

Overview

TaDa provides a set of simple but powerful operations on rows of data.

Key features include:

- **Arithmetic expressions**: Row-wise operations are as simple as string expressions with field names
- **Aggregation**: Any function that operates on an array of values can perform row-wise or column-wise aggregation
- **Data representation**: Handle displaying currencies, floats, integers, and more with ease and arbitrary customization

Note: This library is in early development. The API is subject to change especially as typst adds more support for user-defined types. Backwards compatibility is not guaranteed! Handling of field info, value types, and more may change substantially with more user feedback.

Importing

TaDa is split into several core modules: data display, computed operations, and table manipulation:

```
#import "@preview/tada:0.1.0": ops, display, tabledata
```

- `ops` provides functions like `agg`, `chain`, `collect`, etc. as shown below.
- `display` provides functions like `format-float`, `to-tablex`, etc. that control how rendered content appears.
- `tabledata` provides the `TableData` type and functions like `transpose`, `subset`, `concat`, etc. that directly impact table rows and fields.

Table manipulation

*Note: All following examples wrap rendered content in `#output[...]` blocks. This is purely a helper function for the documentation to make sure the right side's output only renders the current example. It is **not required** in your own code.*

TaDa provides two main ways to construct tables – from columns and from rows:

```
// See `importing tada` above for where these variables
originated
#import ops: *
#import tabledata: *
#import display: *

#let column-data = (
  name: ("Bread", "Milk", "Eggs"),
  price: (1.25, 2.50, 1.50),
  quantity: (2, 1, 3),
)
#let row-data = (
  (name: "Bread", price: 1.25, quantity: 2),
  (name: "Milk", price: 2.50, quantity: 1),
  (name: "Eggs", price: 1.50, quantity: 3),
)
#let td = table-data-from-columns(column-data)
// Equivalent to:
// #let td = TableData(rows: row-data)

// Show using the `table` attribute
#output(td.table)
```

name	price	quantity
Bread	1.25	2
Milk	2.5	1
Eggs	1.5	3

Using `__index`

TaDa will automatically add an `__index` field to each row. This is useful for showing auto-incrementing row numbers and filtering operations:

```
#td.field-info.at("__index").insert("title", "#")
#td.field-info.at("__index").insert("hide", false)
#let td = TableData(..td)
#output(td.table)
```

#	name	price	quantity
0	Bread	1.25	2
1	Milk	2.5	1
2	Eggs	1.5	3

Title formatting

You can pass any `content` as a field's `title`. **Note**: if you pass a string, it will be evaluated as markup.

```
#let fmt = it => heading(outlined: false, upper(it.at(0)) +
it.slice(1))

#let titles = (
  name: (title: fmt), // as a function
  quantity: (title: fmt("Qty")), // as a string
  ..td.field-info,
)
// You can also provide defaults for any unspecified field info
#let defaults = (title: fmt)
#let td = TableData(..td, field-info: titles, field-defaults:
defaults)

#output(td.table)
```

#	Name	Qty	Price
0	Bread	2	1.25
1	Milk	1	2.5
2	Eggs	3	1.5

An important note on table rendering

Typst does not have formal mechanisms yet for object-oriented programming. So, `TableData` is simply a dictionary under the hood. **Therefore**, when calling `td.table` to render a table, it will only show the rows, field data, and tablex keywords passed on initialization of the object. If you insert additional rows or columns, you need to **recreate** a new `TableData` object before calling `.table`, otherwise the new data will not be shown. Alternatively, you can call `to-tablex` on the modified structure which will render with up-to-date information.

```
#let adjusted = td
#adjusted.rows.push((name: "Another product", price: 10.0,
quantity: 2))
// Outdated information!
#output{
  #grid(columns: (lfr, lfr)){
    Updates didn't persist!
    #adjusted.table
  }
  Instead, recreate the `TableData`
  #TableData(..adjusted).table
  // or to-tablex, but this won't auto-populate
  // the __index field
}
}
```

Updates didn't persist!				Instead, recreate the <code>TableData</code>			
#	Name	Qty	Price	#	Name	Qty	Price
0	Bread	2	1.25	0	Bread	2	1.25
1	Milk	1	2.5	1	Milk	1	2.5
2	Eggs	3	1.5	2	Eggs	3	1.5
				3	Another product	2	10

Value formatting

type

Type information can have attached metadata that specifies alignment, display formats, and more. Available types and their metadata are: (

```
string: (default-value: "", display: eval),
float: (display: auto, align: right),
integer: (display: auto, align: right),
percent: (display: format-percent, align: right),
index: (align: right),
```

). While adding your own default types is not yet supported, you can simply defined a dictionary of specifications and pass its keys to the field

```
#let fmt-currency(val) = {
  let sign = if val < 0 {str.from-unicode(0x2212)} else {""}
  let currency = "$"
  [sign#currency]
  format-float(calc.abs(val), precision: 2, pad: true)
}
#let currency-info = (display: fmt-currency, align: right)
#td.field-info.insert("price", (type: "currency"))
#let td = TableData(..td, type-info: ("currency": currency-
info))
#output(td.table)
```

#	Name	Qty	Price
0	Bread	2	\$1.25
1	Milk	1	\$2.50
2	Eggs	3	\$1.50

Transposing

`transpose` is supported, but keep in mind if columns have different types, an error will be a frequent result. To avoid the error, explicitly pass `ignore-types: true`. You can choose whether to keep field names as an additional column by passing a string to `fields-name` that is evaluated as markup:

```
#output{
  #transpose(td, ignore-types: true, fields-name: "").table
}
```

	0	1	2
name	Bread	Milk	Eggs
price	1.25	2.5	1.5
quantity	2	1	3

display

If your type is not available or you want to customize its display, pass a `display` function that formats the value, or a string that accesses `value` in its scope:

```
#td.field-info.at("quantity").insert(
  "display",
  val => ("One", "Two", "Three").at(val - 1),
)
#let td = TableData(..td)
#output(td.table)
```

#	Name	Qty	Price
0	Bread	Two	\$1.25
1	Milk	One	\$2.50
2	Eggs	Three	\$1.50

align etc.

You can pass `align` and `width` to a given field's metadata to determine how content aligns in the cell and how much horizontal space it takes up. In the future, more `tablex` setup arguments will be accepted.

```
#let adjusted = td
#adjusted.field-info.at("name").insert("align", center)
#adjusted.field-info.at("name").insert("width", lfr)
#output{
  #TableData(..adjusted).table
}
```

#	Name	Qty	Price
0	Bread	Two	\$1.25
1	Milk	One	\$2.50
2	Eggs	Three	\$1.50

Deeper `tablex` customization

TaDa uses `tablex` to display the table. So any argument that `tablex` accepts can be passed to `TableData` as well:

```
#let mapper = (index, row) => {
  let fill = if index == 0 {white.darken(15%)} else {none}
  row.map(cell => (..cell, fill: fill))
}
#let td = TableData(
  ..td,
  tablex-kwarg: (
    map-rows: mapper,
    auto-vlines: false,
  ),
)
#output(td.table)
```

#	Name	Qty	Price
0	Bread	Two	\$1.25
1	Milk	One	\$2.50
2	Eggs	Three	\$1.50

Subselection

You can select a subset of fields to display:

```
#output{
  #subset(td, indexes: (0,2), fields: ("__index", "name",
"price")).table
}
```

#	Name	Price
0	Bread	\$1.25
2	Eggs	\$1.50

Rows can also be selected by whether they fulfill a field condition:

```
#output{
  #filter(td, expression: "price < 1.5").table
}
```

#	Name	Qty	Price
0	Bread	Two	\$1.25

Operations

Expressions

The easiest way to leverage **TaDa**'s flexibility is through expressions. They can be strings that treat field names as variables, or functions that take keyword-only arguments.

- **Note!** you must `collect` before showing a table to ensure all expressions are computed:
- **Note!** When passing functions, every field is passed as a named argument to the function. So, make sure to capture unused fields with `..rest` (the name is unimportant) to avoid errors.

```
#let td = with-field(
  td,
  "total",
  expression: "price * quantity",
  type: "currency",
)

// Expressions can build off other expressions, too
#let taxed = with-field(
  td,
  "Tax",
  // Expressions can be functions, too
  expression: (total: none, ..rest) => total * 0.2,
  type: "currency",
)

// Extra field won't show here!
// #output(taxed.table)
// Computed expressions must be collected
#output(collect(taxed).table)
```

#	Name	Qty	Price	Total	Tax
0	Bread	Two	\$1.25	\$2.50	\$0.50
1	Milk	One	\$2.50	\$2.50	\$0.50
2	Eggs	Three	\$1.50	\$4.50	\$0.90

Chaining

It is inconvenient to require several temporary variables as above, or deep function nesting, to perform multiple operations on a table. **TaDa** provides a `chain` function to make this easier:

```
#let totals = chain(td,
  concat.with(
    field: "total",
    expression: "price * quantity",
    title: fmt("Total"),
    type: "currency",
  ),
  concat.with(
    field: "tax",
    expression: "total * 0.2",
    title: fmt("Tax"),
    type: "currency",
  ),
  concat.with(
    field: "after tax",
    expression: "total + tax",
    title: fmt("w/ Tax"),
    type: "currency",
  ),
)
// Don't forget to collect before taking a subset!
collect,
subset.with(
  fields: ("name", "total", "after tax")
),
)
#output(totals.table)
```

Name	Total	W/ Tax
Bread	\$2.50	\$3.00
Milk	\$2.50	\$3.00
Eggs	\$4.50	\$5.40

Aggregation

Row-wise and column-wise reduction is supported through `agg`:

```
#let grand-total = chain(
  totals,
  agg.with(
    using: array.sum,
    fields: "total"
  ),
  // use "item" to extract the value when a table has exactly
  one element
  item
)
#output{
  "Grand total: #fmt-currency(grand-total)*"
}
```

Grand total: \$9.50

It is also easy to aggregate over multiple fields:

```
#let agg-td = agg(
  totals,
  using: array.sum,
  fields: ("total", "after tax"),
  axis: 0,
  title: "##repr(function)\(#field\)*"
)
#output(agg-td.table)
```

sum(total)	sum(after tax)
\$9.50	\$11.40

Roadmap

- ☐ `apply` for value-wise transformations
- ☐ Reconcile whether `field-info` should be required
- ☐ `pivot/melt`
- ☐ `merge/join`