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 can be imported as follows:

From the official packages repository (recommended):

```
#import "@preview/tada:0.1.0"
```

From the source code (not recommended)

Option 1: You can clone the package directly into your project directory:

```
# In your project directory
git clone https://github.com/ntjess/typst-tada.git tada
```

Then import the functionality with `#import "./tada/lib.typ"`

Option 2: If Python is available on your system, use the provided packaging script to install TaDa in typst's `local` directory:

```
# Anywhere on your system
git clone https://github.com/ntjess/typst-tada.git
cd typst-tada
# Replace $XDG_CACHE_HOME with the appropriate directory based on
# https://github.com/typst/packages#downloads
python package.py ./typst.toml "$XDG_CACHE_HOME/typst/packages" \
  --namespace local
```

Now, TaDa is available under the local namespace:

```
#import "@local/tada:0.1.0"
```

Table manipulation

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

```
#let column-data = (
  name: ("Bread", "Milk", "Eggs"),
  price: (1.25, 2.50, 3.50),
  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),
)

// See `importing tada` above for reference
#let td = tada.from-columns(column-data)
// Equivalent to:
// #let td = TableData(rows: row-data)

// Show using `to-tablex`
// #let to-tablex = tada.to-tablex
#to-tablex(td)
```

name	price	quantity
Bread	1.25	2
Milk	2.5	1
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)

#to-tablex(td)
```

Name	Quantity	Price
Bread	2	1.25
Milk	1	2.5
Eggs	3	1.5

Using `__index`

TaDa will automatically add an `__index` field to each row that is hidden by default. If you want it displayed, update its information to set `hide: false`:

```
// You can add new fields or update existing ones using `with-field`
#let td = tada.with-field(td, "__index", hide: false, title:
"#")
// You can also insert attributes directly:
// #td.field-info.__index.insert("hide", false)
// etc.
#to-tablex(td)
```

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

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) = {
  // "negative" sign if needed
  let sign = if val < 0 {str.from-unicode(0x2212)} else {""}
  let currency = "$"
  {#sign#currency}
  tada.display.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))
#to-tablex(td)
```

#	Name	Quantity	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:

```
#to-tablex(
  transpose(td, ignore-types: true, fields-name: "")
)
```

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

`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)
#to-tablex(td)
```

#	Name	Quantity	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", 1fr)
#to-tablex(
  TableData(..adjusted)
)
```

#	Name	Quantity	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-kwargs: (
    map-rows: mapper,
    auto-vlines: false,
  ),
)
#to-tablex(td)
```

#	Name	Quantity	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:

```
#to-tablex(
  subset(td, indexes: (0,2), fields: ("__index", "name",
"price"))
)
```

#	Name	Price
0	Bread	$1.25
2	Eggs	$1.50

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

```
#to-tablex(
  filter(td, expression: "price < 1.5")
)
```

#	Name	Quantity	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!** 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 = tada.with-field(
  td,
  "total",
  expression: "price * quantity",
  type: "currency",
)

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

#to-tablex(taxed)
```

#	Name	Quantity	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 (chain, concat) = (tada.chain, tada.concat)
#let totals = chain(td,
  concat.with(
    field: "total",
    expression: "price * quantity",
    type: "currency",
  ),
  concat.with(
    field: "Tax",
    expression: "total * 0.2",
    type: "currency",
  ),
  concat.with(
    field: "after tax",
    expression: "total + tax",
    title: fmt("w/ Tax"),
    type: "currency",
  ),
  subset.with(
    fields: ("name", "total", "after tax")
  ),
)
#to-tablex(totals)
```

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(
  subset(totals, fields: "total"),
  agg.with(using: array.sum),
  // use `item` to extract the value when a table has exactly
one element
  item
)
// "Output" is a helper function just for capturing example
// outputs. It is not necessary in your code.
#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: "*#rep(function)\(#field\)*"
)
#to-tablex(agg-td)
```

sum(name)	sum(total)	sum(after tax)
BreadMilkEggs	$9.50	$11.40

Roadmap

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