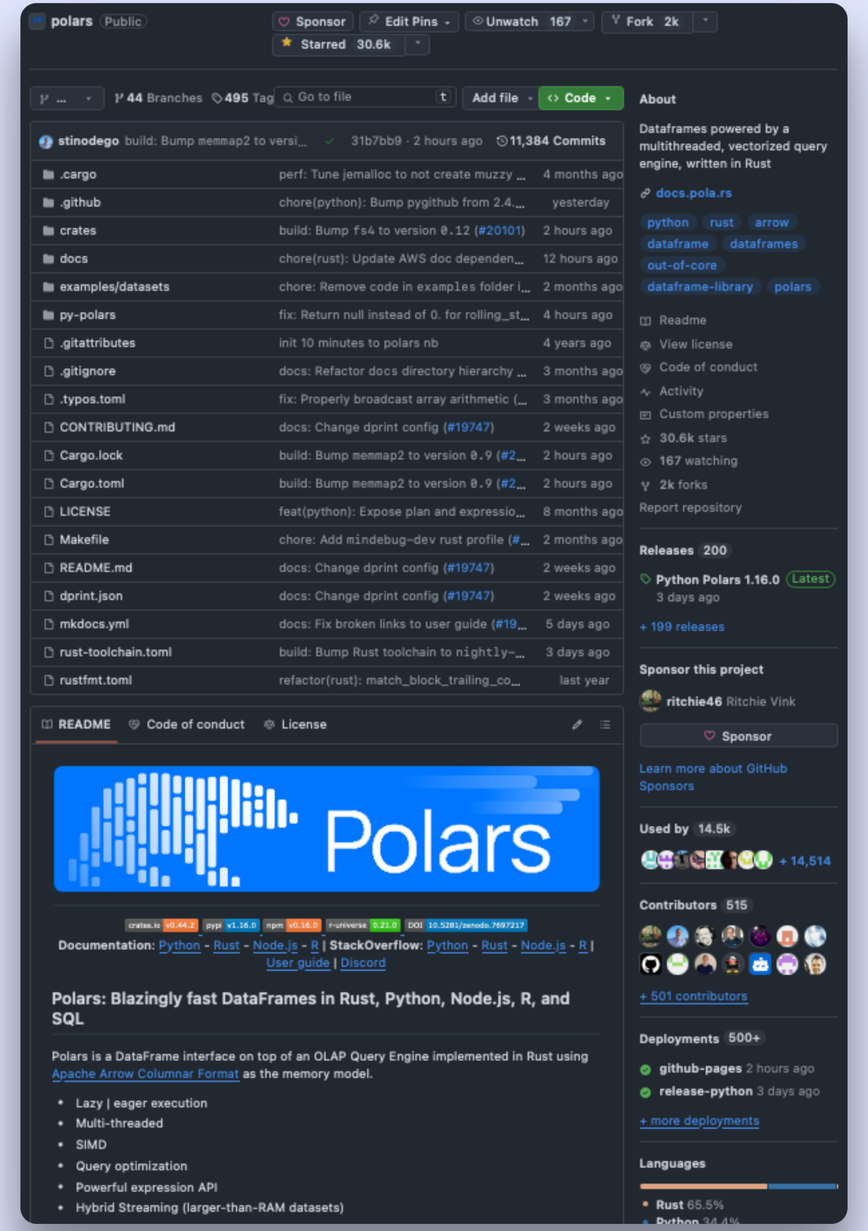
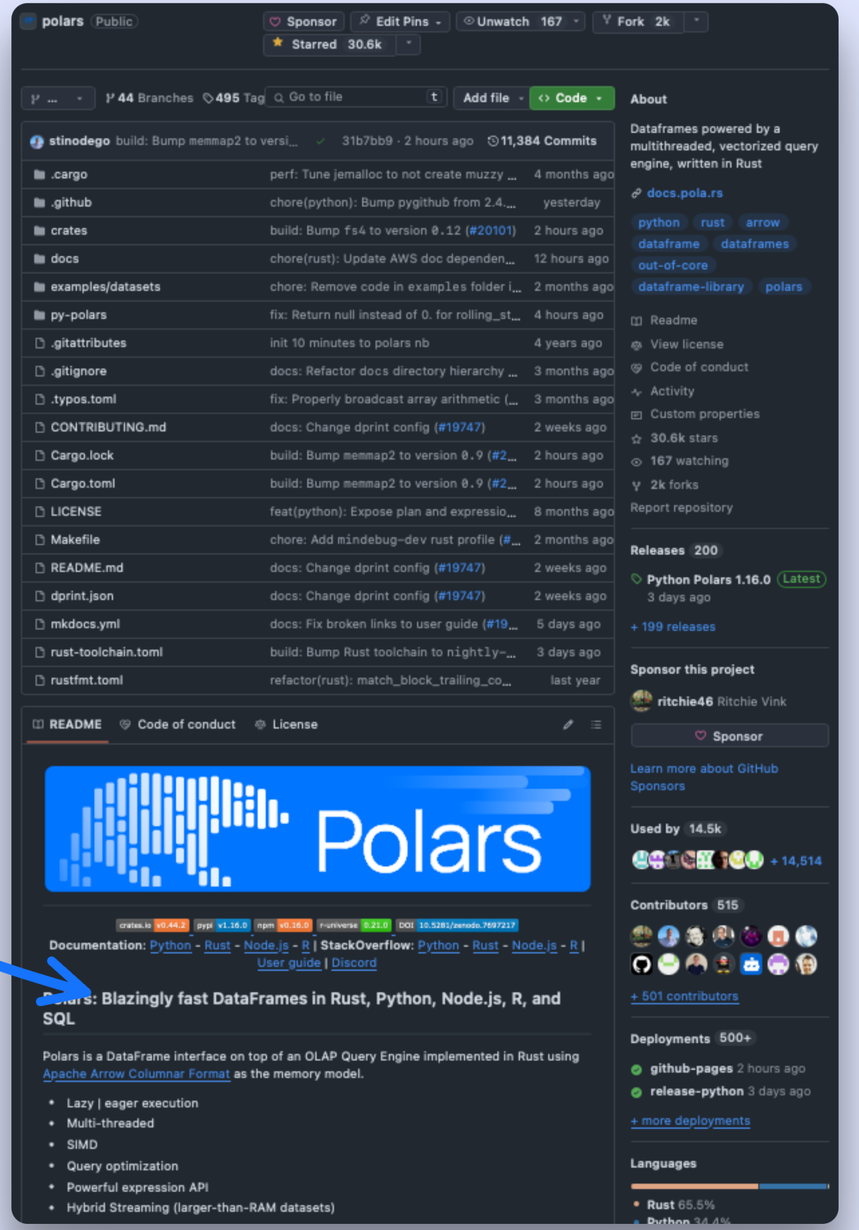
Understanding Polars data types

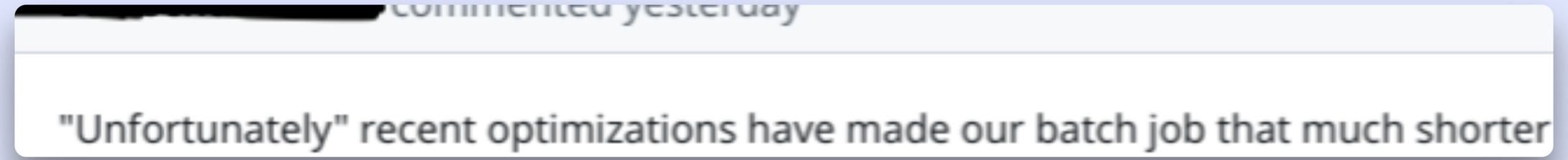
PyData Global 2024

Rodrigo Girão Serrão


polars Public
Sponsor
Edit Pins
Unwatch 167
Fork 2k
Starred 30.6k
44 Branches
495 Tag
Go to file
Add file
Code
stinodego build: Bump memmap2 to versi... 31b7bb9 · 2 hours ago 11,384 Commits
.cargo perf: Tune jemalloc to not create muzzy ... 4 months ago
.github chore(python): Bump pygithub from 2.4.... yesterday
crates build: Bump fs4 to version 0.12 (#20101) 2 hours ago
docs chore(rust): Update AWS doc dependen... 12 hours ago
examples/datasets chore: Remove code in examples folder i... 2 months ago
py-polars fix: Return null instead of 0. for rolling_st... 4 hours ago
.gitattributes init 10 minutes to polars nb 4 years ago
.gitignore docs: Refactor docs directory hierarchy ... 3 months ago
.typos.toml fix: Properly broadcast array arithmetic (... 3 months ago
CONTRIBUTING.md docs: Change dprint config (#19747) 2 weeks ago
Cargo.lock build: Bump memmap2 to version 0.9 (#2... 2 hours ago
Cargo.toml build: Bump memmap2 to version 0.9 (#2... 2 hours ago
LICENSE feat(python): Expose plan and expressio... 8 months ago
Makefile chore: Add mindebug-dev rust profile (#... 2 months ago
README.md docs: Change dprint config (#19747) 2 weeks ago
dprint.json docs: Change dprint config (#19747) 2 weeks ago
mkdocs.yml docs: Fix broken links to user guide (#19... 5 days ago
rust-toolchain.toml build: Bump Rust toolchain to nightly-... 3 days ago
rustfmt.toml refactor(rust): match_block_trailing_co... last year
README
Code of conduct
License
Polars
Documentation: Python - Rust - Node.js - R | StackOverflow: Python - Rust - Node.js - R | User guide | Discord
Polars: Blazingly fast DataFrames in Rust, Python, Node.js, R, and SQL
Polars is a DataFrame interface on top of an OLAP Query Engine implemented in Rust using Apache Arrow Columnar Format as the memory model.
Lazy | eager execution
Multi-threaded
SIMD
Query optimization
Powerful expression API
Hybrid Streaming (larger-than-RAM datasets)
About
Dataframes powered by a multithreaded, vectorized query engine, written in Rust
docs.pola.rs
python rust arrow dataframe dataframes out-of-core dataframe-library polars
Readme
View license
Code of conduct
Activity
Custom properties
30.6k stars
167 watching
2k forks
Report repository
Releases 200
Python Polars 1.16.0 Latest
3 days ago
+ 199 releases
Sponsor this project
ritchie46 Ritchie Vink
Sponsor
Learn more about GitHub Sponsors
Used by 14.5k
+ 14,514
Contributors 515
+ 501 contributors
Deployments 500+
github-pages 2 hours ago
release-python 3 days ago
+ more deployments
Languages
Rust 65.5%

Blazingly fast




commented yesterday
"Unfortunately" recent optimizations have made our batch job that much shorter

Using the right data type?

Using the right data type?

Why bother?

Using the right data type?

- correct semantics

Using the right data type?

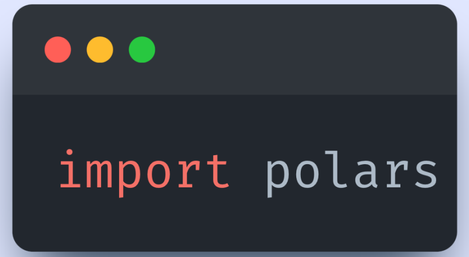
- correct semantics
- memory efficient

Using the right data type?

- correct semantics
- memory efficient
- more performant

Using the right data type?

- correct semantics
- memory efficient
- more performant
- specialised functions


import polars

The data type `Boolean`

The data type `Boolean`

... is like Python's `bool`

The data type `Boolean`

```
N = 73
bools = generate_random_booleans(8 * N)
s = pl.Series(bools, dtype=pl.Boolean)
```

... is like Python's `bool`

The data type `Boolean`

```python
N = 73
bools = generate_random_booleans(8 * N)
s = pl.Series(bools, dtype=pl.Boolean)

print(s.estimated_size())  # ???
```

... is like Python's `bool`

The data type `Boolean`

```python
N = 73
bools = generate_random_booleans(8 * N)
s = pl.Series(bools, dtype=pl.Boolean)

print(s.estimated_size())  # 73
```

... is like Python's `bool`

The data type `Null`

The data type `Null`

… is like Python's `NoneType`

The data type `Null`

~~... is like Python's `NoneType`~~

... is like Python's `None`

The data type `Null`

```
s = pl.Series([1, 2, None, 4])
print(s)
```

... is like Python's `NoneType`

The data type `Null`

```
s = pl.Series([1, 2, None, 4])
print(s)
```

```
shape: (4,)
Series: '' [i64]
[
        1
        2
        null
        4
]
```

... is like Python's `NoneType`

The data type `Null`

```
s = pl.Series([1, 2, None, 4])
print(s)
```

```
shape: (4,)
Series: '' [i64]
[
        1
        2
        null
        4
]
```

... is like Python's `NoneType`

The data type `Null`

```
s = pl.Series([1, 2, None, 4])
print(s.is_null())
```

```
shape: (4,)
Series: '' [bool]
[
        false
        false
        true
        false
]
```

... is like Python's `NoneType`

The data type `Null`

```
s = pl.Series([1, 2, None, 4])
print(s.is_null())
```

```
shape: (4,)
Series: '' [bool]
[
        false
        false
        true
        false
]
```

“Free” because of the validity mask.

… is like Python's `NoneType`

The data type `Null`

“Free”?!

… is like Python's `NoneType`

The data type `Null`

“Free”?! “There is no free lunch.”

… is like Python's `NoneType`

The validity mask is efficiently bit-packed

```
print(
    pl.Series([0, 1, 2, 3])
    .estimated_size()
)  # 32
```

```
print(
    pl.Series([0, 1, None, 3])
    .estimated_size()
)  # 33
```

The validity mask is efficiently bit-packed

```
print(
    pl.Series(range(4))
    .estimated_size()
)  # 32
```

```
print(
    pl.Series([0, 1, None, 3])
    .estimated_size()
)  # 33
```

```
print(
    pl.Series(range(8))
    .estimated_size()
)  # 64
```

```
print(
    pl.Series([0, 1, None, 3,
               4, 5, 6, 7])
    .estimated_size()
)  # ???
```

... is like Python's NoneType

The data type `IntX` / `UIntX`

The data type `IntX` / `UIntX`

… is like Python's `int`

Python supports arbitrarily large integers

```
googol = 10 ** 100
print(googol % 99_999_999_977)  # 11526618770
```

The data type `IntX` / `UIntX`

`IntX` data types:

- `Int8`
- `Int16`
- `Int32`
- `Int64`

... is like Python's `int`

The data type `IntX` / `UIntX`

`IntX` data types:

- `Int8`
- `Int16`
- `Int32`
- `Int64`

Bits available

... is like Python's `int`

The data type `IntX` / `UIntX`

`IntX` data types:

- `Int8`
- `Int16`
- `Int32`
- `Int64`

Bits available

1 for the sign

... is like Python's `int`

The data type `IntX` / `UIntX`

`IntX` data types:

- `Int8`
- `Int16`
- `Int32`
- `Int64`

Bits available

1 for the sign

X - 1

for the value

... is like Python's `int`

The data type `IntX` / `UIntX`

Data types:	Lower limit	Upper limit
• `Int8`	• `-128`	• `127`
• `Int16`	• `-32,768`	• `32,767`
• `Int32`	• `-2,147,483,648`	• `2,147,483,647`
• `Int64`	• `-2**63`	• `2**63 - 1`

... is like Python's `int`

The data type `IntX` / `UIntX`

Data types:	Lower limit	Upper limit
• `Int8`	• `-128`	• `127`
• `Int16`	• `-32,768`	• `32,767`
• `Int32`	• `-2,147,483,648`	• `2,147,483,647`
• `Int64`	• `-2**63`	• `2**63 - 1`

~9 quintillion

... is like Python's `int`

The data type `IntX` / `UIntX`

`UIntX` data types:

- `UInt8`
- `UInt16`
- `UInt32`
- `UInt64`

Bits available

... is like Python's `int`

The data type `IntX` / `UIntX`

`UIntX` data types:

- `UInt8`
- `UInt16`
- `UInt32`
- `UInt64`

Bits available

0 for the sign

... is like Python's `int`

The data type `IntX` / `UIntX`

`UIntX` data types:

- `UInt8`
- `UInt16`
- `UInt32`
- `UInt64`

Bits available

`0` for the sign

`X` for the value

... is like Python's `int`

The data type `IntX` / `UIntX`

⚠️

Overflow/underflow

```
s = pl.Series([0], dtype=pl.UInt64)
print(
    (s - 1).to_list()[0]
)
# 18446744073709551615, ~18 quintillion
```

... is like Python's `int`

The data type `FloatX`

The data type `FloatX`

... is like Python's `float`

`Float32` and `Float64` have the usual rounding issues..?

```
pl.Series([0.1]) + 0.2
"""
shape: (1,)
Series: '' [f64]
[
        0.3
]
"""
```

... is like Python's `float`

Float32 and Float64 have the usual rounding issues!!!

```
with pl.Config(set_float_precision=17):
    pl.Series([0.1]) + 0.2
"""
shape: (1,)
Series: '' [f64]
[
        0.30000000000000004
]
"""
```

Change printing formatting

The data type `FloatX`

`NaN` vs `null`

… is like Python's `float`

NaN vs null

```
pl.Series([1, 0, None]) / 0
```

... is like Python's `float`

NaN vs null

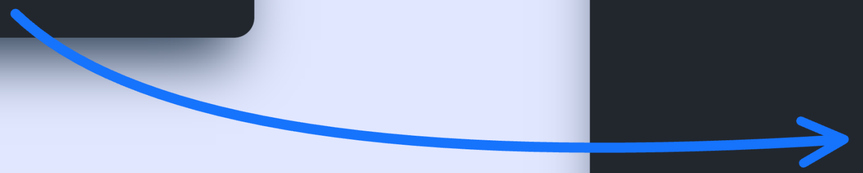
```
pl.Series([1, 0, None]) / 0
```

```
shape: (3,)
Series: '' [f64]
[
        inf
        NaN
        null
]
```

... is like Python's float

NaN vs null

```
pl.Series([1, 0, None]) / 0
```



For missing data

```
shape: (3,)
Series: '' [f64]
[
        inf
        NaN
        null
]
```

NaN vs null

```
pl.Series([1, 0, None]) / 0
```

```
shape: (3,)
Series: '' [f64]
[
        inf
        NaN
        null
]
```

When maths doesn't math 😵‍💫🤯✨

The data type `String`

The data type `String`

... is like Python's `str`

Strings have dozens of efficient functions in `str`:

```
pl.Series(["Hello, world!", "Polars is great"]).str.slice(0, 6)
```

```
shape: (2,)
Series: '' [str]
[
        "Hello,"
        "Polars"
]
```

The data type `String`

Sometimes you'll have a string but don't want one.

... is like Python's `str`

The data type `Date`

The data type `Date`

… is like Python's `datetime.date`

The data type Date

```python
from datetime import date

df = pl.DataFrame({
    "superhero": ["Superman", ... ],
    "first_appearance": [
        date(1938, 4, 18),
        ... ,
    ]
})
```

shape: (... , 2)

superhero	first_appearance
str	date
Superman	1938-04-18
...	...

... is like Python's `datetime.date`

The data type `Time`

The data type `Time`

… is like Python's `datetime.time`

The data type Time

```python
from datetime import time

df = pl.DataFrame({
    "superhero": ["Superman", ... ],
    "avg_wake_up_time": [
        time(5, 30, 0),
        ...
    ]
})
```

```
shape: ( ... , 2)
```

superhero	avg_wake_up_time
str	time
Superman	05:30:00
...	...

... is like Python's `datetime.time`

The data type `Datetime`

The data type `Datetime`

... is like Python's `datetime.datetime`

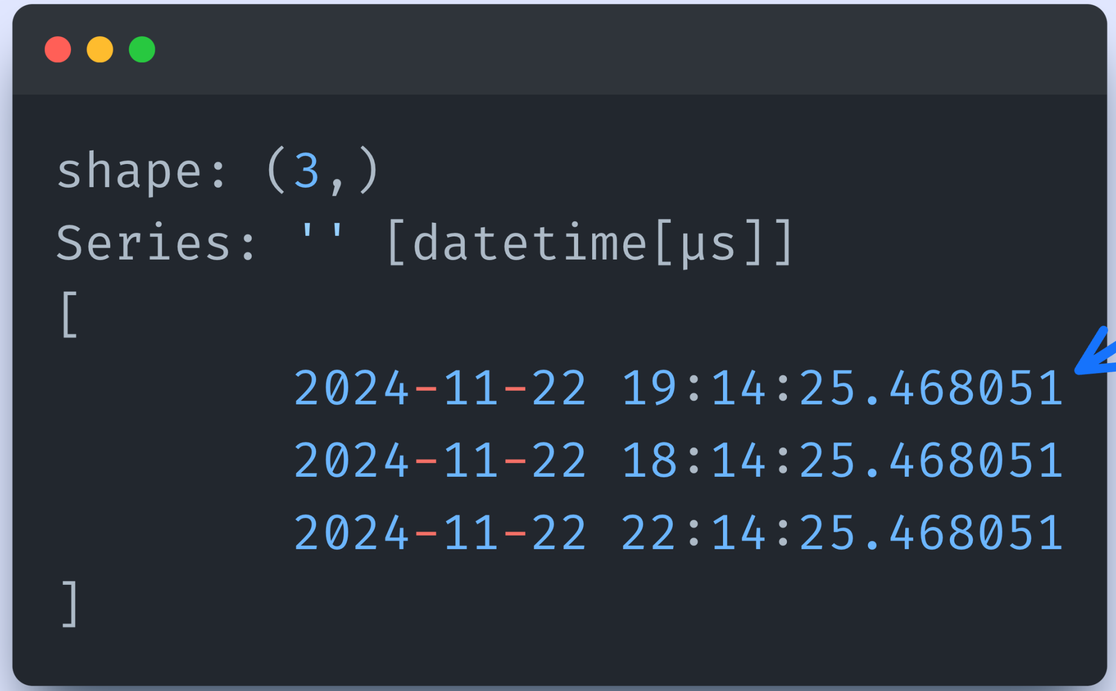
The data type `Datetime`

```python
from datetime import datetime, timedelta, timezone

now = datetime.now()
datetimes = [
    now,
    now.replace(tzinfo=timezone(timedelta(hours=1))),
    now.replace(tzinfo=timezone(timedelta(hours=-3))),
]

s = pl.Series(datetimes)
```

... is like Python's `datetime.datetime`

The data type `Datetime`

Single timezone



```
shape: (3,)
Series: '' [datetime[μs]]
[
        2024-11-22 19:14:25.468051
        2024-11-22 18:14:25.468051
        2024-11-22 22:14:25.468051
]
```

... is like Python's `datetime.datetime`

The data type Datetime

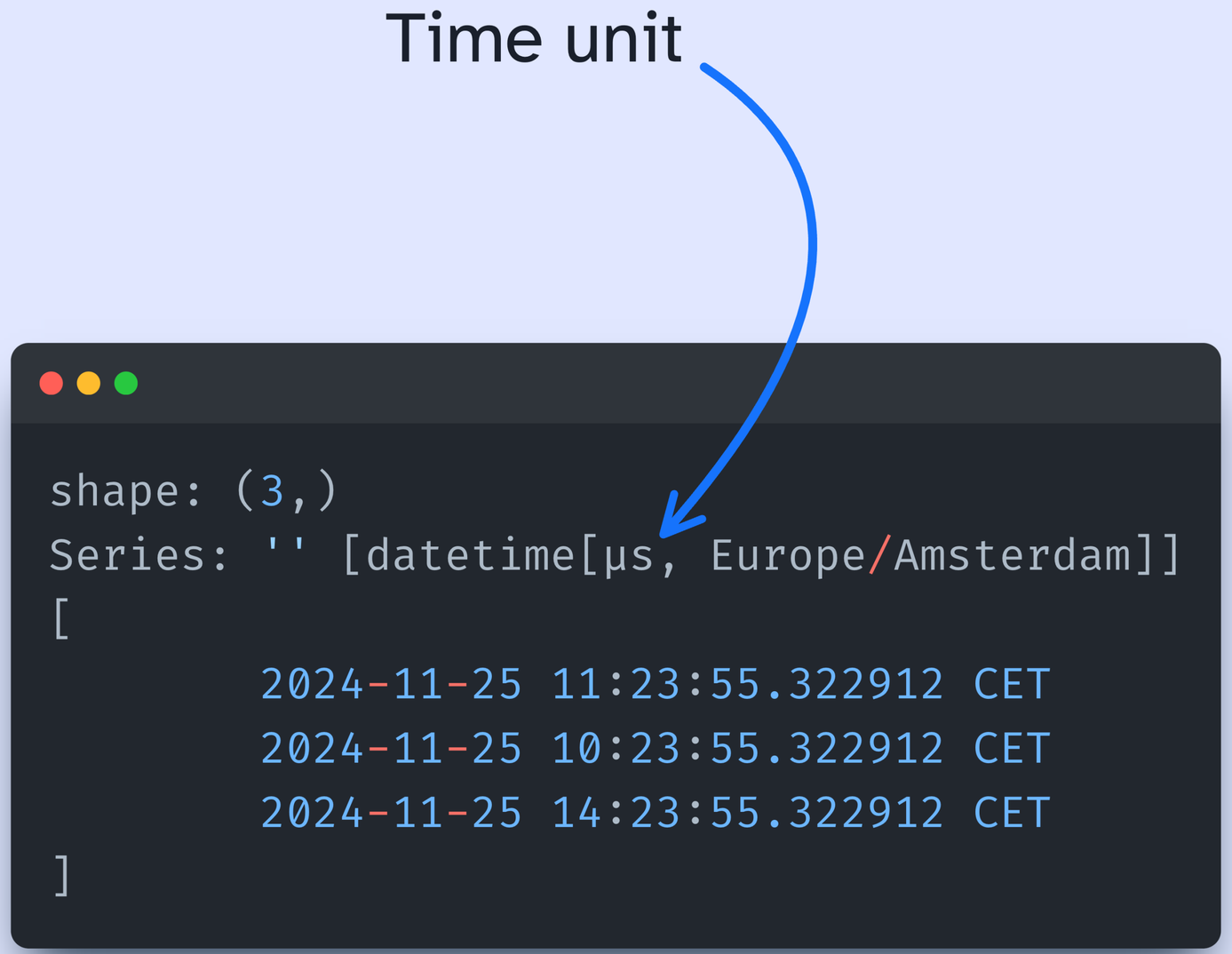
```
s.dt.convert_time_zone("Europe/Amsterdam")
```

Single timezone

```
shape: (3,)
Series: '' [datetime[μs, Europe/Amsterdam]]
[
        2024-11-25 11:23:55.322912 CET
        2024-11-25 10:23:55.322912 CET
        2024-11-25 14:23:55.322912 CET
]
```

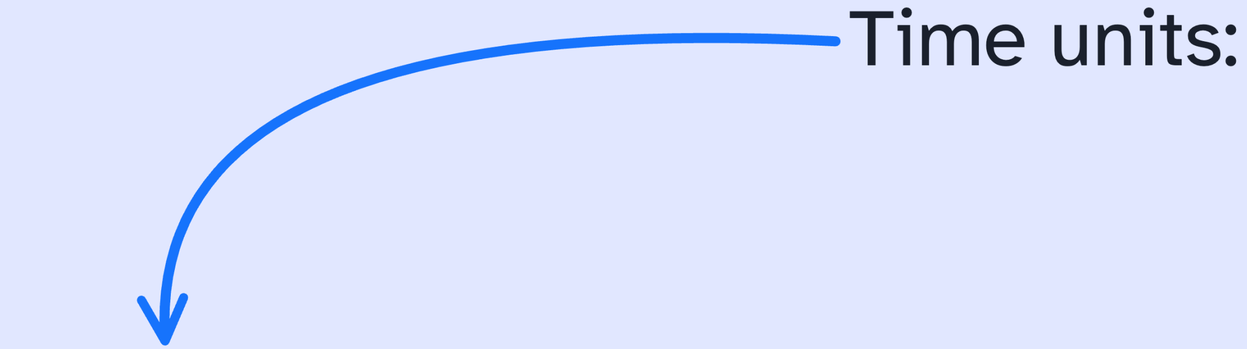
... is like Python's `datetime.datetime`

The data type `Datetime`



... is like Python's `datetime.datetime`

The data type `Datetime`



Millisecond	ms	`10^3`
→ Microsecond	µs/us	`10^6`
Nanosecond	ns	`10^9`

… is like Python's `datetime.datetime`

The data type `Datetime`

Strings → Temporal dtypes

`.str.to_date`	`Date`
`.str.to_datetime`	`Datetime`
`.str.to_time`	`Time`
`.str.strptime`	`Any`

... is like Python's `datetime.datetime`

The data type `Datetime`

Strings ← Temporal dtypes

`.dt.to_string` Any

... is like Python's `datetime.datetime`

The data type `Duration`

The data type `Duration`

… is like Python's `datetime.timedelta`

The data type `Duration`

😴

```
bedtime = pl.Series([
    datetime(2024, 11, 22, 23, 56),
    datetime(2024, 11, 24, 0, 23),
    datetime(2024, 11, 24, 23, 37),
])
```

🥱

```
wake_up = pl.Series([
    datetime(2024, 11, 23, 7, 30),
    datetime(2024, 11, 24, 7, 30),
    datetime(2024, 11, 25, 8, 0),
])
```

... is like Python's `datetime.timedelta`

The data type `Duration`

```
wake_up - bedtime
```

... is like Python's `datetime.timedelta`

The data type `Duration`

```
wake_up - bedtime
```

```
shape: (3,)
Series: '' [duration[μs]]
[
        7h 34m
        7h 7m
        8h 23m
]
```

... is like Python's `datetime.timedelta`

The data type `Binary`

The data type `Binary`

... is like Python's `bytes`

The data type `Binary`

Specialised namespace `bin`

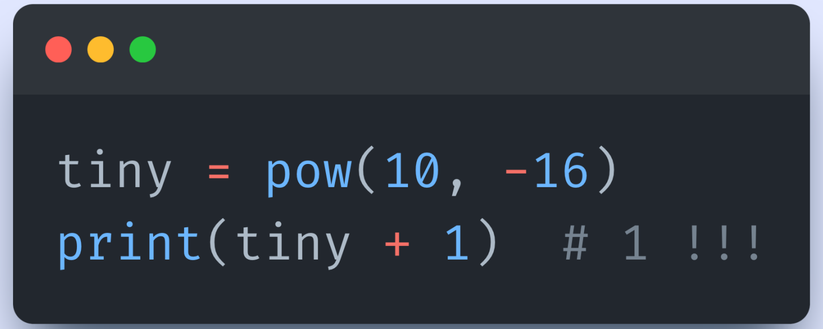
... is like Python's `bytes`

The data type `Decimal`

The data type `Decimal`

... is like Python's `decimal.Decimal`

The data type `Decimal`



```
tiny = pow(10, -16)
print(tiny + 1)  # 1 !!!
```

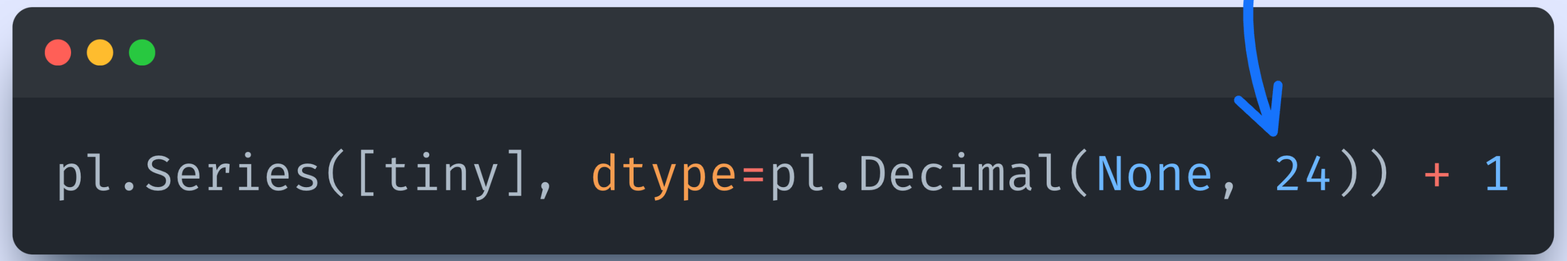
... is like Python's `decimal.Decimal`

The data type `Decimal`

```
pl.Series([tiny], dtype=pl.Decimal(None, 24)) + 1
```

... is like Python's `decimal.Decimal`

The data type `Decimal`



... is like Python's `decimal.Decimal`

The data type `Decimal`

of total digits

```
pl.Series([tiny], dtype=pl.Decimal(None, 24)) + 1
```

... is like Python's `decimal.Decimal`

The data type `Decimal`

```
pl.Series([tiny], dtype=pl.Decimal(None, 24)) + 1
```

```
shape: (1,)
Series: '' [decimal[*,24]]
[
        1.000000000000000100000000
]
```

... is like Python's `decimal.Decimal`

The data type Enum

The data type `Enum`

... is like Python's `enum.StrEnum`

The data type `Enum`

Define valid values

```
valid_values = ["panda", "polar", "brown"]  # 1.
```

... is like Python's `enum.StrEnum`

The data type `Enum`

```python
valid_values = ["panda", "polar", "brown"]  # 1.
bear_enum = pl.Enum(valid_values)  # 2.
```

Create `Enum` variant

... is like Python's `enum.StrEnum`

The data type `Enum`

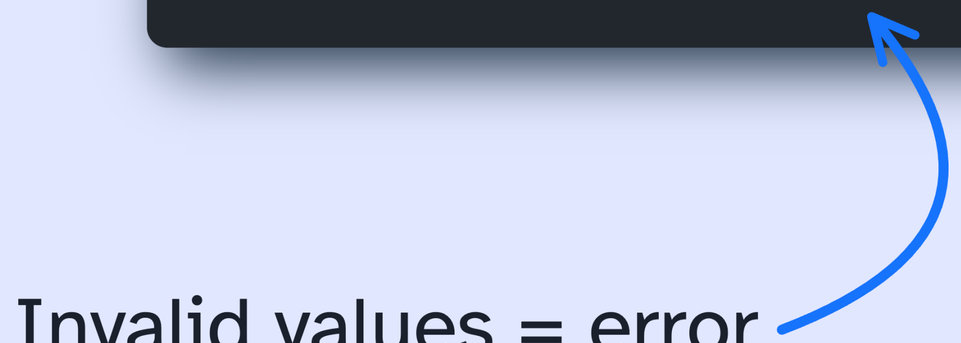
```python
valid_values = ["panda", "polar", "brown"]  # 1.
bear_enum = pl.Enum(valid_values)  # 2.

s = pl.Series(
    ["panda", "polar", "panda", "brown", "panda"],
    dtype=bear_enum,  # 3.
)
```

Use the `Enum` variant

... is like Python's `enum.StrEnum`

The data type `Enum`

```
pl.Series(["pand", "snake"], dtype=bear_enum)
```



Invalid values = error

... is like Python's `enum.StrEnum`

The data type `Enum`

```
education_level = ["High school", "BSc", "MSc", "PhD"]
education_enum = pl.Enum(education_level)
```

Values have a “natural” order

… is like Python's `enum.StrEnum`

The data type `Enum`

```
education_level = ["High school", "BSc", "MSc", "PhD"]
education_enum = pl.Enum(education_level)

people = pl.DataFrame({
    "name": pl.Series(["A", "B", "C", "D"]),
    "degree": pl.Series(
        ["High school", "MSc", "PhD", "MSc"],
        dtype=education_enum,
    ),
})
```

Use the enum variant

... is like Python's `enum.StrEnum`

The data type Enu

```
print(people.filter(pl.col("degree") ⩾ "MSc"))
```

```
shape: (3, 2)
```

name	degree
str	enum
B	MSc
C	PhD
D	MSc

... is like Python's `enum.StrEnum`

The data type `Categorical`

The data type `Categorical`

... is like Python's `enum.StrEnum`

The data type `Categorical`

```
s = pl.Series(
    ["panda", "polar", "pand", "snake", "panda"],
    dtype=pl.Categorical,
)
```

... is like Python's `enum.StrEnum`

The data type `Categorical`

No predefined values

```
s = pl.Series(
    ["panda", "polar", "pand", "snake", "panda"],
    dtype=pl.Categorical,
)
```

... is like Python's `enum.StrEnum`

The data type `Categorical`

No predefined values

```
s = pl.Series(
    ["panda", "polar", "pand", "snake", "panda"],
    dtype=pl.Categorical,
)
print(s.cat.get_categories().to_list())
```

```
['panda', 'polar', 'pand', 'snake']
```

... is like Python's `enum.StrEnum`

The data type `Struct`

The data type `Struct`

... is like Python's `typing.TypedDict`

The data type `Struct`

```
df = pl.DataFrame({
    "name": ["A", "B", "C", "D"],
    "favourite_sport": [
        "basketball", "baseball",
        "soccer", "basketball",
    ],
})
```

... is like Python's `typing.TypedDict`

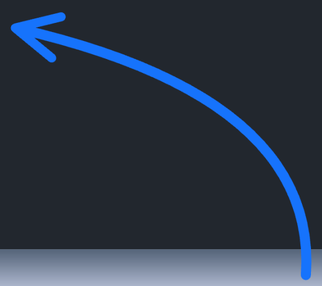
The data type `Struct`

```python
df = pl.DataFrame({
    "name": ["A", "B", "C", "D"],
    "favourite_sport": [
        "basketball", "baseball",
        "soccer", "basketball",
    ],
})
counts = df.select(
    pl.col("favourite_sport")
    .value_counts()
)
print(counts)
```

... is like Python's `typing.TypedDict`

The data type `Struct`

```python
df = pl.DataFrame({
    "name": ["A", "B", "C", "D"],
    "favourite_sport": [
        "basketball", "baseball",
        "soccer", "basketball",
    ],
})
counts = df.select(
    pl.col("favourite_sport")
    .value_counts()
)
print(counts)
```



How many times does each sport appear?

... is like Python's `typing.TypedDict`

The data type Struct

```python
df = pl.DataFrame({
    "name": ["A", "B", "C", "D"],
    "favourite_sport": [
        "basketball", "baseball",
        "soccer", "basketball",
    ],
})
counts = df.select(
    pl.col("favourite_sport")
    .value_counts()
)
print(counts)
```

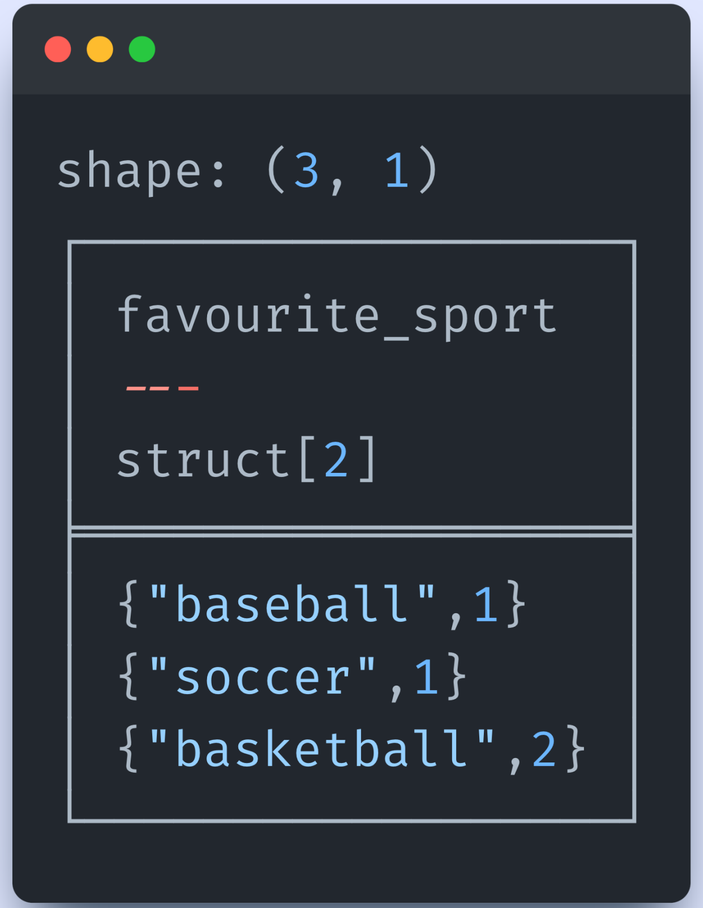
```
shape: (3, 1)
```

favourite_sport
struct[2]
{"baseball",1}
{"soccer",1}
{"basketball",2}

How many times does each sport appear?

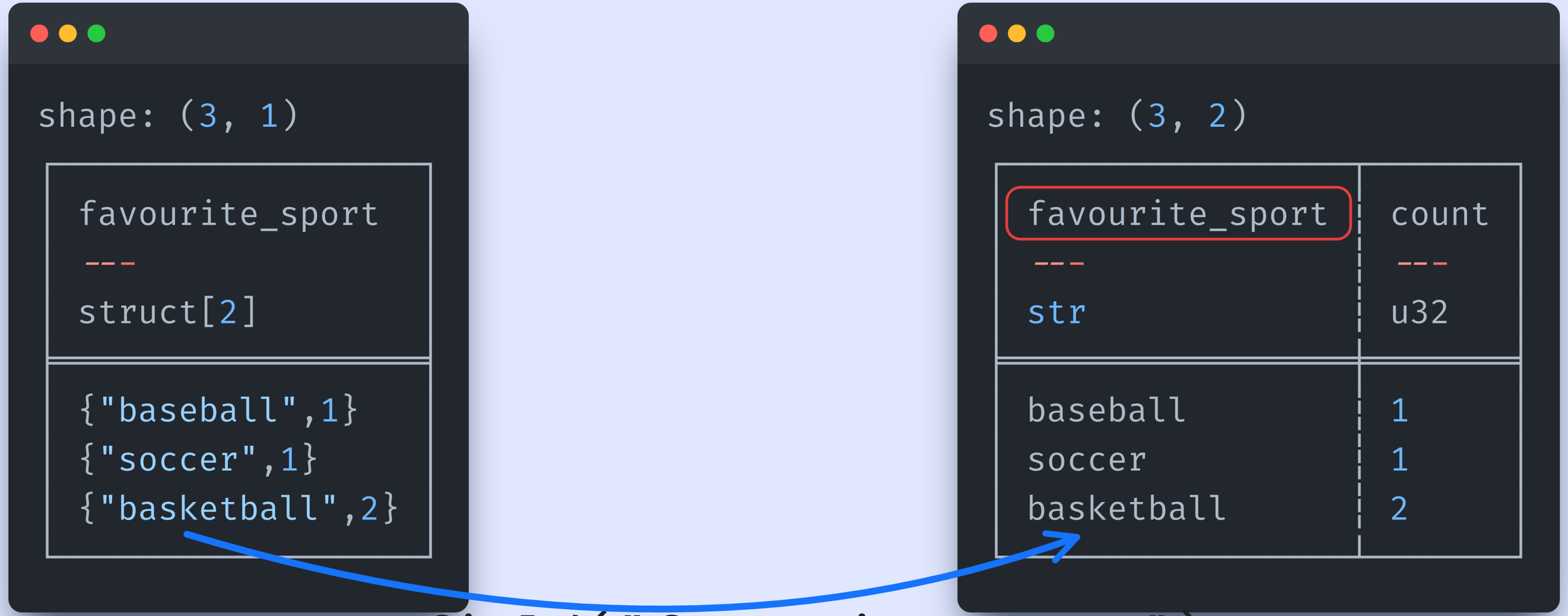
... is like Python's `typing.TypedDict`

The data type `Struct`



... is like Python's `typing.TypedDict`

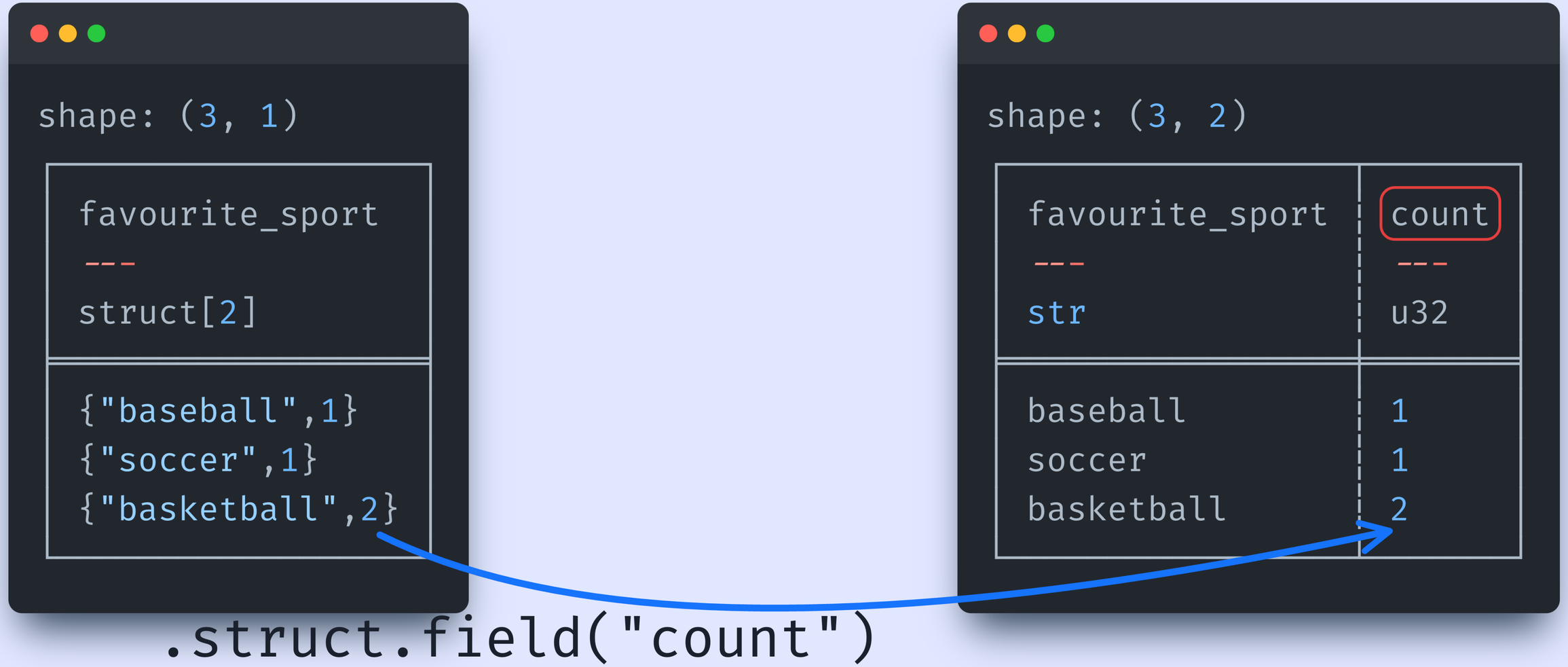
The data type `Struct`




```
.struct.field("favourite_sport")
```

... is like Python's `typing.TypedDict`

The data type `Struct`



... is like Python's `typing.TypedDict`

The data type `List`

The data type `List`

... is like Python's `list`

The data type `List`

```
pl.Series([
    [1, 2, 3],
    [],
    [4, 5],
    [6],
])
```

... is like Python's `list`

The data type `List`

```
pl.Series([
    [1, 2, 3],
    [],
    [4, 5],
    [6],
])
```

```
shape: (4,)
Series: '' [list[i64]]
[
        [1, 2, 3]
        []
        [4, 5]
        [6]
]
```

... is like Python's `list`

The data type `List`

```
pl.Series([
    [1, 2, 3],
    [],
    [4, 5],
    [6],
])
```

Inner type

```
shape: (4,)
Series: '' [list[i64]]
[
        [1, 2, 3]
        []
        [4, 5]
        [6]
]
```

... is like Python's `list`

The data type `List`

Specialised functions in the namespace `list`.

... is like Python's `list`

The data type `Array`

The data type `Array`

... is like Python's `numpy.array`

The data type `Array`

```
pl.Series(
    [
        ["Harry", "Potter"],
        ["Hermione", "Granger"],
        ["Ron", "Weasley"],
    ],
    dtype=pl.Array(pl.String, (2,)),
)
```

... is like Python's `numpy.array`

The data type Array

```
pl.Series(
    [
        ["Harry", "Potter"],
        ["Hermione", "Granger"],
        ["Ron", "Weasley"],
    ],
    dtype=pl.Array(pl.String, (2,)),
)
```

```
shape: (3,)
Series: '' [array[str, 2]]
[
        ["Harry", "Potter"]
        ["Hermione", "Granger"]
        ["Ron", "Weasley"]
]
```

... is like Python's `numpy.array`

The data type `Array`

Inner type

```
pl.Series(
    [
        ["Harry", "Potter"],
        ["Hermione", "Granger"],
        ["Ron", "Weasley"],
    ],
    dtype=pl.Array(pl.String, (2,)),
)
```

```
shape: (3,)
Series: '' [array[str, 2]]
[
        ["Harry", "Potter"]
        ["Hermione", "Granger"]
        ["Ron", "Weasley"]
]
```

... is like Python's `numpy.array`

The data type `Array`

Shape

```
pl.Series(
    [
        ["Harry", "Potter"],
        ["Hermione", "Granger"],
        ["Ron", "Weasley"],
    ],
    dtype=pl.Array(pl.String, (2,)),
)
```

```
shape: (3,)
Series: '' [array[str, 2]]
[
        ["Harry", "Potter"]
        ["Hermione", "Granger"]
        ["Ron", "Weasley"]
]
```

... is like Python's `numpy.array`

The data type `Array`

```
major = np.array(range(24)).reshape(3, 2, 2, 2)
```

4D array

... is like Python's `numpy.array`

The data type Array

```
major = np.array(range(24)).reshape(3, 2, 2, 2)
print(pl.Series(major))
```

4D array

```
shape: (3,)
Series: '' [array[i64, (2, 2, 2)]]
[
        [[[0, 1], [2, 3]], [[4, 5], [6, 7]]]
        [[[8, 9], [10, 11]], [[12, 13], [14, 15]]]
        [[[16, 17], [18, 19]], [[20, 21], [22, 23]]]
]
```

... is like Python's numpy.array

The data type `Array`

```
major = np.array(range(24)).reshape(3, 2, 2, 2)
print(pl.Series(major))
```

```
shape: (3,)
Series: '' [array[i64, (2, 2, 2)]]
[
        [[[0, 1], [2, 3]], [[4, 5], [6, 7]]]
        [[[8, 9], [10, 11]], [[12, 13], [14, 15]]]
        [[[16, 17], [18, 19]], [[20, 21], [22, 23]]]
]
```

3D sub-arrays

... is like Python's `numpy.array`

The data type `Array`

Specialised functions in the namespace `arr`.

... is like Python's `numpy.array`

The data type `Array`

Specialised functions in the namespace `arr`.

Same functions as the ones in `list`.

... is like Python's `numpy.array`

The data type `Array`

Specialised functions in the namespace `arr`.

Same functions as the ones in `list`.

`arr`'s are more efficient.

... is like Python's `numpy.array`

The data type `Object`

The data type `Object`

… is like Python's `object`

The data type `Object`

```
pl.Series([
    enumerate,
    zip,
    max,
    min,
    all,
    any,
    sorted,
])
```

```
shape: (7,)
Series: '' [o][object]
[
        <class 'enumerate'>
        <class 'zip'>
        <built-in function max>
        <built-in function min>
        <built-in function all>
        <built-in function any>
        <built-in function sorted>
]
```

... is like Python's `object`

The data type `Object`

No specialised functions/namespace.

... is like Python's `object`

Using the right data type?

Using the right data type?

- correct semantics

Using the right data type?

- correct semantics
- memory efficient

Using the right data type?

- correct semantics
- memory efficient
- more performant

Using the right data type?

- correct semantics
- memory efficient
- more performant
- specialised functions

https://docs.pola.rs/user-guide/getting-started/

Thank you!

The Polars data type …	is like Python's …
`Boolean`	`bool`
`Int8`, `Int16`, `Int32`, and `Int64`	`int` with restrictions
`UInt8`, `UInt16`, `UInt32`, and `UInt64`	`int` with restrictions
`Float32` and `Float64`	`float`
`Decimal`	`decimal.Decimal`
`String`	`str`
`Binary`	`bytes`
`Date`	`datetime.date`
`Time`	`datetime.time`
`Datetime`	`datetime.datetime`
`Duration`	`datetime.timedelta`
`Array`	`numpy.array`
`List`	`list`
`Categorical`	`enum.StrEnum`
`Enum`	`enum.StrEnum`
`Struct`	`typing.TypedDict`
`Object`	`object`
`Null`	`None`