Intro 2 Polars

```python
import polars as pl
import pandas as pd
import numpy as np
import pyarrow as pa
import plotly.express as px
import string
import random
import os
import sys
%matplotlib inline
import matplotlib.pyplot as plt
from datetime import datetime

# Following two lines only required to view plotly when rendering from VScode.
import plotly.io as pio
# pio.renderers.default = "plotly_mimetype+notebook_connected+notebook"
pio.renderers.default = "plotly_mimetype+notebook"
```

Inspecting polars version

```python
# %pip show polars # check you polars version
# %pip show pandas # check you polars version
```

Motivation

Each of the following, alone(!), is amazing.

1. Small memory footprint
   - Native dtypes: missing, strings.
   - Arrow format in memory.

2. Lazy evaluation allows query Planning.
3. Out of the box parallelism: Fast and informative messages for debugging.
4. Strict typing: This means the dtype of output is defined by the operation and not bu the input. This is both safer, and allows static analysis.

Memory Footprint

Memory Footprint of Storage

Polars vs. Pandas:

```
letters = pl.Series(list(string.ascii_letters))

n = int(10e6)
letter1 = letters.sample(n,with_replacement=True)
letter1.estimated_size(unit='gb')
```

```
0.08381903916597366
```

```
# Pandas with Ver 1.x backend
letter1_pandas = letter1.to_pandas(use_pyarrow_extension_array=False)
letter1_pandas.memory_usage(deep=True, index=False) / 1e9
```

```
0.58
```

The memory footprint of the polars Series is 1/7 of the pandas Series(!). But I did cheat- I used string type data to emphasize the difference. The difference would have been smaller if I had used integers or floats.

```
# # Pandas with Ver 2.x pyarrow backend
letter1_pandas = letter1.to_pandas(use_pyarrow_extension_array=True)
letter1_pandas.memory_usage(deep=True, index=False) / 1e9
```

```
0.09
```

The Pyarrow backend introduced in Pandas> 2.0, narrows the gap between polars and pandas. But polars is still more efficient.

Memory Footprint of Compute

You are probably storing your data to compute with it. Let's compare the memory footprint of computations.

```
# Will run on linux only
# %load_ext memory_profiler
```

```
# %memit -r1 letter1.sort()
```

```
# %memit letter1_pandas.sort_values()
```

```
# %memit -r1 -n1 letter1[10]='a'
```

```
# %memit letter1_pandas[10]='a'
```

Things to notice:

- Operating on existing data consumes less memory in polars than in pandas.
- Changing the data consumes more memory in polars than in pandas. I suspect this has to do with the fact that the arrow memory schema used by polars is optimized. Changing the data, may thus require re-allocation and optimization.

Operating From Disk to Disk

What if my data does not fit into RAM? Turns out you manifest a lazy frame into disk, instead of RAM, thus avoiding the need to load the entire dataset into memory. Alas, the function that does so, sink_parquet(), has currently limited functionality. It is certainly worth keeping an eye on this function, as it matures.

Query Planning

Consider a sort operation that follows a filter operation. Ideally, filter precedes the sort, but we did not ensure this… We now demonstrate that polars' query planner will do it for you. En passant, we see polars is more efficient also without the query planner.

Polars' Eager evaluation, without query planning. Sort then filter.

```
%timeit -n 2 -r 2 letter1.sort().filter(letter1.is_in(['a','b','c']))
```

```
284 ms ± 5.71 ms per loop (mean ± std. dev. of 2 runs, 2 loops each)
```

Polars' Eager evaluation, without query planning. Filter then sort.

```
%timeit -n 2 -r 2 letter1.filter(letter1.is_in(['a','b','c'])).sort()
```

```
121 ms ± 7.84 ms per loop (mean ± std. dev. of 2 runs, 2 loops each)
```

Polars' Lazy evaluation with query planning. Receives sort then filter; executes filter then sort.

```
%timeit -n 2 -r 2 letter1.alias('letters').to_frame().lazy().sort(by='letters').filter(pl.
```

```
110 ms ± 2.71 ms per loop (mean ± std. dev. of 2 runs, 2 loops each)
```

Pandas' eager evaluation in the wrong order: Sort then filter.

```
%timeit -n1 -r1 letter1_pandas.sort_values().loc[lambda x: x.isin(['a','b','c'])]
```

```
1.64 s ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Pandas eager evaluation in the right order: Filter then sort.

```
%timeit -n1 -r1 letter1_pandas.loc[lambda x: x.isin(['a','b','c'])].sort_values()
```

```
148 ms ± 0 ns per loop (mean ± std. dev. of 1 run, 1 loop each)
```

Pandas alternative syntax, just as slow.

```
%timeit -n 2 -r 2 letter1_pandas.loc[letter1_pandas.isin(['a','b','c'])].sort_values()
```

```
145 ms ± 884 µs per loop (mean ± std. dev. of 2 runs, 2 loops each)
```

Things to note:

1. Query planning works!
2. Polars faster than Pandas even in eager evaluation (without query planning).

Parallelism

Polars seamlessly parallelizes over columns (also within, when possible). As the number of columns in the data grows, we would expect fixed runtime until all cores are used, and then linear scaling. The following code demonstrates this idea, using a simple sum-within-column.

```
import time

def scaling_of_sums(n_rows, n_cols):
  # n_cols = 2
  # n_rows = int(1e6)
  A = {}
  A_numpy = np.random.randn(n_rows,n_cols)
  A['numpy'] = A_numpy.copy()
  A['polars'] = pl.DataFrame(A_numpy)
  A['pandas'] = pd.DataFrame(A_numpy)

  times = {}
  for key,value in A.items():
    start = time.time()
    value.sum() # sum over columns
    end = time.time()
    times[key] = end-start # get runtime

  return(times)
```

```
scaling_of_time = {
  p:scaling_of_sums(n_rows= int(1e6),n_cols = p) for p in np.arange(1,16)}
```

```
data = pd.DataFrame(scaling_of_time).T
fig = px.line(
  data,
  labels=dict(
    index="Number of Columns",
    value="Runtime")
)
fig.show()
```

```
Unable to display output for mime type(s): text/html
```

```
Unable to display output for mime type(s): application/vnd.plotly.v1+json, text/html
```

Things to note:

- Pandas is slow.
- Numpy is quite efficient.
- My machine has 8 cores. I would thus expect a fixed timing until 8 columns, and then linear scaling. This is not the case. I suspect that is because parallelism occurs not only between columns, but also within.

```
scaling_of_time_2 = {
  p:scaling_of_sums(n_rows=p ,n_cols = int(1e5)) for p in np.arange(1,16)}
```

```
data = pd.DataFrame(scaling_of_time_2).T
fig = px.line(
  data,
  labels=dict(
    index="Number of Rows",
    value="Runtime")
)
fig.show()
```

```
Unable to display output for mime type(s): application/vnd.plotly.v1+json, text/html
```

Things to note:

- Summing over columns does not parallelize well in polars. This has to do with the fact that arrow stores data in a columnar format.

Speed Of Import

Polar's `read_X` functions are quite faster than Pandas. This is due to better type "guessing" heuristics, and easier mapping between the disk representation and memory representation of the data.

We benchmark by making synthetic data, save it on disk, and reimporting it.

Starting with CSV:

```
n_rows = int(1e5)
n_cols = 10
data_polars = pl.DataFrame(np.random.randn(n_rows,n_cols))
data_polars.write_csv('data/data.csv', has_header = False)
f"{os.path.getsize('data/data.csv')/1e7:.2f} MB on disk"
```

```
'1.96 MB on disk'
```

Import with pandas.

```
%timeit -n2 -r2 data_pandas = pd.read_csv('data/data.csv', header = None)
```

```
79.6 ms ± 678 µs per loop (mean ± std. dev. of 2 runs, 2 loops each)
```

Import with polars.

```
%timeit -n2 -r2 data_polars = pl.read_csv('data/data.csv', has_header = False)
```

```
5.34 ms ± 780 µs per loop (mean ± std. dev. of 2 runs, 2 loops each)
```

Trying parquet format:

```
data_polars.write_parquet('data/data.parquet')
f"{os.path.getsize('data/data.parquet')/1e7:.2f} MB on disk"
```

```
'0.78 MB on disk'
```

```
%timeit -n2 -r2 data_pandas = pd.read_parquet('data/data.parquet')
```

```
8.17 ms ± 3.14 ms per loop (mean ± std. dev. of 2 runs, 2 loops each)
```

```
%timeit -n2 -r2 data_polars = pl.read_parquet('data/data.parquet')
```

```
2.51 ms ± 221 µs per loop (mean ± std. dev. of 2 runs, 2 loops each)
```

Trying Feather format:

```
data_polars.write_ipc('data/data.feather')
f"{os.path.getsize('data/data.feather')/1e7:.2f} MB on disk"
```

```
'0.80 MB on disk'
```

```
%timeit -n2 -r2 data_polars = pl.read_ipc('data/data.feather')
```

```
The slowest run took 5.08 times longer than the fastest. This could mean that an intermediate
173 µs ± 116 µs per loop (mean ± std. dev. of 2 runs, 2 loops each)
```

```
%timeit -n2 -r2 data_pandas = pd.read_feather('data/data.feather')
```

```
2.98 ms ± 812 µs per loop (mean ± std. dev. of 2 runs, 2 loops each)
```

Trying Lance format: TODO: update once supported.

Trying Pickle format:

```
import pickle
pickle.dump(data_polars, open('data/data.pickle', 'wb'))
f"{os.path.getsize('data/data.pickle')/1e7:.2f} MB on disk"
```

```
'0.90 MB on disk'
```

```
%timeit -n2 -r2 data_polars = pickle.load(open('data/data.pickle', 'rb'))
```

```
26 ms ± 107 µs per loop (mean ± std. dev. of 2 runs, 2 loops each)
```

Things to note:

- The difference in speed is quite large between pandas vs. polars.
- When dealing with CSV's, the function `pl.read_csv` reads in parallel, and has better type guessing heuristics.
- The difference in speed is quite large between csv vs. parquet and feather, with feather<parquet<csv.
- Feather is the fastest, but larger on disk. Thus good for short-term storage, and parquet for long-term.
- The fact that pickle isn't the fastest surprised me; but then again, it is not optimized for data.

Speed Of Join

Because pandas is built on numpy, people see it as both an in-memory database, and a matrix/array library. With polars, it is quite clear it is an in-memory database, and not an array processing library (despite having a `pl.dot()` function for inner products). As such, you cannot multiply two polars dataframes, but you can certainly join then efficiently.

Make some data:

```
def make_data(n_rows, n_cols):
  data = np.concatenate(
  (
    np.arange(n_rows)[:,np.newaxis], # index
    np.random.randn(n_rows,n_cols), # values
    ),
    axis=1)

  return data


n_rows = int(1e6)
n_cols = 10
data_left = make_data(n_rows, n_cols)
data_right = make_data(n_rows, n_cols)
```

Polars join:

```
data_left_polars = pl.DataFrame(data_left)
data_right_polars = pl.DataFrame(data_right)

%timeit -n2 -r2 polars_joined = data_left_polars.join(data_right_polars, on = 'column_0',
```

```
66.6 ms ± 5.24 ms per loop (mean ± std. dev. of 2 runs, 2 loops each)
```

Pandas join:

```
data_left_pandas = pd.DataFrame(data_left)
data_right_pandas = pd.DataFrame(data_right)

%timeit -n2 -r2 pandas_joined = data_left_pandas.merge(data_right_pandas, on = 0, how = 'i
```

```
174 ms ± 9.32 ms per loop (mean ± std. dev. of 2 runs, 2 loops each)
```

The NYC Taxi Dataset

```
path = 'data/NYC' # Data from https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page
file_names = os.listdir(path)
```

Pandas query syntax:

```
%%time
taxi_pandas = pd.read_parquet(path)

query = '''
    passenger_count > 0 and
    passenger_count < 5 and
    trip_distance >= 0 and
    trip_distance <= 10 and
    fare_amount >= 0 and
    fare_amount <= 100 and
    tip_amount >= 0 and
    tip_amount <= 20 and
    total_amount >= 0 and
    total_amount <= 100
    '''.replace('\n', '')
taxi_pandas.query(query).groupby('passenger_count').agg({'tip_amount':'mean'})
```

```
CPU times: user 1.12 s, sys: 1.07 s, total: 2.2 s
Wall time: 1.06 s
```

passenger_count	tip_amount
1.0	2.096363
2.0	2.120294
3.0	2.074437
4.0	2.054331

Well, the `loc` syntax is usually faster than the `query` syntax:

```
%%time
taxi_pandas = pd.read_parquet(path)

ind = (
```

```python
    taxi_pandas['passenger_count'].between(1,4)
    & taxi_pandas['trip_distance'].between(0,10)
    & taxi_pandas['fare_amount'].between(0,100)
    & taxi_pandas['tip_amount'].between(0,20)
    & taxi_pandas['total_amount'].between(0,100)
)
(
    taxi_pandas[ind]
    .groupby('passenger_count')
    .agg({'tip_amount':'mean'})
)
```

```
CPU times: user 1.13 s, sys: 948 ms, total: 2.08 s
Wall time: 838 ms
```

	tip_amount
passenger_count	
1.0	2.096363
2.0	2.120294
3.0	2.074437
4.0	2.054331

Polars

```python
%%time
q = (
    pl.scan_parquet(f'{path}/*.parquet')
    .filter(
        (pl.col('passenger_count') > 0) &
        (pl.col('passenger_count') < 5) &
        (pl.col('trip_distance') >= 0) &
        (pl.col('trip_distance') <= 10) &
        (pl.col('fare_amount') >= 0) &
        (pl.col('fare_amount') <= 100) &
        (pl.col('tip_amount') >= 0) &
        (pl.col('tip_amount') <= 20) &
        (pl.col('total_amount') >= 0) &
        (pl.col('total_amount') <= 100)
    )
```

```python
    .groupby('passenger_count')
    .agg([pl.mean('tip_amount')])
    )
q.collect()
```

```
CPU times: user 486 ms, sys: 108 ms, total: 594 ms
Wall time: 96.4 ms
```

```
PARTITIONED DS
```

passenger_count f64	tip_amount f64
1.0	2.096363
2.0	2.120294
4.0	2.054331
3.0	2.074437

```python
q.show_graph()
```

Things to note:

- Pandas `loc` syntax is faster than `query`; both considerably slower than polars.
- I only have 2 parquet files. When I run the same with more files, despite my 16GB of RAM, **pandas will crash my python kernel**.
- From the query graph I see import is done in parallel, and filtering done at scanning time!
- Warning: The `pl.scan_paquet()` function will not work with a glob if files are in a remote data lake (e.g. S3). More on that later…

Moving Forward…

If this motivational section has convinced you to try polars instead of pandas, here is a more structured intro.

Getting Help

Before we dive in, you should be aware of the following references for further help:

1. A github page.
2. A user guide.
3. A very active community on Discord.
4. The API reference.
5. A Stack-Overflow tag.
6. Cheat-sheet for pandas users.

Warning: Be careful of AI assistants such as Github-Copilot, TabNine, etc. Polars is still very new, and they may give you pandas completions instead of polars.

Polars Series

Much like pandas, polars' fundamental building block is the series. A series is a column of data, with a name, and a dtype.

Series-Object Housekeeping

Construct a series

```
s = pl.Series("a", [1, 2, 3])
s
```

a
i64
1
2
3

Make pandas series for comparison:

```
s_pandas = pd.Series([1, 2, 3], name = "a")
```

```
type(s)
```

```
polars.series.series.Series
```

```
type(s_pandas)
```

```
pandas.core.series.Series
```

```
s.dtype
```

```
Int64
```

```
s_pandas.dtype
```

```
dtype('int64')
```

Renaming a series; will be very useful when operating on dataframe columns.

```
s.alias("b")
```

b
i64
1
2
3

```
s.clone()
```

a
i64
1
2
3

```
s.clone().append(pl.Series("a", [4, 5, 6]))
```

a
i64
1
2
3
4
5
6

Note: `series.append` operates in-place. That is why we cloned the series first.

Flatten a list of lists using `explode()`; this will not work for more than 2 levels of nesting.

```
pl.Series("a", [[1, 2], [3, 4], [9, 10]]).explode()
```

a
i64
1
2
3
4
9
10

```
s.extend_constant(666, n=2)
```

a
i64
1
2
3
666
666

```
s.rechunk()
```

a
i64
1
2
3

```
s.rename("b", in_place=False) # has an in_place option. Unlike .alias()
```

b
i64
1
2
3

```
s.to_dummies()
```

a_1	a_2	a_3
u8	u8	u8
1	0	0
0	1	0
0	0	1

```
s.clear() # creates an empty series, with same dtype. Previously called s.cleared()
```

a
i64

Constructing a series of floats, for later use.

```
f = pl.Series("a", [1., 2., 3.])
f
```

a
f64
1.0
2.0
3.0

```
f.dtype
```

```
Float64
```

Memory Representation of Series

Object size in memory. Super useful for profiling:

```
s.estimated_size(unit="gb")
```

```
2.2351741790771484e-08
```

```
s.chunk_lengths() # what is the length of each memory chunk?
```

```
[3]
```

Filtering and Subsetting

```
s[0] # same as s.__getitem__(0)
```

```
1
```

Filtering with `[` and Booleans will not work:

```
s[[True, False, True]]
```

```
NotImplementedError: Unsupported idxs datatype.
```

Filtering with a Polars Boolean series, worked in previous versions of polars (<=15). Currently (16) does not.

```python
s[pl.Series("a", [True, False, True])]
```

```
ValueError: Cannot __getitem__ on Series of dtype: 'Int64' with argument: 'shape: (3,)
Series: 'a' [bool]
[
    true
    false
    true
]' of type: '<class 'polars.series.series.Series'>'.
```

Filtering with a pandas (Boolean) series will not work (why should it?), nor with a numpy array.

For an easy transition to work with lazy dataframes and query planning (Section **??**), you may want to prefer the `filter` method, which can actually take a polars series, or list of booleans (but not a pandas series or numpy array):

```python
s.filter(pl.Series("a", [True, False, True])) # works
```

a
i64
1
3

```python
s.filter([True, False, True])
```

a
i64
1
3

```python
s.head(2)
```

a
i64
1

a
i64
2

```python
s.limit(2)
```

a
i64
1
2

```python
s.tail(2)
```

a
i64
2
3

```python
s.sample(2, with_replacement=False)
```

a
i64
1
3

```python
s.take([0, 2]) # same as s[0,2] and pandas .iloc[[0,2]]
```

a
i64
1
3

```python
s.slice(1, 2) # same as pandas .iloc[1:2]
```

a
i64
2
3

```
s.take_every(2)
```

a
i64
1
3

Aggregations

```
s.sum()
```

6

```
s.min()
```

1

```
s.arg_min()
```

0

```
s.max()
```

3

```
s.arg_max()
```

2

```python
s.mean()
```

```
2.0
```

```python
s.median()
```

```
2.0
```

```python
s.entropy()
```

```
-4.68213122712422
```

```python
s.describe()
```

statistic str	value f64
"count"	3.0
"null_count"	0.0
"mean"	2.0
"std"	1.0
"min"	1.0
"max"	3.0
"median"	2.0
"25%"	1.0
"75%"	3.0

```python
s.value_counts()
```

a i64	counts u32
1	1
2	1
3	1

Object Transformations

```python
pl.Series("a",[1,2,3,4]).reshape(dims = (2,2))
```

```
TypeError: reshape() got an unexpected keyword argument 'dims'
```

```python
s.shift(1)
```

a
i64
null
1
2

```python
s.shift(-1)
```

a
i64
2
3
null

```python
s.shift_and_fill(1, 999)
```

```
TypeError: shift_and_fill() takes 2 positional arguments but 3 were given
```

Mathematical Transformations

```python
s.abs()
```

a
i64
1
2
3

```
s.sin()
```

a
f64
0.841471
0.909297
0.14112

```
s.exp()
```

a
f64
2.718282
7.389056
20.085537

```
s.hash()
```

a
u64
6364136223846793005
12728272447693586010
645664597830827398

```
s.log()
```

a
f64
0.0
0.693147
1.098612

```
s.peak_max()
```

bool
false
false
true

```
s.sqrt()
```

a f64
1.0
1.414214
1.732051

Comparisons

```
s.clip_max(2)
```

a i64
1
2
2

```
s.clip_min(1)
```

a i64
1
2
3

```
s.clip(1,2) # AKA Winsorizing
```

a
i64
1
2
2

You cannot round integers, but you can round floats.

```
f.round(2)
```

a
f64
1.0
2.0
3.0

```
f.ceil()
```

a
f64
1.0
2.0
3.0

```
f.floor()
```

a
f64
1.0
2.0
3.0

Search

```
s.is_in(pl.Series([1, 10]))
```

a
bool
true
false
false

```
s.is_in([1, 10])
```

a
bool
true
false
false

Things to note:

- `is_in()` in polars has an underscore, unlike `isin()` in pandas.

Apply

Applying your own function:

```
s.apply(lambda x: x + 1)
```

a
i64
2
3
4

Using your own functions comes with a performance cost:

```
s1 = pl.Series(np.random.randn(int(1e5)))
```

Adding 1 with apply:

```
%timeit -n2 -r2 s1.apply(lambda x: x + 1)
```

```
12.1 ms ± 617 µs per loop (mean ± std. dev. of 2 runs, 2 loops each)
```

Adding 1 without apply:

```
%timeit -n2 -r2 s1+1
```

```
73.3 µs ± 10 µs per loop (mean ± std. dev. of 2 runs, 2 loops each)
```

Cummulative Operations

```
s.cummax()
```

a
i64
1
2
3

```
s.cumsum()
```

a
i64
1
3
6

```
s.cumprod()
```

a
i64
1
2
6

```
s.ewm_mean(com=0.5)
```

a
f64
1.0
1.75
2.615385

Sequential Operations

```
s.diff()
```

a
i64
null
1
1

```
s.pct_change()
```

a
f64
null
1.0
0.5

Windowed Operations

```
s.rolling_apply(
  pl.sum,
  window_size=2)
```

a
i64
null
3
5

Not all functions will work within a `rolling_apply`! Only polars' functions will.

```
s.rolling_apply(np.sum, window_size=2) # will not work
```

Some rolling functions have been prepared for you.

```
s.rolling_max(window_size=2)
```

a
i64
null
2
3

Logical Aggregations

```
b = pl.Series("a", [True, True, False])
b.dtype
```

```
Boolean
```

```
b.all()
```

```
False
```

```
b.any()
```

True

Uniques and Duplicates

```
s.is_duplicated()
```

a
bool
false
false
false

```
s.is_unique()
```

a
bool
true
true
true

```
s.n_unique()
```

3

```
pl.Series([1,2,3,4,1]).unique_counts()
```

u32
2
1
1

u32
1

The first appearance of a value in a series:

```
pl.Series([1,2,3,1]).is_first()
```

bool
true
true
true
false

dtypes

Note. Unlike pandas, polars' test functions have an underscore: `is_numeric()` instead of `isnumeric()`.

Testing

```
s.is_numeric()
```

```
True
```

```
s.is_float()
```

```
False
```

```
s.is_utf8()
```

```
False
```

```
s.is_boolean()
```

```
False
```

```
s.is_temporal() # previously called .is_datelike()
```

```
False
```

Casting

```
s.cast(pl.Int32)
```

a
i32
1
2
3

Things to note:

- The dtypes to cast to are **polars** dtypes. Don't try `s.cast("int32")`, `s.cast(np.int32)`, or `s.cast(pd.int)`
- `cast()` is polars' equivalent of pandas' `astype()`.
- For a list of dtypes see the official documentation.

Optimizing dtypes

Find the most efficient dtype for a series:

```
s.shrink_dtype() # like pandas pd.to_numeric(..., downcast="...") and pandas_dtype_efficie
```

a
i8
1
2

a
i8
3

Also see here.

Shrink the memory allocation to the size of the actual data (in place).

```
s.shrink_to_fit()
```

a
i64
1
2
3

Ordering and Sorting

```
s.sort()
```

a
i64
1
2
3

```
s.reverse()
```

a
i64
3
2
1

```
s.rank()
```

a
f32
1.0
2.0
3.0

```
s.arg_sort()
```

a
u32
0
1
2

`arg_sort()` returns the indices that would sort the series. Same as R's `order()`.

```
(s.sort() == s[s.arg_sort()]).all()
```

```
True
```

`arg_sort()` can also be used to return the original series from the sorted one:

```
(s == s[s[s.arg_sort()].arg_sort()]).all()
```

```
True
```

```
s.shuffle(seed=1) # random permutation
```

a
i64
2
1
3

Missing

Pandas users will be excited to know that thanks to arrow, polars has built in missing value support for all(!) dtypes. This has been a long awaited feature in the Python data science ecosystem with implications on speed, memory, style and more. The Polars Userguide has a great overview of the topic from which we collect some take-homes:

- `np.nan` is also supported along `pl.Null`, but is not considered as a missing value by polars. This has implications on null counts, statistical aggregations, etc.
- `pl.Null`, and `np.nan`s have their own separate functions for imputing, counting, etc.

PS - Arrow support is also expected in Pandas 2.0.

```
m = pl.Series("a", [1, 2, None, np.nan])
```

```
m.is_null() # checking for None's. Like pandas .isna()
```

a bool
false
false
true
false

```
m.is_nan() # checking for np.nan's
```

a bool
false
false
null
true

For comparison with pandas:

```
m_pandas = pd.Series([1, 2, None, np.nan])
```

```
m_pandas.isna()
```

```
0    False
1    False
2     True
3     True
dtype: bool
```

```
m_pandas.isnull() # alias for pd.isna()
```

```
0    False
1    False
2     True
3     True
dtype: bool
```

```
# Polars
m1 = pl.Series("a", [1, None, 2, ]) # python native None
m2 = pl.Series("a", [1, np.nan, 2, ]) # numpy's nan
m3 = pl.Series("a", [1, float('nan'), 2, ]) # python's nan

# Pandas
m4 = pd.Series([1, None, 2 ])
m5 = pd.Series([1, np.nan, 2, ])
m6 = pd.Series([1, float('nan'), 2, ])
```

```
[m1.sum(), m2.sum(), m3.sum(), m4.sum(), m5.sum(), m6.sum()]
```

```
[3, nan, nan, 3.0, 3.0, 3.0]
```

Things to note:

- Aggregating pandas and polars series behave differently w.r.t. missing values:
  - Both will ignore `None`; which is unsafe.
  - Polars will not ignore `np.nan`; which is safe. Pandas is unsafe w.r.t. `np.nan`, and will ignore it.

Filling missing values; `None` and `np.nan` are treated differently:

```
m1.fill_null(0)
```

a
i64
1
0
2

```python
m2.fill_null(0)
```

a
f64
1.0
NaN
2.0

```python
m2.fill_nan(0)
```

a
f64
1.0
0.0
2.0

```python
m1.drop_nulls()
```

a
i64
1
2

```python
m1.drop_nans()
```

a
i64
1
null
2

```
m2.drop_nulls()
```

a
f64
1.0
NaN
2.0

```
m1.interpolate()
```

a
i64
1
1
2

```
m2.interpolate() # np.nan is not considered missing, so why interpolate?
```

a
f64
1.0
NaN
2.0

Export To Other Python Objects

The current section deals with exports to other python objects in memory. See Section **??** for exporting to disk.

```python
s.to_frame()
```

a
i64
1
2
3

```python
s.to_list()
```

```
[1, 2, 3]
```

```python
s.to_numpy() # useful for preparing data for learning with scikit-learn
```

```
array([1, 2, 3])
```

```python
s.to_pandas()
```

```
0    1
1    2
2    3
Name: a, dtype: int64
```

```python
s.to_arrow() # useful for preparing data for learning with XGBoost. Maybe sklearn in the f
```

```
<pyarrow.lib.Int64Array object at 0x290eb0ee0>
[
  1,
  2,
  3
]
```

Strings

Like Pandas, accessed with the `.str` attribute.

```python
st = pl.Series("a", ["foo", "bar", "baz"])
```

```python
st.str.n_chars() # gets number of chars. In ASCII this is the same as lengths()
```

a
u32
3
3
3

```python
st.str.lengths() # gets number of bytes in memory
```

a
u32
3
3
3

```python
st.str.concat("-")
```

a
str
"foo-bar-baz"

```python
st.str.contains("foo|tra|bar")
```

a
bool
true
true
false

```python
st.str.count_match(pattern= 'o') # count literal metches
```

a
u32
2
0
0

Regex is supported. The `r` prefix in `r"<regex pattern>"` is useful for emphasizing regular expressions, but not really necessary (more about it here).

```
st.str.count_match(pattern=r"\w") # \w is regex for alphanumeric
```

a
u32
3
3
3

```
st.str.ends_with("oo")
```

a
bool
true
false
false

```
st.str.starts_with("fo")
```

a
bool
true
false
false

To extract the **first** appearance of a pattern, use `extract`:

```python
url = pl.Series("a", [
            "http://vote.com/ballon_dor?candidate=messi&ref=polars",

            "http://vote.com/ballon_dor?candidate=jorginho&ref=polars",

            "http://vote.com/ballon_dor?candidate=ronaldo&ref=polars"
            ])
```

```python
url.str.extract(r"=(\w+)", 1) # "=(\w+)" is read: match an equality, followed by any numbe
```

a str
"messi"
"jorginho"
"ronaldo"

To extract **all** appearances of a pattern, use `extract_all`:

```python
url.str.extract_all("=(\w+)")
```

a list[str]
["=messi", "=polars"]
["=jorginho", "=polars"]
["=ronaldo", "=polars"]

```python
st.str.ljust(8, "*")
```

a str
"foo*****"
"bar*****"
"baz*****"

```python
st.str.rjust(8, "*")
```

a str
"*****foo"
"*****bar"
"*****baz"

```
st.str.lstrip('f')
```

a str
"oo"
"bar"
"baz"

```
st.str.rstrip('r')
```

a str
"foo"
"ba"
"baz"

Replacing first appearance of a pattern:

```
st.str.replace(r"o+", "ZZ")
```

a str
"fZZ"
"bar"
"baz"

Replace all appearances of a pattern:

```
st.str.replace_all("o", "ZZ")
```

a str
"fZZZZ"
"bar"
"baz"

String to list of strings. Number of splits inferred.

```
st.str.split(by="o")
```

a list[str]
["f", "", ""]
["bar"]
["baz"]

```
st.str.split(by="a", inclusive=True)
```

a list[str]
["foo"]
["ba", "r"]
["ba", "z"]

String to dict of strings. Number of **splits** fixed.

```
st.str.split_exact("a", 2)
```

a struct[3]
{"foo",null,null}
{"b","r",null}
{"b","z",null}

String to dict of strings. **Length of output** fixed.

```
st.str.splitn("a", 4)
```

a struct[4]
{"foo",null,null,null}
{"b","r",null,null}
{"b","z",null,null}

Strip white spaces.

```
pl.Series(['   ohh   ','   yeah   ']).str.strip()
```

 str
"ohh"
"yeah"

```
st.str.to_uppercase()
```

a str
"FOO"
"BAR"
"BAZ"

```
st.str.to_lowercase()
```

a str
"foo"
"bar"
"baz"

```
st.str.zfill(5)
```

a str
"00foo"
"00bar"
"00baz"

```
st.str.slice(offset=0, length=2)
```

a str
"fo"
"ba"
"ba"

Date and Time

There are 4 datetime dtypes in polars:

1. Date: A date, without hours. Generated with `pl.Date()`.
2. Datetime: Date and hours. Generated with `pl.Datetime()`.
3. Time: Hour of day. Generated with `pl.Time()`.
4. Duration: As the name suggests. Similar t o `timedelta` in pandas. Generated with `pl.Duration()`.

Warning: Python has a sea of modules that support datetimes. A partial list includes: datetime module, extensions in dateutil, numpy, pandas, arrow, the deprecated scikits.timeseries and certainly others. Be aware of the dtype you are using, and the accompanying methods.

Time Range

```
from datetime import datetime, timedelta

start = datetime(year= 2001, month=2, day=2)
stop = datetime(year=2001, month=2, day=3)

date = pl.date_range(
  low=start,
  high=stop,
```

```
    interval=timedelta(seconds=500*61))
date
```

```
/var/folders/91/c3y_9h950pb0gq8c8sdytk1r0000gn/T/ipykernel_85263/2166444983.py:6: Deprecatio

`low` is deprecated as an argument to `date_range`; use `start` instead.

/var/folders/91/c3y_9h950pb0gq8c8sdytk1r0000gn/T/ipykernel_85263/2166444983.py:6: Deprecatio

`high` is deprecated as an argument to `date_range`; use `end` instead.
```

datetime[s]
2001-02-02 00:00:00
2001-02-02 08:28:20
2001-02-02 16:56:40

Things to note:

- How else could I have constructed this series? What other types are accepted as `low` and `high`?
- `pl.date_range` may return a series of dtype `Date` or `Datetime`. This depens of the granularity of the inputs.

```
date.dtype
```

```
Datetime(time_unit='us', time_zone=None)
```

Cast to different time unit. May be useful when joining datasets, and the time unit is different.

```
date.dt.cast_time_unit(tu="ms")
```

```
TypeError: cast_time_unit() got an unexpected keyword argument 'tu'
```

Extract Time Sub-Units

```
date.dt.second()
```

u32
0
20
40

```
date.dt.minute()
```

u32
0
28
56

```
date.dt.hour()
```

u32
0
8
16

```
date.dt.day()
```

u32
2
2
2

```
date.dt.week()
```

u32
5
5
5

```
date.dt.weekday()
```

u32
5
5
5

```
date.dt.month()
```

u32
2
2
2

```
date.dt.year()
```

i32
2001
2001
2001

```
date.dt.ordinal_day() # day in year
```

u32
33
33
33

```
date.dt.quarter()
```

u32
1
1
1

Durations

Equivalent to Pandas `period` dtype.

```
diffs = date.diff()
diffs
```

duration[s]
null
8h 28m 20s
8h 28m 20s

```
diffs.dtype
```

```
Duration(time_unit='us')
```

```
diffs.dt.seconds()
```

i64
null
30500
30500

```
diffs.dt.minutes()
```

i64
null
508
508

```
diffs.dt.days()
```

i64
null
0
0

```
diffs.dt.hours()
```

i64
null
8
8

Date Aggregations

Note that aggregating dates, returns a `datetime` type object.

```
date.dt.max()
```

```
datetime.datetime(2001, 2, 2, 16, 56, 40)
```

```
date.dt.min()
```

```
datetime.datetime(2001, 2, 2, 0, 0)
```

```
date.dt.mean()
```

```
datetime.datetime(2001, 2, 2, 8, 28, 20)
```

```
date.dt.median()
```

```
datetime.datetime(2001, 2, 2, 8, 28, 20)
```

Date Transformations

Notice the syntax of `offset_by`. It is similar to R's `lubridate` package.

```
date.dt.offset_by(by="1y2m20d")
```

datetime[s]
2002-02-22 00:02:00
2002-02-22 08:30:20
2002-02-22 16:58:40

Negative offset is also allowed.

```
date.dt.offset_by(by="-1y2m20d")
```

datetime[s]
2000-01-12 23:58:00
2000-01-13 08:26:20

datetime[s]
2000-01-13 16:54:40

```
date.dt.round("1y")
```

datetime[s]
2001-01-01 00:00:00
2001-01-01 00:00:00
2001-01-01 00:00:00

```
date2 = date.dt.truncate("30m") # round to period
pd.crosstab(date,date2)
```

col_0 row_0	2001-02-02 00:00:00	2001-02-02 08:00:00	2001-02-02 16:30:00
2001-02-02 00:00:00	1	0	0
2001-02-02 08:28:20	0	1	0
2001-02-02 16:56:40	0	0	1

From Date to String

```
date.dt.strftime("%Y-%m-%d")
```

str
”2001-02-02”
”2001-02-02”
”2001-02-02”

From String to Datetime

```
sd = pl.Series(
    "date",
    [
        "2021-04-22",
        "2022-01-04 00:00:00",
        "01/31/22",
        "Sun Jul  8 00:34:60 2001",
    ],
)
```

Parse into `Date` type.

```
sd.str.strptime(datatype= pl.Date, fmt="%F", strict=False)
```

```
/var/folders/91/c3y_9h950pb0gq8c8sdytk1r0000gn/T/ipykernel_85263/3102708485.py:1: Deprecatio

`datatype` is deprecated as an argument to `strptime`; use `dtype` instead.

/var/folders/91/c3y_9h950pb0gq8c8sdytk1r0000gn/T/ipykernel_85263/3102708485.py:1: Deprecatio

`fmt` is deprecated as an argument to `strptime`; use `format` instead.
```

date
date
2021-04-22
null
null
null

```
sd.str.strptime(pl.Date, "%D", strict=False)
```

date
date
null
null
2022-01-31

date
date
null

Parse into `Datetime` type.

```
sd.str.strptime(pl.Datetime, "%F %T",strict=False)
```

date
datetime[s]
null
2022-01-04 00:00:00
null
null

```
sd.str.strptime(pl.Datetime, "%a %h %d %T %Y",strict=False)
```

date
datetime[s]
null
null
null
2001-07-08 00:35:00

Parse into `Time` type.

```
sd.str.strptime(pl.Time, "%a %h %d %T %Y",strict=False)
```

date
time
null
null
null
00:35:00

Comparing Series

```
s.series_equal(pl.Series("a", [1, 2, 3]))
```

```
True
```

DataFrames

General:

1. There is no row index (like R's `data.frame`, `data.table`, and `tibble`; unlike Python's `pandas`).
2. Will not accept duplicate column names (unlike pandas).

DataFrame-Object Hosekeeping

A frame can be created as you would expect. From a dictionary of series, a numpy array, a pandas sdataframe, or a list of polars (or pandas) series, etc.

```
df = pl.DataFrame({
  "integer": [1, 2, 3],
  "date": [
    (datetime(2022, 1, 1)),
    (datetime(2022, 1, 2)),
    (datetime(2022, 1, 3))],
    "float":[4.0, 5.0, 6.0],
    "string": ["a", "b", "c"]})

df
```

	integer i64	date datetime[s]	float f64	string str
1		2022-01-01 00:00:00	4.0	"a"
2		2022-01-02 00:00:00	5.0	"b"
3		2022-01-03 00:00:00	6.0	"c"

```
print(df)
```

```
shape: (3, 4)

 integer   date                  float   string
 ---       ---                   ---     ---
 i64       datetime[s]           f64     str

 1         2022-01-01 00:00:00   4.0     a
 2         2022-01-02 00:00:00   5.0     b
 3         2022-01-03 00:00:00   6.0     c
```

Things to note:

1. The frame may be printed with Jupter's styling, or as ASCII with a `print()` statement.
2. Shape, and dtypes, are part of the output.

```
df.columns
```

```
['integer', 'date', 'float', 'string']
```

```
df.shape
```

```
(3, 4)
```

```
df.height # probably more useful than df.shape[0]
```

```
3
```

```
df.width
```

```
4
```

```
df.schema # similar to pandas info()
```

```
{'integer': Int64,
 'date': Datetime(time_unit='us', time_zone=None),
 'float': Float64,
 'string': Utf8}
```

```
df.with_row_count()
```

	row_nr u32	integer i64	date datetime[s]	float f64	string str
0	1	2022-01-01 00:00:00	4.0	"a"	
1	2	2022-01-02 00:00:00	5.0	"b"	
2	3	2022-01-03 00:00:00	6.0	"c"	

Add a single column

```
df.with_columns(
    pl.Series("new", [1, 2, 3])
    ) # replaces the now-deprecated function `df.with_column()`
```

integer i64	date datetime[s]	float f64	string str	new i64
1	2022-01-01 00:00:00	4.0	"a"	1
2	2022-01-02 00:00:00	5.0	"b"	2
3	2022-01-03 00:00:00	6.0	"c"	3

Add multiple columns

```
df.with_columns(
  pl.Series("new1", [1, 2, 3]),
  pl.Series("new2", [4, 5, 6])
  )
```

integer i64	date datetime[s]	float f64	string str	new1 i64	new2 i64
1	2022-01-01 00:00:00	4.0	"a"	1	4
2	2022-01-02 00:00:00	5.0	"b"	2	5
3	2022-01-03 00:00:00	6.0	"c"	3	6

```
df.clone() # deep copy
```

	integer i64	date datetime[s]	float f64	string str
1		2022-01-01 00:00:00	4.0	"a"
2		2022-01-02 00:00:00	5.0	"b"
3		2022-01-03 00:00:00	6.0	"c"

The following commands make changes in place; I am thus creating a copy of `df`.

```
df_copy = df.clone() # making a copy since
df_copy.insert_at_idx(1, pl.Series("new", [1, 2, 3]))
```

	integer i64	new i64	date datetime[s]	float f64	string str
1		1	2022-01-01 00:00:00	4.0	"a"
2		2	2022-01-02 00:00:00	5.0	"b"
3		3	2022-01-03 00:00:00	6.0	"c"

```
df_copy.replace_at_idx(0, pl.Series("new2", [1, 2, 3]))
```

	new2 i64	new i64	date datetime[s]	float f64	string str
1	1		2022-01-01 00:00:00	4.0	"a"
2	2		2022-01-02 00:00:00	5.0	"b"
3	3		2022-01-03 00:00:00	6.0	"c"

```
df_copy.replace('float', pl.Series("new_float", [4.0, 5.0, 6.0]))
```

	new2 i64	new i64	date datetime[s]	float f64	string str
1	1		2022-01-01 00:00:00	4.0	"a"
2	2		2022-01-02 00:00:00	5.0	"b"
3	3		2022-01-03 00:00:00	6.0	"c"

```python
def foo(frame):
  return frame.with_columns(pl.Series("new", [1, 2, 3]))
df.pipe(foo)
```

	integer i64	date datetime[s]	float f64	string str	new i64
1		2022-01-01 00:00:00	4.0	"a"	1
2		2022-01-02 00:00:00	5.0	"b"	2
3		2022-01-03 00:00:00	6.0	"c"	3

```python
df.is_empty()
```

```
False
```

```python
df.clear() # make empty copy. replaced .cleared()
```

integer i64	date datetime[s]	float f64	string str

```python
df.clear().is_empty()
```

```
True
```

```python
df.rename({'integer': 'integer2'})
```

	integer2 i64	date datetime[s]	float f64	string str
1		2022-01-01 00:00:00	4.0	"a"
2		2022-01-02 00:00:00	5.0	"b"
3		2022-01-03 00:00:00	6.0	"c"

Convert to Other Python Objects

To Pandas

```
df.to_pandas()
```

	integer	date	float	string
0	1	2022-01-01	4.0	a
1	2	2022-01-02	5.0	b
2	3	2022-01-03	6.0	c

To Numpy

```
df.to_numpy()
```

```
array([[1, datetime.datetime(2022, 1, 1, 0, 0), 4.0, 'a'],
       [2, datetime.datetime(2022, 1, 2, 0, 0), 5.0, 'b'],
       [3, datetime.datetime(2022, 1, 3, 0, 0), 6.0, 'c']], dtype=object)
```

To List

```
df.get_columns() # columns as list of polars series
```

```
[shape: (3,)
 Series: 'integer' [i64]
 [
    1
    2
    3
 ],
 shape: (3,)
 Series: 'date' [datetime[s]]
 [
    2022-01-01 00:00:00
    2022-01-02 00:00:00
    2022-01-03 00:00:00
 ],
 shape: (3,)
 Series: 'float' [f64]
 [
    4.0
```

```
    5.0
    6.0
 ],
 shape: (3,)
 Series: 'string' [str]
 [
    "a"
    "b"
    "c"
 ]]
```

```
df.rows() # rows as list of tuples
```

```
[(1, datetime.datetime(2022, 1, 1, 0, 0), 4.0, 'a'),
 (2, datetime.datetime(2022, 1, 2, 0, 0), 5.0, 'b'),
 (3, datetime.datetime(2022, 1, 3, 0, 0), 6.0, 'c')]
```

To Python Dict

```
df.to_dict() # columns as dict of polars series
```

```
{'integer': shape: (3,)
 Series: 'integer' [i64]
 [
    1
    2
    3
 ],
 'date': shape: (3,)
 Series: 'date' [datetime[ s]]
 [
    2022-01-01 00:00:00
    2022-01-02 00:00:00
    2022-01-03 00:00:00
 ],
 'float': shape: (3,)
 Series: 'float' [f64]
 [
    4.0
```

```
    5.0
    6.0
],
'string': shape: (3,)
Series: 'string' [str]
[
    "a"
    "b"
    "c"
]}
```

Dataframe in Memory

```
df.estimated_size(unit="gb")
```

```
9.96515154838562e-08
```

```
df.n_chunks() # number of ChunkedArrays in the dataframe
```

```
1
```

```
df.rechunk() # ensure contiguous memory layout
```

	integer i64	date datetime[s]	float f64	string str
1	2022-01-01 00:00:00	4.0	"a"	
2	2022-01-02 00:00:00	5.0	"b"	
3	2022-01-03 00:00:00	6.0	"c"	

```
df.shrink_to_fit() # reduce memory allocation to actual size
```

	integer i64	date datetime[s]	float f64	string str
1	2022-01-01 00:00:00	4.0	"a"	
2	2022-01-02 00:00:00	5.0	"b"	

	integer i64	date datetime[s]	float f64	string str
3	2022-01-03 00:00:00	6.0	"c"	

Statistical Aggregations

```
df.describe()
```

describe str	integer f64	date str	float f64	string str
"count"	3.0	"3"	3.0	"3"
"null_count"	0.0	"0"	0.0	"0"
"mean"	2.0	null	5.0	null
"std"	1.0	null	1.0	null
"min"	1.0	"2022-01-01 00:…	4.0	"a"
"max"	3.0	"2022-01-03 00:…	6.0	"c"
"median"	2.0	null	5.0	null
"25%"	1.0	null	4.0	null
"75%"	3.0	null	6.0	null

Compare to pandas:

```
df.to_pandas().describe()
```

	integer	date	float
count	3.0	3	3.0
mean	2.0	2022-01-02 00:00:00	5.0
min	1.0	2022-01-01 00:00:00	4.0
25%	1.5	2022-01-01 12:00:00	4.5
50%	2.0	2022-01-02 00:00:00	5.0
75%	2.5	2022-01-02 12:00:00	5.5
max	3.0	2022-01-03 00:00:00	6.0
std	1.0	NaN	1.0

Things to note:

- Comparing to pandas:

- Polars will summarize all columns even if they are not numeric.
- The statistics returned are different.

Statistical aggregations operate column-wise (and in parallel).

```
df.max()
```

	integer	date	float	string
	i64	datetime[s]	f64	str
3		2022-01-03 00:00:00	6.0	”c”

```
df.min()
```

	integer	date	float	string
	i64	datetime[s]	f64	str
1		2022-01-01 00:00:00	4.0	”a”

```
df.mean()
```

integer	date	float	string
f64	datetime[s]	f64	str
2.0	null	5.0	null

```
df.median()
```

integer	date	float	string
f64	datetime[s]	f64	str
2.0	null	5.0	null

```
df.sum()
```

integer i64	date datetime[s]	float f64	string str
6	null	15.0	null

```
df.std()
```

integer f64	date datetime[s]	float f64	string str
1.0	null	1.0	null

```
df.quantile(0.1)
```

integer f64	date datetime[s]	float f64	string str
1.0	null	4.0	null

Extraction

1. If you are used to pandas, recall there is no index. There is thus no need for `loc` vs. `iloc`, `reset_index()`, etc. See here for a comparison of extractors between polars and pandas.
2. Filtering and selection is possible with the `[` operator, or the `filter()` and `select()` methods. The latter is recommended to facilitate query planning (discussed in Section **??**).

Single cell extraction.

```
df[0,0] # like pandas .iloc[]
```

```
1
```

Slicing along rows.

```
df[0:1]
```

integer i64	date datetime[s]	float f64	string str
1	2022-01-01 00:00:00	4.0	”a”

Slicing along columns.

```
df[:,0:1]
```

integer i64
1
2
3

Selecting Columns

Column selection by label

```
df.select("integer")
# or df['integer']
# or df[:,'integer']
```

integer i64
1
2
3

Select columns with list of labels

```
df.select(["integer", "float"])
# or df[['integer', 'float']]
```

integer i64	float f64
1	4.0

integer	float
i64	f64
2	5.0
3	6.0

As of polars>=15.0.0, you don't have to pass a list:

```
df.select("integer", "float")
```

integer	float
i64	f64
1	4.0
2	5.0
3	6.0

Column slicing by label

```
df[:,"integer":"float"]
```

integer	date	float
i64	datetime[s]	f64
1	2022-01-01 00:00:00	4.0
2	2022-01-02 00:00:00	5.0
3	2022-01-03 00:00:00	6.0

Note: `df.select()` does not support slicing ranges such as `df.select("integer":"float")`.

Get a column as a 1D polars frame.

```
df.get_column('integer')
```

integer
i64
1
2
3

Get a column as a polars series.

```
df.to_series(0)
```

integer i64
1
2
3

```
df.find_idx_by_name('float')
```

```
2
```

```
df.drop("integer")
```

date datetime[s]	float f64	string str
2022-01-01 00:00:00	4.0	"a"
2022-01-02 00:00:00	5.0	"b"
2022-01-03 00:00:00	6.0	"c"

`df.drop()` not have an `inplace` argument. Use `df.drop_in_place()` instead.

pl.col()

The `pl.col()` is **super important** for referencing columns. It will be used to select columns within a `df.select()` context, and to transform columns within a `df.with_columns()` context. It may extract a single column, a list, a particular (polars) dtype, a regex pattern, or simply all columns.

When exctracting along dtype, use polars' dtypes, not pandas' dtypes. For example, use `pl.Int64` instead of `np.int64`.

Select along dtype

```
df.select(pl.col(pl.Int64))
```

integer
i64
1
2
3

```
df.select(pl.col(pl.Float64))
```

float
f64
4.0
5.0
6.0

```
df.select(pl.col(pl.Utf8))
```

string
str
"a"
"b"
"c"

List of dtypes

```
df.select(pl.col([pl.Int64, pl.Float64]))
```

integer	float
i64	f64
1	4.0
2	5.0
3	6.0

Regular Expression

```
df.select(pl.col("*")) # same as df.select(pl.all())
```

	integer i64	date datetime[s]	float f64	string str
1		2022-01-01 00:00:00	4.0	"a"
2		2022-01-02 00:00:00	5.0	"b"
3		2022-01-03 00:00:00	6.0	"c"

```
df.select(pl.col("*").exclude("integer"))
```

date datetime[s]	float f64	string str
2022-01-01 00:00:00	4.0	"a"
2022-01-02 00:00:00	5.0	"b"
2022-01-03 00:00:00	6.0	"c"

```
df.select(pl.col("*").exclude(pl.Float64))
```

	integer i64	date datetime[s]	string str
1		2022-01-01 00:00:00	"a"
2		2022-01-02 00:00:00	"b"
3		2022-01-03 00:00:00	"c"

```
df.select(pl.col("^.*te.*$")) # regex matching anything with a "te"
```

	integer i64	date datetime[s]
1		2022-01-01 00:00:00
2		2022-01-02 00:00:00
3		2022-01-03 00:00:00

Filtering Rows

```
df.head(2)
```

	integer i64	date datetime[s]	float f64	string str
1	2022-01-01 00:00:00	4.0	"a"	
2	2022-01-02 00:00:00	5.0	"b"	

```
df.limit(2) # same as pl.head()
```

	integer i64	date datetime[s]	float f64	string str
1	2022-01-01 00:00:00	4.0	"a"	
2	2022-01-02 00:00:00	5.0	"b"	

```
df.tail(1)
```

	integer i64	date datetime[s]	float f64	string str
3	2022-01-03 00:00:00	6.0	"c"	

```
df.take_every(2)
```

	integer i64	date datetime[s]	float f64	string str
1	2022-01-01 00:00:00	4.0	"a"	
3	2022-01-03 00:00:00	6.0	"c"	

```
df.slice(offset=1, length=1)
```

	integer i64	date datetime[s]	float f64	string str
2	2022-01-02 00:00:00	5.0	"b"	

```
df.sample(1)
```

	integer	date	float	string
	i64	datetime[s]	f64	str
2	2022-01-02 00:00:00	5.0	”b”	

```
df.row(1) # get row as tuple
```

```
(2, datetime.datetime(2022, 1, 2, 0, 0), 5.0, 'b')
```

Row filtering by label

```
df.filter(pl.col("integer") == 2)
```

	integer	date	float	string
	i64	datetime[s]	f64	str
2	2022-01-02 00:00:00	5.0	”b”	

Things to note:

- The `[` operator does not support indexing with boolean such as `df[df["integer"] == 2]`.
- The `filter()` method is recommended over `[` by the authors of polars, to facilitate lazy evaluation (discussed later).

Selecting A Single Item

Exctracts the first element as a scalar. Useful when you output a single number as a frame object.

```
pl.DataFrame([1]).item() # notice the output is not a frame, rather, a scalar.
```

```
1
```

Uniques and Duplicates

```
df.is_unique()
```

bool
true
true
true

```
df.is_duplicated()
```

bool
false
false
false

```
df.unique() # same as pd.drop_duplicates()
```

integer i64	date datetime[s]	float f64	string str
3	2022-01-03 00:00:00	6.0	"c"
2	2022-01-02 00:00:00	5.0	"b"
1	2022-01-01 00:00:00	4.0	"a"

```
df.n_unique()
```

3

Missing

```
df_with_nulls = df.with_columns(
    pl.Series("missing", [3, None, np.nan]),
)
```

```
df_with_nulls.null_count() # same as pd.isnull().sum()
```

integer u32	date u32	float u32	string u32	missing u32
0	0	0	0	1

```
df_with_nulls.drop_nulls() # same as pd.dropna()
```

	integer i64	date datetime[s]	float f64	string str	missing f64
1	2022-01-01 00:00:00	4.0	”a”	3.0	
3	2022-01-03 00:00:00	6.0	”c”	NaN	

Note: There is no `drop_nan()` method. See here for workarounds.

```
df_with_nulls.fill_null(0) # same as pd.fillna(0)
```

	integer i64	date datetime[s]	float f64	string str	missing f64
1	2022-01-01 00:00:00	4.0	”a”	3.0	
2	2022-01-02 00:00:00	5.0	”b”	0.0	
3	2022-01-03 00:00:00	6.0	”c”	NaN	

But recall that `None` and `np.nan` are not the same thing.

```
df_with_nulls.fill_nan(99)
```

	integer i64	date datetime[s]	float f64	string str	missing f64
1	2022-01-01 00:00:00	4.0	"a"	3.0	
2	2022-01-02 00:00:00	5.0	"b"	null	
3	2022-01-03 00:00:00	6.0	"c"	99.0	

```
df_with_nulls.interpolate()
```

	integer i64	date datetime[s]	float f64	string str	missing f64
1	2022-01-01 00:00:00	4.0	"a"	3.0	
2	2022-01-02 00:00:00	5.0	"b"	NaN	
3	2022-01-03 00:00:00	6.0	"c"	NaN	

Transformations

- The general idea of colum trasformation is to wrap all transformations in a `with_columns()` method, and the select colums to operat on with `pl.col()`.
- Previous versions of polars used `df.with_column()` and `df.with_columns()`. The `with_column()` method is now deprecated.
- The output column will have the same name as the input, unless you use the `alias()` method to rename it.
- The `with_columns()` is called a **polars context**.
- The flavor of the `with_columns()` context is similar to pandas' `assign()`.
- One can use `df.iter_rows()` to get an iterator over rows.

```
df.with_columns(
    pl.col("integer") * 2,
    pl.col("integer").alias("integer2"),
    integer3 = pl.col("integer") * 3
)
```

	integer i64	date datetime[s]	float f64	string str	integer2 i64	integer3 i64
2	2022-01-01 00:00:00	4.0	"a"	1	3	
4	2022-01-02 00:00:00	5.0	"b"	2	6	
6	2022-01-03 00:00:00	6.0	"c"	3	9	

Things to note:

- The columns `integer` is multiplied by 2 in place, because no `alias` is used.
- The column `integer` is copied, by renaming it to `integer2`.
- As of polars version >15.. (I think), you can use `=` to assign. That is how `integer3` is created.
- You cannot use `[` to assign! This would not have worked `df['integer3'] = df['integer'] * 2`

If a selection returns multiple columns, all will be transformed:

```
df.with_columns(
    pl.col([pl.Int64,pl.Float64])*2
)
```

integer i64	date datetime[s]	float f64	string str
2	2022-01-01 00:00:00	8.0	"a"
4	2022-01-02 00:00:00	10.0	"b"
6	2022-01-03 00:00:00	12.0	"c"

```
df.with_columns(
    pl.all().cast(pl.Utf8)
)
```

integer str	date str	float str	string str
"1"	"2022-01-01 00:...	"4.0"	"a"
"2"	"2022-01-02 00:...	"5.0"	"b"
"3"	"2022-01-03 00:...	"6.0"	"c"

Apply your own lambda function.

```
df.select([pl.col("integer"), pl.col("float")]).apply(lambda x: x[0] + x[1])
```

apply f64
5.0

apply
f64
7.0
9.0

As usual, using your own functions may have a very serious toll on performance:

```
df_big = pl.DataFrame(np.random.randn(1000000, 2), schema=["a", "b"]) # previous versions
```

```
%timeit -n2 -r2 df_big.sum(axis=1)
```

```
683 µs ± 357 µs per loop (mean ± std. dev. of 2 runs, 2 loops each)
```

```
%timeit -n2 -r2 df_big.apply(lambda x: x[0] + x[1])
```

```
218 ms ± 448 µs per loop (mean ± std. dev. of 2 runs, 2 loops each)
```

How would numpy and pandas deal with this row-wise summation?

```
df.shift(1)
```

integer	date	float	string
i64	datetime[s]	f64	str
null	null	null	null
1	2022-01-01 00:00:00	4.0	"a"
2	2022-01-02 00:00:00	5.0	"b"

```
df.shift_and_fill(1, 'WOW')
```

```
TypeError: shift_and_fill() takes 2 positional arguments but 3 were given
```

Sorting

```
df.sort(by=["integer","float"])
```

integer i64	date datetime[s]	float f64	string str
1	2022-01-01 00:00:00	4.0	"a"
2	2022-01-02 00:00:00	5.0	"b"
3	2022-01-03 00:00:00	6.0	"c"

```
df.reverse()
```

integer i64	date datetime[s]	float f64	string str
3	2022-01-03 00:00:00	6.0	"c"
2	2022-01-02 00:00:00	5.0	"b"
1	2022-01-01 00:00:00	4.0	"a"

Joins

High level:

- `df.hstack()` for horizontal concatenation; like pandas `pd.concat([],axis=1)` or R's `cbind`.
- `df.vstack()` for vertical concatenation; like pandas `pd.concat([],axis=0)` or R's `rbind`.
- `df.merge_sorted()` for vertical stacking, with sorting.
- `pl.concat()`, which is similar to the previous two, but with memory re-chunking. `pl.concat()` also allows diagonal concatenation, if columns are not shared.
- `df.extend()` for vertical concatenation, but with memory re-chunking. Similar to `df.vstack().rechunk()`.
- `df.join()` for joins; like pandas `pd.merge()` or `df.join()`.

For more on the differences between these methods, see here.

hstack

```
new_column = pl.Series("c", np.repeat(1, df.height))

df.hstack([new_column])
```

	integer i64	date datetime[s]	float f64	string str	c i64
1		2022-01-01 00:00:00	4.0	"a"	1
2		2022-01-02 00:00:00	5.0	"b"	1
3		2022-01-03 00:00:00	6.0	"c"	1

vstack

```
df2 = pl.DataFrame({
  "integer": [1, 2, 3],
  "date": [
    (datetime(2022, 1, 4)),
    (datetime(2022, 1, 5)),
    (datetime(2022, 1, 6))],
    "float":[7.0, 8.0, 9.0],
    "string": ["d", "d", "d"]})


df.vstack(df2)
```

	integer i64	date datetime[s]	float f64	string str
1		2022-01-01 00:00:00	4.0	"a"
2		2022-01-02 00:00:00	5.0	"b"
3		2022-01-03 00:00:00	6.0	"c"
1		2022-01-04 00:00:00	7.0	"d"
2		2022-01-05 00:00:00	8.0	"d"
3		2022-01-06 00:00:00	9.0	"d"

Concatenation

```
pl.concat([df, df2])
# equivalent to:
# pl.concat([df, df2], how='vertical', rechunk=True, parallel=True)
```

	integer i64	date datetime[s]	float f64	string str
1	2022-01-01 00:00:00	4.0	"a"	
2	2022-01-02 00:00:00	5.0	"b"	
3	2022-01-03 00:00:00	6.0	"c"	
1	2022-01-04 00:00:00	7.0	"d"	
2	2022-01-05 00:00:00	8.0	"d"	
3	2022-01-06 00:00:00	9.0	"d"	

```
pl.concat([df,new_column.to_frame()], how='horizontal')
```

	integer i64	date datetime[s]	float f64	string str	c i64
1	2022-01-01 00:00:00	4.0	"a"	1	
2	2022-01-02 00:00:00	5.0	"b"	1	
3	2022-01-03 00:00:00	6.0	"c"	1	

extend

```
df.extend(df2) # like vstack, but with memory re-chunking. Similar to df.vstack().rechunk(
```

	integer i64	date datetime[s]	float f64	string str
1	2022-01-01 00:00:00	4.0	"a"	
2	2022-01-02 00:00:00	5.0	"b"	
3	2022-01-03 00:00:00	6.0	"c"	
1	2022-01-04 00:00:00	7.0	"d"	
2	2022-01-05 00:00:00	8.0	"d"	
3	2022-01-06 00:00:00	9.0	"d"	

merge_sorted

```
df.merge_sorted(df2, key="integer") # vstacking with sorting.
```

integer i64	date datetime[s]	float f64	string str
1	2022-01-01 00:00:00	4.0	"a"
1	2022-01-04 00:00:00	7.0	"d"
2	2022-01-02 00:00:00	5.0	"b"
2	2022-01-05 00:00:00	8.0	"d"
3	2022-01-03 00:00:00	6.0	"c"
1	2022-01-04 00:00:00	7.0	"d"
2	2022-01-05 00:00:00	8.0	"d"
3	2022-01-06 00:00:00	9.0	"d"
3	2022-01-06 00:00:00	9.0	"d"

Caution: Joining along rows is possible only if matched columns have the same dtype. Timestamps may be tricky because they may have different time units. Recall that timeunits may be cast before joining using `series.dt.cast_time_unit()`:

```
df.with_columns(
    pl.col(pl.Datetime("ns")).dt.cast_time_unit(tu="ms")
)
```

If you cannot arrange schema before concatenating, use a diagonal concatenation:

```
pl.concat(
    [df,new_column.to_frame()],
    how='diagonal')
```

integer i64	date datetime[s]	float f64	string str	c i64
1	2022-01-01 00:00:00	4.0	"a"	null
2	2022-01-02 00:00:00	5.0	"b"	null
3	2022-01-03 00:00:00	6.0	"c"	null
1	2022-01-04 00:00:00	7.0	"d"	null
2	2022-01-05 00:00:00	8.0	"d"	null
3	2022-01-06 00:00:00	9.0	"d"	null
null	null	null	null	1

	integer	date	float	string	c
	i64	datetime[s]	f64	str	i64
null	null		null	null	1
null	null		null	null	1

join

```
df.join(df2, on="integer", how="left")
```

integer	date	float	string	date_right	float_right	string_right
i64	datetime[s]	f64	str	datetime[s]	f64	str
1	2022-01-01 00:00:00	4.0	"a"	2022-01-04 00:00:00	7.0	"d"
2	2022-01-02 00:00:00	5.0	"b"	2022-01-05 00:00:00	8.0	"d"
3	2022-01-03 00:00:00	6.0	"c"	2022-01-06 00:00:00	9.0	"d"
1	2022-01-04 00:00:00	7.0	"d"	2022-01-04 00:00:00	7.0	"d"
2	2022-01-05 00:00:00	8.0	"d"	2022-01-05 00:00:00	8.0	"d"
3	2022-01-06 00:00:00	9.0	"d"	2022-01-06 00:00:00	9.0	"d"

Things to note:

- Repeating column names have been suffixed with "_right".
- Unlike pandas, there are no indices. The `on`/`left_on`/`right_on` argument is always required.
- `how=` may take the following values: 'inner', 'left', 'outer', 'semi', 'anti', 'cross'.
- The join is super fast, as demonstrated in Section above.

join_asof

```
df.join_asof(
    df2,
    left_on="date",
    right_on='date',
    by="integer",
    strategy="backward",
    tolerance='1w')
```

```
argument in operation 'asof_join' is not explicitly sorted

- If your data is ALREADY sorted, set the sorted flag with: '.set_sorted()'.
- If your data is NOT sorted, sort the 'expr/series/column' first.

This might become an error in a future version.

argument in operation 'asof_join' is not explicitly sorted

- If your data is ALREADY sorted, set the sorted flag with: '.set_sorted()'.
- If your data is NOT sorted, sort the 'expr/series/column' first.

This might become an error in a future version.
```

integer i64	date datetime[s]	float f64	string str	float_right f64	string_right str
1	2022-01-01 00:00:00	4.0	"a"	null	null
2	2022-01-02 00:00:00	5.0	"b"	null	null
3	2022-01-03 00:00:00	6.0	"c"	null	null
1	2022-01-04 00:00:00	7.0	"d"	7.0	"d"
2	2022-01-05 00:00:00	8.0	"d"	8.0	"d"
3	2022-01-06 00:00:00	9.0	"d"	9.0	"d"

Things to note:

- Yes! `merge_asof()` is also available.
- The `strategy=` argument may take the following values: 'backward', 'forward'.
- The `tolerance=` argument may take the following values: '1w', '1d', '1h', '1m', '1s', '1ms', '1us', '1ns'.

Reshaping

High level:

- `df.transpose()` as the name suggests.
- `df.melt()` for wide to long.
- `df.pivot()` for long to wide.
- `df.explode()` for breaking strings into rows.
- `df.unstack()`

```
df.transpose()
```

column_0 str	column_1 str	column_2 str	column_3 str	column_4 str	column_5 str
"1"	"2"	"3"	"1"	"2"	"3"
"2022-01-01 00:...	"2022-01-02 00:...	"2022-01-03 00:...	"2022-01-04 00:...	"2022-01-05 00:...	"2022-01-06 0
"4.0"	"5.0"	"6.0"	"7.0"	"8.0"	"9.0"
"a"	"b"	"c"	"d"	"d"	"d"

Wide to Long

```
# The following example is adapted from Pandas documentation: https://pandas.pydata.org/do

np.random.seed(123)
wide = pl.DataFrame({
    'famid': ["11", "12", "13", "2", "2", "2", "3", "3", "3"],
    'birth': [1, 2, 3, 1, 2, 3, 1, 2, 3],
    'ht1': [2.8, 2.9, 2.2, 2, 1.8, 1.9, 2.2, 2.3, 2.1],
    'ht2': [3.4, 3.8, 2.9, 3.2, 2.8, 2.4, 3.3, 3.4, 2.9]})

wide.head(2)
```

famid str	birth i64	ht1 f64	ht2 f64
"11"	1	2.8	3.4
"12"	2	2.9	3.8

```
wide.melt(
  id_vars=['famid', 'birth'],
  value_vars=['ht1', 'ht2'],
  variable_name='treatment',
  value_name='height').sample(5)
```

famid str	birth i64	treatment str	height f64
"3"	3	"ht2"	2.9

famid str	birth i64	treatment str	height f64
"3"	2	"ht1"	2.3
"2"	1	"ht2"	3.2
"3"	3	"ht1"	2.1
"2"	3	"ht2"	2.4

Break strings into rows.

```
wide.explode(columns=['famid']).limit(5)
```

famid str	birth i64	ht1 f64	ht2 f64
"1"	1	2.8	3.4
"1"	1	2.8	3.4
"1"	2	2.9	3.8
"2"	2	2.9	3.8
"1"	3	2.2	2.9

Long to Wide

```
# Example adapted from https://stackoverflow.com/questions/5890584/how-to-reshape-data-fro

long = pl.DataFrame({
    'id': [1, 1, 1, 2, 2, 2, 3, 3, 3],
    'treatment': ['A', 'A', 'B', 'A', 'A', 'B', 'A', 'A', 'B'],
    'height': [2.8, 2.9, 2.2, 2, 1.8, 1.9, 2.2, 2.3, 2.1]
    })

long.limit(5)
```

id i64	treatment str	height f64
1	"A"	2.8
1	"A"	2.9
1	"B"	2.2
2	"A"	2.0

id	treatment	height
i64	str	f64
2	"A"	1.8

```
long.pivot(
  index='id', # index in the wide format
  columns='treatment', # defines columns in the wide format
  values='height')
```

```
/var/folders/91/c3y_9h950pb0gq8c8sdytk1r0000gn/T/ipykernel_85263/1142736381.py:1: Deprecatio

In a future version of polars, the default `aggregate_function` will change from `'first'` to
```

id	A	B
i64	f64	f64
1	2.8	2.2
2	2.0	1.9
3	2.2	2.1

```
long.unstack(step=2) # works like a transpose, and then wrap rows. Change the `step=` to g
```

id_0	id_1	id_2	id_3	id_4	treatment_0	treatment_1	treatment_2	treatment_3	treatment_4
i64	i64	i64	i64	i64	str	str	str	str	str
1	1	2	3	3	"A"	"B"	"A"	"A"	"B"
1	2	2	3	null	"A"	"A"	"B"	"A"	null

Groupby

Grouping over categories:

- `df.partion_by()` will return a list of frames.
- `df.groupby()` for grouping. Just like pandas, only parallelized, etc. The output will have the length of the number of groups.
- `over()` will assign each row the aggregate in the group. Like pandas `groupby.transform`. The output will have the same length as the input.

Grouping over time:

- `df.grouby_rolling()` for rolling window grouping, a.k.a. a sliding window. Each row will be assigned the aggregate in the window.
- `df.groupby_dynamic()` for dynamic grouping. Each period will be assigned the agregate in the period. The output may have more rows than the input.

After grouping:

- `df.groupby().agg()` for aggregating.
- `df.groupby().apply()` for applying a function to each group.
- `df.groupby().count()` for counting.
- `df.groupby().first()` for getting the first row of each group.
- …

See the API reference for the various options. Also see the user guide for more details.

```
df2 = pl.DataFrame({
    "integer": [1, 1, 2, 2, 3, 3],
    "float": [1.0, 2.0, 3.0, 4.0, 5.0, 6.0],
    "string": ["a", "b", "c", "d", "e", "f"],
    "datetime": [
        (datetime(2022, 1, 4)),
        (datetime(2022, 1, 4)),
        (datetime(2022, 1, 4)),
        (datetime(2022, 1, 9)),
        (datetime(2022, 1, 9)),
        (datetime(2022, 1, 9))],
})
```

```
df2.partition_by("integer")
```

```
[shape: (2, 4)

  integer  float  string  datetime
  ---      ---    ---     ---
  i64      f64    str     datetime[ s]

  1        1.0    a       2022-01-04 00:00:00
  1        2.0    b       2022-01-04 00:00:00
                          ,
 shape: (2, 4)
```

```
 integer   float   string   datetime
 ---       ---     ---      ---
 i64       f64     str      datetime[ s]

 2         3.0     c        2022-01-04 00:00:00
 2         4.0     d        2022-01-09 00:00:00
                            ,
shape: (2, 4)

 integer   float   string   datetime
 ---       ---     ---      ---
 i64       f64     str      datetime[ s]

 3         5.0     e        2022-01-09 00:00:00
 3         6.0     f        2022-01-09 00:00:00
                           ]
```

```
groupper = df2.groupby("integer")
groupper.count()
```

integer i64	count u32
1	2
2	2
3	2

```
groupper.sum()
```

integer i64	float f64	string str	datetime datetime[s]
2	7.0	null	null
3	11.0	null	null
1	3.0	null	null

Groupby a fixed time window with `df.groupby_dynamic()`:

```
(
  df2
```

```
    .groupby_dynamic(index_column="datetime", every="1d")
    .agg(pl.col("float").sum())
)
```

```
argument in operation 'groupby_dynamic' is not explicitly sorted

- If your data is ALREADY sorted, set the sorted flag with: '.set_sorted()'.
- If your data is NOT sorted, sort the 'expr/series/column' first.

This might become an error in a future version.
```

datetime datetime[s]	float f64
2022-01-04 00:00:00	6.0
2022-01-09 00:00:00	15.0

If you do not want a single summary per period, rather, a window at each datapoint, use `df.groupby_rolling()`:

```
(
  df2
  .groupby_rolling(index_column="datetime", period='1d')
  .agg(pl.col("float").sum())
)
```

```
argument in operation 'groupby_rolling' is not explicitly sorted

- If your data is ALREADY sorted, set the sorted flag with: '.set_sorted()'.
- If your data is NOT sorted, sort the 'expr/series/column' first.

This might become an error in a future version.
```

datetime datetime[s]	float f64
2022-01-04 00:00:00	1.0

datetime datetime[s]	float f64
2022-01-04 00:00:00	3.0
2022-01-04 00:00:00	6.0
2022-01-09 00:00:00	4.0
2022-01-09 00:00:00	9.0
2022-01-09 00:00:00	15.0

Over

You may be familar with pandas `groupby().transform()`, which will return a frame with the same row-count as its input. You may be familiar with Postgres SQL window function. You may not be familiar with either, and still want to aggregate within group, but propagate the result to all group members. Polars' `over()` is the answer.

```
df.with_columns(
  pl.col("float").sum().over("string").alias("sum")
).limit(5)
```

integer i64	date datetime[s]	float f64	string str	sum f64
1	2022-01-01 00:00:00	4.0	"a"	4.0
2	2022-01-02 00:00:00	5.0	"b"	5.0
3	2022-01-03 00:00:00	6.0	"c"	6.0
1	2022-01-04 00:00:00	7.0	"d"	24.0
2	2022-01-05 00:00:00	8.0	"d"	24.0

Careful: `over()` should follow the aggregation. The following will not fail, but return the wrong result:

```
df.with_columns(
  pl.col("float").over("string").sum().alias("sum")
).limit(5)
```

integer i64	date datetime[s]	float f64	string str	sum f64
1	2022-01-01 00:00:00	4.0	"a"	39.0
2	2022-01-02 00:00:00	5.0	"b"	39.0

	integer i64	date datetime[s]	float f64	string str	sum f64
3	2022-01-03 00:00:00	6.0	"c"	39.0
1	2022-01-04 00:00:00	7.0	"d"	39.0
2	2022-01-05 00:00:00	8.0	"d"	39.0

Processing Multiple Frames Simultanously

Q: What if you want to access a column from frame `df`, when processing frame `df2`?
A: Just join them.
Q: What if they are not joinable?
A: Use a diagonal join. Q: Can't I just add a search-space into the lazy query? A: Ahhh! Use `df.with_context()`.

```
df3 = pl.Series("blah", [100,2,3]).to_frame()

q = (
    df.lazy()
    .with_context( # add colums of df2 to the search space
        df3.lazy()
        )
    .with_columns(
        pl.col('float').map_dict(remapping={4.0:None}, default=100).fill_null(pl.col('blah
        )
    )

q.collect()
```

	integer i64	date datetime[s]	float f64	string str	float2 f64
1	2022-01-01 00:00:00	4.0	"a"	35.0
2	2022-01-02 00:00:00	5.0	"b"	100.0
3	2022-01-03 00:00:00	6.0	"c"	100.0
1	2022-01-04 00:00:00	7.0	"d"	100.0
2	2022-01-05 00:00:00	8.0	"d"	100.0
3	2022-01-06 00:00:00	9.0	"d"	100.0

Things to note:

- `with_context()` is a lazy operation. This is great news, since it means both frames will benefit from query planning, etc.
- `with_context()` will not copy the data, but rather, add a reference to the data.
- Why not use `pl.col('blah').mean()` within the `map_dict()`? That is indeed more reasonable. It simply did not work.
- Try it yourself: Can you use multiple `with_context()`?

Query Planning and Optimization

The take-home of this section, is that polar can take advantage of half-a-century's worth of research in query planning and optimization. You will not have to think about the right order of operations, or the right data structures to use. Rather, replace the polars dataframe with a polars lazy-dataframe, state all the operations you want, and just finish with a `collect()`. Polars will take care of the rest, and provide you with the tools to understand its plan.

We will not go into the details of the difference between a lazy and a non-lazy dataframe. Just assume a lazy frame allows everything a non-lazy frame can do, but it does not execute the operations until you call `collect()`. This is not entirely true, but you will get an informative error if you try to do something that is not supported.

Get your lazy dataframe:

```
df_lazy = df.lazy()
```

State all your operations:

```
q = (
  df_lazy
  .filter(pl.col("float") > 2.0)
  .filter(pl.col("float") > 3.0)
  .filter(pl.col("float") > 7.0)
  .select(["integer"])
  .sort("integer")
)
```

And now visualize the query.

```
q # same as q.show_graph(optimized=False)
```

```
<polars.LazyFrame object at 0x29F419C10>
```

```python
q.show_graph(optimized=True)
```

Things to note:

- You will need Graphviz installed to visualize the query plan.
- To understand the plan, you need some terminology from relational databases. Namely:
  - A *selection* is a polars' filter, i.e. subset of rows, marked in the graph with a σ.
  - A *projection* is polars seletion, i.e. a subset of columns, marked in the graph with a π.
- The optimized plan removes redudancies, and orders the operations in the most efficient way.

You can now execute the plan with a `collect()`:

```python
q.collect()
```

integer i64
2
3

```python
q.describe_plan()
```

```
/var/folders/91/c3y_9h950pb0gq8c8sdytk1r0000gn/T/ipykernel_85263/3358036380.py:1: Deprecatio

`LazyFrame.describe_plan` has been deprecated; Please use `LazyFrame.explain` instead

'SORT BY [col("integer")]\n   SELECT [col("integer")] FROM\n    FILTER [(col("float")) > (7.
```

Inspecting, Profiling, and Debugging a Query

For early stopping (debugging?) you can replace `collect()` with `fetch()`:

```python
q.fetch(2)
```

integer
i64
2
3

You can inspect the data at any point in the query. `df.inspect()` will print the state of a single node in the query graph: [TODO: replace with example with multiple nodes]

```
q = (
    pl.scan_parquet(f'{path}/*.parquet')
    .filter(
        (pl.col('passenger_count') > 0) &
        (pl.col('passenger_count') < 5) &
        (pl.col('trip_distance') > 0) &
        (pl.col('trip_distance') < 10) &
        (pl.col('fare_amount') > 0) &
        (pl.col('fare_amount') < 100) &
        (pl.col('tip_amount') > 0) &
        (pl.col('tip_amount') < 20) &
        (pl.col('total_amount') > 0) &
        (pl.col('total_amount') < 100)
    )
    .inspect() # here is the inspect
    .groupby('passenger_count')
    .agg([pl.mean('tip_amount')])
    )
q.collect()
```

```
shape: (3_537_967, 5)

 tip_amount   passenger_count   trip_distance   fare_amount   total_amount
 ---          ---               ---             ---           ---
 f64          f64               f64             f64           f64

 3.65         2.0               3.8             14.5          21.95
 4.0          1.0               2.1             8.0           13.3
 1.76         1.0               0.97            7.5           10.56
 3.0          1.0               4.3             23.5          30.3
 …            …                 …               …             …
 3.5          2.0               1.7             8.0           15.3
 2.26         1.0               1.2             7.5           13.56
```

```
4.86          1.0              5.62           20.5          29.16
2.36          1.0              1.9            8.0           14.16
```

PARTITIONED DS

passenger_count f64	tip_amount f64
1.0	2.701872
4.0	2.782241
2.0	2.749387
3.0	2.715053

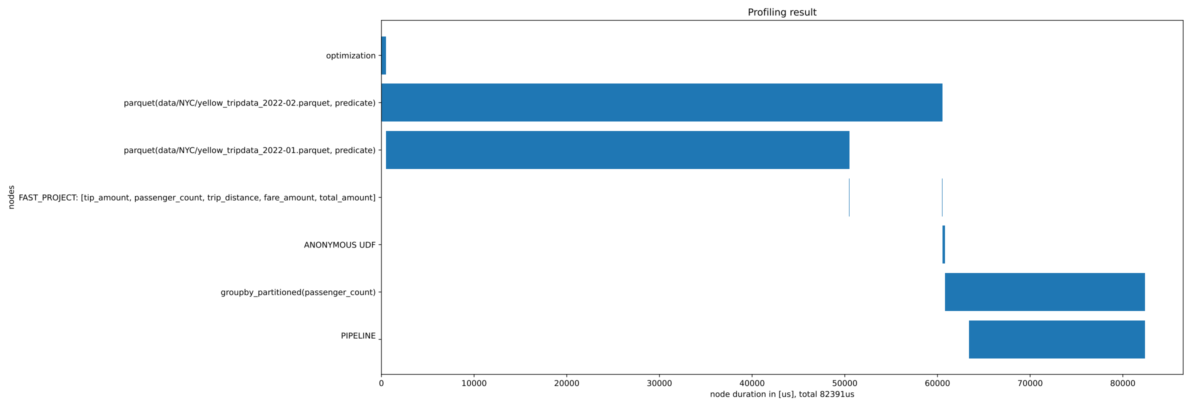
You can profile the execution of a query with `df.profile()`:

```
q.profile(show_plot=True)
```

PARTITIONED DS

shape: (3_537_967, 5)

tip_amount --- f64	passenger_count --- f64	trip_distance --- f64	fare_amount --- f64	total_amount --- f64
3.65	2.0	3.8	14.5	21.95
4.0	1.0	2.1	8.0	13.3
1.76	1.0	0.97	7.5	10.56
3.0	1.0	4.3	23.5	30.3
…	…	…	…	…
3.5	2.0	1.7	8.0	15.3
2.26	1.0	1.2	7.5	13.56
4.86	1.0	5.62	20.5	29.16
2.36	1.0	1.9	8.0	14.16




```
(shape: (4, 2)

  passenger_count   tip_amount
  ---               ---
  f64               f64

  1.0               2.701872
  4.0               2.782241
  2.0               2.749387
  3.0               2.715053
                 ,
 shape: (8, 3)

  node                                  start   end
  ---                                   ---     ---
  str                                   u64     u64

  optimization                          0       491
  parquet(data/NYC/yellow_tripdata…     33      60520
  parquet(data/NYC/yellow_tripdata…     491     50482
  FAST_PROJECT: [tip_amount, passe…     50486   50490
  FAST_PROJECT: [tip_amount, passe…     60528   60530
  ANONYMOUS UDF                         60571   60830
  groupby_partitioned(passenger_co…     60831   82396
  PIPELINE                              63419   82391

                              )
```

Exporting a Query

You can export your query, as a JSON file.

```
q.write_json("query.json") # export
```

```
ValueError: Error("the enum variant FunctionNode::Opaque cannot be serialized", line: 0, colu
```

This is how the query will look on disk:

```
import json
json.loads(open("query.json").read())# inspect
```

```
JSONDecodeError: Expecting value: line 1 column 3376 (char 3375)
```

You can now load it and run it.

```
pl.LazyFrame.read_json("query.json").collect()
```

```
ValueError: Error("EOF while parsing a value", line: 1, column: 3375)
```

SQL Flavor

If you are a hardcore SQL user, you may want to use the SQL flavor of polars. The following syntax is experimental, and may change.

```
sql = pl.SQLContext()
sql.register("lazy_frame", lazy_frame) # register the lazy frame as a table

sql.query("""
    SELECT passenger_count, AVG(tip_amount) FROM lazy_frame
    WHERE passenger_count < 3
    GROUP BY passenger_count
    """) # query the table
```

```
NameError: name 'lazy_frame' is not defined
```

I/O

You will find that polars is blazing fast at reading and writing data. This is due to:

1. Very good heuristics/rules implemented in the `read_csv` function.
2. The use of Apache Arrow as an internal data structure, which maps seamlesly to the parquet file format.
3. Parallelism, whenever possible.
4. Lazy scans/imports, which allows the materialization only of required data; i.e., filters and projections are executed at scan time.

Import

High level:

- `pl.read_X()` will read a file into a non-lazy frame.
- `pl.scan_X()` will read a file into a lazy frame.
- You can use globs to import multiple files but:
  - You may need to teak schema manually.
  - Filesystme operations are handeled by fsspec, which may open only the first file when using globs in remote filesystems (e.g. S3). This is discussed in Section **??**.

From a Single File

Let’s firs make a csv to import:

```
df.write_csv("df.csv")
```

Import the csv into a non-lazy frame:

```
pl.read_csv("df.csv")
```

integer i64	date str	float f64	string str
1	”2022-01-01T00:...	4.0	”a”
2	”2022-01-02T00:...	5.0	”b”
3	”2022-01-03T00:...	6.0	”c”
1	”2022-01-04T00:...	7.0	”d”
2	”2022-01-05T00:...	8.0	”d”

	integer	date	float	string
	i64	str	f64	str
3	"2022-01-06T00:...	9.0	"d"	

Importing as a lazy frame:

```
df_lazy = pl.scan_csv("df.csv")
```

Things become interesting when you manipulate the lazy frame before materializing it:

```
q = (
  df_lazy
  .filter(pl.col("float") > 2.0)
  .filter(pl.col("float") > 3.0)
  .filter(pl.col("float") > 7.0)
  .select(["integer"])
  .sort("integer")
)

q.show_graph(optimized=True)
```

```
q.collect()
```

integer
i64
2
3

Things to note:

- From the graph we see that the filtering (σ) is done at scan time, and not after the materialization of the data. This is crucial for processing datasets that are larger then memory.
- To get the actual data, we naturally need to `collect()`.

Cleary, .csv is not the only format that can be read. It is possibly the least recommended. Other file types can be found here and include:

- Excel.
- Arrow IPC: A binary format for storing columnar data.
- Feather (V2): Multiple IPC files with a shared schema.
- Parquet (non-partitioned): A tabular file format (not columnar) that is optimized for long-term storage, more compressed than Feather.
- JSON: Short for JavaScript Object Notation, a textual data-interchange format (like XML).
- Avro: A binary row-based format. Good for streaming.

Each of the above formats has a non-lazy reader using `pl.read_*` and a lazy reader using `pl.scan_*`.

Currently unsuported formats:

- Feather (V1).
- HDF5.

From Multiple Files in Your Filesystem

Most of today's datasets will span more than a single file on disk. Polar supports reading from multiple files in your file system (as opposed to a remote datalake such as S3), and will automatically merge them into a single dataframe. There are, however, many file formats, and each has its own way of partitioning the data. Multi-file storage supported by polars (at the time of writing):

1. Parquet (partitioned): A collection of files with a common schema, partitioned as folders on disk.
2. Delta-Lake: If your data is saves as many parquet files on S3, a failed copy operation may "break" the data. Systems that protect data from such failures (failed copy is only an example) are called "transactional systems", and the garantees they provide are called "ACID". A Delta-Lake, is a piece of open source software, that manages your queries to give your data-lake the ACID properties.
3. Arrow Dataset: A collection of files (csv, parquet, feather, etc) with a common schema.

TODO: https://pola-rs.github.io/polars-book/user-guide/multiple_files/intro.html

Arbitrary Collection of Files

You can always scan from some arbitrary collection of files and concatenate the result.

```
path = 'data/NYC' # Data from https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page
file_names = os.listdir(path)
file_names
```

```
['.DS_Store',
 'yellow_tripdata_2022-01.parquet',
 'yellow_tripdata_2022-02.parquet']
```

```
df_lazy_list = []
for file in file_names:
    df_lazy_list.append(
        pl.scan_parquet(f'{path}/{file}')
        )
```

```
ArrowErrorException: ExternalFormat("File out of specification: The file must end with PAR1")
```

With a list of lazy frames you can proceed by concatenating into a single lazy frame using `pl.concat()`, or collecting them into a list of eager frames using `pl.collect_all()`. The best option depends on your use case.

Things to note:

- The arrow data format uses caching for string and categorical data (i.e. pl.Series). If importing multiple files, such as multiple parquet/feather files, or an arrow dataset, different files may be cached differnetly. This will cause an error when trying to concatenate the dataframes. To avoid this, you can disable string caching, or enforce joint caching of all files.
- Dataframes may have incompatible schema, as discussed in Section **??** above. You may need to manually adjust the schema before concatenating.

Here is an example that deals with both issues:

```
with pl.StringCache(): # Enforce joint caching of all files
  df_lazy_list = []
  for file in file_names:
    lazy_frame = (
        pl.scan_parquet(f'{path}/{file}') # read a lazy frame
        .with_columns(
            pl.col(pl.Datetime('ns')).dt.cast_time_unit('ms')
            ) # ensure joinable time units
    )
    df_lazy_list.append(lazy_frame)

q= (
    pl.concat(df_lazy_list) # concat into into a single lazy frame
    .filter(pl.col('passenger_count') < 3)
```

```
    .groupby('passenger_count')
    .agg([pl.mean('tip_amount')])
)
q.collect() # execute query
```

```
ArrowErrorException: ExternalFormat("File out of specification: The file must end with PAR1")
```

Partitioned Parquet

The code snipped above (@sec- multiple_files) is fully generalizable wrt the files you import and what you do to them. Most often, you don't need such generality. For instance, when importing multiple parquet files form the local file system, the `pl.read_parquet()` function will allow you to use globs. The above may thus read:

```
with pl.StringCache(): # Enforce joint caching of all files
  lazy_frame = pl.scan_parquet(f'{path}/*.parquet')

q= (
    lazy_frame # concat into into a single lazy frame
    .filter(pl.col('passenger_count') < 3)
    .groupby('passenger_count')
    .agg([pl.mean('tip_amount')])
)
q.collect() # execute query
```

```
PARTITIONED DS
```

passenger_count f64	tip_amount f64
1.0	2.400686
0.0	2.273948
2.0	2.579188

Apache Arrow Dataset

TODO: The exampe below deals with partitioned parquet, and not arrow datasets. Fix.

An Apache Arrow dataset is a collection of parquet files, with an index file. It is a very efficient way to store data on disk, and to read it in parallel.

Writing an Arrow dataset:

```
# Write df as an arrow dataset:
df.to_pandas().to_parquet(
    "df",
    engine="pyarrow",
    partition_cols=["integer"])

os.listdir("df") # inspect folder on disk
```

```
['integer=1', 'integer=2', 'integer=3']
```

```
# inspect partitions
[os.listdir(f"df/{x}/") for x in os.listdir("df")]
```

```
[['1559a42060594111899080O2b1b05e14-0.parquet'],
 ['1559a42060594111899080O2b1b05e14-0.parquet'],
 ['1559a42060594111899080O2b1b05e14-0.parquet']]
```

```
import pyarrow.dataset as ds
dset = ds.dataset("df", format="parquet")  # define folder as dataset
pl.scan_ds(dset).collect() # import
```

```
/var/folders/91/c3y_9h950pb0gq8c8sdytk1r0000gn/T/ipykernel_85263/385436092.py:3: Deprecation

`scan_ds` has been renamed; this redirect is temporary, please use `scan_pyarrow_dataset` ins
```

date datetime[s]	float f64	string str
2022-01-01 00:00:00	4.0	"a"
2022-01-04 00:00:00	7.0	"d"
2022-01-02 00:00:00	5.0	"b"
2022-01-05 00:00:00	8.0	"d"
2022-01-03 00:00:00	6.0	"c"
2022-01-06 00:00:00	9.0	"d"

Things to note:

- We used pandas to write the arrow dataset. It seemed easier than the pyarrow syntax.
- The `partition_cols` argument is used to partition the dataset on disk. Each partition is a parquet file (or another partition).
- Reading from the web (not from the local filesystem) is slightly different. TODO: add reference.

Multiple CSVs

TODO: `pl.read_csv_batched()`

From Multiple Files on a Remote Datalake

If you are coming from Pandas, reading from a remote datalake (say S3), and a local filesystem may feel the same. This is because the authors of pandas went to great lengths to make the API feel the same. At the time of writing, if you give polars a remote glob, it will only read the first file (ref).

Your current options for reading multiple files stored remotely are:

1. Read one file at a time, and concatenate the results, or use the `pl.scan_parquet()` as in @sec- multiple_files.
2. Use third party functionality that can link to multiple remote files. Luckily, the pyarrow library gives you this functionality. See here for an example.

Reading from a Database

See here.

Serverless

See here for working in serverless environments such as AWS Lambda.

Export to Disk

Well, there is not much to say here; just look for `pl.write_*` functions. Alternatively, export to pandas, arrow, numpy, and use their exporters.

Plotting

To get an intuition of what you may expect in this chapter you should know the following. There are various approaches to plotting in python:

1. The object oriented, where a dataframe has a plotting method. E.g. `df.plot()`. The method may use a single, or even multiple backends. Such is the pandas dataframe, which may use a matplotlib, plotly, or bokeh backend.
2. The functional method, where a plotting function takes a dataframe as an argument. E.g. `plot(df)`. Such are the matplotlib, seaborn, and plotly functions, which may take pandas dataframes as inputs.

Plotting support in polars thus boils down to the folowing questions: (1) Do polars dataframes have a plotting method? With which backend? (2) Can plotting functions take polars dataframes as inputs?

The answer to the first is negative. Polars dataframes do not have a plotting method, and it seems they are not planned to have one (TODO: add reference). The answer to the second is “almost yes”. Any plotting function that can take an iterable such as a list, or numpy 1D arrays, will work. Either becaus polars series are iterable, or because one can convert them (to arrow or numpy being the fastest).

Passing polars frames may cause trouble. You may expect to use a `plot(df, x='col1', y='col2')` syntax; it may work if `df` is a pandas dataframe, but not with polars. Support of this syntax does not depend on polars developers, rather, on the plotting function developpers. I suspect that the plotly and bokeh teams will eventually supprts polars. I do not know about the seaborm, or matplotlib teams.

The current state of affairs:

- Plotly, matplotlib, and seaborn support polars series as input.
- Matplotlib and seaborn support polars frames as input. Plotly (5.12.0) does not.

Plotly Functions

The `iris` dataset is provided by plotly as a pandas frame. We convert it to a polars frame.

```
iris = pl.DataFrame(px.data.iris())
iris.head()
```

sepal_length f64	sepal_width f64	petal_length f64	petal_width f64	species str	species_id i64
5.1	3.5	1.4	0.2	”setosa”	1

sepal_length f64	sepal_width f64	petal_length f64	petal_width f64	species str	species_id i64
4.9	3.0	1.4	0.2	”setosa”	1
4.7	3.2	1.3	0.2	”setosa”	1
4.6	3.1	1.5	0.2	”setosa”	1
5.0	3.6	1.4	0.2	”setosa”	1

```
fig = px.scatter(
    x=iris["sepal_width"].to_list(),
    y=iris["sepal_length"].to_list())
fig.show()
```

```
Unable to display output for mime type(s): application/vnd.plotly.v1+json, text/html
```

But wait! Maybe a polars series is “array-like” and can be used as input? Yes it can!

```
fig = px.scatter(
    x=iris["sepal_width"],
    y=iris["sepal_length"])
fig.show()
```

```
Unable to display output for mime type(s): application/vnd.plotly.v1+json, text/html
```

Can a polars frame be used as input? No it can not. The following will currently not work:

```
fig = px.scatter(
    data_frame=iris,
    x="sepal_width",
    y="sepal_length")
fig.show()
```

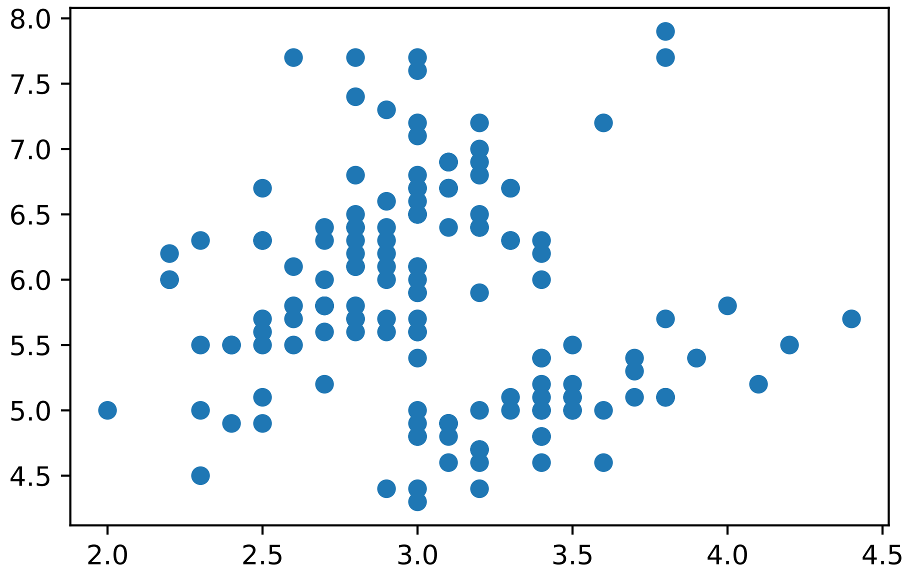
Matplotlib Functions

The above discussion applies to matplotlib functions as well; with the exception that matplotlib functions already support polars frames as input.

Inputing polars series:

```
fig, ax = plt.subplots()
ax.scatter(
    x=iris["sepal_width"],
    y=iris["sepal_length"])
```

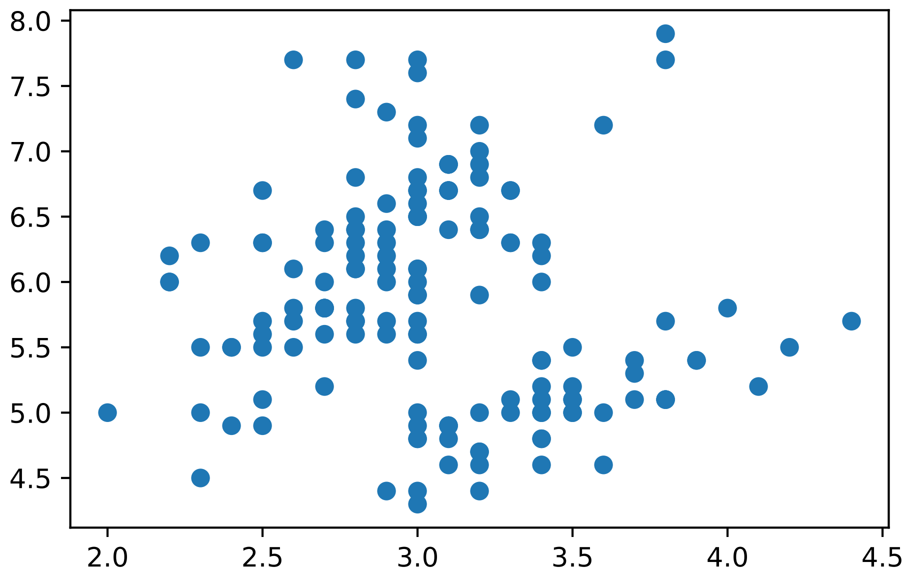
```
<matplotlib.collections.PathCollection at 0x290f88eb0>
```



Inputing polars frames:

```
fig, ax = plt.subplots()
ax.scatter(
    data=iris,
    x="sepal_width",
    y="sepal_length")
```

```
<matplotlib.collections.PathCollection at 0x296ce0370>
```

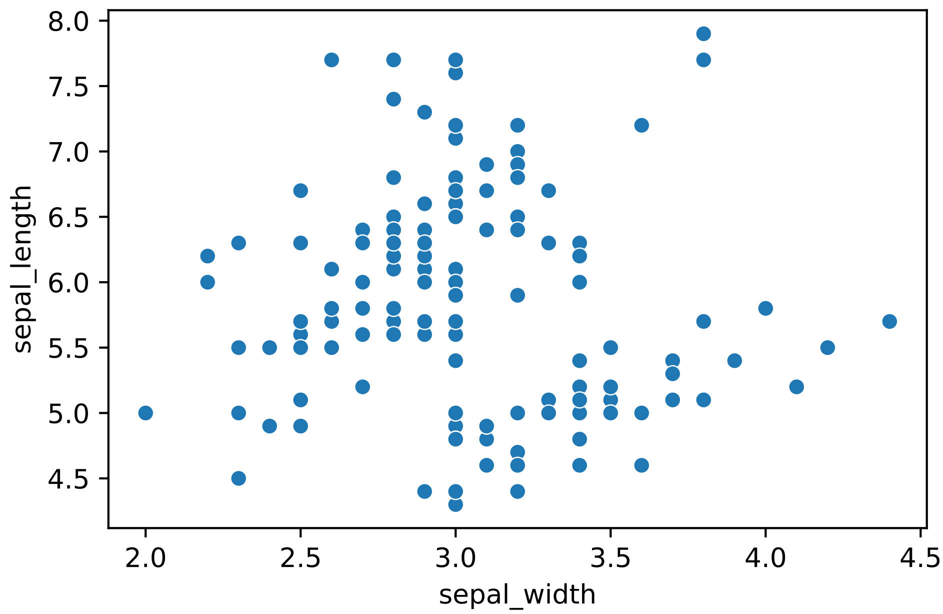


Seborn Functions

Because Seaborn uses a matplotlib backend, the above discussion applies to seaborn functions as well.

```
import seaborn as sns
sns.scatterplot(
    data=iris,
    x="sepal_width",
    y="sepal_length")
```

```
<AxesSubplot: xlabel='sepal_width', ylabel='sepal_length'>
```



Polars and ML

"How do to machine learning with polars?" is not a well defined question. ML can be done with many libraries, and the answer depends on the library you are using. One possibility is converting polars dataframes to a numpy arrays. This is very easy when dealing with numerical data. Converting `pl.Utf8` and `pl.Categorical` dtypes is a bit more involved, but still possible. For instance, by using `polars.DataFrame.to_dummies()`, `polars.get_dummies()`, or `polars.Series.to_dummies()`.

But wait! Isn't the conversion to numpy an expensive operation? Not terribly, but there is a better way. At the time of writing, ML libraries such as scikit-learn and xgboost, do not support polars dataframes as inputs. XGboost, however, does support arrow dataframes. This is great news since converting polars to arrow is just passing a pointer. See an example here.

Polars and Patsy

Patsy is a python library for describing statistical models (especially linear models and generalized linear models) and building design matrices.

```
import patsy as pt
#make a dataframe
data_pandas = pd.DataFrame(
```

```
    np.random.randn(100, 3),
    columns=["y", "x1", "x2"])
```

Use patsy to make a design matrix X, and a target vector y from a pandas dataframe.

```
formula = 'y ~ x1 + x2'
y, X = pt.dmatrices(formula, data_pandas)

X[:3]
```

```
array([[ 1.        ,  0.99734545,  0.2829785 ],
       [ 1.        , -0.57860025,  1.65143654],
       [ 1.        , -0.42891263,  1.26593626]])
```

```
y[:3]
```

```
array([[-1.0856306 ],
       [-1.50629471],
       [-2.42667924]])
```

Does the same work with polars? Yes!

```
data_polars= pl.DataFrame(data_pandas)
X, y = pt.dmatrices(formula, data_polars)
X[:3]
```

```
array([[-1.0856306 ],
       [-1.50629471],
       [-2.42667924]])
```

Effect Coding and Contrasts

There are many ways to encode categorical variables. For predictions, dummy coding is enough. If you want to discuss and infer on effect sizes, you may want to use other coding schemes.

One way to go about is to use the category_encoders library.

We start by making some categorical data.

```python
import string
import random
cat = pl.Series(
    name="cat",
    values=random.choices(
        population=string.ascii_letters[:5],
        k=data_polars.height)
    ).to_frame()
data_polars = data_polars.hstack(cat)
data_polars.head()
```

y	x1	x2	cat
f64	f64	f64	str
-1.085631	0.997345	0.282978	"c"
-1.506295	-0.5786	1.651437	"a"
-2.426679	-0.428913	1.265936	"e"
-0.86674	-0.678886	-0.094709	"a"
1.49139	-0.638902	-0.443982	"b"

The category encoders currently expects pandas dataframes as input, and does not support polars dataframes.

```python
import category_encoders as ce
encoder = ce.HelmertEncoder()
encoder.fit(data_polars.to_pandas())
```

```
/Users/johnros/workspace/polars_demo/.venv/lib/python3.9/site-packages/category_encoders/base

Intercept column might not be added anymore in future releases (c.f. issue #370)

HelmertEncoder(cols=['cat'],
               mapping=[{'col': 'cat',
                         'mapping':     cat_0  cat_1  cat_2  cat_3
 1   -1.0   -1.0   -1.0   -1.0
 2    1.0   -1.0   -1.0   -1.0
 3    0.0    2.0   -1.0   -1.0
 4    0.0    0.0    3.0   -1.0
 5    0.0    0.0    0.0    4.0
-1    0.0    0.0    0.0    0.0
-2    0.0    0.0    0.0    0.0}])
```