tytable

Typst tables from Polars DataFrames

Documentation

Version 0.1.dev1+g05e5b9ba7

05e5b9b · built 2026-07-16

Contents

1 Introduction

tytable is a small Python library that turns Polars DataFrames into Typst tables. It is inspired by R's tinytable package and mirrors most of its styling power, including image and sparkline support plus a Jupyter HTML preview.

1.1 What is Typst?

Typst is a markup-based typesetting system: you write a plain-text `.typ` file containing prose, headings, equations, figures, and layout instructions, then compile it into a PDF. It fills a role similar to LaTeX, with a compact modern syntax and a fast compiler. A tiny Typst document looks like this:

```
#set page(paper: "a4")
= Results

The experiment completed successfully.
```

Typst is excellent at page layout, but an analysis usually starts somewhere else: in Python, with a Polars DataFrame produced by a query, aggregation, or model. Rewriting that data by hand as Typst table cells is repetitive and error-prone. It also leaves you to solve escaping, number formatting, spanning headers, footnotes, image paths, and consistent styling over and over.

1.2 Why tytable exists

Tytable is the bridge between those two jobs:

Polars		**tytable**		**Typst**
data and computation	→	table intent	→	document and PDF

Python continues to own the data. Typst continues to own the document. Tytable only describes how the DataFrame becomes a table: format these values, style those cells, add these groups, and emit a self-contained Typst fragment. The same object can also render HTML for a notebook preview or ASCII for a terminal.

The core idea is to record *intent* by chaining methods. Tytable replays that intent at render time, after the final table shape is known. This keeps the Python code readable and avoids generating Typst markup by hand.

Install the latest release from PyPI:

```
uv add tytable
```

The equivalent pip command is `pip install tytable`. Install the optional images extra when generating plots or sparklines. Existing image files can be embedded without additional Python dependencies:

```
uv add "tytable[images]"
```

2 Start here: your first table

Begin with a Polars DataFrame and pass it to `tt`. That is enough to make a table—there is no style configuration to learn first.

SOURCE

```python
"""The smallest complete tytable example."""

import polars as pl

from tytable import tt

data = pl.DataFrame(
    {
        "Product": ["Widget", "Gadget", "Gizmo"],
        "Price": [9.99, 14.50, 3.25],
        "In stock": [120, 0, 54],
    }
)

table = tt(data)
table.save("build/01_basic.typ")
```

RESULT

Product	Price	In stock
Widget	9.99	120
Gadget	14.5	0
Gizmo	3.25	54

In Jupyter, the last two lines can simply be:

```python
table = tt(data)
table
```

Jupyter shows an HTML preview. In a script, `.save("catalog.typ")` writes the Typst fragment shown above. Saving does not compile a PDF; install the Typst CLI before using `typst compile` locally.

2.1 Make one column easier to read

Now add one formatting decision. Column names are used directly, so this reads as "show Price with two decimal places":

```python
table = tt(data).fmt(j="Price", digits=2)
```

2.2 Add one visual cue

Styling is another chained instruction. A header treatment is a useful first one because it does not depend on the number of data rows:

```
table = (
    tt(data)
    .fmt(j="Price", digits=2)
    .style(i="header", bold=True, background="#17324d", color="white")
)
```

2.3 Put it in a report

Add figure metadata when the table becomes part of a larger document, then save it:

SOURCE

```
"""A first report-ready table: format, style, caption, and label."""

import polars as pl

from tytable import tt

data = pl.DataFrame(
    {
        "Product": ["Widget", "Gadget", "Gizmo"],
        "Price": [9.99, 14.50, 3.25],
        "In stock": [120, 0, 54],
    }
)

(
    tt(data, caption="Product catalog", label="product-catalog")
    .fmt(j="Price", digits=2)
    .style(i="header", bold=True, background="#17324d", color="white")
    .save("build/tables/catalog.typ")
)
```

RESULT — ready for the report

Product	Price	In stock
Widget	9.99	120
Gadget	14.50	0
Gizmo	3.25	54

Table 2: Product catalog

The generated file can be `#include`-ed in a Typst report. Refer to the numbered figure there with `@product-catalog`.

2.4 The storyline from here

The rest of this guide grows that first table one concern at a time:

1. understand rows, columns, and chaining;
2. format values and rename display headers;
3. add styling, groups, themes, and layout;
4. embed plots and assemble a complete report table;

5. turn the same chain into a reusable, typed Python component;
6. use the task-oriented API reference to find the method for a job.

3 Core concepts

Four conventions make tytable chains predictable before you begin combining formatting, styling, and grouping directives.

3.1 Row indexing is 0-based

`i=0` is the first *data* row (the row **after** the column-name header). Use `i="header"` for the column-name row and `i="body"` for every table-body row. The explicit `"body"` selector is useful when you want to leave headers unchanged. Negative integers select column-group header rows (`-1` is the innermost row, immediately above the column-name header; increasingly negative values move upward). Row-group separator rows are addressed with `i="groupi"`, column-group rows with `i="groupj"`.

3.2 Select columns by name

`j="Score"` is the preferred form; `j=0` selects the first column by position. Both `i` and `j` also accept a *list* of strings or integers to target several rows or columns in one call: `j=["Q1 Rev", "Q1 Cost"]`, `i=["header", "body"]`. `i` additionally accepts Polars expressions, boolean series, and callables for data-driven selection (see the **Styling** section).

`.fmt()` and `.style()` use the intersection of `i` and `j`, so combining them targets individual cells—for example, `.fmt(i=1, j="Score", digits=1)` formats only the `Score` cell in the second data row, and `.style(i=1, j="Score",`
`bold=True)` styles only that cell.

3.3 Everything returns `self`

`.style()`, `.fmt()`, `.group()`, `.set_name()`, and the `.theme_*()` methods all return the table, so you chain them. `.theme(fn)` applies a custom theme callable. `.render()` and `.save()` are terminal.

3.4 Evaluation is lazy

Styling, formatting, grouping, and plotting are recorded as *intent* and replayed in a fixed order at render time. Row indices always refer to the final, visible table.

4 Renaming columns

`.set_name()` renames column headers for display without touching the underlying Polars `DataFrame` — the original frame is never modified. This is useful when the Polars column names are machine-friendly identifiers but you want human-readable headers in the rendered table, or when you need a header that Polars would reject as a column name (such as an empty string `""` or a duplicate).

Two calling modes:

- **Per-column**: `.set_name(j, name=...)` renames the column(s) selected by `j`. `j` follows the same selector rules as `.style()` / `.fmt()` (name, integer position, a list, or a regex with `regex=True`; see `.style()`). `name` is a single `str` (applied to every matched column) or a `list[str]` with one entry per match.
- **Full-list replace**: `.set_name(name=[...])` (omit `j`) replaces every column header at once — the list length must equal the column count.

After renaming, subsequent `j` selectors use the *new* display names; the old polars column name no longer matches. The example starts from `grp`, `val_1`, `val_2` and replaces them with `""`, `Revenue`, `Cost`

— then formats the renamed columns by their new names. Their alignment continues to follow the numeric dtypes of the underlying source columns:

SOURCE

```
"""
Renaming columns with ``.set_name()``.
"""

import polars as pl

from tytable import tt

df = pl.DataFrame(
    {
        "grp": ["North", "South", "East", "West"],
        "val_1": [12450.5, 9800.0, 15200.25, 7300.75],
        "val_2": [8100.0, 6200.0, 9900.0, 5100.0],
    }
)

(
    tt(df, caption="Renaming columns for display", width=1)
    .set_name(name=["", "Revenue", "Cost"])
    .fmt(j=["Revenue", "Cost"], digits=2)
    .style(i="header", bold=True, background="#2c3e50", color="white")
    .save("build/15_set_name.typ")
)
```

RESULT

	Revenue	Cost
North	12450.50	8100.00
South	9800.00	6200.00
East	15200.25	9900.00
West	7300.75	5100.00

Table 3: Renaming columns for display

For one-off renames at construction time, `tt(df, colnames_override={old: new})` does the same thing without a chained call.

5 Formatting

Cell values can be formatted in three complementary ways. Pick whichever suits the column, or mix them across columns in the same table.

5.1 In Polars

The most capable option: do everything *before* passing the dataframe to `tt()`. Polars expressions can round, cast numbers to strings, swap the decimal delimiter, prepend a currency symbol, add thousands separators, and fill nulls — anything Polars can express, the table will render. Here revenue

is rounded to two decimals, formatted as USD with thousands separators, and nulls filled with an em dash, all before `tt()` ever sees the data:

Because that Polars expression converts `Revenue` to a string column, the example explicitly restores right alignment with `.style(align="r")`. In contrast, `.fmt()` preserves the source dtype and therefore needs no alignment style for numeric columns.

SOURCE

```
"""Polars-first formatting — currency, thousands separators, and nulls handled before
tt()."""

import polars as pl

from tytable import tt

df = pl.DataFrame(
    {
        "Region": ["North", "South", "East", "West"],
        "Revenue": [12450.5, 9800.0, 15200.25, None],
        "Status": ["active", "active", None, "paused"],
    }
)

# Round, format as USD currency with thousands separators, and fill nulls —
# all in polars, so tt() only ever sees strings.
df = df.with_columns(
    pl.col("Revenue")
    .round(2)
    .map_elements(lambda x: f"${x:,.2f}", return_dtype=pl.Utf8)
    .fill_null("—"),
    pl.col("Status").fill_null("n/a"),
)

(
    tt(df, caption="Regional revenue (formatted in polars)")
    # Revenue is now a string column, so restore numeric-style alignment explicitly.
    .style(j="Revenue", align="r")
    .save("build/11_format_polars.typ")
)
```

RESULT

Region	Revenue	Status
North	$12,450.50	active
South	$9,800.00	active
East	$15,200.25	n/a
West	—	paused

Table 4: Regional revenue (formatted in polars)

(Tytable's per-cell Typst escaping — `escape=True` by default — still applies to whatever strings Polars produces, so characters like `$` are escaped for you.)

5.2 With `.fmt()`

For quick, in-table transforms that stay inside the `tt()` chain, without reaching back into polars:

- `digits` — fixed decimal places (`num_fmt="decimal"`), significant figures (`num_fmt="significant"`), or typeset scientific notation (`num_fmt="scientific"`)
- `replace` — replace missing/null/NaN values with a string or a `{old: new}` mapping
- `linebreak` — choose a literal input marker to replace with a native line break. For example, `linebreak="\n"` translates newline characters to a single `\` in Typst or `<br>` in HTML; use another marker such as `"|"` when that is more convenient for the source data
- `math` — typeset selected values as Typst equations
- `escape` — per-cell Typst escaping (on by default via `tt(escape=True)`)

Formatting text and equations does not require disabling safe escaping. This example treats `Formula` as Typst math and uses `|` inside `Detail` to mark where a native line break should appear:

```
df = pl.DataFrame({
    "Formula": ["x^2 + y^2", "sum_(i=1)^n i"],
    "Detail": ["first line|second line", "one line"],
})

table = (
    tt(df)
    .fmt(j="Formula", math=True)
    .fmt(j="Detail", linebreak="|")
)
```

Math mode is Typst-specific; HTML and ASCII retain the original value. Line-break replacement targets Typst and HTML, while ASCII retains the marker.

The larger example below puts the built-ins side by side: decimal, significant, and scientific numbers, missing-value replacement, and Typst math. Each `.fmt()` call targets the column that needs that particular transformation.

SOURCE

```
"""Formatting example — numeric, replacement, and Typst math formats."""

import polars as pl

from tytable import tt

df = pl.DataFrame(
    {
        "Region": ["North", "South", "East", "West"],
        "Revenue": [12450.5, 9800.0, 15200.25, None],
        "Significant": [3.14159, 0.00123, 987.654, None],
        "Scientific": [3141.59, 0.00123, 987654.0, None],
        "Equation": [
            "e^(i pi) + 1 = 0",
            "sum_(k=1)^n k = n(n+1)/2",
            "integral_0^infinity e^(-x) dif x = 1",
            "mat(1, 2; 3, 4)",
```

```
        ],
        "Status": ["active", "active", None, "paused"],
    }
)

(
    tt(df, caption="Regional revenue")
    .fmt(j="Revenue", digits=2)
    .fmt(j="Significant", digits=3, num_fmt="significant")
    .fmt(j="Scientific", digits=2, num_fmt="scientific")
    .fmt(j="Equation", math=True)
    .fmt(j="Revenue", replace={"null": "—"})
    .fmt(j="Status", replace={"null": "n/a"})
    .save("build/02_format.typ")
)
```

RESULT

Region	Revenue	Significant	Scientific	Equation	Status
North	12450.50	3.14	3.14×10^{3}	$e^{i\pi} + 1 = 0$	active
South	9800.00	0.00123	1.23×10^{-3}	$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$	active
East	15200.25	988	9.88×10^{5}	$\int_{0}^{\infty} e^{-x}\,\mathrm{d}x = 1$	n/a
West	—			$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$	paused

Table 5: Regional revenue

5.3 With `.fmt(fn=...)`

For anything the built-ins don't cover, pass a callable to `fn`. It runs *column-wise*: tytable hands it the current string values of the selected column (as a `list`) and expects a `list` of the same length back. This makes it easy to implement transforms that depend on magnitude — for example, abbreviating large numbers into a human-readable scale where `201818` becomes `"201.8 thousand"` and `2729179` becomes `"2.7 million"`:

SOURCE

```
"""Custom formatting function — abbreviate large numbers (1028 -> "1.0 thousand")."""

import polars as pl

from tytable import tt


def humanize(values: list[str]) -> list[str]:
    """Abbreviate large numbers into a human-readable scale."""

    def scale(n: float) -> str:
        if abs(n) >= 1_000_000:
            return f"{n / 1_000_000:.1f} million"
        if abs(n) >= 1_000:
            return f"{n / 1_000:.1f} thousand"
```

```
        return str(int(n))

    return [scale(float(v)) for v in values]


df = pl.DataFrame(
    {
        "City": ["Geneva", "Zurich", "Lugano"],
        "Population": [201818, 415367, 2729179],
    }
)

(
    tt(df, caption="City populations, human-readable")
    .fmt(j="Population", fn=humanize)
    .save("build/10_format_fn.typ")
)
```

RESULT

City	Population
Geneva	201.8 thousand
Zurich	415.4 thousand
Lugano	2.7 million

Table 6: City populations, human-readable

The Mizani package is the closest Python equivalent to R's `scales`. Its vectorized label callables cover currencies, percentages, scientific notation, dates, and more. Because `.fmt(fn=...)` supplies strings, a small typed adapter converts the values to numbers first. A following `.fmt(escape=True)` safely escapes symbols introduced by the external formatter when required by the output backend, such as `$` in Typst:

SOURCE

```
"""Mizani formatters — scales-style currency and percentage labels."""

import polars as pl
from mizani.labels import label_currency, label_percent

from tytable import tt

currency_label = label_currency(prefix="$", precision=0, big_mark=",")
percent_label = label_percent(precision=1)


def format_currency(values: list[str]) -> list[str]:
    """Adapt tytable's string column to Mizani's numeric label callable."""

    return list(currency_label([float(value) for value in values]))
```

```python
def format_percent(values: list[str]) -> list[str]:
    """Adapt tytable's string column to Mizani's numeric label callable."""

    return list(percent_label([float(value) for value in values]))


data = pl.DataFrame(
    {
        "Region": ["North", "South", "East", "West"],
        "Revenue": [1_284_000, 936_000, 1_512_000, 1_067_000],
        "Margin": [0.128, -0.064, 0.173, 0.091],
    }
)

(
    tt(data, caption="Labels formatted with Mizani")
    .fmt(j="Revenue", fn=format_currency)
    .fmt(j="Margin", fn=format_percent)
    .fmt(j=["Revenue", "Margin"], escape=True)
    .save("build/10_mizani.typ")
)
```

RESULT

Region	Revenue	Margin
North	$1,284,000	12.8%
South	$936,000	-6.4%
East	$1,512,000	17.3%
West	$1,067,000	9.1%

Table 7: Labels formatted with Mizani

Mizani is an optional development dependency used for this documentation example; it is not installed with tytable at runtime.

6 Styling

Styling directives control the appearance of cells and table-adjacent text without changing the underlying values.

By default, text columns and their headers are left-aligned, while columns with Polars numeric dtypes and their headers are right-aligned. Alignment is inferred from the source dtype before `.fmt()` changes the displayed text, so formatted numbers, replacement marks, currencies, and percentages stay aligned. An explicit `.style(align=...)` directive takes precedence. Column-group headers are centered.

6.1 Styling cells

Apply per-cell styling through selectors `i` (rows) and `j` (columns). Supported properties: `bold`, `italic`, `underline`, `strikeout`, `monospace`, `smallcaps`, `color`, `background`, `fontsize`, `align` (`l`/`c`/`r`), `alignv` (`t`/`m`/`b`), `indent`, `colspan`, `rowspan`, and per-side borders (`line="tblr"` in any combination, with `line_color` / `line_width`).

Any number of these properties can be combined in a single `.style()` call when they share the same `i/j` selector — e.g. `style(j="Score", align="c", background="#eee", bold=True)` is one directive rather than three separate calls. (Value formatting such as `digits` belongs to `.fmt()`, a separate pipeline, and so always needs its own call.)

Use `i="body"` when a style should apply to every table-body row while leaving the header rows alone. The example below uses it to draw the left and right borders around the body; its header has a separate style.

When `j` selects several columns, `align` and `alignv` also accept a *per-column string* with one shorthand character per selected column — e.g. `align="llr"` left-aligns the first two and right-aligns the third.

SOURCE

```
"""Styling example — typography, colour, dtype-aware alignment, and per-side
borders."""

import polars as pl

from tytable import tt

df = pl.DataFrame(
    {
        "Region": ["North", "South", "East", "West"],
        "Sales": [12450.5, 9800.0, 15200.25, 7300.75],
        "Growth": [0.12, -0.04, 0.21, 0.08],
    }
)

(
    tt(df, caption="Quarterly sales by region")
    .fmt(j="Sales", digits=2)
    .fmt(j="Growth", digits=2)
    .style(i="header", bold=True, color="white", background="#2c3e50", line="b")
    .style(i=2, bold=True, color="#27ae60")
    .style(i=1, color="#c0392b")
    .style(i=0, line="b", line_color="#bdc3c7")
    .style(i="body", line="lr", line_width=0.05)
    .save("build/03_style.typ")
)
```

RESULT

Region	Sales	Growth
North	12450.50	0.12
South	9800.00	-0.04
East	**15200.25**	**0.21**
West	7300.75	0.08

Table 8: Quarterly sales by region

6.2 Rotated headers for compact columns

Long labels can make otherwise small numeric columns unnecessarily wide. Select only those header cells with `i="header"`, rotate them, and leave the first descriptive column horizontal. `alignv="b"` pins the rotated labels to the bottom of the header row, next to the values they describe (and keeps the first header on the same baseline); `align="l"` sets the rotated labels' horizontal anchor. Explicit narrow widths then keep the numeric columns compact.

SOURCE

```python
"""Rotated headers — keep long labels over compact numeric columns."""

import polars as pl

from tytable import tt

df = pl.DataFrame(
    {
        "Model": ["Baseline", "Compact", "Ensemble"],
        "Accuracy": [0.81, 0.84, 0.87],
        "Precision": [0.78, 0.83, 0.86],
        "Recall": [0.85, 0.82, 0.88],
        "F1 score": [0.81, 0.82, 0.87],
        "Parameters (M)": [2.4, 1.1, 7.3],
    }
)

compact_columns = ["Accuracy", "Precision", "Recall", "F1 score", "Parameters (M)"]

(
    tt(
        df,
        caption="Rotated labels keep numeric columns compact",
        width=["3cm", "1.25cm", "1.25cm", "1.25cm", "1.25cm", "1.25cm"],
    )
    .fmt(j=compact_columns, digits=2)
    .style(i="header", bold=True, align="l", alignv="b", line="b")
    .style(i="header", j=compact_columns, rotate=-55)
    .style(i="body", j=compact_columns, align="c")
    .save("build/03_rotated_headers.typ")
)
```

RESULT

Model	Accuracy	Precision	Recall	F1 score	Parameters (M)
Baseline	0.81	0.78	0.85	0.81	2.40
Compact	0.84	0.83	0.82	0.82	1.10
Ensemble	0.87	0.86	0.88	0.87	7.30

Table 9: Rotated labels keep numeric columns compact

6.3 Caption and notes

The special selectors `i="caption"` and `i="notes"` style the table caption and footnotes. These are not grid cells, so the styling is applied as inline text markup — Typst `text(...)` / `#strong[...]` / `#smallcaps[...]`, or HTML `<span>` plus `<b>` / `<i>` / … — rather than through the cell style grid. This mirrors R tinytable's `style_tt(i="caption", …)` / `i="notes", …`.

Typst output is wrapped in a `figure` by default. Pass `label="product-scores"` to attach `<product-scores>` to that figure, then reference the numbered table with `@product-scores` in the surrounding Typst document. Pass `figure=False` when an unnumbered table without figure semantics is more appropriate. Because captions and numbered labels are figure features, combining `figure=False` with either `caption` or `label` raises `ValueError`.

```
(
    tt(
        df,
        caption="Product scores",
        label="product-scores",
        notes=["Source: Q3 report"],
    )
    .style(i="caption", bold=True, color="#c0392b", fontsize=1.2)
    .style(i="notes", italic=True, color="blue", align="c")
)
```

The text-level properties apply: `bold`, `italic`, `underline`, `strikeout`, `monospace`, `smallcaps`, `color`, `fontsize` (plus `align`, `background`, and `indent` for notes). Use `output=` to restrict styling to one backend, e.g. `output=("typst",)`.

6.4 Target notes to cells

A plain string in `notes` is an unmarked footer note. Use a dictionary when a note should point back to one or more cells. Its `text` value is the footer text, while `i` selects rows and `j` selects columns. Row selectors follow the normal conventions (`0` is the first data row and `"header"` is the column-name row); column names are preferred over positions.

If no `marker` is supplied, targeted notes receive superscript numbers in note order. Set `marker` explicitly for a symbol or label such as `"*"`; the same marker appears at every selected cell and beside the footer text. For a cell target, supply `i`; omitting `j` targets every column in those rows. When both selectors are lists, tytable marks their cross-product, not pairwise row/column coordinates.

SOURCE

```
"""Targeted notes — automatic numbering, explicit markers, and list selectors."""

import polars as pl

from tytable import tt

df = pl.DataFrame(
    {
        "Region": ["North", "South", "West"],
        "Actual": [128, 117, 136],
        "Forecast": [125, 121, 136],
    }
)

(
    tt(
        df,
        caption="Quarterly revenue (USD thousands)",
        notes=[
            {
                "text": "Actual includes a one-time contract.",
                "i": 0,
                "j": "Actual",
            },
            {
                "text": "Forecast values are provisional.",
                "marker": "*",
                "i": [0, 1, 2],
                "j": "Forecast",
            },
            "Source: Finance planning model.",
        ],
    )
    .theme_striped()
    .style(i="header", bold=True)
    .style(i="notes", italic=True, color="#52606d")
    .save("build/04_targeted_notes.typ")
)
```

RESULT

Region	Actual	Forecast
North	128[1]	125[*]
South	117	121[*]
West	136	136[*]

[1] *Actual includes a one-time contract.*

[*] *Forecast values are provisional.*

Source: Finance planning model.

Table 10: Quarterly revenue (USD thousands)

6.5 List selectors

`i` and `j` accept a list of strings as well as integers, so you can name several rows or columns in one call without repeating yourself. A list-of-strings `j` selector like `j=["Revenue", "Cost", "Growth %"]` is self-documenting and resilient to column reordering — no need to track integer positions.

The same works for `i`: `i=["header", "body"]` styles the column-name row and every data row in a single directive.

SOURCE

```
"""
List selectors — targeting multiple rows and columns by name in one call.

``i`` and ``j`` both accept a list of strings, so you can apply the same
formatting or styling to several columns or rows without repeating yourself.
Unlike integer lists, string lists read as plain-language descriptions:
``j=["Revenue", "Cost"]`` instead of ``j=[0, 2]``.

This example uses list-of-strings ``j`` selectors to format several numeric
columns in one call, and a list-of-strings ``i`` selector to make both the
column-header row and all data rows bold at once. The numeric columns are
right-aligned automatically from their Polars dtypes.
"""

import polars as pl

from tytable import tt

df = pl.DataFrame(
    {
        "Product": ["Widget", "Gadget", "Gizmo", "Thing"],
        "Revenue": [12450.5, 9800.0, 15200.25, 7300.75],
        "Cost": [8100.0, 6200.0, 9900.0, 5100.0],
        "Growth %": [12.3, -3.1, 18.7, 5.2],
    }
)

(
    tt(df, caption="List selectors — strings as row/column targets", width=1)
    .style(i=["header", "body"], bold=True)
    .fmt(j=["Revenue", "Cost"], digits=0)
    .fmt(j=["Growth %"], digits=1)
    .style(i="header", background="#2c3e50", color="white")
    .save("build/13_list_selectors.typ")
)
```

RESULT

Product	Revenue	Cost	Growth %
Widget	**12450**	**8100**	**12.3**
Gadget	**9800**	**6200**	**-3.1**
Gizmo	**15200**	**9900**	**18.7**
Thing	**7301**	**5100**	**5.2**

Table 11: List selectors — strings as row/column targets

6.6 Data-driven row selectors

Instead of hard-coding row numbers, select rows by value. `i` accepts three dynamic forms evaluated against the original DataFrame at render time:

- A *Polars expression*: `i=(pl.col("Growth %") > 0) & (pl.col("Profit") > 0)`
- A boolean `pl.Series`: `i=pl.Series("review", [False, True, False, True])`
- A Python callable: `i=lambda row: row["Profit"] < 0`

All three work with `.style()`, `.fmt()`, `.plot()`, and `.images()`. The expression or callable runs against the *original* `DataFrame` (before row-group insertion), so the column names you use are always the original ones. Polars expressions may combine any number of columns: the example marks a row green and bold only when growth is positive **and** profit is above zero. It also uses an explicit review mask and a callable that targets only the `Profit` cell in loss-making rows. South deliberately combines positive growth (`3.1`) with a loss (`-44`), demonstrating why both sides of the expression matter.

SOURCE

```
"""Data-driven selectors — style financial results without fixed row numbers.

This example demonstrates all three dynamic selector forms:
1. A multi-column Polars expression highlights profitable growth.
2. A boolean Series marks rows selected for manual review.
3. A Python callable identifies individual loss-making cells.
"""

import polars as pl

from tytable import tt

df = pl.DataFrame(
    {
        "Region": ["North", "South", "East", "West"],
        "Revenue": [1284, 936, 1512, 1067],
        "Cost": [1150, 980, 1290, 1120],
        "Growth %": [8.2, 3.1, 12.0, -4.5],
        "Profit": [134, -44, 222, -53],
    }
)

(
    tt(df, caption="Regional performance review", width=1)
    .fmt(j=["Revenue", "Cost", "Profit"], digits=0)
    .fmt(j="Growth %", digits=1)
```

```
    .style(i="header", bold=True, background="#2c3e50", color="white")
    .style(
        i=(pl.col("Growth %") > 0) & (pl.col("Profit") > 0),
        bold=True,
        color="#087e8b",
    )
    .style(i=pl.Series("review", [False, True, False, True]), background="#fff4cc")
    .style(i=lambda row: row["Profit"] < 0, j="Profit", bold=True, color="#b23a48")
    .save("build/14_data_driven.typ")
)
```

RESULT

Region	Revenue	Cost	Growth %	Profit
North	**1284**	**1150**	**8.2**	**134**
South	936	980	3.1	**-44**
East	**1512**	**1290**	**12.0**	**222**
West	1067	1120	-4.5	**-53**

Table 12: Regional performance review

7 Grouping

Grouping adds visual hierarchy by placing spanning labels above related columns or separator labels between related data rows.

7.1 Column groups

Column groups add *spanning header rows* above the regular column names, so you can label clusters of related columns. The simplest way is to pass an explicit delimiter: `.group(delimiter="_")` splits every column name on that string and turns the shared prefix into a group. In the example below the dataframe has four columns named `Q1_revenue`, `Q1_cost`, `Q2_revenue`, and `Q2_cost`; the underscore split yields two groups — `Q1` spanning the first two columns and `Q2` spanning the last two. For full control you can instead pass a dict mapping each label to its column positions, e.g. `.group(j={"Group A": [0, 1], "Group B": [2, 3]})`. These spanning header rows are then addressable through the special selector `i="groupj"`, for example to style every column-group label in one call.

7.2 Row groups

Row groups insert a *labelled separator row* before a given data row, visually breaking the table into sections. Pass `i` as a `{label: row}` dict where `row` is the 0-based data row the divider should precede. The example calls `.group(i={"Division B": 1})` to place a “Division B” divider in front of the second data row. That separator row is then addressable through the special selector `i="groupi"` — used here to render its label bold on a light grey background.

SOURCE

```
"""Grouping example — spanning column headers and row-group separator rows."""

import polars as pl
```

```python
from tytable import tt

df = pl.DataFrame(
    {
        "Q1_revenue": [100, 200],
        "Q1_cost": [40, 80],
        "Q2_revenue": [120, 220],
        "Q2_cost": [50, 90],
    }
)

(
    tt(df, caption="Half-year financials")
    .group(delimiter="_")
    .group(i={"Division B": 1})
    .style(i="groupi", bold=True, background="#f0f0f0")
    .save("build/04_group.typ")
)
```

RESULT

Q1		Q2	
revenue	cost	revenue	cost
Q1_revenue	Q1_cost	Q2_revenue	Q2_cost
100	40	120	50
Division B			
200	80	220	90

Table 13: Half-year financials

8 Themes

Themes bundle reusable appearance and layout choices behind chainable methods. The subsections below cover styling and composition, resizing, and pagination.

8.1 Styling and composition

Restrained top, header, and bottom rules give every new table the `default` booktab treatment. Add striping with `.theme_striped()` or cell borders with `.theme_grid()`. Themes stack, so both can be combined with the default rules and the layout themes described below. Explicit `.style()` calls retain precedence over the deferred default and striped styles.

`.theme_empty()` is the escape hatch for a blank slate. It clears all themes, styles, formats, prepare hooks, and theme-level Typst options recorded before it, so call it immediately after `tt(...)` and before adding anything that should remain:

```python
table = tt(df).theme_empty().fmt(j="Score", digits=2).style(i="header", bold=True)
```

Use `.theme(custom_theme)` for a custom callable accepting the table and returning it (or `None`). The public `THEMES` registry remains available for discovery and advanced composition, but normal application code should prefer the typed methods.

The gallery compares the default, stacked striped and grid treatments, and the unstyled result:

SOURCE

```python
"""Themes gallery — every built-in theme applied to the same data."""

import polars as pl

from tytable import tt

df = pl.DataFrame(
    {
        "City": ["Berlin", "Paris", "Rome", "Madrid"],
        "Pop. (M)": [3.85, 2.16, 2.87, 3.32],
        "Area (km²)": [891, 105, 1285, 604],
    }
)

tt(df, caption="default theme").save("build/05_theme_default.typ")
tt(df, caption="default + striped
themes").theme_striped().save("build/05_theme_striped.typ")
tt(df, caption="default + grid themes").theme_grid().save("build/05_theme_grid.typ")
tt(df, caption="empty theme").theme_empty().save("build/05_theme_empty.typ")
```

GALLERY — default

City	Pop. (M)	Area (km²)
Berlin	3.85	891
Paris	2.16	105
Rome	2.87	1285
Madrid	3.32	604

Table 14: default theme

GALLERY — striped

City	Pop. (M)	Area (km²)
Berlin	3.85	891
Paris	2.16	105
Rome	2.87	1285
Madrid	3.32	604

Table 15: default + striped themes

GALLERY — grid

City	Pop. (M)	Area (km²)
Berlin	3.85	891
Paris	2.16	105
Rome	2.87	1285
Madrid	3.32	604

Table 16: default + grid themes

GALLERY — empty

City	Pop. (M)	Area (km²)
Berlin	3.85	891
Paris	2.16	105
Rome	2.87	1285
Madrid	3.32	604

Table 17: empty theme

8.2 Resize

The `resize` theme scales a table to fit a target size, expressed as a fraction of the available page area. It wraps the rendered fragment in a Typst `#layout(size => …)` block that measures the table and rescales it by a uniform factor. This is useful when a wide table would otherwise overflow the text column.

Three knobs control `.theme_resize()`:

- `width` — target width as a fraction of the page content width (`1` = full width). Used unless `height` is given.
- `height` — target height as a fraction of the page content height. When set, height drives the scaling and width follows proportionally.
- `direction` — `"down"` only shrinks oversized tables, `"up"` only expands undersized ones, `"both"` (default) always scales to the target.

```
from tytable import tt

# Shrink only if wider than 95% of the page; leave smaller tables alone.
tt(df).theme_resize(width=0.95, direction="down")

# Always scale to the full page width.
tt(df).theme_resize()
```

SOURCE

```
"""Resize theme — scale a wide table down to fit the page width."""

import polars as pl

from tytable import tt

# A wide frame whose natural width would overflow the text column.
```

```
df = pl.DataFrame(
    {
        "Region": ["North", "South", "East"],
        "Product category": ["Enterprise", "Consumer", "Public sector"],
        "January revenue": [12450, 9810, 7320],
        "February revenue": [13120, 10105, 7980],
        "March revenue": [11980, 11240, 8455],
        "April revenue": [14250, 12010, 9020],
        "May revenue": [15110, 12640, 9440],
        "June revenue": [15890, 13120, 9870],
        "Forecast status": ["Above target", "On track", "Above target"],
        "Commentary": ["Strong finish", "Recovery continuing", "Growth market"],
    }
)

# direction="down" only shrinks when the table is wider than `width` of the
# page; smaller tables are left untouched. Use direction="both" to always scale.
(
    tt(df, caption='Resized to fit (width=0.95, direction="down")')
    .theme_resize(width=0.95, direction="down")
    .save("build/12_resize.typ")
)
```

RESULT

Region	Product category	January revenue	February revenue	March revenue	April revenue	May revenue	June revenue	Forecast status	Commentary
North	Enterprise	12450	13120	11980	14250	15110	15890	Above target	Strong finish
South	Consumer	9810	10105	11240	12010	12640	13120	On track	Recovery continuing
East	Public sector	7320	7980	8455	9020	9440	9870	Above target	Growth market

Table 18: Resized to fit (width=0.95, direction="down")

8.3 Multipage tables

Typst figures normally keep their contents on one page. For a long table, `.theme_multipage()` makes tytable figures breakable while preserving their caption, numbering, label, and reference semantics. The complete header block, including column-group rows, repeats at the top of each page by default:

```
tt(df, caption="Long results", label="long-results").theme_multipage()
```

Disable header repetition when the surrounding document supplies its own page context:

```
tt(df).theme_multipage(repeat_headers=False)
```

The generated Typst show rule is scoped to figures whose kind is `"tytable"`, so images and other figures later in the document retain their own page-break behaviour.

9 Column widths

The `width` parameter of `tt()` accepts several forms: a single fraction spread evenly, a per-column list of fractions, a Typst/HTML unit string, or `None` for auto. You may mix all three in one list. Pass `width=1` for a *full-width* table — the fraction is split across columns so the table fills the available content width (e.g. `width=0.5` covers half).

SOURCE

```
"""Column-width example — fractions, fixed units, and auto sizing."""

import polars as pl

from tytable import tt

df = pl.DataFrame(
    {
        "Metric": ["Revenue", "Cost", "Profit", "Margin"],
        "Value": [12450.5, 9800.0, 2650.5, 0.21],
        "Note": [
            "Q3 total, excludes rebates",
            "Fixed + variable",
            "After tax",
            "Ratio of profit to revenue",
        ],
    }
)

(
    tt(df, caption="Mixed column widths", width=[0.2, 0.15, None])
    .fmt(j="Value", digits=2)
    .save("build/06_widths.typ")
)
```

RESULT

Metric	Value	Note
Revenue	12450.50	Q3 total, excludes rebates
Cost	9800.00	Fixed + variable
Profit	2650.50	After tax
Margin	0.21	Ratio of profit to revenue

Table 19: Mixed column widths

Pin the first column to a fixed Typst length and let the rest share the remaining space with `1fr`, so the table spans the full content width while the label column stays fixed:

SOURCE

```
"""
Column-width example — full-width table with a fixed first column.

The first column is pinned to an absolute width; the remaining columns use Typst
fractional units (``1fr``) so they share whatever width is left, making the
table span the full content width.
"""

import polars as pl

from tytable import tt
```

```
df = pl.DataFrame(
    {
        "Model": ["A-100", "B-200", "C-300", "D-400"],
        "Accuracy": [0.923, 0.941, 0.918, 0.955],
        "Latency (ms)": [12.4, 9.8, 15.1, 11.0],
        "Notes": ["baseline", "best accuracy", "fastest", "recommended"],
    }
)

(
    tt(df, caption="Full width, fixed first column", width=["3.5cm", "1fr", "1fr",
"1fr"])
    .fmt(j="Accuracy", digits=3)
    .fmt(j="Latency (ms)", digits=1)
    .style(i="header", bold=True)
    .save("build/09_widths_fixed.typ")
)
```

RESULT

Model	Accuracy	Latency (ms)	Notes
A-100	0.923	12.4	baseline
B-200	0.941	9.8	best accuracy
C-300	0.918	15.1	fastest
D-400	0.955	11.0	recommended

Table 20: Full width, fixed first column

10 Images & sparklines

Use `.images()` to embed existing files and `.plot()` to generate graphics from cell values. This example adds three committed SVG country flags, then supplies a plotting function `fun(values) -> matplotlib.figure.Figure` for the sparkline column. Tytable handles generated PNG saving and path management. Only `.plot()` requires the `images` extra; `.images()` embeds existing files without additional Python dependencies.

SOURCE

```
"""
Images & sparklines example — embedding existing files and generated plots.

Requires the `images` extra:  pip install tytable[images]
"""

import matplotlib.pyplot as plt
import polars as pl

from tytable import tt
```

```
def sparkline(values, *, color="#2c3e50", **kw):
    fig, ax = plt.subplots(figsize=(2.4, 0.8), dpi=150)
    ax.plot(range(len(values)), values, color=color, lw=3, solid_joinstyle="round")
    ax.set_axis_off()
    ax.margins(y=0.15)
    fig.tight_layout(pad=0)
    return fig


df = pl.DataFrame(
    {
        "Flag": ["DE", "FR", "IT"],
        "Country": ["Germany", "France", "Italy"],
        "Score": [85.43, 72.10, 91.87],
        "Trend": [[1, 3, 2, 5, 4], [5, 4, 3, 2, 1], [1, 2, 3, 4, 5]],
    }
)

(
    tt(df, caption="Country scores with flags and trend sparklines")
    .theme_striped()
    .fmt(j="Score", digits=2)
    # Existing image paths are relative to the saved build/07_images.typ fragment.
    .images(
        j="Flag",
        paths=[
            "../assets/flags/de.svg",
            "../assets/flags/fr.svg",
            "../assets/flags/it.svg",
        ],
        height=1.2,
    )
    .plot(j="Trend", fun=sparkline, height=1.5, color="#2c3e50")
    .style(j="Score", align="c")
    .style(
        i="header",
        bold=True,
        background="#2c3e50",
        color="white",
        line="b",
        line_width=0.08,
    )
    .save("build/07_images.typ", assets="assets")
)
```

RESULT

Flag	Country	Score	Trend
	Germany	85.43	
	France	72.10	
	Italy	91.87	

Table 21: Country scores with flags and trend sparklines

11 Putting it together

A feature-rich table built without any image dependencies — combining explicit column groups, a row-group separator, numeric formatting, a full-width layout, and targeted styling. A list selector formats all numeric columns in one call; their right alignment comes automatically from their Polars dtypes.

SOURCE

```
"""
Full report example — combining groups, formatting, styling, and widths.

A feature-rich table built without any image dependencies: explicit column
groups (dict form), a row-group separator, numeric formatting, a full-width
layout, and targeted styling. Numeric alignment follows the source dtypes.
"""

import polars as pl

from tytable import tt

df = pl.DataFrame(
    {
        "Store": ["01 Downtown", "02 Midtown", "03 Harborside", "04 Eastside"],
        "Q1 Rev": [12450.5, 9800.0, 15200.25, 7300.75],
        "Q1 Cost": [8100.0, 6200.0, 9900.0, 5100.0],
        "Q2 Rev": [13100.0, 9600.0, 16050.5, 7900.0],
        "Q2 Cost": [8300.0, 6000.0, 10100.0, 5400.0],
    }
)

(
    tt(df, caption="Store performance summary", width=1)
    .group(j={"Q1": ["Q1 Rev", "Q1 Cost"], "Q2": ["Q2 Rev", "Q2 Cost"]})
    .group(i={"Coastal district": 2})
    .fmt(j=["Q1 Rev", "Q1 Cost", "Q2 Rev", "Q2 Cost"], digits=2)
    .style(i="header", bold=True, color="white", background="#2c3e50")
    .style(i="groupj", bold=True, background="#ecf0f1")
    .style(i="groupi", bold=True, background="#f0f0f0")
    .style(j="Store", bold=True)
    .style(i=0, background="#fdf2e9")
    .save("build/08_full_report.typ")
)
```

RESULT

	Q1		Q2	
Store	Q1 Rev	Q1 Cost	Q2 Rev	Q2 Cost
01 Downtown	12450.50	8100.00	13100.00	8300.00
02 Midtown	9800.00	6200.00	9600.00	6000.00
Coastal district				
03 Harborside	15200.25	9900.00	16050.50	10100.00
04 Eastside	7300.75	5100.00	7900.00	5400.00

Table 22: Store performance summary

12 Advanced Python: reusable table components

The `tt()` factory is the shortest way to create a table, while `TyTable` is the public class it returns. Import the class when a larger program needs type annotations, reusable theme functions, dependency injection, or a report component with a stable `build() -> TyTable` contract:

```
from tytable import TyTable, tt

def add_brand_style(table: TyTable) -> TyTable:
    return table.style(i="header", bold=True, background="#17324d", color="white")

table: TyTable = add_brand_style(tt(df))
```

Using `tt(df)` is still preferred to spelling `TyTable(df)` directly: the factory is the stable construction entry point, and the annotation exposes the useful class contract without coupling application code to implementation details.

12.1 A report component, end to end

The following script separates four concerns that tend to become tangled in a larger report: data preparation in Polars, design tokens, table construction, and filesystem output. The `build()` method returns a fully configured `TyTable`, so callers can preview it in a notebook, render it to a string, add a one-off style, or save it themselves.

SOURCE

```
"""Reusable report component — type-safe configuration around ``TyTable``."""

from dataclasses import dataclass
from pathlib import Path

import polars as pl

from tytable import TyTable, tt


@dataclass(frozen=True)
class ReportStyle:
    """A small design token set shared by every table in a report."""

    ink: str = "#17324d"
```

```
    accent: str = "#087e8b"
    pale: str = "#eaf6f6"
    warning: str = "#b23a48"


def apply_report_style(table: TyTable, style: ReportStyle) -> TyTable:
    """A typed theme function: accept and return the public ``TyTable`` class."""

    return (
        table.style(i="header", bold=True, color="white", background=style.ink)
        .style(i="groupj", bold=True, color=style.ink, background=style.pale)
        .style(i="caption", bold=True, color=style.ink, fontsize=1.15)
        .style(i="notes", italic=True, color="#52606d")
    )


class QuarterlyReportTable:
    """Turn a raw frame into the project's standard quarterly table."""

    def __init__(self, data: pl.DataFrame, style: ReportStyle | None = None) -> None:
        self.data = data
        self.style = style or ReportStyle()

    def build(self) -> TyTable:
        numeric = ["Revenue", "Target", "Variance"]
        table = (
            tt(
                self.data,
                caption="Quarterly performance",
                label="quarterly-performance",
                notes=["Variance is actual revenue minus target."],
                width=["3.2cm", "1fr", "1fr", "1fr"],
            )
            .group(j={"Actual": ["Revenue"], "Plan": ["Target", "Variance"]})
            .fmt(j=numeric, digits=0)
            .style(i=pl.col("Variance") < 0, j="Variance", bold=True,
color=self.style.warning)
            .style(i=pl.col("Variance") >= 0, j="Variance", bold=True,
color=self.style.accent)
        )
        return apply_report_style(table, self.style)

    def save(self, directory: Path) -> None:
        """Keep filesystem policy outside the table-building method."""

        self.build().save(str(directory / "quarterly-performance.typ"))


df = pl.DataFrame(
    {
        "Region": ["North", "South", "East", "West"],
        "Revenue": [1_284_000, 936_000, 1_512_000, 1_067_000],
```

```
        "Target": [1_200_000, 1_000_000, 1_450_000, 1_100_000],
    }
).with_columns((pl.col("Revenue") - pl.col("Target")).alias("Variance"))

QuarterlyReportTable(df).build().save("build/16_advanced_pipeline.typ")
```

RESULT

	Actual	Plan	
Region	**Revenue**	**Target**	**Variance**
North	1284000	1200000	**84000**
South	936000	1000000	**-64000**
East	1512000	1450000	**62000**
West	1067000	1100000	**-33000**

Variance is actual revenue minus target.

Table 23: **Quarterly performance**

12.2 Design rules for larger programs

- **Accept and return `TyTable` in reusable styling functions.** This is the simplest custom-theme interface and keeps those functions easy to test.
- **Keep `build()` separate from `save()`.** Building describes the table; saving decides where report artifacts belong. This also makes HTML previews and tests (`table.render("html")`) straightforward.
- **Build fresh variants when they should diverge.** Chaining methods mutate the table and return `self`. If a print version and a web version need different directives, call the component's `build()` method twice rather than trying to copy internal state.
- **Let Polars own data preparation.** Derived columns, sorting, aggregation, joins, and input validation belong before `tt(...)`; tytable should describe presentation intent.
- **Test rendered contracts at the right level.** Assert the prepared DataFrame separately, then use a small snapshot of `render("typst")` or `render("html")` for the table layer.

12.3 Developing and debugging a table

Keep the function that builds a table separate from the code that fetches data, assembles the full report, and writes production artifacts. A small preview script can then call that function with representative data, without running the rest of the application:

```
# scripts/preview_sales_table.py
import polars as pl

from reports.sales_table import build_sales_table

sample = pl.DataFrame({
    "Region": ["North", "South"],
    "Revenue": [12500.0, 9875.5],
})
table = build_sales_table(sample)
```

```
print(table)
table.save("build/sales-preview.html")
table.save("build/sales-preview.typ")
```

`print(table)` works because every `TyTable` has an ASCII representation. It is the fastest way to check the visible rows and columns, formatted values, names, row groups, notes, and alignment from a terminal. Open `build/sales-preview.html` in a browser when visual styling matters; HTML output does not require Jupyter. See Alternative backends for the capabilities and limitations of both previews.

For the authoritative Typst result, make a tiny wrapper document next to the generated fragment:

```
// build/preview.typ
#set page(paper: "a4", margin: 2.5cm)
#set text(size: 10pt)
#include "sales-preview.typ"
```

Then leave Typst watching that wrapper in one terminal:

```
typst watch build/preview.typ build/preview.pdf
```

After changing the Python builder, run the preview script again. Typst notices the regenerated fragment and recompiles the PDF. This gives a short feedback loop while still testing the real paged backend and its surrounding page width, fonts, and document settings.

12.4 Data from a web source

Polars can read a stable CSV endpoint directly, and the resulting DataFrame is no different to tytable. For reproducible reports, download or cache the input and record its retrieval date (or a content hash) instead of making every docs or CI build depend on the network:

```
from pathlib import Path
import polars as pl
from tytable import tt

cache = Path("data/indicator.csv")
if not cache.exists():
    frame = pl.read_csv(STABLE_CSV_URL)
    frame.write_csv(cache)

df = pl.read_csv(cache)
tt(df, notes=["Source: provider name; retrieved 2026-07-15"]).save("build/
indicator.typ")
```

In production, pin a versioned URL when the provider offers one and validate the expected columns before building the table. This keeps a harmless upstream schema change from silently reshaping a report.

13 Alternative backends

Typst is the primary output format, but a `TyTable` can render the same recorded intent as HTML or fixed-width terminal text. These backends are useful for development, tests, and applications that

do not need a paged document. They are independent renderers, not conversions of the generated Typst, so use a compiled Typst preview for final layout decisions.

13.1 HTML

`table.render("html")` returns an HTML table fragment as a string. Jupyter calls the same renderer automatically when a `TyTable` is the last value in a cell, but the backend does not depend on Jupyter. In a regular script, save a browser preview directly:

```
table.save("build/preview.html")
```

The HTML backend represents captions, notes, row and column groups, images, spans, alignment, and most visual cell styling. It is also convenient for web applications and snapshot tests that need inspectable markup:

```
html = table.render("html")
assert "Total" in html
```

HTML layout follows browser and CSS rules. Page-oriented Typst behavior such as multipage tables, repeated page headers, and exact Typst sizing is therefore not reproduced, and browser output should not be treated as a pixel-accurate preview of the final PDF.

13.2 ASCII terminal output

`print(table)`, `repr(table)`, and `table.render("ascii")` produce a fixed-width plain-text table. An interactive Python prompt also shows this representation when the table is the value of an expression:

```
table = build_sales_table(sample)
print(table)

text = table.render("ascii")
assert "Revenue" in text
```

Formatting, renamed columns, row groups, and horizontal alignment are preserved. Captions and notes are emitted as plain text, with targeted note markers written as `[1]`, `[*]`, and so on. Cells are limited to 60 terminal columns and truncated with an ellipsis; wide and combining Unicode characters are measured by their terminal display width.

ASCII output intentionally omits properties that plain text cannot represent reliably, including colors, backgrounds, font styles, rotation, and line styling. Column-group headers and general row or column spans are not currently represented. Use HTML for a quick visual check and compiled Typst when exact layout matters. `.save()` infers only HTML or Typst from the destination suffix; write the string returned by `.render("ascii")` yourself when a text artifact is needed.

14 Table showcase

A polished table usually needs less decoration than expected: one strong header colour, a quiet secondary band for column groups, restrained striping, right-aligned numbers, and a single highlight that communicates the decision. The dtype-aware defaults supply that numeric alignment; the example only records styles that differ from those defaults. This model scorecard combines those choices with a targeted footnote and a Typst label for cross-referencing.

SOURCE

```
"""Showcase — a publication-ready model comparison table."""

import polars as pl

from tytable import tt

df = pl.DataFrame(
    {
        "Model": ["Linear", "Forest", "Boosted", "Neural"],
        "Accuracy": [0.842, 0.913, 0.927, 0.919],
        "F1": [0.816, 0.901, 0.921, 0.914],
        "Latency": [1.2, 8.7, 4.1, 12.6],
        "Parameters": [24, 18_400, 7_800, 52_100],
    }
)

(
    tt(
        df,
        caption="Model selection scorecard",
        label="model-scorecard",
        notes=[
            {"text": "Highest validation accuracy.", "i": 2, "j": "Accuracy"},
            "Latency measured on the same CPU batch (milliseconds; lower is better).",
        ],
        width=["2.8cm", "1fr", "1fr", "1fr", "1.2fr"],
    )
    .theme_empty()
    .group(j={"Quality": ["Accuracy", "F1"], "Cost": ["Latency", "Parameters"]})
    .fmt(j=["Accuracy", "F1"], digits=3)
    .fmt(j="Latency", digits=1)
    .style(i="groupj", bold=True, color="#153243", background="#dbeff0")
    .style(i="header", bold=True, color="white", background="#153243")
    .style(j="Model", bold=True)
    .style(i=[1, 3], background="#f4f7f8")
    .style(i=2, bold=True, background="#d7f0ea", line="tb", line_color="#087e8b")
    .style(i=2, j="Accuracy", color="#087e8b")
    .style(i="caption", bold=True, color="#153243", fontsize=1.2)
    .style(i="notes", italic=True, color="#52606d")
    .save("build/17_showcase.typ")
)
```

RESULT — publication-ready scorecard

	Quality		Cost	
Model	Accuracy	F1	Latency	Parameters
Linear	0.842	0.816	1.2	24
Forest	0.913	0.901	8.7	18400
Boosted	**0.927**[1]	**0.921**	**4.1**	**7800**
Neural	0.919	0.914	12.6	52100

[1] *Highest validation accuracy.*

Latency measured on the same CPU batch (milliseconds; lower is better).

Table 24: **Model selection scorecard**

For other visual idioms, compare the sparse financial table in **Putting it together**, the built-in variants in **Themes**, and the trend column in **Images & sparklines**. Together they cover a publication table, a dense report table, and a compact dashboard table without requiring a separate rendering system.

15 Task-oriented API reference

Start here when you know the task but not the method. Methods marked **chainable** mutate the `TyTable` and return `self`; output methods are terminal.

Task	Use	Result
Create	`tt(...)`	`TyTable`
Style cells	`.style(...)`	chainable
Format values	`.fmt(...)`	chainable
Group rows/ columns	`.group(...)`	chainable
Rename headers	`.set_name(...)`	chainable
Add a built-in theme	`.theme_striped()` / `.theme_resize(...)`	chainable
Apply a custom theme	`.theme(fn)`	chainable
Add plots/images	`.plot(...)` / `.images(...)`	chainable
Post-process output	`.finalize(...)`	chainable
Get a string	`.render(...)`	`str` (terminal)
Write a file	`.save(...)`	`None` (terminal)

15.1 Selector cheat sheet

`.style()`, `.fmt()`, `.plot()`, and `.images()` share the same selectors; `.set_name()` shares the column selector. Names are exact matches by default.

Selector	Example	Meaning
`i`	`0`, `2`, `[0, 2]`	0-based data row(s)
`i`	`"header"`, `"body"`	column names or all data rows
`i`	`"groupi"`, `"groupj"`	row- or column-group header rows

Selector	Example	Meaning
`i`	`-1`, `-2`	column-group rows, from innermost upward
`i`	`pl.col("Score") > 80`	Polars expression evaluated on source data
`i`	`pl.Series(...)`	boolean mask with one value per source row
`i`	`lambda row: ...`	predicate receiving a row dictionary
`j`	`"Score"`, `0`	column name (preferred) or position
`j`	`["Revenue", "Cost"]`	several columns in one directive

Set `regex=True` on an individual method call to treat its string `j` selectors as `re.search` patterns. `.style()` also accepts `i="caption"` and `i="notes"`; those targets support text-oriented properties rather than cell spans or borders.

15.2 Creating a table

CREATE

```
tt(
    data, *, figure=True, caption=None, label=None, notes=None, width=None,
height=None,
    gutter=2, colnames=True, colnames_override=None, rownames=False, digits=None,
    escape=True, finalize=None,
) -> TyTable
```

`data` is a Polars `DataFrame` and is cloned on construction. The constructor options fall into four groups:

Concern	Options	Notes
Figure	`figure`, `caption`, `label`, `notes`	captions and labels require `figure=True`
Layout	`width`, `height`, `gutter`	`width=1` fills the line; lists set each column
Headers	`colnames`, `colnames_override`	show and rename display headers
Values	`digits`, `escape`	global numeric precision and safe markup
Behaviour	`finalize`	initial output callback
Reserved	`rownames`	present for parity; not implemented

`width` accepts a fraction, a Typst length string, or one entry per column (fractions, strings such as `"3cm"` / `"1fr"`, and `None` may be mixed). `gutter` accepts points as a number, a unit string, or `None`. A note is a string or a dictionary with `text`, `marker`, `i`, and `j` keys.

`TyTable(...)` has the same constructor options, but application code should normally construct with `tt(...)` and use `TyTable` for annotations.

15.3 Formatting and structure

STYLE

```
.style(
    i=None, j=None, *, regex=False, bold=None, italic=None, underline=None,
    strikeout=None, monospace=None, smallcaps=None, color=None, background=None,
    fontsize=None, align=None, alignv=None, indent=None, colspan=None,
rowspan=None,
    rotate=None, line=None, line_color=None, line_width=0.1, line_trim=None,
    output=None,
) -> TyTable
```

Combines any properties sharing one selector. `align` uses `l`/`c`/`r`, `alignv` uses `t`/`m`/`b`, `rotate` is degrees, and `line` is any combination of `t`/`b`/`l`/`r`. With several columns, `align="llr"` assigns one alignment per column. `fontsize`, `indent`, and `line_width` are in `em`. `output` can restrict a directive to a tuple such as `("typst",)`.

FORMAT

```
.fmt(
    i=None, j=None, *, regex=False, digits=None, num_fmt='decimal', replace=None,
    escape=False, fn=None, linebreak=None, math=False, output=None,
) -> TyTable
```

Transforms values in this order: `digits`, `fn`, `replace`, `linebreak`, `math`, then `escape`. `num_fmt` is `"decimal"`, `"significant"`, or `"scientific"`; `replace` may blank missing values, supply a replacement string, or map old values to new ones. `linebreak` is a literal marker replaced for Typst and HTML output. `math=True` wraps Typst values in math delimiters without changing HTML or ASCII. `fn` receives one column as `list[str]` and must return a list of the same length.

GROUP

```
.group(i=None, j=None, *, delimiter=None) -> TyTable
```

For row groups, pass `{label: row}` or a list with one group value per data row. For spanning column headers, pass `{label: [columns]}` as `j`, or pass a literal string as `delimiter` to split every column name. `j` and `delimiter` are mutually exclusive.

RENAME DISPLAY HEADERS

```
.set_name(j=None, *, regex=False, name) -> TyTable
```

With `j`, `name` is one display name or a list matching the selected columns. Without `j`, pass the complete list of display names. The DataFrame remains unchanged; later `j` selectors use the new display names.

CUSTOM THEME

```
.theme(fn) -> TyTable
```

`fn` is a callable `theme(table) -> TyTable | None`. Built-in themes use the typed methods below; they stack on the implicit default theme.

ADD STRIPES

```
.theme_striped() -> TyTable
```

ADD A GRID

```
.theme_grid() -> TyTable
```

RESET STYLING

```
.theme_empty() -> TyTable
```

This reset is destructive and order-sensitive. Call it immediately after `tt(...)`; constructor-level figure, row-height, and gutter settings survive.

ROTATE

```
.theme_rotate(angle=90, i=None, j=None) -> TyTable
```

With no selectors, rotates the whole table. Pass `i` and/or `j` to rotate selected cell content instead.

RESIZE

```
.theme_resize(width=1, height=None, direction='both') -> TyTable
```

Scales Typst output by width or height. `direction` is `"down"`, `"up"`, or `"both"`.

SPAN PAGES

```
.theme_multipage(*, repeat_headers=True) -> TyTable
```

Makes the Typst figure breakable. Header and column-group rows repeat on each page unless `repeat_headers=False`.

15.4 Plots and images

Install the `images` extra for both methods. Media is materialized when the table renders or saves, not when the directive is recorded.

GENERATE PLOTS

```
.plot(
    i=None, j=None, *, regex=False, fun=None, data=None, height=1.0, height_px=400,
    width_px=1200, color='black', xlim=None, output=None,
) -> TyTable
```

`j` and `fun` are required. The callable receives the typed cell value (or the matching `data` entry) and returns a Matplotlib `Figure` or `plotnine` plot. Pixel dimensions control PNG generation for both backends and override a returned Matplotlib figure's canvas size; `height` independently controls the displayed cell size.

EMBED FILES

```
.images(i=None, j=None, *, regex=False, paths=None, height=1.0, output=None) ->
TyTable
```

`j` and `paths` are required. Paths are assigned row-major across selected cells. Use `.save(..., assets=...)` to control where referenced files live.

15.5 Rendering and output

POST-PROCESS

```
.finalize(fn) -> TyTable
```

Registers `fn(rendered: str, output: str) -> str`. Callbacks run in registration order after any renderer and are useful for narrowly scoped integration markup.

RENDER STRING

```
.render(output='typst') -> str
```

`output` is `"typst"`, `"html"`, or `"ascii"`. Rendering resolves all recorded intent and runs finalizers. The same table can be rendered more than once. See Alternative backends for HTML and ASCII usage, including terminal previews with `print(table)`.

SAVE FILE

```
.save(path, assets=None) -> None
```

Creates parent directories and infers HTML from `.html` / `.htm`; other suffixes produce Typst. `assets` is relative to the output file and defaults to a sibling `tytable_assets/` directory.

16 Using a generated table in Typst

`.save()` writes a Typst content fragment, so place it in the document with `#include` (not `#import`, which is for importing named definitions):

```
// report.typ
#set page(paper: "a4", margin: 2.5cm)
#set text(size: 10pt)

= Product catalog

The current catalog is shown in @product-catalog.
```

```
#include "build/tables/catalog.typ"
```

Build the fragment first, then compile the parent document:

```
uv run python make_tables.py
typst compile report.typ report.pdf
```

The include path is relative to the `.typ` file containing the `#include`. Generated image references need a little more care: they must resolve within the *Typst project root* (the directory tree available to `typst compile`). Make the assets location explicit in Python:

```
.save("build/tables/products.typ", assets="../assets/products")
```

Images then land in `build/assets/products/` and the `.typ` references `../assets/products/...`, which resolves correctly from `build/tables/` when compiled as part of the parent. Without an explicit `assets=`, images land in a `tytable_assets/` folder next to the output file.

17 Coming from R tinytable

If you already use R's `tinytable`, tytable should feel familiar: create a table, then layer on formatting, styling, grouping, and themes. The main adjustments are Python method chaining, 0-based row indices, and selecting columns by name.

R (`tinytable`)	**Python (`tytable`)**
`tt(data)`	`tt(df)` — Polars DataFrame
`style_tt(x, ...)`	`.style(...)`
`format_tt(x, ...)`	`.fmt(...)`
`group_tt(x, ...)`	`.group(...)`
`theme_tt(x, ...)`	`.theme_striped()` / `.theme_grid()` / other theme methods
`print(x, "typst")`	`.render("typst")`
`save_tt(x, "out.typ")`	`.save("out.typ")`
`x %>% format(...) %>% ...`	`.fmt(...).style(...)` — method chain
`colnames(x) <- c(...)`	`.set_name(name=[...])`
1-based rows; 0 = colnames	**0-based** data rows; `i="header"`
column by integer position	column by **name** (preferred)