64
1
2
3
4
5

```
pl.Series("bools", [True, False, True, False
```

shape: (5,)

bools
bool
true
false
true
false
true

```
pl.Series("dbls", [1., 2., 3., 4., 5.])
```

shape: (5,)

dbls
f64
1.0
2.0
3.0
4.0
5.0

```
pl.Series("strs", ["A", "B", "C", "D", "E"]
```

shape: (5,)

strs
str
"A"
"B"
"C"
"D"
"E"

Missing values

In Polars, missing data is represented by the value `null`. This missing value `null` is used for all data types, including numerical types.

```python
pl.Series("ints",
  [1, 2, 3, None])
```

shape: (4,)

ints
i64
1
2
3
null

```python
pl.Series("bools",
  [True, False, True, None]
```

shape: (4,)

bools
bool
true
false
true
null

```python
pl.Series("ints",
  [1, 2, 3, np.nan])
```

```
TypeError: unexpected value
while building Series of type
Int64; found value of type
Float64: NaN

Hint: Try setting
`strict=False` to allow
passing data with mixed types.
```

```python
pl.Series("dbls",
  [1., 2., 3., None])
```

shape: (4,)

dbls
f64
1.0
2.0
3.0
null

```python
pl.Series("strs",
  ["A", "B", "C", None])
```

shape: (4,)

strs
str
"A"
"B"
"C"
null

```python
pl.Series("dbls",
  [1., 2., 3., np.nan])
```

shape: (4,)

dbls
f64
1.0
2.0
3.0
NaN

Missing value checking

Checking for missing values can be done via the `is_null()` method

```
pl.Series("ints",
  [1, 2, 3, None]).is_null()
```

shape: (4,)

ints
bool
false
false
false
true

```
pl.Series("dbls",
  [1., 2., 3., np.nan]).is_null()
```

shape: (4,)

dbls
bool
false
false
false
false

```
pl.Series("dbls",
  [1., 2., 3., None]).is_null()
```

shape: (4,)

dbls
bool
false
false
false
true

```
pl.Series("bools",
  [True, False, True, None]).is_null()
```

shape: (4,)

bools
bool
false
false
false
true

DataFrames

Data Frames can be constructed in the same was as Pandas,

```
df = pl.DataFrame(
  {
    "name":   ["anna","bob","carol", "dave", "erin"],
    "id":     np.random.randint(100, 999, 5),
    "weight": np.random.normal(70, 20, 5),
    "height": np.random.normal(170, 15, 5),
    "date":   pd.date_range(start='2/1/2025', periods=5, freq='D')
  },
  schema_overrides = {"id": pl.UInt16, "weight": pl.Float32}
)
df
```

shape: (5, 5)

name	id	weight	height	date
str	u16	f32	f64	datetime[ns]
"anna"	202	79.477219	162.607949	2025-02-01 00:00:00
"bob"	535	97.369003	175.888696	2025-02-02 00:00:00
"carol"	960	51.663464	156.06223	2025-02-03 00:00:00
"dave"	370	67.517059	171.197477	2025-02-04 00:00:00
"erin"	206	29.780743	167.607252	2025-02-05 00:00:00

Expressions

Polars makes use of lazy evaluation to improve its flexibility and computational performance.

```
bmi_expr = pl.col("weight") / (pl.col("height") ** 2)
bmi_expr
```

[(col("weight")) / (col("height").pow([dyn int: 2]))]

This represents a potential computation that can be executed later. Much of the power of Polars comes from the ability to chain together / compose these expressions.

Contexts

Contexts are the environments in which expressions are evaluated - examples of common contexts include: `select`, `with_columns`, `filter`, and `group_by`.

```
df.select(bmi = bmi_expr)
```

shape: (5, 1)

bmi
f64
0.003006
0.003147
0.002121
0.002304
0.00106

```
df.with_columns(bmi = bmi_expr)
```

shape: (5, 6)

name	id	weight	height	date	bmi
str	u16	f32	f64	datetime[ns]	f64
"anna"	202	79.477219	162.607949	2025-02-01 00:00:00	0.003006
"bob"	535	97.369003	175.888696	2025-02-02 00:00:00	0.003147
"carol"	960	51.663464	156.06223	2025-02-03 00:00:00	0.002121
"dave"	370	67.517059	171.197477	2025-02-04 00:00:00	0.002304
"erin"	206	29.780743	167.607252	2025-02-05 00:00:00	0.00106

filter()

```
df.filter(
  pl.col("height") > 160,
  pl.col("id") < 500
)
```

shape: (3, 5)

name	id	weight	height	date
str	u16	f32	f64	datetime[ns]
"anna"	202	79.477219	162.607949	2025-02-01 00:00:00
"dave"	370	67.517059	171.197477	2025-02-04 00:00:00
"erin"	206	29.780743	167.607252	2025-02-05 00:00:00

```
df.filter(
  (pl.col("height") > 160) |
  (pl.col("id") < 500)
)
```

shape: (4, 5)

name	id	weight	height	date
str	u16	f32	f64	datetime[ns]
"anna"	202	79.477219	162.607949	2025-02-01 00:00:00
"bob"	535	97.369003	175.888696	2025-02-02 00:00:00
"dave"	370	67.517059	171.197477	2025-02-04 00:00:00
"erin"	206	29.780743	167.607252	2025-02-05 00:00:00

`group_by()` & `agg()`

```
df.group_by(
  id_range = pl.col("id") - pl.col("id") % 100
).agg(
  pl.len(),
  pl.col("name"),
  bmi_expr.alias("bmi"),
  pl.col("weight", "height").mean().name.prefix("avg_"),
  med_height = pl.col("height").median()
)
```

shape: (4, 7)

id_range	len	name	bmi	avg_weight	avg_height	med_height
u16	u32	list[str]	list[f64]	f32	f64	f64
300	1	["dave"]	[0.002304]	67.517059	171.197477	171.197477
500	1	["bob"]	[0.003147]	97.369003	175.888696	175.888696
900	1	["carol"]	[0.002121]	51.663464	156.06223	156.06223
200	2	["anna", "erin"]	[0.003006, 0.00106]	54.628983	165.107601	165.107601

More expression expansion

```python
num_cols = pl.col(pl.Float64, pl.Float32)

df.with_columns(
  ((num_cols - num_cols.mean())/num_cols.std()).name.suffix("_std")
)
```

shape: (5, 7)

name	id	weight	height	date	weight_std	height_std
str	u16	f32	f64	datetime[ns]	f32	f64
"anna"	202	79.477219	162.607949	2025-02-01 00:00:00	0.552878	-0.529878
"bob"	535	97.369003	175.888696	2025-02-02 00:00:00	1.243865	1.201382
"carol"	960	51.663464	156.06223	2025-02-03 00:00:00	-0.521299	-1.383169
"dave"	370	67.517059	171.197477	2025-02-04 00:00:00	0.090973	0.589841
"erin"	206	29.780743	167.607252	2025-02-05 00:00:00	-1.366417	0.121824

NYC Taxi Data

```python
df = pl.scan_parquet(
  "~/Scratch/nyctaxi/*_fix.parquet"
)
df
```

naive plan: (run **LazyFrame.explain(optimized=True)** to see the optimized plan)

Parquet SCAN [/Users/rundel/Scratch/nyctaxi/yellow_tripdata_2020-01_fix.parquet, ... 58 other sources]

PROJECT */19 COLUMNS

```python
df.select(pl.len()).collect()
```

shape: (1, 1)

len
u32
171021073

```python
df.select(pl.len()).explain()
```

```
'FAST COUNT (Parquet) [/Users/rundel/Scratch/nyctaxi/yellow_tripdata_2020-01_fix.parquet, ... 58 other sources] as "len"\n  DF []; PROJECT */0 COLUMNS'
```

Large lazy queries

```
zone_lookup = pl.read_csv(
  "https://d37ci6vzurychx.cloudfront.net/misc/taxi_zone_lookup.csv"
).rename(
  {"LocationID": "pickup_zone"}
)
```

```
query = (
  df
  .filter(pl.col("trip_distance") > 0)
  .rename({"PULocationID": "pickup_zone"})
  .group_by("pickup_zone")
  .agg(
    num_rides = pl.len(),
    avg_fare_per_mile = (pl.col("fare_amount") / pl.col("trip_distance")).mean().round(2)
  ).join(
    zone_lookup.lazy(),
    on = "pickup_zone",
    how = "left"
  )
  .sort("pickup_zone")
)
```

Plan

```
query
```

naive plan: (run **LazyFrame.explain(optimized=True)** to see the optimized plan)

SORT BY [col("pickup_zone")]

LEFT JOIN:

LEFT PLAN ON: [col("pickup_zone").strict_cast(Int64)]

AGGREGATE

[len().alias("num_rides"), [(col("fare_amount")) / (col("trip_distance"))].mean().round().alias("avg_fare_per_mile")] BY [col("pickup_zone")] FROM

RENAME

FILTER [(col("trip_distance")) > (0.0)] FROM

Parquet SCAN [/Users/rundel/Scratch/nyctaxi/yellow_tripdata_2020-01_fix.parquet, ... 58 other sources]

PROJECT */19 COLUMNS

RIGHT PLAN ON: [col("pickup_zone").strict_cast(Int64)]

DF ["pickup_zone", "Borough", "Zone", "service_zone"]; PROJECT */4 COLUMNS

END LEFT JOIN

Result

```
query.collect()
```

shape: (263, 6)

pickup_zone	num_rides	avg_fare_per_mile	Borough	Zone	service_zone
i32	u32	f64	str	str	str
1	6022	2205.09	"EWR"	"Newark Airport"	"EWR"
2	149	4.93	"Queens"	"Jamaica Bay"	"Boro Zone"
3	5812	11.98	"Bronx"	"Allerton/Pelham Gardens"	"Boro Zone"
4	225977	9.9	"Manhattan"	"Alphabet City"	"Yellow Zone"
5	891	20.02	"Staten Island"	"Arden Heights"	"Boro Zone"
…	…	…	…	…	…
261	818903	9.25	"Manhattan"	"World Trade Center"	"Yellow Zone"
262	2336728	7.93	"Manhattan"	"Yorkville East"	"Yellow Zone"
263	3531531	7.76	"Manhattan"	"Yorkville West"	"Yellow Zone"
264	1245641	22.66	"Unknown"	"N/A"	"N/A"
265	275794	66.35	"N/A"	"Outside of NYC"	"N/A"

Performance

```
%timeit query.collect()
```

```
1.14 s ± 62 ms per loop (mean ± std. dev. of 7 runs, 1 loop
each)
```