# Dataframely

A declarative, 🐻‍❄️-native data frame validation library

**Oliver Borchert**

May 14, 2025

# About me

**Oliver Borchert**

@borchero

in/borchero

- BSc Computer Science, MSc Data Engineering & Analytics @ TUM
- Wrote my Master thesis at AWS
- Joined QuantCo in Munich in July 2022
- Working on machine learning & data engineering
- Excited about open-source software and avidly contributing

quantco

- We help companies turn data into decisions by combining economics, engineering, and AI
- We maintain dozens of data pipelines processing billions of data points
- Pipelines are built on top of open-source software and we are heavily contributing to it

# Where we started…

Trying to understand our legacy data pipeline 😵‍💫


We 💖 polars :)

🚀 Rewrite in **polars** for improved performance

**How can we make the pipeline easy to understand and easy to debug?**

# Why dataframely?

😵‍💫 **Maintaining** code without knowledge of data frame contents is **time-consuming**

💥 **Erroneous** or unexpected **data** can lead to **costly data pipeline failures**

💪 **Validating** assumptions about data frames **increases pipeline robustness**

👓 **Type annotations** for data frame contents greatly **improve code legibility**

`dataframely` — A declarative, 🐻‍❄️-native data frame validation library

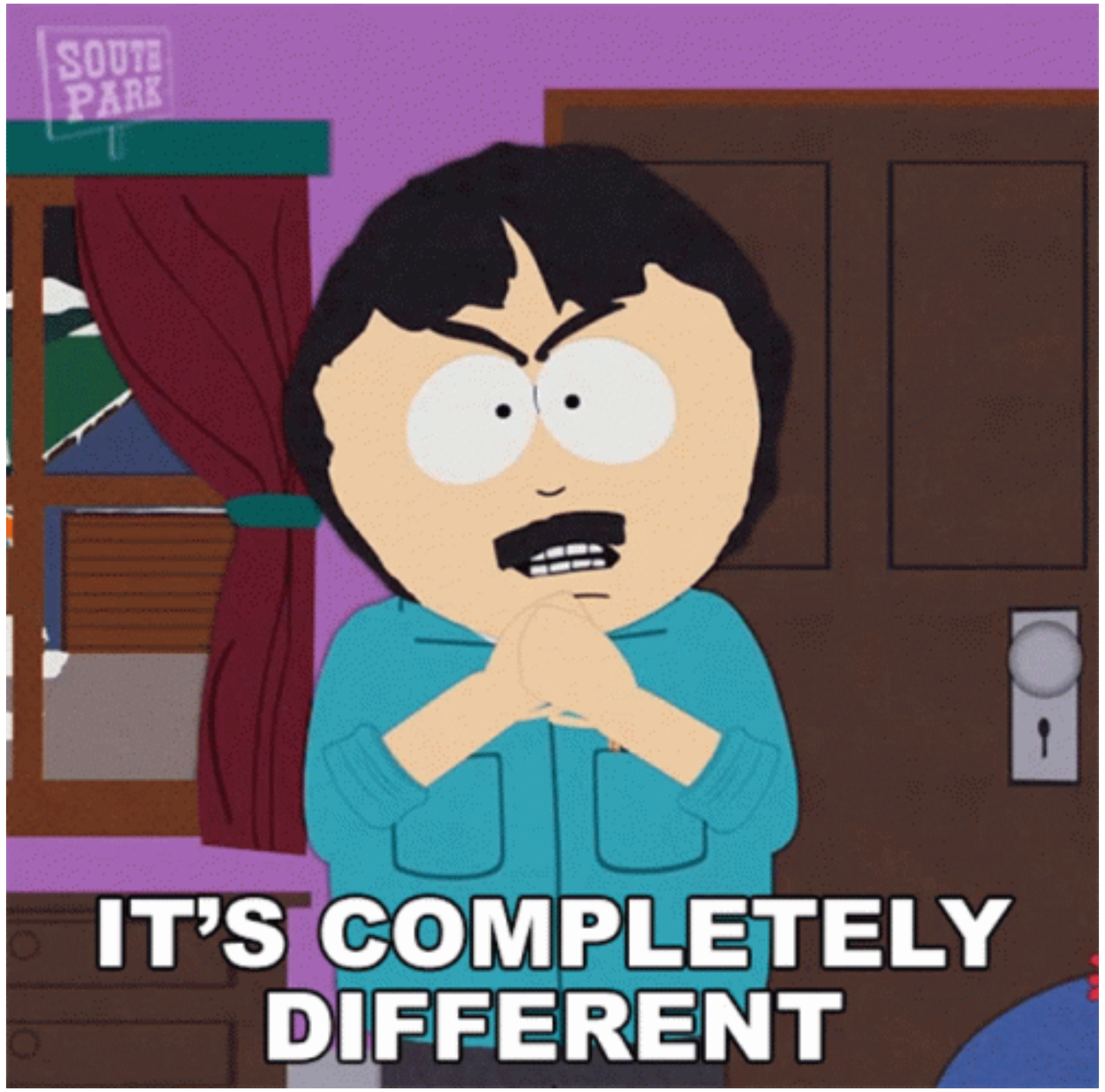
# Why not use existing libraries?

**Dataframely extends the scope of existing libraries with many advanced features!**

| | `pandera[polars]` | `patito` | `dataframely` |
|---|---|---|---|
| Validation with simple column constraints | ✅ | ✅ | ✅ |
| Full support for polars data types | ✅ | | ✅ |
| Lazy validation | ✅ | ✅ | ✅ |
| Composite primary keys | ✅ | 🟠 | ✅ |
| Validation on groups of rows | 🟠 | ❌ | ✅ |
| Validation of interdependent data frames | ❌ | ❌ | ✅ |
| Soft-validation for production use cases | ❌ | ❌ | ✅ |
| Structured validation failure info | 🟠 | ❌ | ✅ |
| Export to SQL schema / Usage as a DSL | ❌ | ❌ | ✅ |
| Data generation for unit testing | ❌ | 🟠 | ✅ |

Unmaintained 😢

# What about Great Expectations?




We talk about this today! :)

**Data Validation ≠ Data Validation**

| Row-level | Data frame-level |
|---|---|
| • Row-level validation result<br>• Can confidently be used on subsets of the data | • Binary validation result for an entire dataset<br>• Allows to check distributions of column values |

# Running Example: Hospital Claims

Each table is typically represented by a data frame

## Relational Data Model

| Hospital Invoices | |
|---|---|
| `invoice_id` | `data` |
| `001` | `...` |
| `002` | `...` |
| `...` | `...` |
| `...` | `...` |

1:N relationship

| Diagnoses | |
|---|---|
| `invoice_id` | `diagnosis` |
| `001` | `R26.8` |
| `002` | `U50.50` |
| `002` | `E87.6` |
| `...` | `...` |

In order to write **easily understandable** code that processes this data **reliably**, all you need is…


import dataframely as dy
imgflip.com

# Defining a Schema

*Schemas* allow to define names and datatypes of all data frame columns

```python
class InvoiceSchema(dy.Schema):
    invoice_id = dy.String()
    admission_date = dy.Date()
    discharge_date = dy.Date()
    received_at = dy.Datetime()
    amount = dy.Decimal()
```

# Defining a Schema

*Parameters* of column types can be used to express value constraints

```python
class InvoiceSchema(dy.Schema):
    invoice_id = dy.String(primary_key=True)
    admission_date = dy.Date(nullable=False)
    discharge_date = dy.Date(nullable=False)
    received_at = dy.Datetime(nullable=False)
    amount = dy.Decimal(nullable=False, min_exclusive=Decimal(0))
```

# Defining a Schema

`@dy.rule()` can be used to express cross-column constraints

```python
class InvoiceSchema(dy.Schema):
    invoice_id = dy.String(primary_key=True)
    admission_date = dy.Date(nullable=False)
    discharge_date = dy.Date(nullable=False)
    received_at = dy.Datetime(nullable=False)
    amount = dy.Decimal(nullable=False, min_exclusive=Decimal(0))

    @dy.rule()
    def discharge_after_admission() -> pl.Expr:
        return pl.col("discharge_date") >= pl.col("admission_date")

    @dy.rule()
    def received_at_after_discharge() -> pl.Expr:
        return pl.col("received_at").dt.date() >= pl.col("discharge_date")
```

# Validating Data

`.validate()` allows to check schema-compliance of a data frame

```
invoices: pl.DataFrame

df_valid = InvoiceSchema.validate(invoices)
```

Invalid data in any row raises a RuleValidationError with failure details

```
RuleValidationError: 1 rules failed validation:
 * Column 'amount' failed validation for 1 rules:
   - 'min_exclusive' failed for 1 rows
```

Hard failures are problematic in production!

# Filtering Data in Production

`.filter()` partitions the data frame into "good" and "bad" rows

```
invoices: pl.DataFrame

good, failure = InvoiceSchema.filter(invoices, cast=True)
```

Try to cast input data types to the expected types

This method **never** raises an exception — but `failure` can be used to investigate

```
failure.counts()
failure.cooccurrence_counts()
failure.invalid()
```

# How does `filter` work in practice?

| invoice_id | admission_date | discharge_date | received_at | amount |
|---|---|---|---|---|
| str | date | date | datetime[µs] | f64 |
| "001" | 2025-01-01 | 2025-01-04 | 2025-01-05 00:00:00 | 0.0 |
| "002" | 2025-01-05 | 2025-01-07 | 2025-01-08 00:00:00 | 200.0 |
| "003" | 2025-01-01 | 2025-01-01 | 2025-01-02 00:00:00 | 400.0 |

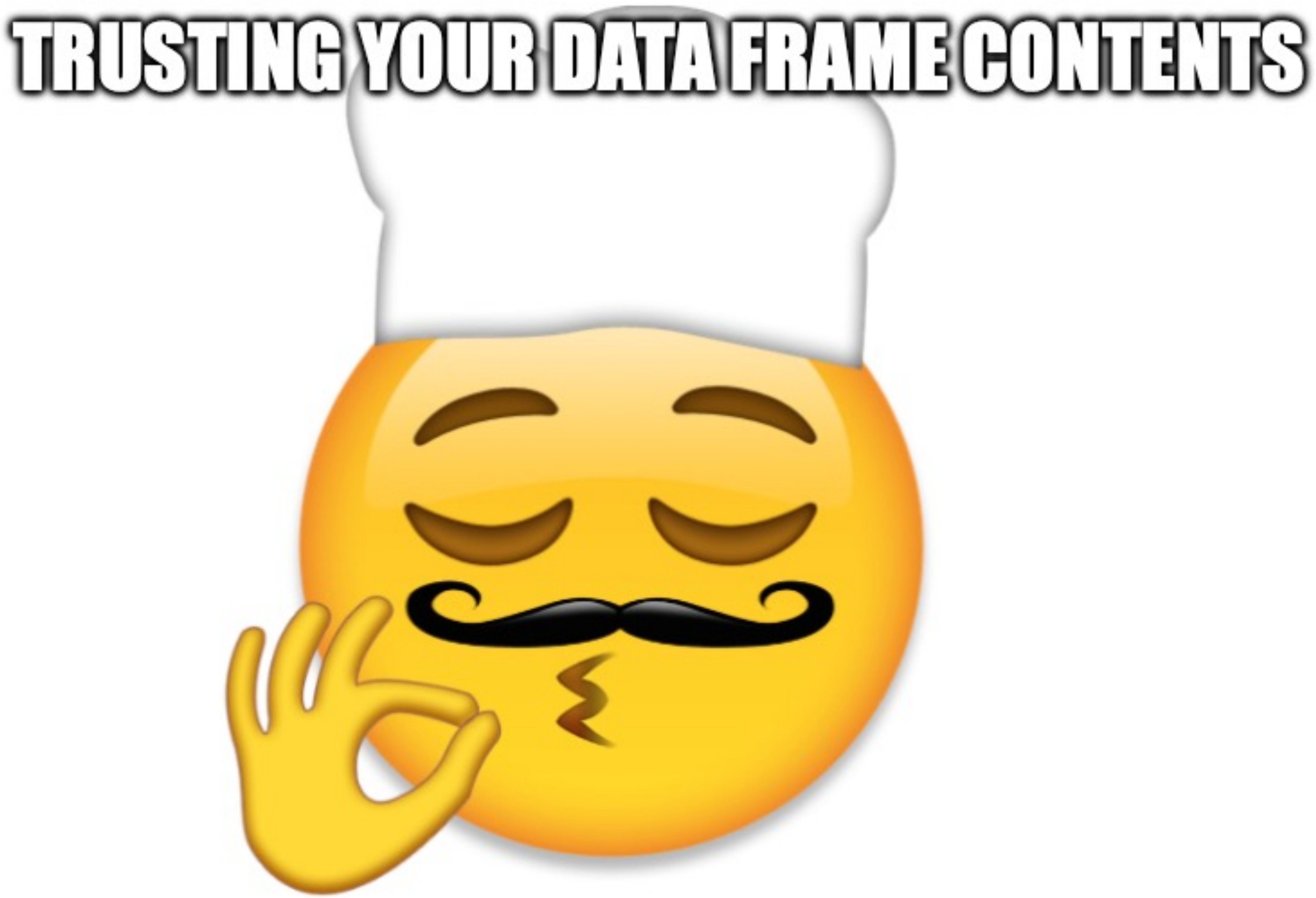
`.filter(cast=True)`

`good`

| invoice_id | admission_date | discharge_date | received_at | amount |
|---|---|---|---|---|
| str | date | date | datetime[µs] | decimal[*,0] |
| "002" | 2025-01-05 | 2025-01-07 | 2025-01-08 00:00:00 | 200 |
| "003" | 2025-01-01 | 2025-01-01 | 2025-01-02 00:00:00 | 400 |

`failure.invalid()`

| invoice_id | admission_date | discharge_date | received_at | amount |
|---|---|---|---|---|
| str | date | date | datetime[µs] | decimal[*,0] |
| "001" | 2025-01-01 | 2025-01-04 | 2025-01-05 00:00:00 | 0 |


TRUSTING YOUR DATA FRAME CONTENTS

# “Typed” Data Frames

`.validate()` and `.filter()` return “typed” data frames instead of a plain `pl.DataFrame`

```python
df_valid: dy.DataFrame[InvoiceSchema] = InvoiceSchema.validate(invoices)
```

**Pure** typing construct
→ no special runtime type

Similar to classic type hints:

- Code is **much** easier to understand/reason about
- mypy helps to ensure data frames with correct type are passed

```python
def build_feature(
  invoices: dy.DataFrame[InvoiceSchema],
) -> dy.DataFrame[FeatureSchema]: ...
```

**“Design by contract”**
Express pre-/post-conditions in schemas

# Defining a second Schema

```python
class DiagnosisSchema(dy.Schema):
    invoice_id = dy.String(primary_key=True)
    diagnosis_code = dy.String(primary_key=True, regex=r"^[A-Z][0-9]{2,4}$")
    is_main = dy.Bool(nullable=False)
```

# Defining a second Schema

`@dy.rule()` can *also* be used to express constraints across *rows*

```
class DiagnosisSchema(dy.Schema):
    invoice_id = dy.String(primary_key=True)
    diagnosis_code = dy.String(primary_key=True, regex=r"^[A-Z][0-9]{2,4}$")
    is_main = dy.Bool(nullable=False)

    @dy.rule(group_by=["invoice_id"])
    def exactly_one_main_diagnosis() -> pl.Expr:
        return pl.col("is_main").sum() == 1
```

# Defining a group of data frames

*Collections* allow to define interdependent groups of data frames

```python
class HospitalClaims(dy.Collection):
    invoices: dy.LazyFrame[InvoiceSchema]
    diagnoses: dy.LazyFrame[DiagnosisSchema]
```

# Defining a group of data frames

`@dy.filter()` can be used to express constraints across collection members

```
class HospitalClaims(dy.Collection):
    invoices: dy.LazyFrame[InvoiceSchema]
    diagnoses: dy.LazyFrame[DiagnosisSchema]

    @dy.filter()
    def at_least_one_diagnosis_per_invoice(self) -> pl.LazyFrame:
        return self.invoices.join(
            self.diagnoses.select(pl.col("invoice_id").unique()),
            on="invoice_id",
            how="inner",
        )
```

Returns all shared primary keys that ought to be kept!

# Validating & Filtering Collections

Just like schemas, collections can be validated and filtered

```python
invoices: pl.DataFrame
diagnoses: pl.DataFrame
inputs = {"invoices": invoices, "diagnoses": diagnoses}

claims: HospitalClaims = HospitalClaims.validate(inputs)
claims, failure = HospitalClaims.filter(inputs)
# `failure` is `TypedDict` with failure info for "invoices" and "diagnoses"
```

# Schemas 🤝 Unit Tests

Easily sample data that adheres to a schema or collection

**"Stubs"**: Empty data frames with valid schema

```
InvoiceSchema.create_empty()
```

**Random data**: Automatic generation even with custom checks via fuzzy sampling

```
df = InvoiceSchema.sample(10)
df.shape
# >>> (10, 5)
```

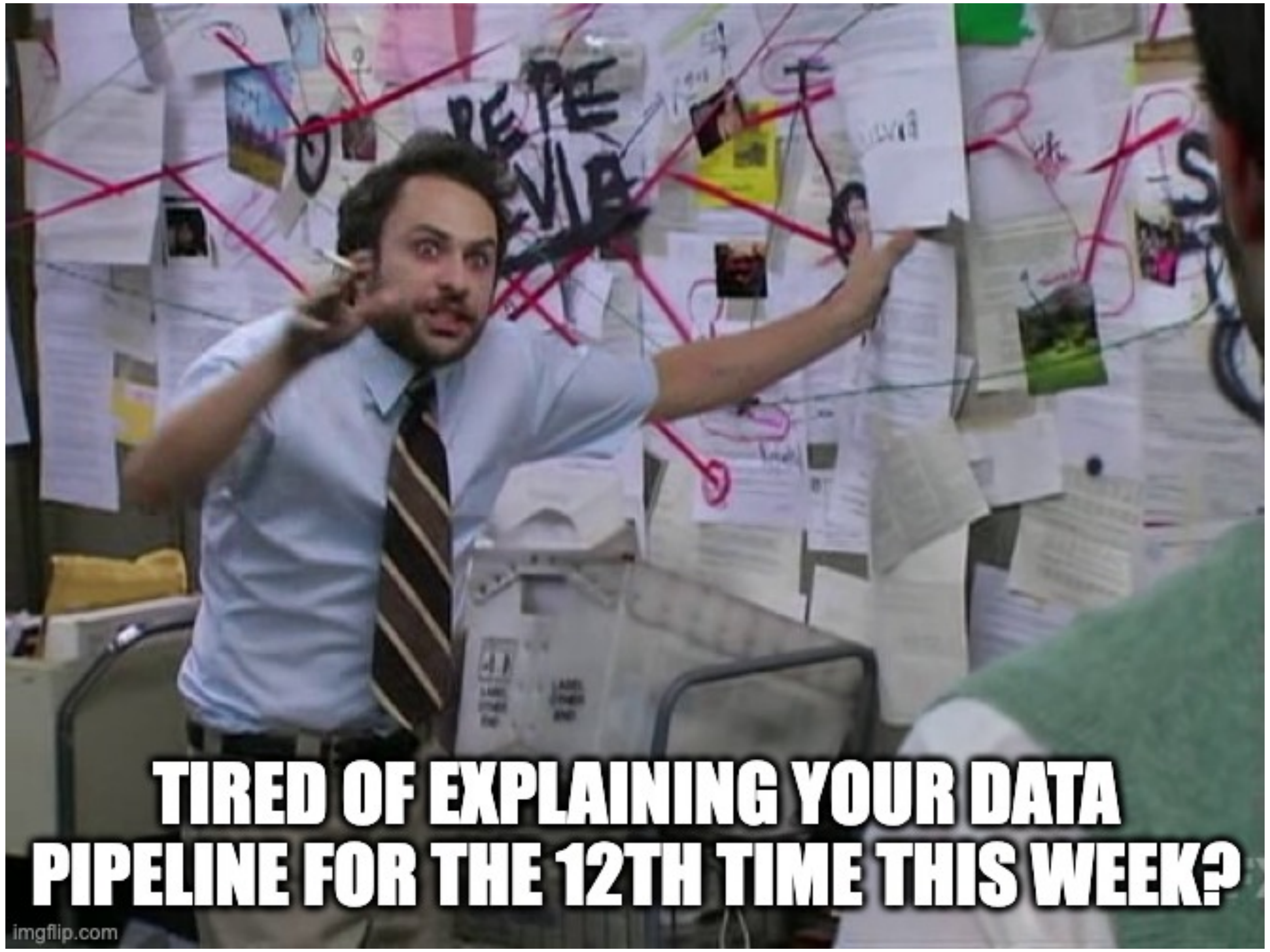
**Value overrides**: Sample with user-defined values

```
df = InvoiceSchema.sample(
  overrides={
    "amount": [100, 500, 1000]
  }
)
df.shape
# >>> (3, 5)
```

*Stubs*, *random data* and *value overrides* are also supported for **collections**!

# Real-World Experience

- 🚀 Validating data frames has greatly improved **legibility** and **robustness** of our code
- 📝 **Statically typed APIs** define contracts that increase code correctness and quality
- 🔎 Filtering has made **introspection** of **pipeline failures** more efficient & effective
- 🧪 Setting up sample data for **unit tests** has become *much* easier




**Check out dataframely…**

…and let us know what you think! 🎉

`github.com/Quantco/dataframely`

# Schemas 🤝 SQL

Easily create SQL tables with appropriate data types for a schema

```
class DiagnosisSchema(dy.Schema):
    invoice_id = dy.String(primary_key=True)
    diagnosis_code = dy.String(primary_key=True, regex=r"^[A-Z][0-9]{2,4}$")
    is_main = dy.Bool(nullable=False)
```

You can attach custom metadata for more complex logic

Inferred automatically!

```
import sqlalchemy as sa

table = sa.Table(
    "diagnosis",
    sa.MetaData(),
    *DiagnosisSchema.sql_schema(dialect=engine.dialect),
)
```

```
CREATE TABLE diagnosis (
    invoice_id     TEXT,
    diagnosis_code VARCHAR(5),
    is_main        BOOLEAN,

    PRIMARY KEY (invoice_id, diagnosis_code)
)
```