Overview

Ta|Da 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

Table manipulation

Ta|Da provides two main ways to construct tables – from columns and from rows:

Hello world

```
#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`

Ta|Da 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 titles = (
  name: (title: fmt("Name")),
  price: (title: fmt("Price")),
  quantity: (title: fmt("Qty")),
  ..td.field-info,
)
#let td = TableData(..td, field-info: titles)
#output(td.table)
```

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

Value formatting

`type`

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

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

```
#td.field-info.at("price").insert("type", "currency")
#let td = TableData(..td)
#output(td.table)
```

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

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

Currency and decimal locales

You can account for your locale by updating `default-currency`, `default-hundreds-separator`, and `default-decimal`:

```
#output[
  American: #format-currency(12.5)

  #default-currency.update("€")
  European: #format-currency(12.5)
]
```

American: $12.50

European: €12.50

These changes will also impact how `currency` and `float` types are displayed in a table.

`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	Price	Qty
0	Bread	€1.25	Two
1	Milk	€2.50	One
2	Eggs	€1.50	Three

`align` etc.

You can pass any other keywords accepted in the `tablex` constructor such as `align`, `fill`, `width`, etc.:

```
#output[
  #td.field-info.at("name").insert("fill", red)
  #TableData(..td).table
]
```

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

`tablex` customization

Ta|Da 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-kwargs: (
    map-rows: mapper,
    auto-vlines: false,
  ),
)
#output(td.table)
```

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

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	Price	Qty
0	Bread	€1.25	Two

Operations

Expressions

The easiest way to leverage Ta|Da'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",
  title: fmt("Total"),
  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,
  title: fmt("Tax"),
  type: "currency",
)

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

#	Name	Price	Qty	Total	Tax
0	Bread	€1.25	Two	€2.50	€0.50
1	Milk	€2.50	One	€2.50	€0.50
2	Eggs	€1.50	Three	€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. Ta|Da 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
Milk	€2.50	€3
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: #format-currency(grand-total)*
]
```

Grand total: €9.50

It is also easy to aggregate over multiple fields:

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

#output(agg-rows.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`