pandas / polars

Lecture 10

Dr. Colin Rundel

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

```
df_wide.index
```

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

```
df_wide.columns
```

```
Index(['cases', 'pop'], dtype='str', name=
```

```
( 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

```
1 cereal = pd.read_csv("https://sta663-sp26.github.io/slides/data/cereal.csv"); cereal
```

	name	mfr	type	calories	sugars	rating
0	100% Bran	Nabisco	Cold	70	6	68.402973
1	100% Natural Bran	Quaker Oats	Cold	120	8	33.983679
2	All-Bran	Kellogg's	Cold	70	5	59.425505
3	All-Bran with Extra Fiber	Kellogg's	Cold	50	0	93.704912
4	Almond Delight	Ralston Purina	Cold	110	8	34.384843
...	...	...	...	...	...	...
72	Triples	General Mills	Cold	110	3	39.106174
73	Trix	General Mills	Cold	110	12	27.753301
74	Wheat Chex	Ralston Purina	Cold	100	3	49.787445
75	Wheaties	General Mills	Cold	100	3	51.592193
76	Wheaties Honey Gold	General Mills	Cold	110	8	36.187559

77 rows × 6 columns

groupby

Groups can be created within a DataFrame via `groupby()` - this returns a `DataFrameGroupBy` object that contains information about the groups and how to access them.

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

```
<pandas.api.typing.DataFrameGroupBy object at 0x133472f90>
```

```
1 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
```

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

```
{'General Mills': [5, 7, 11, 12, 13, 14, 18, 22, 31, 36, 40, 42, 47, 51, 59, 69, 70, 7
```

groupby and aggregation methods

These groups can be used by the standard aggregation methods (e.g. `sum()`, `mean()`, `std()`, etc.)

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

```
TypeError: dtype 'str' does not support operation 'mean'
```

```
1 ( cereal
2   .groupby("type")
3   .mean(numeric_only=True)
4 )
```

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

```
1 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()`

```
1 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

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

	name	mfr	type	calories	sugars	rating
9	Bran Flakes	Post	Cold	90	5	53.313813
27	Fruit & Fibre Dates; Walnuts; and Oats	Post	Cold	120	10	40.917047
29	Fruity Pebbles	Post	Cold	110	12	28.025765
30	Golden Crisp	Post	Cold	100	15	35.252444
32	Grape Nuts Flakes	Post	Cold	100	5	52.076897
33	Grape-Nuts	Post	Cold	110	3	53.371007
34	Great Grains Pecan	Post	Cold	120	4	45.811716
37	Honey-comb	Post	Cold	110	11	28.742414
52	Post Nat. Raisin Bran	Post	Cold	120	14	37.840594

Iterating over groups

`DataFrameGroupBy` objects can also be iterated over:

```
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
                     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
```

Aggregation

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

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

```
TypeError: dtype 'str' does not support operation 'mean'
```

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

mfr	calories	sugars	rating
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),

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

```
TypeError: mean() got an unexpected keyword argu
```

```python
( 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 × 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.

```
( 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 × 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	type	calories	sugars	rating
2	All-Bran	Kellogg's	Cold	70	5	59.425505
3	All-Bran with Extra Fiber	Kellogg's	Cold	50	0	93.704912
5	Apple Cinnamon Cheerios	General Mills	Cold	110	10	29.509541
6	Apple Jacks	Kellogg's	Cold	110	14	33.174094
7	Basic 4	General Mills	Cold	130	8	37.038562
11	Cheerios	General Mills	Cold	110	1	50.764999
12	Cinnamon Toast Crunch	General Mills	Cold	120	9	19.823573
13	Clusters	General Mills	Cold	110	7	40.400208
14	Cocoa Puffs	General Mills	Cold	110	13	22.736446
16	Corn Flakes	Kellogg's	Cold	100	2	45.863324
17	Corn Pops	Kellogg's	Cold	110	12	35.782791
18	Count Chocula	General Mills	Cold	110	13	22.396513

	name	mfr	type	calories	sugars	rating
19	Cracklin' Oat Bran	Kellogg's	Cold	110	7	40.448772
21	Crispix	Kellogg's	Cold	110	3	46.895644
22	Crispy Wheat & Raisins	General Mills	Cold	100	10	36.176196
24	Froot Loops	Kellogg's	Cold	110	13	32.207582
25	Frosted Flakes	Kellogg's	Cold	110	11	31.435973
26	Frosted Mini-Wheats	Kellogg's	Cold	100	7	58.345141
28	Fruitful Bran	Kellogg's	Cold	120	12	41.015492
31	Golden Grahams	General Mills	Cold	110	9	23.804043
36	Honey Nut Cheerios	General Mills	Cold	110	10	31.072217
38	Just Right Crunchy Nuggets	Kellogg's	Cold	110	6	36.523683
39	Just Right Fruit & Nut	Kellogg's	Cold	140	9	36.471512
40	Kix	General Mills	Cold	110	3	39.241114
42	Lucky Charms	General Mills	Cold	110	12	26.734515
46	Mueslix Crispy Blend	Kellogg's	Cold	160	13	30.313351
47	Multi-Grain Cheerios	General Mills	Cold	100	6	40.105965
48	Nut&Honey Crunch	Kellogg's	Cold	120	9	29.924285
49	Nutri-Grain Almond-Raisin	Kellogg's	Cold	140	7	40.692320
50	Nutri-grain Wheat	Kellogg's	Cold	90	2	59.642837
51	Oatmeal Raisin Crisp	General Mills	Cold	130	10	30.450843
53	Product 19	Kellogg's	Cold	100	3	41.503540
58	Raisin Bran	Kellogg's	Cold	120	12	39.259197
59	Raisin Nut Bran	General Mills	Cold	100	8	39.703400
60	Raisin Squares	Kellogg's	Cold	90	6	55.333142

	name	mfr	type	calories	sugars	rating
62	Rice Krispies	Kellogg's	Cold	110	3	40.560159
66	Smacks	Kellogg's	Cold	110	15	31.230054
67	Special K	Kellogg's	Cold	110	3	53.131324
69	Total Corn Flakes	General Mills	Cold	110	3	38.839746
70	Total Raisin Bran	General Mills	Cold	140	14	28.592785
71	Total Whole Grain	General Mills	Cold	100	3	46.658844
72	Triples	General Mills	Cold	110	3	39.106174
73	Trix	General Mills	Cold	110	12	27.753301
75	Wheaties	General Mills	Cold	100	3	51.592193
76	Wheaties Honey Gold	General Mills	Cold	110	8	36.187559


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.
> - Provides a consistent and predictable API.
> - Adheres to a strict schema (data-types should be known before running the query).

```
1 import polars as pl
2 pl.__version__
```

```
'1.38.1'
```

Series

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

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

shape: (5,)

ints
i64
1
2
3
4
5

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

shape: (5,)

bools
bool
true
false
true
false
true

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

shape: (5,)

dbls
f64
1.0
2.0
3.0
4.0
5.0

```
1 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.

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

shape: (4,)

ints
i64
1
2
3
null

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

shape: (4,)

bools
bool
true
false
true
null

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

```
TypeError: unexpected value while I

Hint: Try setting `strict=False` t
```

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

shape: (4,)

dbls
f64
1.0
2.0
3.0
null

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

shape: (4,)

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

```
1 pl.Series("dbls",
2   [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

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

shape: (4,)

ints
bool
false
false
false
true

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

shape: (4,)

dbls
bool
false
false
false
false

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

shape: (4,)

dbls
bool
false
false
false
true

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

shape: (4,)

bools
bool
false
false
false
true

`null` vs `NaN`

In Polars, `null` and `NaN` are distinct concepts: `null` represents missing data while `NaN` is a valid IEEE 754 floating point value. This distinction matters for aggregation and filtering.

```
1 s_null = pl.Series("x", [1., 2., None])
```

```
1 s_null.is_null()
```

shape: (3,)

x
bool
false
false
true

```
1 s_null.is_nan()
```

shape: (3,)

x
bool
false
false
null

```
1 s_null.sum()
```

3.0

```
1 s_null.mean()
```

1.5

```
1 s_null.fill_null(0)
```

shape: (3,)

x
f64
1.0
2.0
0.0

```
1 s_nan = pl.Series("x", [1., 2., np.nan])
```

```
1 s_nan.is_null()
```

shape: (3,)

x
bool
false
false
false

```
1 s_nan.is_nan()
```

shape: (3,)

x
bool
false
false
true

```
1 s_nan.sum()
```

nan

```
1 s_nan.mean()
```

nan

```
1 s_nan.fill_nan(0)
```

shape: (3,)

x
f64
1.0
2.0
0.0

DataFrames

Data Frames can be constructed in the same way 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[μs]
"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.

```
1 bmi_expr = pl.col("weight") / ((pl.col("height")/100) ** 2)
2 bmi_expr
```

[(col("weight")) / ([(col("height")) / (dyn int: 100)].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`.

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

shape: (5, 1)

bmi
f64
30.057933
31.473487
21.212307
23.036622
10.601075

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

shape: (5, 6)

name	id	weight	height	date	bmi
str	u16	f32	f64	datetime[μs]	f64
"anna"	202	79.477219	162.607949	2025-02-01 00:00:00	30.057933
"bob"	535	97.369003	175.888696	2025-02-02 00:00:00	31.473487
"carol"	960	51.663464	156.06223	2025-02-03 00:00:00	21.212307
"dave"	370	67.517059	171.197477	2025-02-04 00:00:00	23.036622
"erin"	206	29.780743	167.607252	2025-02-05 00:00:00	10.601075

filter()

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

shape: (3, 5)

name	id	weight	height	date
str	u16	f32	f64	datetime[μs]
"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[μs]
"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"]	[23.036622]	67.517059	171.197477	171.197477
900	1	["carol"]	[21.212307]	51.663464	156.06223	156.06223
500	1	["bob"]	[31.473487]	97.369003	175.888696	175.888696
200	2	["anna", "erin"]	[30.057933, 10.601075]	54.628983	165.107601	165.107601

More expression expansion

```
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[μs]	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

`sort()` & `sort_by()`

`sort()` sorts a DataFrame by one or more columns. `sort_by()` is the expression-level equivalent for use within contexts.

```
1 df.sort("height", descending=True)
```

shape: (5, 5)

name	id	weight	height	date
str	u16	f32	f64	datetime[μs]
"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
"anna"	202	79.477219	162.607949	2025-02-01 00:00:00
"carol"	960	51.663464	156.06223	2025-02-03 00:00:00

```
1 df.sort("id", "height")
```

shape: (5, 5)

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

```
1 df.select(
2   pl.col("name", "id", "height").sort_by("height
3 )
```

shape: (5, 3)

name	id	height
str	u16	f64
"bob"	535	175.888696
"dave"	370	171.197477
"erin"	206	167.607252
"anna"	202	162.607949
"carol"	960	156.06223

Pivoting - long to wide

```
1 df_long
```

shape: (12, 4)

country	year	type	count
str	i64	str	str
"A"	1999	"cases"	"0.7K"
"A"	1999	"pop"	"19M"
"A"	2000	"cases"	"2K"
"A"	2000	"pop"	"20M"
"B"	1999	"cases"	"37K"
…	…	…	…
"B"	2000	"pop"	"174M"
"C"	1999	"cases"	"212K"
"C"	1999	"pop"	"1T"
"C"	2000	"cases"	"213K"
"C"	2000	"pop"	"1T"

```
1 df_long.pivot(
2   on="type",
3   index=["country","year"],
4   values="count"
5 )
```

shape: (6, 4)

country	year	cases	pop
str	i64	str	str
"A"	1999	"0.7K"	"19M"
"A"	2000	"2K"	"20M"
"B"	1999	"37K"	"172M"
"B"	2000	"80K"	"174M"
"C"	1999	"212K"	"1T"
"C"	2000	"213K"	"1T"

Wide to long (unpivot)

In polars, the equivalent to `melt()` is `unpivot()`.

```
1 df_wide
```

shape: (3, 3)

country	1999	2000
str	str	str
"A"	"0.7K"	"2K"
"B"	"37K"	"80K"
"C"	"212K"	"213K"

```
1 df_wide.unpivot(
2   on=["1999", "2000"],
3   index="country",
4   variable_name="year",
5   value_name="cases"
6 )
```

shape: (6, 3)

country	year	cases
str	str	str
"A"	"1999"	"0.7K"
"B"	"1999"	"37K"
"C"	"1999"	"212K"
"A"	"2000"	"2K"
"B"	"2000"	"80K"
"C"	"2000"	"213K"

NYC Taxi Example

NYC Taxi Data

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

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

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

PROJECT */19 COLUMNS

ESTIMATED ROWS: 377895472

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

shape: (1, 1)

len
u32
171021073

Query plan

Just like with DuckDB and sqlite we can ask polars to show us the query plan for a given expression.

```
1 print( df.select(pl.len()).explain() )
```

```
SELECT [len()]
  Parquet SCAN [/Users/rundel/Scratch/nyctaxi/fix/yellow_tripdata_2020-01_fix.parquet,
  PROJECT */19 COLUMNS
  ESTIMATED ROWS: 377895472
```

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

```
1 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").cast(Int64)]

AGGREGATE[maintain_order: false]

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

FROM

SELECT [col("VendorID"), col("tpep_pickup_datetime"), col("tpep_dropoff_datetime"), col("passenger_count"), col("trip_distance"), col("RatecodeID"), col("store_and_fwd_flag"), col("PULocationID").alias("pickup_zone"), col("DOLocationID"), col("payment_type"), col("fare_amount"), col("extra"), col("mta_tax"), col("tip_amount"), col("tolls_amount"), col("improvement_surcharge"), col("total_amount"), col("congestion_surcharge"), col("Airport_fee")]

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

FROM

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

PROJECT */19 COLUMNS

ESTIMATED ROWS: 377895472

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

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

END LEFT JOIN

Result

```
1 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"

pandas equivalent

```python
# pyarrow is required for reading parquet files

zone_lookup_pd = pd.read_csv(
  "https://d37ci6vzurychx.cloudfront.net/misc/taxi_zone_lookup.csv"
).rename(columns={"LocationID": "pickup_zone"})

def pd_query():
    df = pd.read_parquet("~/Scratch/nyctaxi/fix/")
    filtered = (
      df[df["trip_distance"] > 0]
      .rename(columns={"PULocationID": "pickup_zone"})
      .assign(fare_per_mile=lambda x: x["fare_amount"] / x["trip_distance"])
    )
    return (
      filtered
      .groupby("pickup_zone")
      .agg(
        num_rides=("fare_per_mile", "size"),
        avg_fare_per_mile=("fare_per_mile", "mean")
      )
      .round({"avg_fare_per_mile": 2})
      .reset_index()
      .merge(zone_lookup_pd, on="pickup_zone", how="left")
```

Performance

polars

```
1 %timeit query.collect()
```

```
1.10 s ± 72 ms per loop (mean ± std. dev. of 3 runs, 1 loop each)
```

pandas

```
1 %timeit pd_query()
```

```
150.8 s ± 5.9 s per loop (mean ± std. dev. of 3 runs, 1 loop each)
```