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"
```

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
```

```
letter1_pandas = letter1.to_pandas()
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.

Memory Footprint of Compute

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

```
%load_ext memory_profiler
```

```
%memit letter1.sort()
```

```
peak memory: 559.01 MiB, increment: 229.99 MiB
```

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

```
peak memory: 721.22 MiB, increment: 391.38 MiB
```

```
%memit letter1[10]='a'
```

```
peak memory: 494.54 MiB, increment: 78.50 MiB
```

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

```
peak memory: 416.71 MiB, increment: 0.00 MiB
```

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']))
```

```
773 ms ± 55.1 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()
```

```
270 ms ± 40.2 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.
```

```
198 ms ± 1.7 ms per loop (mean ± std. dev. of 2 runs, 2 loops each)
```

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

%timeit -n 2 -r 2 letter1_pandas.sort_values().loc[lambda x: x.isin([‘a’,‘b’,‘c’])]

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

::: {.cell execution_count=13}
``` {.python .cell-code}
%timeit -n 2 letter1_pandas.loc[lambda x: x.isin(['a','b','c'])].sort_values()

696 ms ± 18.7 ms per loop (mean ± std. dev. of 7 runs, 2 loops each)
```

:::

Pandas alternative syntax, just as slow.

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

```
699 ms ± 17.5 ms 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.
```

```python
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()
    end = time.time()
    times[key] = end-start

  return(times)
```

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

```python
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 wonder why?

Speed Of Import

Polar's `read_x` functions are quite faster than Pandas. This is due to better type "guessing" heuristics, and to native support of the parquet file format.

We now make synthetic data, save it as csv or parquet, and reimport it with polars and pandas.

Starting with CSV:

```python
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)
os.path.getsize('data/data.csv')
```

```
19632742
```

Import with pandas.

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

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

Import with polars.

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

```
16.1 ms ± 1.09 ms per loop (mean ± std. dev. of 2 runs, 2 loops each)
```

Trying parquet format:

```python
data_polars.write_parquet('data/data.parquet')
os.path.getsize('data/data.parquet')
```

```
7794866
```

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

```
21.7 ms ± 13 ms per loop (mean ± std. dev. of 2 runs, 2 loops each)
```

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

```
7.9 ms ± 854 µs per loop (mean ± std. dev. of 2 runs, 2 loops each)
```

Trying Feather format:

```
data_polars.write_ipc('data/data.feather')
os.path.getsize('data/data.feather')
```

```
8002191
```

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

```
135 µs ± 60.5 µs per loop (mean ± std. dev. of 2 runs, 2 loops each)
```

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

```
5.3 ms ± 1.03 ms per loop (mean ± std. dev. of 2 runs, 2 loops each)
```

```
import pickle
pickle.dump(data_polars, open('data/data.pickle', 'wb'))
os.path.getsize('data/data.pickle')
```

```
9001101
```

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

```
12.5 ms ± 508 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. Certainly when dealing with csv, but also with other formats.
- I dare argue that polars' type guessing is better, but I am not demonstrating it here.
- The difference in speed is quite large between csv vs. parquet and feather, with feather<parquet<csv.
- The fact that pickle isn't the fastest surprised me.
- Feather is the fastest, but larger on disk. Thus good for short-term storage, and parquet for long-term.

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',
```

```
242 ms ± 29.2 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
```

```
733 ms ± 103 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

```
%%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 2.7 s, sys: 1.12 s, total: 3.82 s
Wall time: 1.27 s
```

```
/home/johnros/workspace/practicing_python_2/venv/lib/python3.10/site-packages/IPython/core/fo

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

	tip_amount
passenger_count	
1.0	2.701872
2.0	2.749387
3.0	2.715053
4.0	2.782241

Polars

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

```
CPU times: user 913 ms, sys: 269 ms, total: 1.18 s
Wall time: 208 ms
```

```
PARTITIONED DS
```

shape: (4, 2)

passenger_count

tip_amount

f64

f64

3.0

2.715053

2.0

2.749387

1.0

2.701872

4.0

2.782241

```
q.show_graph()
```

Things to note:

- Polars is much faster.
- I only have 2 parquet files. When I run the same with more files, pandas will crash my python kernel.
- You will soon fall in love with the polars query syntax.
- 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. 1. A user guide. 1. A very active community on Discord. 1. The API reference. 1. A Stack-Overflow tag. 1. Cheat-sheet for pandas users.

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. In the following we:

1. Create a series and demonstrate basic operations on it.
2. Demonstrate the various dtypes.
3. Discuss missing values.
4. Filter a series.

Series Housekeeping

Construct a series

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

shape: (3,)

a

i64

1

2

3

Make pandas series for comparison:

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

```
type(s)
```

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

```python
type(s_pandas)
```

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

```python
s.dtype
```

```
Int64
```

```python
s_pandas.dtype
```

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

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

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

shape: (3,)

b

i64

1

2

3

```python
s.clone()
```

shape: (3,)

a

i64

1

2

3

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

shape: (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()`.

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

shape: (6,)

a

i64

1

2

3

4

9

10

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

shape: (5,)

a

i64

1

2

3

666

666

```
s.new_from_index()
```

```
s.rechunk()
```

shape: (3,)

a

i64

1

2

3

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

shape: (3,)

b

i64

1

2

3

```
s.to_dummies()
```

shape: (3, 3)

a_1

a_2

a_3

u8

u8

u8

1

0

0

0

1

0

0

0

1

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

```
/tmp/ipykernel_98877/844840155.py:1: DeprecationWarning:
`Series.cleared` has been renamed; this redirect is temporary, please use `.clear` instead
```

shape: (0,)

a

i64

Constructing a series of floats, for later use.

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

shape: (3,)

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]
```

```
1
```

Filtering with a Polars (Boolean) series will work:

```
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.internals.series.series.Series'>'.
```

Filtering with a pandas (Boolean) series will not work (why should it?)

```python
s[pd.Series([True, False, True])]
```

```
ValueError: Cannot __getitem__ on Series of dtype: 'Int64' with argument: '0     True
1    False
2     True
dtype: bool' of type: '<class 'pandas.core.series.Series'>'.
```

Filtering with a numpy (Boolean) array will work (but I can't see why you would want to):

```python
s[np.array([True, False, True])]
```

```
ValueError: Cannot __getitem__ on Series of dtype: 'Int64' with argument: '[ True False  Tru
```

Filtering with a Boolean list will not work:

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

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

BUT, 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
```

shape: (2,)

a

i64

1

3

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

shape: (2,)

a

i64

1

3

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

shape: (2,)

a

i64

1

2

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

shape: (2,)

a

i64

1

2

Negative indexing is not supported:

```python
s.head(-1)
s.limit(-1)
```

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

shape: (2,)

a

i64

2

3

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

shape: (2,)

a

i64

1

2

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

shape: (2,)

a

i64

1

3

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

shape: (2,)

a

i64

2

3

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

shape: (2,)

a

i64

1

3

Aggregations

```
s.sum()
```

```
6
```

```
s.min()
```

```
1
```

```
s.arg_min()
```

```
0
```

```
s.mean()
```

```
2.0
```

```
s.median()
```

```
2.0
```

```
s.entropy()
```

```
-4.68213122712422
```

```
s.describe()
```

shape: (6, 2)

statistic

value

str

f64

"min"

1.0

"max"

3.0

"null_count"

0.0

"mean"

2.0

"std"

1.0

"count"

3.0

```
s.value_counts()
```

shape: (3, 2)

a

counts

i64

u32

1

1

3

1

2

1

Object Transformations

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

shape: (2,)

a

list[i64]

[1, 2]

[3, 4]

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

shape: (3,)

a

i64

null

1

2

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

shape: (3,)

a

i64

2

3

null

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

shape: (3,)

a

i64

999

1

2

Mathematical Transformations

```
s.abs()
```

shape: (3,)

a

i64

1

2

3

```
s.sin()
```

shape: (3,)

a

f64

0.841471

0.909297

0.14112

```
s.exp()
```

shape: (3,)

a

f64

2.718282

7.389056

20.085537

```
s.hash()
```

shape: (3,)

a

u64

13321499719149775801

8196255364589999986

3071011010030224171

```
s.log()
```

shape: (3,)

a

f64

0.0

0.693147

1.098612

```
s.peak_max()
```

shape: (3,)

bool

false

false

true

```
s.sqrt()
```

shape: (3,)

a

f64

1.0

1.414214

1.732051

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

shape: (3,)

a

i64

1

2

2

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

shape: (3,)

a

i64

1

2

3

You cannot round integers, but you can round floats.

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

shape: (3,)

a

f64

1.0

2.0

3.0

```
f.ceil()
```

shape: (3,)

a

f64

1.0

2.0

3.0

```
f.floor()
```

shape: (3,)

a

f64

1.0

2.0

3.0

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

shape: (3,)

a

bool

true

false

false

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

shape: (3,)

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)
```

shape: (3,)

a

i64

2

3

4

Using your own functions comes with a performance cost:

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

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

```
233 µs ± 118 µs per loop (mean ± std. dev. of 2 runs, 2 loops each)
```

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

```
19.1 ms ± 2.07 ms per loop (mean ± std. dev. of 2 runs, 2 loops each)
```

Cummulative Operations

```
s.cummax()
```

shape: (3,)

a

i64

1

2

3

```
s.cumsum()
```

shape: (3,)

a

i64

1

3

6

```
s.cumprod()
```

shape: (3,)

a

i64

1

2

6

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

shape: (3,)

a

f64

1.0

1.75

2.615385

Sequential Operations

```
s.diff()
```

shape: (3,)

a

i64

null

1

1

```
s.pct_change()
```

shape: (3,)

a

f64

null

1.0

0.5

Windowed Operations

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

shape: (3,)

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
```

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

shape: (3,)

a

i64

null

2

3

```
s.clip(1, 2)
```

shape: (3,)

a

i64

1

2

2

```
s.clone()
```

shape: (3,)

a

i64

1

2

3

```
# check equality with clone
s == s.clone()
```

shape: (3,)

a

bool

true

true

true

Booleans

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

```
Boolean
```

```
b.all()
```

```
False
```

```
b.any()
```

```
True
```

Uniques and Duplicates

```
s.is_duplicated()
```

shape: (3,)

a

bool

false

false

false

```
s.is_unique()
```

shape: (3,)

a

bool

true

true

true

```
s.n_unique()
```

```
3
```

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

shape: (4,)

u32

2

1

1

1

The first appearance of a value in a series:

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

shape: (5,)

bool

true

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_datelike()
```

```
/tmp/ipykernel_98877/830626930.py:1: DeprecationWarning:

`Series.is_datelike` has been renamed; this redirect is temporary, please use `.is_temporal`

False
```

Compare with Pandas Type Checkers:

```
pd.api.types.is_string_dtype(s_pandas)
```

```
False
```

```
pd.api.types.is_string_dtype(s)
```

```
False
```

Casting

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

shape: (3,)

a

i32

1

2

3

Things to note:

- `s.cast()` is an in place operation. If you want to keep the original series, you can use `s.cast(pl.Int32).clone()`.
- `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()
```

shape: (3,)

a

i8

1

2

3

Also see here.

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

```
s.shrink_to_fit()
```

shape: (3,)

a

i64

1

2

3

Ordering and Sorting

```
s.sort()
```

shape: (3,)

a

i64

1

2

3

```
s.reverse()
```

shape: (3,)

a

i64

3

2

1

```
s.rank()
```

shape: (3,)

a

f32

1.0

2.0

3.0

```
s.arg_sort()
```

shape: (3,)

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()]
```

shape: (3,)

a

bool

true

true

true

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

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

shape: (3,)

a

bool

true

true

true

```
s.shuffle(seed=1)
```

shape: (3,)

a

i64

2

1

3

Missing

Pandas users will be excited to know that 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 performance and syntax.

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

shape: (4,)

a

bool

false

false

true

false

```
m.is_nan()
```

shape: (4,)

a

bool

false

false

null

true

```
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
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:

- The use of `is_null()` instead of pandas `isna()`.
- Polars supports `np.nan` but that is a different dtype than `None` (which is a `Null` type). `None` is not considered
- 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)
```

shape: (3,)

a

i64

1

0

2

```
m1.interpolate()
```

shape: (3,)

a

i64

1

1

2

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

shape: (3,)

a

f64

1.0

NaN

2.0

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

shape: (3,)

a

f64

1.0

0.0

2.0

```
m1.drop_nulls()
```

shape: (2,)

a

i64

1

2

```
m1.drop_nans()
```

shape: (3,)

a

i64

1

null

2

```
m2.drop_nulls()
```

shape: (3,)

a

f64

1.0

NaN

2.0

Export

```
s.to_frame()
```

shape: (3, 1)

a

i64

1

2

3

```
s.to_list()
```

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

```
s.to_numpy()
```

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

```
s.to_pandas()
```

```
/home/johnros/workspace/practicing_python_2/venv/lib/python3.10/site-packages/IPython/core/fo

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

	a
0	1
1	2
2	3

```
s.to_arrow()
```

```
<pyarrow.lib.Int64Array object at 0x7f067c0fbb20>
[
  1,
  2,
  3
]
```

Strings

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

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

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

shape: (3,)

a

u32

3

3

3

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

shape: (3,)

a

u32

3

3

3

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

shape: (1,)

a

str

"foo-bar-baz"

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

shape: (3,)

a

bool

true

true

false

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

shape: (3,)

a

u32

2

0

0

Count pattern matches. Notice the `r"<regex pattern>"` syntax for regex (more about it here).

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

shape: (3,)

a

u32

3

3

3

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

shape: (3,)

a

bool

true

false

false

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

shape: (3,)

a

bool

true

false

false

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

```
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"
            ])

url.str.extract(r"=(\w+)", 1)
```

shape: (3,)

a

str

"messi"

"jorginho"

"ronaldo"

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

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

shape: (3,)

a

list[str]

["=messi", "=polars"]

["=jorginho", "=polars"]

["=ronaldo", "=polars"]

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

shape: (3,)

a

str

"foo*****"

"bar*****"

"baz*****"

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

shape: (3,)

a

str

"*****foo"

"*****bar"

"*****baz"

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

shape: (3,)

a

str

"oo"

"bar"

"baz"

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

shape: (3,)

a

str

"foo"

"ba"

"baz"

Replacing first appearance of a pattern:

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

shape: (3,)

a

str

"fZZo"

"bar"

"baz"

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

shape: (3,)

a

str

"fZZ"

"bar"

"baz"

Replace all appearances of a pattern:

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

shape: (3,)

a

str

"fZZZZ"

"bar"

"baz"

String to list of strings. Number of spits inferred.

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

shape: (3,)

a

list[str]

["f", "", ""]

["bar"]

["baz"]

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

shape: (3,)

a

list[str]

["foo"]

["ba", "r"]

["ba", "z"]

String to dict of strings. Number of splits fixed.

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

shape: (3,)

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)
```

shape: (3,)

a

struct[4]

{"foo",null,null,null}

{"b","r",null,null}

{"b","z",null,null}

Strip white spaces.

```
st.str.rjust(8, " ").str.strip()
```

shape: (3,)

a

str

"foo"

"bar"

"baz"

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

shape: (3,)

a

str

"FOO"

"BAR"

"BAZ"

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

shape: (3,)

a

str

"foo"

"bar"

"baz"

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

shape: (3,)

a

str

"00foo"

"00bar"

"00baz"

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

shape: (3,)

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. Duration: As the name suggests. Similar t o `timedelta` in pandas. Generated with `pl.Duration()`.
4. Time: Hour of day. Generated with `pl.Time()`.

Converting from Strings

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

shape: (4,)

date

date

2021-04-22

null

null

null

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

shape: (4,)

date

date

null

2022-01-04

null

null

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

shape: (4,)

date

date

null

null

2022-01-31

null

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
```

shape: (3,)

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(tu='us', tz=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")
```

shape: (3,)

datetime[ms]

2001-02-02 00:00:00

2001-02-02 08:28:20

2001-02-02 16:56:40

From Date to String

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

shape: (3,)

str

"2001-02-02"

"2001-02-02"

"2001-02-02"

Ecxtract Time Sub-Units

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

shape: (3,)

u32

0

20

40

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

shape: (3,)

u32

0

28

56

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

shape: (3,)

u32

0

8

16

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

shape: (3,)

u32

2

2

2

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

shape: (3,)

u32

5

5

5

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

shape: (3,)

u32

5

5

5

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

shape: (3,)

u32

2

2

2

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

shape: (3,)

i32

2001

2001

2001

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

shape: (3,)

u32

33

33

33

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

shape: (3,)

u32

1

1

1

Durations

Equivalent to Pandas `period` dtype.

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

shape: (3,)

duration[s]

null

8h 28m 20s

8h 28m 20s

```
diffs.dtype
```

```
Duration(tu='us')
```

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

shape: (3,)

i64

null

30500

30500

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

shape: (3,)

i64

null

508

508

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

shape: (3,)

i64

null

0

0

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

shape: (3,)

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)
```

I have no idea what is an "average date", but it can be computed.

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

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

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

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

Data Transformations

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

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

shape: (3,)

datetime[s]

2002-02-22 00:02:00

2002-02-22 08:30:20

2002-02-22 16:58:40

Nagative offset is also allowed.

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

shape: (3,)

datetime[s]

2000-01-12 23:58:00

2000-01-13 08:26:20

2000-01-13 16:54:40

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

shape: (3,)

datetime[s]

2001-01-01 00:00:00

2001-01-01 00:00:00

2001-01-01 00:00:00

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

```
/home/johnros/workspace/practicing_python_2/venv/lib/python3.10/site-packages/IPython/core/fo

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

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

Comparing Series

```python
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 duplicat 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
```

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”

```
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)
```

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

```
3
```

```python
df.width
```

```
4
```

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

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

```python
df.with_row_count()
```

shape: (3, 5)

row_nr

integer

date

float

string

u32

i64

datetime[s]

f64

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_column(pl.Series("new", [1, 2, 3]))
```

```
/tmp/ipykernel_98877/324654970.py:1: DeprecationWarning:

`DataFrame.with_column` has been renamed; this redirect is temporary, please use `.with_colur
```

shape: (3, 5)

integer

date

float

string

new

i64

datetime[s]

f64

str

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])])
```

shape: (3, 6)

integer

date

float

string

new1

new2

i64

datetime[s]

f64

str

i64

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
```

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"

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]))
```

shape: (3, 5)

integer

new

date

float

string

i64

i64

datetime[s]

f64

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]))
```

shape: (3, 5)

new2

new

date

float

string

i64

i64

datetime[s]

f64

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]))
```

shape: (3, 5)

new2

new

date

float

string

i64

i64

datetime[s]

f64

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"

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

```
/tmp/ipykernel_98877/3455956199.py:2: DeprecationWarning:

`DataFrame.with_column` has been renamed; this redirect is temporary, please use `.with_colum
```

shape: (3, 5)

integer

date

float

string

new

i64

datetime[s]

f64

str

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

```
df.is_empty()
```

```
False
```

```
df.cleared() # make empty copy
```

```
/tmp/ipykernel_98877/3487364701.py:1: DeprecationWarning:

`DataFrame.cleared` has been renamed; this redirect is temporary, please use `.clear` instead
```

shape: (0, 4)

integer

date

float

string

i64

datetime[s]

f64

str

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

```
/tmp/ipykernel_98877/3664706437.py:1: DeprecationWarning:

`DataFrame.cleared` has been renamed; this redirect is temporary, please use `.clear` instead

True
```

Renaming columns can be done with `rename()`. Later, we will see it may also be done with an `alias()` statement withing a `with_columns()` context.

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

shape: (3, 4)

integer2

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"

Convert to Other Python Objects

To Pandas

```
df.to_pandas()
```

```
/home/johnros/workspace/practicing_python_2/venv/lib/python3.10/site-packages/IPython/core/fo

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

	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 Python List

```
df.
```

```
SyntaxError: invalid syntax (791285630.py, line 1)
```

To Python Dict

```
df.to_dict()
```

```
{'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"
 ]}
```

To Python Tuple

Dataframe in Memory

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

```
9.965151548385662e-08
```

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

```
1
```

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

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"

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

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”

Statistical Aggregations

```
df.describe()
```

shape: (7, 5)

describe

integer

date

float

string

str

f64

str

f64

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

Compare to pandas:

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

```
/home/johnros/workspace/practicing_python_2/venv/lib/python3.10/site-packages/IPython/core/fo

In future versions `DataFrame.to_latex` is expected to utilise the base implementation of `St
```

	integer	float
count	3.0	3.0
mean	2.0	5.0
std	1.0	1.0
min	1.0	4.0
25%	1.5	4.5
50%	2.0	5.0
75%	2.5	5.5
max	3.0	6.0

Things to note:

- Polas will summarize all columns, even if they are not numeric.
- The statistics returned are different.
- In the following, Polars will always return a frame with the same number of columns as the original frame; pandas would have returned columns only where the operation is defined, and omit NAs.

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

```
df.max()
```

shape: (1, 4)

integer

date

float

string

i64

datetime[s]

f64

str

3

2022-01-03 00:00:00

6.0

"c"

```
df.min()
```

shape: (1, 4)

integer

date

float

string

i64

datetime[s]

f64

str

1

2022-01-01 00:00:00

4.0

"a"

```
df.mean()
```

shape: (1, 4)

integer

date

float

string

f64

datetime[s]

f64

str

2.0

null

5.0

null

```
df.median()
```

shape: (1, 4)

integer

date

float

string

f64

datetime[s]

f64

str

2.0

null

5.0

null

```
df.sum()
```

shape: (1, 4)

integer

date

float

string

i64

datetime[s]

f64

str

6

null

15.0

null

```
df.std()
```

shape: (1, 4)

integer

date

float

string

f64

datetime[s]

f64

str

1.0

null

1.0

null

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

shape: (1, 4)

integer

date

float

string

f64

datetime[s]

f64

str

1.0

null

4.0

null

Exctraction

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 with pandas.
2. Filtering and selection is possible with the `[` operator, or the `filter()` and `select()` methods. The latter is recommended to facilitate lazy evaluation (discussed later).

Single cell extraction.

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

```
1
```

Slicing along rows.

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

shape: (1, 4)

integer

date

float

string

i64

datetime[s]

f64

str

1

2022-01-01 00:00:00

4.0

"a"

Slicing along columns.

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

shape: (3, 1)

integer

i64

1

2

3

Filtering Rows

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

shape: (2, 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"

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

shape: (2, 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"

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

shape: (1, 4)

integer

date

float

string

i64

datetime[s]

f64

str

3

2022-01-03 00:00:00

6.0

"c"

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

shape: (2, 4)

integer

date

float

string

i64

datetime[s]

f64

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)
```

shape: (1, 4)

integer

date

float

string

i64

datetime[s]

f64

str

2

2022-01-02 00:00:00

5.0

"b"

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

shape: (1, 4)

integer

date

float

string

i64

datetime[s]

f64

str

1

2022-01-01 00:00:00

4.0

"a"

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

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

```
df.rows() # all 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')]
```

Row filtering by label

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

shape: (1, 4)

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 Columns

Column selection by label

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

shape: (3, 1)

integer

i64

1

2

3

Multiple column selection by label

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

shape: (3, 2)

integer

float

i64

f64

1

4.0

2

5.0

3

6.0

As of polars>15.., you don't have to pass a list:

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

shape: (3, 2)

integer

float

i64

f64

1

4.0

2

5.0

3

6.0

Column slicing by label

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

shape: (3, 3)

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: Slicing with `df.select()` does not support ranges such as `df.select("integer":"float")`; only lists of column names.

Get a column as a 1D polars frame.

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

shape: (3,)

integer

i64

1

2

3

Get a column as a polars series.

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

shape: (3,)

integer

i64

1

2

3

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

```
2
```

```
df.get_columns() # get a list of 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.drop("integer")
```

shape: (3, 3)

date

float

string

datetime[s]

f64

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"

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

Select along dtype

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

shape: (3, 1)

integer

i64

1

2

3

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

shape: (3, 1)

float

f64

4.0

5.0

6.0

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

shape: (3, 1)

string

str

"a"

"b"

"c"

Things to note:

- The `pl.col()` function will be very useful for referencing columns in a dataframe. 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`.

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()
```

shape: (3,)

bool

true

true

true

```
df.is_duplicated()
```

shape: (3,)

bool

false

false

false

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

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"

```
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()
```

shape: (1, 5)

integer

date

float

string

missing

u32

u32

u32

u32

u32

0

0

0

0

1

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

shape: (2, 5)

integer

date

float

string

missing

i64

datetime[s]

f64

str

f64

1

2022-01-01 00:00:00

4.0

"a"

3.0

3

2022-01-03 00:00:00

6.0

"c"

NaN

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

shape: (3, 5)

integer

date

float

string

missing

i64

datetime[s]

f64

str

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)
```

shape: (3, 5)

integer

date

float

string

missing

i64

datetime[s]

f64

str

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()
```

shape: (3, 5)

integer

date

float

string

missing

i64

datetime[s]

f64

str

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()`.
- 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
```

```
)
```

shape: (3, 6)

integer

date

float

string

integer2

integer3

i64

datetime[s]

f64

str

i64

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
)
```

shape: (3, 4)

integer

date

float

string

i64

datetime[s]

f64

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)
)
```

shape: (3, 4)

integer

date

float

string

str

str

str

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 labda function.

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

shape: (3, 1)

apply

f64

5.0

7.0

9.0

But wait- using your own functions may have a very serious toll on performance:

```
df_big = pl.DataFrame(np.random.randn(1000000, 2), columns=["a", "b"])
%timeit -n2 -r2 df_big.sum(axis=1)
```

```
4.99 ms ± 102 µs per loop (mean ± std. dev. of 2 runs, 2 loops each)
```

```
/tmp/ipykernel_98877/3759046676.py:1: DeprecationWarning:

`columns` is deprecated as an argument to `__init__`; use `schema` instead.
```

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

```
370 ms ± 28.3 ms per loop (mean ± std. dev. of 2 runs, 2 loops each)
```

PS- How would numpy and pandas deal with this row-wise summation? Numpy would be fastest, and pandas lag considerably behind.

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

shape: (3, 4)

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')
```

shape: (3, 4)

integer

date

float

string

str

str

str

str

"WOW"

"WOW"

"WOW"

"WOW"

"1"

"2022-01-01 00:...

"4.0"

"a"

"2"

"2022-01-02 00:...

"5.0"

"b"

Sorting

```
df.sort("integer")
```

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"

```
df.reverse()
```

shape: (3, 4)

integer

date

float

string

i64

datetime[s]

f64

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.
- `pl.concat()`, which is similar to the previous to, but with memory re-chunking.
- `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])
```

shape: (3, 5)

integer

date

float

string

c

i64

datetime[s]

f64

str

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)
```

shape: (6, 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"

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)
```

shape: (6, 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"

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')
```

shape: (3, 5)

integer

date

float

string

c

i64

datetime[s]

f64

str

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(
```

shape: (6, 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"

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.
```

shape: (9, 4)

integer

date

float

string

i64

datetime[s]

f64

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_column(
    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')
```

shape: (9, 5)

integer

date

float

string

c

i64

datetime[s]

f64

str

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

null

null

null

null

1

null

null

null

null

1

join

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

shape: (6, 7)

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 the Section Section above.

join_asof

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

shape: (6, 6)

integer

date

float

string

float_right

string_right

i64

datetime[s]

f64

str

f64

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

```
df.transpose()
```

shape: (4, 6)

column_0

column_1

column_2

column_3

column_4

column_5

str

str

str

str

str

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 00:...

”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)
```

shape: (2, 4)

famid

birth

ht1

ht2

str

i64

f64

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)
```

shape: (5, 4)

famid

birth

treatment

height

str

i64

str

f64

"2"

1

”ht2”

3.2

”3”

3

”ht2”

2.9

”2”

2

”ht2”

2.8

”3”

1

”ht2”

3.3

”3”

2

”ht2”

3.4

Break strings into rows.

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

shape: (5, 4)

famid

birth

ht1

ht2

str

i64

f64

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)
```

shape: (5, 3)

id

treatment

height

i64

str

f64

1

"A"

2.8

1

"A"

2.9

1

"B"

2.2

2

"A"

2.0

2

"A"

1.8

```
long.pivot(
  index='id',
  columns='treatment',
  values='height')
```

shape: (3, 3)

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
```

shape: (2, 15)

id_0

id_1

id_2

id_3

id_4

treatment_0

treatment_1

treatment_2

treatment_3

treatment_4

height_0

height_1

height_2

height_3

height_4

i64

i64

i64

i64

i64

str

str

str

str

str

f64

f64

f64

f64

f64

1

1

2

3

3

"A"

"B"

"A"

"A"

"B"

2.8

2.2

1.8

2.2

2.1

1

2

2

3

null

"A"

"A"

"B"

"A"

null

2.9

2.0

1.9

2.3

null

Groupby

```
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)),
```

```python
        (datetime(2022, 1, 9)),
        (datetime(2022, 1, 9)),
        (datetime(2022, 1, 9))],
})
```

```python
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
                             ]
```

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

shape: (3, 2)

integer

count

i64

u32

3

2

2

2

1

2

```
groupper.sum()
```

shape: (3, 4)

integer

float

string

datetime

i64

f64

str

datetime[s]

1

3.0

null

2074-01-07 00:00:00

2

7.0

null

2074-01-12 00:00:00

3

11.0

null

2074-01-17 00:00:00

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

```
(
  df2
  .groupby_dynamic(index_column="datetime", every="1d")
  .agg(pl.col("float").sum())
)
```

shape: (2, 2)

datetime

float

datetime[s]

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())
)
```

shape: (6, 2)

datetime

float

datetime[s]

f64

2022-01-04 00:00:00

1.0

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

```
/tmp/ipykernel_98877/13300274.py:1: DeprecationWarning:
```

```
`DataFrame.with_column` has been renamed; this redirect is temporary, please use `.with_colu
```

shape: (5, 5)

integer

date

float

string

sum

i64

datetime[s]

f64

str

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 computing expression. The following will not fail, but return the wrong result:

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

```
/tmp/ipykernel_98877/1359718742.py:1: DeprecationWarning:

`DataFrame.with_column` has been renamed; this redirect is temporary, please use `.with_colu
```

shape: (5, 5)

integer

date

float

string

sum

i64

datetime[s]

f64

str

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

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

What if you want to access a column from frame `df`, when processing frame `df2`? This is what `df.with_context()` will do.

```
q = (
    df.lazy()
    .with_context( # add colums of df2 to the search space
        df2.select(pl.all().suffix("_2")).lazy()
        )
    .select(
        (pl.col("float") + pl.col("float_2")).alias('sum') # sum from the two frames
```

```
        )
    )

q.collect()
```

shape: (6, 1)

sum

f64

5.0

7.0

9.0

11.0

13.0

15.0

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.
- 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 0x7F06B67022F0>
```

```
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 subset of rows, marked in the graph with a σ.
  - A *projection* is 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()`:

```
q.collect()
```

shape: (2, 1)

integer

i64

2

3

```
q.describe_plan()
```

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

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

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

shape: (2, 1)

integer

i64

2

3

Inspecting, Profiling, and Debugging a Query

You can inspect the data at any point in the query. `df.inspect()` will print the output of a single node in the query graph:

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

```
shape: (2, 2)

 integer   float
 ---       ---
 i64       f64
```

```
 2          8.0
 3          9.0
```

shape: (2, 1)

integer

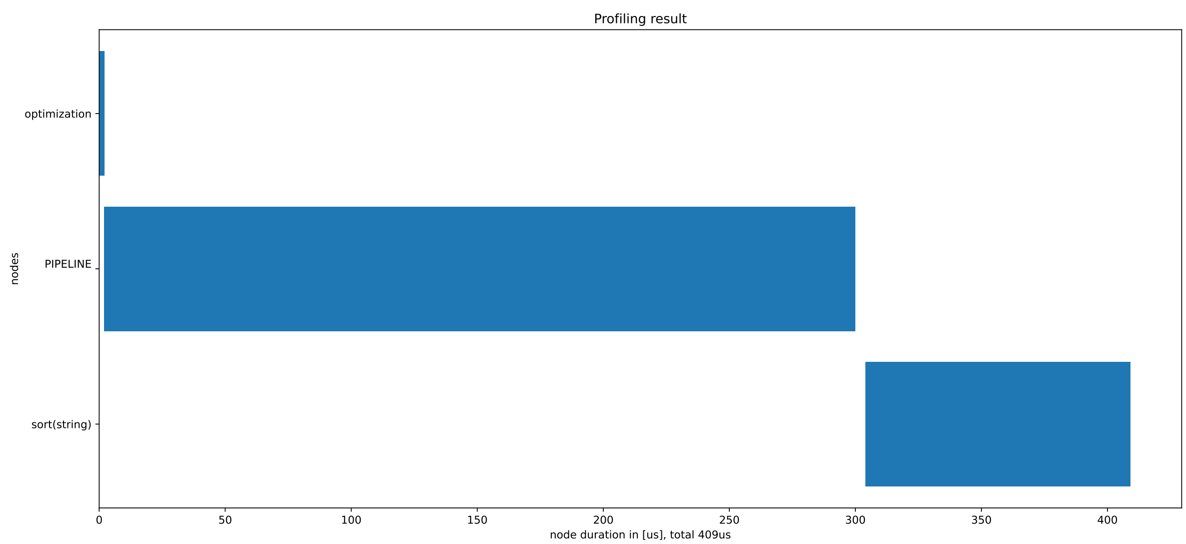
i64

2

3

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

```
(
    df_lazy
    .groupby('string')
    .agg(pl.col('float').sum())
    .sort('string')
    .profile(show_plot=True)
)
```




```
(shape: (4, 2)

 string   float
```

```
  ---     ---
  str     f64

  a       4.0
  b       5.0
  c       6.0
  d       24.0
          ,
 shape: (3, 3)

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

  optimization  0       2
  PIPELINE      2       300

  sort(string)  304     409
                )
```

Exporting a Query

You can export your query, as a JSON file.

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

This is how the query will look on disk:

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


```
{'Sort': {'input': {'Projection': {'expr': [{'Column': 'integer'}],
    'input': {'Selection': {'input': {'Selection': {'input': {'Selection': {'input': {'DataF
              'datatype': 'Int64',
              'values': [1, 2, 3, 1, 2, 3]},
             {'name': 'date',
              'datatype': {'Datetime': ['Microseconds', None]},
              'values': [1640995200000000,
               1641081600000000,
               1641168000000000,
               1641254400000000,
```

```
                1641340800000000,
                1641427200000000]},
              {'name': 'float',
               'datatype': 'Float64',
               'values': [4.0, 5.0, 6.0, 7.0, 8.0, 9.0]},
              {'name': 'string',
               'datatype': 'Utf8',
               'values': ['a', 'b', 'c', 'd', 'd', 'd']}]},
            'schema': {'inner': {'integer': 'Int64',
              'date': {'Datetime': ['Microseconds', None]},
              'float': 'Float64',
              'string': 'Utf8'}},
            'output_schema': None,
            'projection': None,
            'selection': None}},
          'predicate': {'BinaryExpr': {'left': {'Column': 'float'},
            'op': 'Gt',
            'right': {'Literal': {'Float64': 2.0}}}}}},
        'predicate': {'BinaryExpr': {'left': {'Column': 'float'},
          'op': 'Gt',
          'right': {'Literal': {'Float64': 3.0}}}}}},
      'predicate': {'BinaryExpr': {'left': {'Column': 'float'},
        'op': 'Gt',
        'right': {'Literal': {'Float64': 7.0}}}}}},
    'schema': {'inner': {'integer': 'Int64'}}}},
  'by_column': [{'Column': 'integer'}],
  'args': {'descending': [False], 'nulls_last': False, 'slice': None}}}
```


You can now load it and run it.

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

shape: (2, 1)

integer

i64

2

3

SQL Flavor

If you are a hardcore SQL user, you may want to use the SQL flavor of polars.

```
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 'sql' 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

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")
```

shape: (6, 4)

integer

date

float

string

i64

str

f64

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"

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()
```

shape: (2, 1)

integer

i64

2

3

Things to note:

- From the graph we see that the filtering (

$$\sigma$$

) is done at scan time, and not after the materialization of the data.
- 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): A portable columnar file format that is optimized for storing data in a fast and efficient manner, utilizing the Arrow IPC format internally.
- 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.
- Avro: A binary row-based format.

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

Not suported formats: - Feather (V1). - HDF5: Currently not supported.

From Multiple Files in The 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 {#sec- multiple_files}

Without any format-specific utilities, you can always read 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
```

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

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

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()`. Which one to use 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. The latter will look like this:
- As discussed in the Section Section above.

Here is an example of a full import:

```python
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
```

shape: (3, 2)

passenger_count

tip_amount

f64

f64

1.0

2.400686

0.0

2.273948

2.0

2.579188

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
```

shape: (3, 2)

passenger_count

tip_amount

f64

f64

1.0

2.400686

0.0

2.273948

2.0

2.579188

Apache Arrow Dataset

An Apache Arrow dataset is a collection of parquet files, with a common schema. 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=3', 'integer=1', 'integer=2']
```

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

```
[['24255c3bcf2048c785b93fd5ef1bd53e-0.parquet'],
 ['24255c3bcf2048c785b93fd5ef1bd53e-0.parquet'],
 ['24255c3bcf2048c785b93fd5ef1bd53e-0.parquet']]
```

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

shape: (6, 3)

date

float

string

datetime[s]

f64

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.
- 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). I expect this to change in the near future.

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.

See here for working in serverless environments.

Reading from a Database

See here.

Export

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 approaches to plotting in python:

1. The object oriented, where a dataframe has a plotting method. 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. 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 numpy 1D arrays as inputs, can take a polars series after a zero copy conversion with `pl.Series.to_numpy()`. Some plotting functions will not even require the conversion to numpy (and will handle it internally).

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.

Interesting: How do I know if a python function can take a polars frame as input?

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()
```

shape: (5, 6)

sepal_length

sepal_width

petal_length

petal_width

species

species_id

f64

f64

f64

f64

str

i64

5.1

3.5

1.4

0.2

”setosa”

1

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

Plotly’s `scatter()` can take x and y as str, int or Series or array-like. The following will naturally work:

```
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 not work:

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

Matplotlib Functions

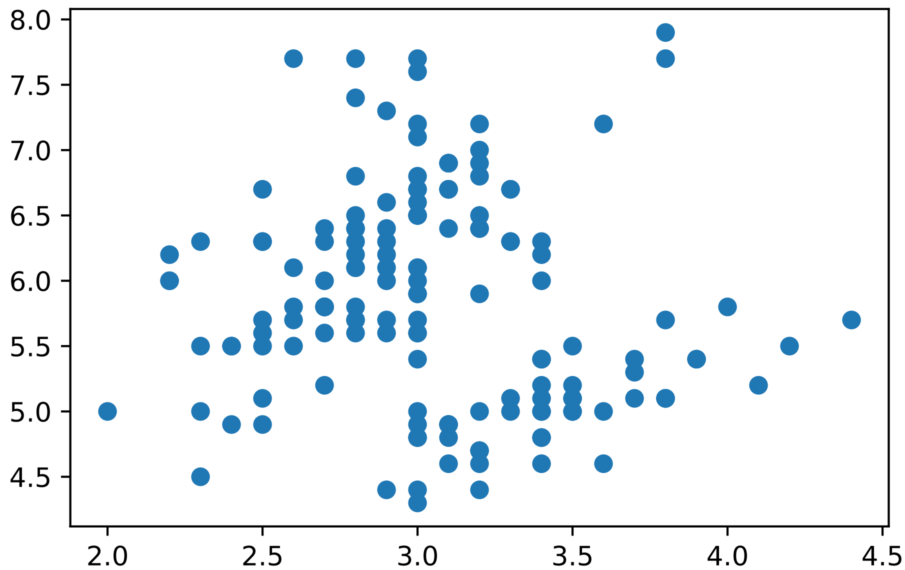
The above discussion applies to matplotlib functions as well; with the exception that matplotlib functions already support polars frames as input.

Inputing polars series:

```
fig, ax = plt.subplots()
ax.scatter(
```

```
    x=iris["sepal_width"],
    y=iris["sepal_length"])
```

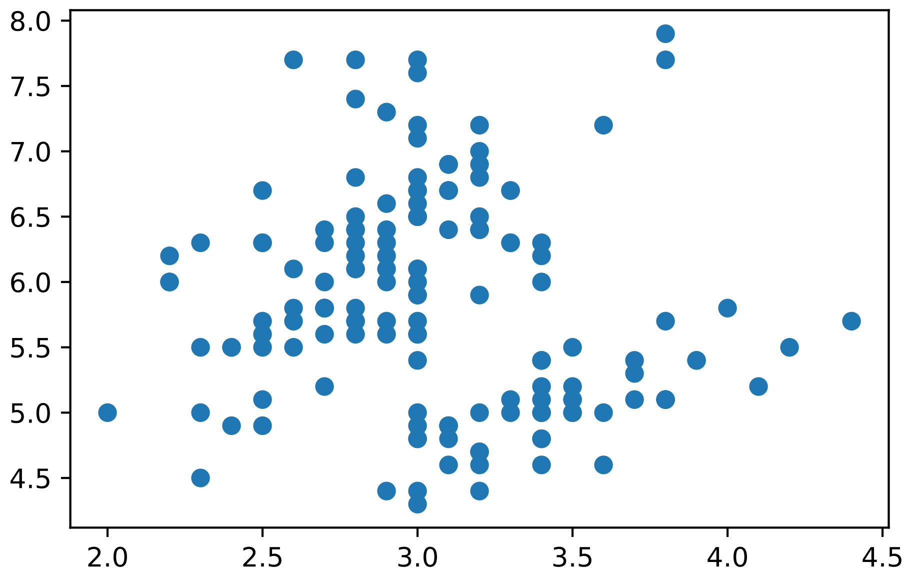
```
<matplotlib.collections.PathCollection at 0x7f06387e6800>
```



Inputing polars frames:

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

```
<matplotlib.collections.PathCollection at 0x7f0552377070>
```

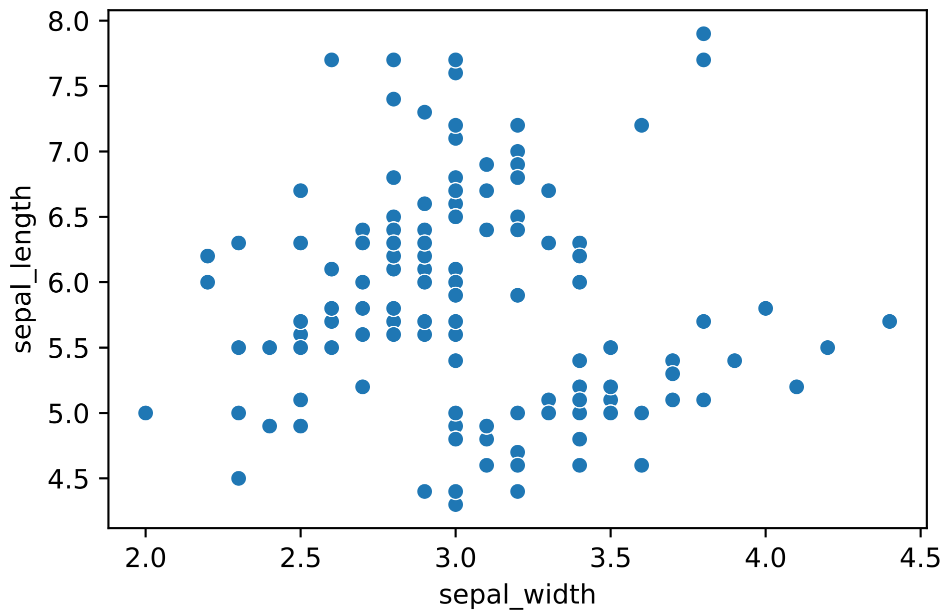


Seborn Functions

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.

```
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()
```

shape: (5, 4)

y

x1

x2

cat

f64

f64

f64

str

-1.085631

0.997345

0.282978

"c"

-1.506295

-0.5786

1.651437

”c”

-2.426679

-0.428913

1.265936

”b”

-0.86674

-0.678886

-0.094709

”d”

1.49139

-0.638902

-0.443982

”d”

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

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

```
/home/johnros/workspace/practicing_python_2/venv/lib/python3.10/site-packages/category_encode

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}])
```

Config

```
list(dir(pl.Config))
```

```
['__class__',
 '__delattr__',
 '__dict__',
 '__dir__',
 '__doc__',
 '__enter__',
 '__eq__',
 '__exit__',
 '__format__',
 '__ge__',
 '__getattribute__',
 '__gt__',
 '__hash__',
 '__init__',
 '__init_subclass__',
 '__le__',
 '__lt__',
 '__module__',
 '__ne__',
 '__new__',
 '__reduce__',
 '__reduce_ex__',
 '__repr__',
 '__setattr__',
 '__sizeof__',
 '__str__',
 '__subclasshook__',
 '__weakref__',
 'load',
 'restore_defaults',
 'save',
 'set_ascii_tables',
 'set_auto_structify',
 'set_fmt_float',
 'set_fmt_str_lengths',
 'set_tbl_cell_alignment',
 'set_tbl_cols',
```

```
'set_tbl_column_data_type_inline',
'set_tbl_dataframe_shape_below',
'set_tbl_formatting',
'set_tbl_hide_column_data_types',
'set_tbl_hide_column_names',
'set_tbl_hide_dataframe_shape',
'set_tbl_hide_dtype_separator',
'set_tbl_rows',
'set_tbl_width_chars',
'set_verbose',
'state']
```