NEXT-GENERATION ETL

PyDataFlow

A declarative, method-chaining engine that bridges the gap between the intuition of visual ETL designers and the raw power of Python.

Headless Design

Mimics the logic of visual tools like Alteryx without the GUI overhead. Build strictly linear, easy-to-audit pipelines.

Fluent Syntax

Readable, English-like method chaining. Self-documenting code that describes *what* to do, not just *how*.

Production Ready

Built on Pandas for speed, designed for extensibility. Handles CSV, Excel, and Parquet with unified error handling.

THE PHILOSOPHY

Strictly Linear

Data engineering is messy. Your code shouldn't be. By enforcing a strict, immutable method-chaining pattern, PyDataFlow eliminates "spaghetti code" side effects and makes complex pipelines instantly readable.

SOURCE		LOGIC		SINK
`.input_data()`	→	`.transform()`	→	`.output_data()`

THE TRANSLATION LAYER

Syntax Mapping (Core)

CONCEPT	PYDATAFLOW	PYTHON / PANDAS
Horizontal Merge	`.join()`	`pd.merge()`
Vertical Stack	`.union()`	`pd.concat()`
Aggregation	`.summarize()`	`.groupby().agg()`

THE TRANSLATION LAYER

Syntax Mapping (Reshaping)

CONCEPT	PYDATAFLOW	PYTHON / PANDAS
Long to Wide	**.cross_tab()**	.pivot_table()
Wide to Long	**.transpose()**	.melt()
Ingestion	**.input_data()**	pd.read_csv / _excel

THE FRAMEWORK

PyDataFlow

Headless ETL for Python. A fluent, method-chaining engine that bridges the gap between visual transformation tools and raw pandas scripting.

Pattern: Fluent Interface

```
# A Complete Linear Workflow
(
    PyDataFlow.input_data("source.csv")
    .filter("Region == 'North'")
    .formula("Total", "Qty * Price")
    .output_data("report.xlsx")
)
```

Feature: Method Chaining

— ARCHITECTURAL PATTERN

The Split Problem

Strict linear pipelines cannot split in memory. To create branches (e.g., separating "High Value" vs "Low Value" flows), we use the **Materialization Pattern**.

1. **Save** the common state to a file.
2. **Start** two new chains reading that file.

```
# 1. Main Pipeline & Checkpoint
stream.process().output_data("checkpoint.parquet")
```

checkpoint.parquet

```
# 2. Branch A
input_data("checkpoint")
.filter("Region='North'")
.output("North.csv")
```

```
# 3. Branch B
input_data("checkpoint")
.filter("Region='South'")
.output("South.csv")
```

primary.csv

secondary.xlsx

— ARCHITECTURAL PATTERN

Reuniting Flows

Bring isolated datasets back together. Use **Inline Ingestion** to load a secondary file directly into a merge primitive, keeping the syntax linear and clean.

> Supports both **Horizontal Enrichment** (Join) and **Vertical Stacking** (Union).

```
# Load Primary → Merge Secondary

(
    PyDataFlow.input_data("primary.csv")

# The secondary file is ingested inside the join

    .join(
        other="secondary.xlsx",
        on="CustomerID",
        type="left"
    )
```

— ARCHITECTURAL PATTERN

Vertical Stacking

Consolidate fragmented data. After branching logic separates data into "North" and "South" files, use a **Union** to stack them back into a single master dataset.

> Essential for **Monthly Consolidations** and **Reuniting Split Flows**.

branch_north.csv

branch_south.csv

```
# Ingest North → Stack South

(
    PyDataFlow.input_data("branch_north.csv")

# The south branch is appended to the bottom

    .union(
        other="branch_south.csv"
    )

    .output_data("consolidated_regions.parquet")
)
```

ENTRY PRIMITIVE

.input_data()

The pipeline entry point. Ingests raw data from CSV, Excel, or Parquet files, initializing the PyDataFlow engine.

filepath: "sales_2024.csv"

```
# Start Flow with CSV
PyDataFlow.input_data(
    filepath="sales_2024.csv"
)
```

EXIT PRIMITIVE

.output_data()

The pipeline finale. Commits your transformed DataFrame to a permanent file on disk. Supports CSV, Excel, JSON, and Parquet.

filepath: "final_results.xlsx"

```
# Save to Excel
.output_data(
    filepath="final_results.xlsx"
)
```

DEBUGGING PRIMITIVE

.sample()

Instantly reduce dataset size for rapid testing. Keeps the top N rows to verify logic without processing millions of records.

n: 50

```
# Preview Top 50 Rows
.sample(
    n=50
)
```

ORDERING PRIMITIVE

.sort()

Organize your data sequence. Reorder rows based on the values in one or more columns. Essential for ranking, Pareto analysis, and time-series view.

columns: ["Date", "Sales"]

```
# Sort by Date (Asc) then Sales (Desc)
.sort(
    columns=["Date", "Sales"],
    ascending=[True, False]
)
```

ascending: [True, False]

COLUMN LOGIC PRIMITIVE

.select()

Refine your dataset structure. Pick specific columns to retain and rename specific fields to clearer business terms.

keep: ["ID", "Amt"]

```
# Clean up dataset
.select(
    keep=["OrderID", "Amt"],
    rename={"Amt": "Revenue"}
)
```

rename: {"Amt": "Rev"}

ROW LOGIC PRIMITIVE

.filter()

Precision data selection. Retain only the rows that match your specific boolean criteria, removing noise and focusing on the signal.

expression: "Logic String"

```
# Keep High Value Orders in North
.filter(
    expression="Region == 'North' and Sales > 100"
)
```

AGGREGATION PRIMITIVE

.summarize()

Transform granular transactions into meaningful insights. Group data by dimensions and calculate metrics like sums, averages, or counts.

group_by: ["Region"]

```
# Calculate Total Revenue by Region
.summarize(
    group_by=["Region"],
    aggregations={"Revenue": "sum"}
)
```

aggregations: {"Revenue": "sum"}

DEDUPLICATION PRIMITIVE

.unique()

Removes duplicate rows based on specific columns. Ensures data integrity by keeping only the first instance of a record.

columns: ["CustomerID", "Date"]

```
# Remove Duplicate Orders
.unique(
    columns=["CustomerID", "OrderDate"]
)
```

DATE LOGIC PRIMITIVE

.datetime_parse()

Converts a string column into a proper Date object, enabling sorting, date math, and time-series analysis.

col: "OrderDate"

```
# Convert Strings to Dates
.datetime_parse(
    col="OrderDate",
    format="%Y-%m-%d"
)
```

format: "%Y-%m-%d"

TIME LOGIC PRIMITIVE

.datetime_diff()

Calculates the difference between two dates in specific units (e.g., days).
Essential for calculating Lead Times and performing OTIF analysis.

start_col: "OrderDate"

```
# Calculate Days to Ship
.datetime_diff(
    new_col="Lead_Time",
    start_col="OrderDate",
    end_col="ShipDate",
    unit="days"
)
```

end_col: "ShipDate"

ANALYTICS PRIMITIVE

.running_total()

Calculates the cumulative sum of a column. The total accumulates row-by-row and resets when the group changes. Essential for Pareto and ABC analysis.

group_by: ["Year"]

```
# Cumulative Revenue by Year
.running_total(
    group_by=["Year"],
    column="Revenue",
    rename="Cumulative_Rev"
)
```

rename: "Cumulative_Rev"

TRANSFORMATION PRIMITIVE

.formula()

The mathematical engine of PyDataFlow. Calculate new insights using Python-style string expressions.

col: "Profit"

```
# Calculate Profit Margin
.formula(
    "Profit",
    "Revenue * (1 - Margin_Cost)"
)
```

expr: Calculation

BLENDING PRIMITIVE

.union()

Stack rows vertically. Append data from another stream onto the bottom of the current dataset, ideal for consolidating monthly files.

other: "february_sales.csv"

```
# Stack February Data
.union(
    other="february_sales.csv"
)
```

BLENDING PRIMITIVE

.join()

Merge disparate datasets into a unified view. Horizontally combine streams based on shared keys, just like a SQL JOIN.

other: "customers.csv"

```
# Enrich Sales with Customer Details
.join(
    other="customers.csv",
    on="CustomerID",
    type="left"
)
```

type: "left"

RESHAPING PRIMITIVE

.cross_tab()

Pivots data from Long (vertical) to Wide (horizontal). Turns unique values in a column into new headers.

header_col: "Month"

```
# Pivot: Accounts vs Months
.cross_tab(
    group_by=["Account"],
    header_col="Month",
    value_col="Amount"
)
```

value_col: "Amount"

RESHAPING PRIMITIVE

.transpose()

Unpivot data from Wide to Long. Melts monthly columns into a clean database format, reversing the effect of a pivot table.

key_columns: ["Product"]

```
# Melt Months into Rows
.transpose(
    key_columns=["Product", "Region"]
)
```

CATEGORY 01

Input / Output

The essential bookends of every pipeline. Seamlessly ingest raw files and commit transformed results to disk. Supports CSV, Excel, and Parquet.

Ingest: .input_data()

```
# The Lifecycle of a Data Flow
PyDataFlow.input_data("raw_source.parquet")
    .... # Transformations happen here
    .output_data("clean_target.xlsx")
```

Commit: .output_data()

CATEGORY 02

Preparation

The art of refining raw data. Filter noise, curate columns, deduplicate records, and sort sequences to prepare a clean foundation for analysis.

Refine: .filter() & .unique()

```
# Clean and Curate
(
    stream
    .select(keep=["ID", "Date", "Value"])
    .filter("Value > 0")
    .unique(columns=["ID"])
    .sort(columns=["Date"])
)
```

Organize: .sort() & .select()

CATEGORY 03

Blending

Merge disparate data sources. Combine datasets horizontally (enriching rows) or vertically (stacking periods) to create a unified view of the business.

Merge: .join()

```
# Enrich and Append
(
    stream
    .join(other="budget.csv", on="Dept", type="left")
    .union(other="prev_year_data.csv")
)
```

Stack: .union()

CATEGORY 04

Calculation

Derive new insights. Perform mathematical operations, logical evaluations, and handle complex date arithmetic to transform raw signals into business metrics.

Logic: .formula()

```
# Business Logic

stream
.formula(
    col="Total",
    expr="Qty * Price"
)
.formula(
    col="Flag",
    expr="'Yes' if Total > 100 else 'No'"
)
```

```
# Time Calculations

stream
.datetime_parse(
    col="OrderDate"
)
.datetime_diff(
    new_col="Days_Late",
    start="OrderDate",
    end="ShipDate"
)
```

Time: .datetime_parse() & .diff()

CATEGORY 05

Aggregation

Compress raw transactions into actionable intelligence. Group data by dimensions to calculate sums, averages, counts, and cumulative running totals.

Group: .summarize()

```
# Analytics & Metrics

stream
.summarize(
    group_by=["Region"],
    aggregations={
        "Sales": "sum",
        "Qty": "mean"
    }
)
```

```
# Cumulative Growth

stream
.running_total(
    group_by=["Year"],
    column="Revenue",
    rename="Total_Run"
)
```

Accumulate: .running_total()

CATEGORY 06

Reshaping

Change the shape of your data. Pivot "Long" database rows into "Wide" financial reports, or unpivot wide Excel inputs back into clean database format.

Pivot: .cross_tab()

```
# Create P&L Report

stream
.cross_tab(
    group_by=["Account"],
    header_col="Month",
    value_col="Amount"
)
```

```
# Normalize Budget

stream
.transpose(
    key_columns=[
        "Product",
        "Region"
    ]
)
```

Melt: .transpose()

Ready to Build

You now possess the complete grammar of data engineering. Combine these 16 primitives to solve any transformation challenge.

> START CODING

THANK YOU

Questions?

Feel free to ask any questions about the architecture or implementation.

Zbigniew Galar

zbigniew.galar@gmail.com