Contents

Overview

TaDa provides a set of simple but powerful operations on rows of data. A full manual is available online: https://github.com/ntjess/typst-tada/blob/v0.2.0/docs/manual.pdf

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:0.2.0"
```

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 `showman` to install TaDa in typst's `local` directory:

```
# Anywhere on your system
git clone https://github.com/ntjess/typst-tada.git
cd typst-tada

# Can be done in a virtual environment
pip install "git+https://github.com/ntjess/showman.git"
showman package ./typst.toml
```

Now, TaDa is available under the local namespace:

```
#import "@local/tada:0.2.0"
```

Table adjustment

Creation

TaDa provides three main ways to construct tables – from columns, rows, or records.

- **Columns** are a dictionary of field names to column values. Alternatively, a 2D array of columns can be passed to `from-columns`, where `values.at(0)` is a column (belongs to one field).
- **Records** are a 1D array of dictionaries where each dictionary is a row.
- **Rows** are a 2D array where `values.at(0)` is a row (has one value for each field). Note that if `rows` are given without field names, they default to (0, 1, ..n).

```
#let column-data = (
  name: ("Bread", "Milk", "Eggs"),
  price: (1.25, 2.50, 1.50),
  quantity: (2, 1, 3),
)
#let record-data = (
  (name: "Bread", price: 1.25, quantity: 2),
  (name: "Milk", price: 2.50, quantity: 1),
  (name: "Eggs", price: 1.50, quantity: 3),
)
#let row-data = (
  ("Bread", 1.25, 2),
  ("Milk", 2.50, 1),
  ("Eggs", 1.50, 3),
)

#import tada: TableData
#let td = TableData(data: column-data)
// Equivalent to:
#let td2 = tada.from-records(record-data)
// _Not_ equivalent to (since field names are unknown):
#let td3 = tada.from-rows(row-data)

#to-table(td)
#to-table(td2)
#to-table(td3)
```

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

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

0	1	2
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).replace("_", " ")
  )
}

#let titles = (
  // As a function
  name: (title: fmt),
  // As a string
  quantity: (title: fmt("Qty")),
)
#let td = TableData(..td, field-info: titles)

#to-table(td)
```

Name	price	**Qty**
Bread	1.25	2
Milk	2.5	1
Eggs	1.5	3

Adapting default behavior

You can specify defaults for any field not explicitly populated by passing information to `field-defaults`. Observe in the last example that `price` was not given a title. We can indicate it should be formatted the same as `name` by passing `title: fmt` to `field-defaults`. **Note** that any field that is explicitly given a value will not be affected by `field-defaults` (i.e., `quantity` will retain its string title "Qty")

```
#let defaults = (title: fmt)
#let td = TableData(..td, field-defaults: defaults)
#to-table(td)
```

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

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`:

```
// Use the helper function `update-fields` to update multiple fields
// and/or attributes
#import tada: update-fields
#let td = update-fields(
  td, __index: (hide: false, title: "\#")
)
// You can also insert attributes directly:
// #td.field-info.__index.insert("hide", false)
// etc.
#to-table(td)
```

#	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-value: "", align: left)`
- **content** : `(display: format-content, align: left)`
- **float** : `(align: right)`
- **integer** : `(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 currency-info = (
  display: tada.display.format-usd, align: right
)
#td.field-info.insert("price", (type: "currency"))
#let td = TableData(..td, type-info: ("currency": currency-info))
#to-table(td)
```

#	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:

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

	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),
)

#let td = TableData(..td)
#to-table(td)
```

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

`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 `table` setup arguments will be accepted.

```
#let adjusted = update-fields(
  td, name: (align: center, width: 1.4in)
)
#to-table(adjusted)
```

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

Deeper `table` customization

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

```
#let mapper = (x, y) => {
  if y == 0 {rgb("#8888")} else {none}
}
#let td = TableData(
  ..td,
  table-kwargs: (
    fill: mapper, stroke: (x: none, y: black)
  ),
)
#to-table(td)
```

#	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 or rows to display:

```
#import tada: subset
#to-table(
  subset(td, indexes: (0,2), fields: ("name", "price"))
)
```

Name	Price
Bread	$1.25
Eggs	$1.50

Note that `indexes` is based on the table's `__index` column, *not* it's positional index within the table:

```
#let td2 = td
#td2.data.insert("__index", (1, 2, 2))
#to-table(
  subset(td2, indexes: 2, fields: ("__index", "name"))
)
```

#	Name
2	Milk
2	Eggs

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

```
#to-table(
  tada.filter(td, expression: "price < 1.5")
)
```

#	Name	Price	Qty
0	Bread	$1.25	Two

Concatenation

Concatenating rows and columns are both supported operations, but only in the simple sense of stacking the data. Currently, there is no ability to join on a field or otherwise intelligently merge data.

- `axis: 0` places new rows below current rows
- `axis: 1` places new columns to the right of current columns
- Unless you specify a fill value for missing values, the function will panic if the tables do not match exactly along their concatenation axis.
- You cannot stack with `axis: 1` unless every column has a unique field name.

```
#import tada: stack

#let td2 = TableData(
  data: (
    name: ("Cheese", "Butter"),
    price: (2.50, 1.75),
  )
)
#let td3 = TableData(
  data: (
    rating: (4.5, 3.5, 5.0, 4.0, 2.5),
  )
)

// This would fail without specifying the fill
// since `quantity` is missing from `td2`
#let stack-a = stack(td, td2, missing-fill: 0)
#let stack-b = stack(stack-a, td3, axis: 1)
#to-table(stack-b)
```

Name	Price	Qty	Rating
Bread	1.25	Two	4.5
Milk	2.5	One	3.5
Eggs	1.5	Three	5
Cheese	2.5	/	4
Butter	1.75	/	2.5

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 make-dict(field, expression) = {
  let out = (:)
  out.insert(
    field,
    (expression: expression, type: "currency"),
  )
  out
}

#let td = update-fields(
  td, ..make-dict("total", "price * quantity" )
)

#let tax-expr(total: none, ..rest) = { total * 0.2 }
#let taxed = update-fields(
  td, ..make-dict("tax", tax-expr),
)

#to-table(
  subset(taxed, fields: ("name", "total", "tax"))
)
```

Name	Total	Tax
Bread	$2.50	$0.50
Milk	$2.50	$0.50
Eggs	$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. Furthermore, when you need to compute several fields at once and don't need extra field information, you can use `add-expressions` as a shorthand:

```
#import tada: chain, add-expressions
#let totals = chain(td,
  add-expressions.with(
    total: "price * quantity",
    tax: "total * 0.2",
    after-tax: "total + tax",
  ),
  subset.with(
    fields: ("name", "total", "after-tax")
  ),
  // Add type information
  update-fields.with(
    after-tax: (type: "currency", title: fmt("w/ Tax")),
  ),
)
#to-table(totals)
```

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

Sorting

You can sort by ascending/descending values of any field, or provide your own transformation function to the `key` argument to customize behavior further:

```
#import tada: sort-values
#to-table(sort-values(
  td, by: "quantity", descending: true
))
```

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

Aggregation

Column-wise reduction is supported through `agg`, using either functions or string expressions:

```
#import tada: agg, item
#let grand-total = chain(
  totals,
  agg.with(after-tax: array.sum),
  // use "item" to extract exactly one element
  item
)
// "Output" is a helper function just for these docs.
// It is not necessary in your code.
#output[
  *Grand total: #tada.display.format-usd(grand-total)*
]
```

Grand total: $11.40

It is also easy to aggregate several expressions at once:

```
#let agg-exprs = (
  "# items": "quantity.sum()",
  "Longest name": "[#name.sorted(key: str.len).at(-1)]",
)
#let agg-td = tada.agg(td, ..agg-exprs)
#to-table(agg-td)
```

# items	Longest name
6	Bread

Functions in `tabledata.typ`

TableData

Constructs a `TableData()` object from a dictionary of columnar data. See examples in the overview above for metadata examples.

Parameters

```
TableData(
  data: dictionary,
  field-info: dictionary,
  type-info: dictionary,
  field-defaults: dictionary,
  table-kwargs: dictionary,
  ..reserved
) -> TableData
```

data `dictionary`

A dictionary of arrays, each representing a column of data. Every column must have the same length. Missing values are represented by `none`.

Default: `none`

field-info `dictionary`

A dictionary of dictionaries, each representing the properties of a field. The keys of the outer dictionary must match the keys of `data`. The keys of the inner dictionaries are all optional and can contain: A dictionary of dictionaries, each representing the properties of a field. The keys of the outer dictionary must match the keys of `data`. The keys of the inner dictionaries are all optional and can contain:

- `type` (string): The type of the field. Must be one of the keys of `type-info`. Defaults to `auto`, which will attempt to infer the type from the data.
- `title` (string): The title of the field. Defaults to the field name, title-cased.
- `display` (string): The display format of the field. Defaults to the display format for the field's type.
- `expression` (string, function): A string or function containing a Python expression that will be evaluated for each row to compute the value of the field. The expression can reference any other field in the table by name.
- `hide` (boolean): Whether to hide the field from the table. Defaults to `false`.

Default: `(:)`

type-info `dictionary`

A dictionary of dictionaries, each representing the properties of a type. These properties will be populated for a field if its type is given in `field-info` and the property is not specified already.

Default: `(:)`

field-defaults `dictionary`

Default values for every field if not specified in `field-info`.

Default: `(:)`

table-kwargs `dictionary`

Keyword arguments to pass to `table()`.

Default: `(:)`

..reserved

Reserved for future use; currently discarded.

add-expressions

Shorthand to easily compute expressions on a table.

```
#let td = TableData(data: (a: (1, 2, 3)))
#to-table(add-expressions(td, b: `a + 1`))
```

a	b
1	2
2	3
3	4

Parameters

```
add-expressions(
  td: TableData,
  ..expressions: any
)
```

td `TableData`

The table to add expressions to

..expressions `any`

An array of expressions to compute.

- Positional arguments are converted to (`value`: (expression: `value`))
- Named arguments are converted to (`key`: (expression: `value`))

count

Returns a `TableData()` with a single `count` column and one value – the number of rows in the table.

```
#let td = TableData(data: (a: (1, 2, 3),
b: (3, 4, none)))
#to-table(count(td))
```

count
3

Parameters

```
count(td: TableData) -> TableData
```

td `TableData`

The table to count

drop

Similar to `subset()`, but drops the specified fields and/or indexes instead of keeping them.

```
#let td = TableData(data: (a: (1, 2), b:
(3, 4), c: (5, 6)))
#to-table(drop(td, fields: ("a", "c"),
indexes: (0,)))
```

b
4

Parameters

```
drop(
  td: TableData,
  fields: array str auto,
  indexes: array int auto
) -> TableData
```

td `TableData`

The table to drop fields from

fields `array` or `str` or `auto`

The field or fields to drop. If `auto`, no fields are dropped

Default: `none`

indexes `array` or `int` or `auto`

The index or indexes to drop. If `auto`, no indexes are dropped

Default: `none`

from-columns

Constructs a `TableData()` object from a list of column-oriented data and their field info.

```
#let data = (
  (1, 2, 3),
  (4, 5, 6),
)
#let mk-tbl(..args) = to-table(from-columns(..args))
#set align(center)
#grid(columns: 2, column-gutter: 1em)[
  Auto names:
  #mk-tbl(data)
][
  User names:
  #mk-tbl(data, field-info: ("a", "b"))
]
```

Auto names:

0	1
1	4
2	5
3	6

User names:

a	b
1	4
2	5
3	6

Parameters

```
from-columns(
  columns: array,
  field-info: dictionary,
  ..metadata: dictionary
) -> TableData
```

columns `array`

A list of arrays, each representing a column of data. Every column must have the same length and columns.len() must match field-info.keys().len()

field-info `dictionary`

See the `field-info` argument to `TableData()` for handling dictionary types. If an array is passed, it is converted to a dictionary of (key1: (:), …).

Default: `auto`

..metadata `dictionary`

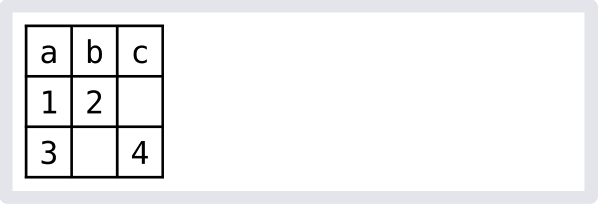
Forwarded directly to `TableData()`

from-records

Constructs a `TableData()` object from a list of records.

A record is a dictionary of key-value pairs, Records may contain different keys, in which case the resulting `TableData()` will contain the union of all keys present with `none` values for missing keys.

```
#let records = (
  (a: 1, b: 2),
  (a: 3, c: 4),
)
#to-table(from-records(records))
```



a	b	c
1	2	
3		4

Parameters

```
from-records(
  records: array,
  ..metadata: dictionary
) -> TableData
```

records `array`

A list of dictionaries, each representing a record. Every record must have the same keys.

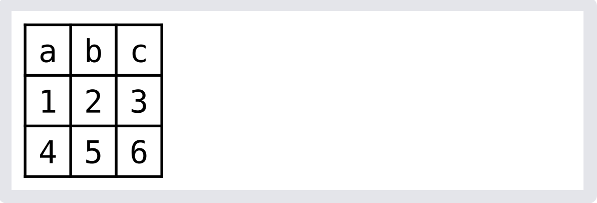
..metadata `dictionary`

Forwarded directly to `TableData()`

from-rows

Constructs a `TableData()` object from a list of row-oriented data and their field info.

```
#let data = (
  (1, 2, 3),
  (4, 5, 6),
)
#to-table(from-rows(data, field-info:
("a", "b", "c")))
```



a	b	c
1	2	3
4	5	6

Parameters

```
from-rows(
  rows: array,
  field-info: dictionary,
  ..metadata: dictionary
) -> TableData
```

rows `array`

A list of arrays, each representing a row of data. Every row must have the same length and rows.at(0).len() must match field-info.keys().len()

field-info `dictionary`

See the `field-info` to `from-columns()`()

Default: `auto`

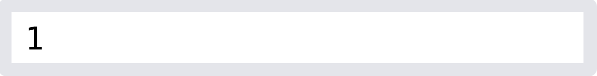
..metadata `dictionary`

Forwarded directly to `TableData()`

item

Extracts a single value from a `TableData()` that has exactly one field and one row.

```
#let td = TableData(data: (a: (1,)))
#item(td)
```



1

Parameters

```
item(td: TableData) -> any
```

td `TableData`

The table to extract a value from

stack

Stacks two tables on top of or next to each other.

```
#let td = TableData(data: (a: (1, 2), b:
(3, 4)))
#let other = TableData(data: (c: (7, 8),
d: (9, 10)))
#grid(columns: 2, column-gutter: 1em)[
  #to-table(stack(td, other, axis: 1))
][
  #to-table(stack(
    td, other, axis: 0, missing-fill: -4
  ))
]
```

a	b	c	d
1	3	7	9
2	4	8	10

a	b	c	d
1	3	−4	−4
2	4	−4	−4
−4	−4	7	9
−4	−4	8	10

- td (TableData): The table to stack on
- other (TableData): The table to stack
- axis (int): The axis to stack on. 0 will place `other` below `td`, 1 will place `other` to the right of `td`. If `missing-fill` is not specified, either the number of rows or fields must match exactly along the chosen axis.
  - **Note!** If `axis` is 1, `other` may not have any field names that are already in `td`.
- missing-fill (any): The value to use for missing fields or rows. If `auto`, an error will be raised if the number of rows or fields don't match exactly along the chosen axis.

Parameters

```
stack(
  td,
  other,
  axis,
  missing-fill
) -> TableData
```

subset

Creates a new `TableData()` with only the specified fields and/or indexes.

```
#let td = TableData(data: (a: (1, 2), b:
(3, 4), c: (5, 6)))
#to-table(subset(td, fields: ("a", "c"),
indexes: (0,)))
```

a	c
1	5

Parameters

```
subset(
  td: TableData,
  indexes: array str auto,
  fields: array int auto,
) -> TableData
```

td `TableData`

The table to subset

indexes `array` or `str` or `auto`

The field or fields to keep. If `auto`, all fields are kept

Default: `auto`

fields `array` or `int` or `auto`

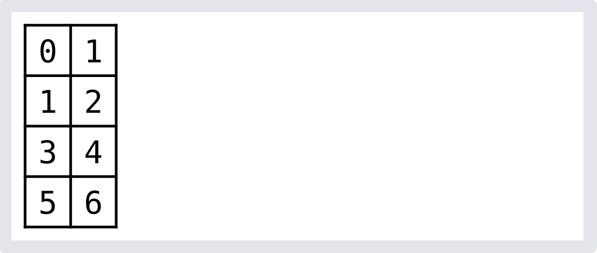
The index or indexes to keep. If `auto`, all indexes are kept

Default: `auto`

transpose

Converts rows into columns, discards field info, and uses `__index` as the new fields.

```
#let td = TableData(data: (a: (1, 2), b:
(3, 4), c: (5, 6)))
#to-table(transpose(td))
```



0	1
1	2
3	4
5	6

Parameters

```
transpose(
  td,
  fields-name: str none,
  ignore-types: bool,
  ..metadata: dictionary
) -> TableData
```

td

- td (TableData): The table to transpose

fields-name `str` or `none`

The name of the field containing the new field names. If `none`, the new fields are named `0`, `1`, etc.

Default: `none`

ignore-types `bool`

Whether to ignore the types of the original table and instead use `content` for all fields. This is useful when not all columns have the same type, since a warning will occur when multiple types are encountered in the same field otherwise.

Default: `false`

..metadata `dictionary`

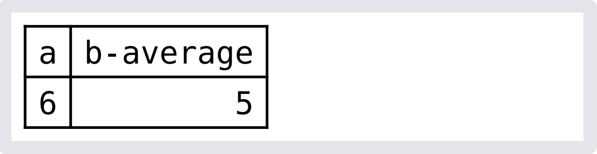
Forwarded directly to `TableData()`

Functions in `ops.typ`

agg

Performs an aggregation across entire data columns.

```
#let td = TableData(data: (a: (1, 2, 3),
b: (4, 5, 6)))
#to-table(agg(td, a: array.sum, b-average:
"b.sum() / b.len()"))
```



a	b-average
6	5

Parameters

```
agg(
  td: TableData,
  field-info: dictionary,
  ..field-func-map: dictionary
) -> TableData
```

td `TableData`

The table to aggregate

field-info `dictionary`

Optional overrides to the initial table's field info. This is useful in case an aggregation function changes the field's type or needs a new display function.

Default: `(:)`

..field-func-map `dictionary`

A mapping of field names to aggregation functions or expressions. Expects a function accepting named arguments, one for each field in the table. The return value will be placed in a single cell.

- Note! If the assigned name for a function matches an existing field, *and* a function (not a string) is passed, the behavior changes: Instead, the function must take one *positional* argument and only receives values for the field it's assigned to. For instance, in a table with a field `price`, you can easily calculate the total price by calling `agg(td, price: array.sum)`. If this behavior was not enabled, this would be `agg(td, price: (price: none, ..rest) => price.sum()`.
- Columns will have their missing (`none`) values removed before being passed to the function or expression.

chain

Sequentially applies a list of table operations to a given table.

The operations can be any function that takes a TableData object as its first argument. It is recommended when applying many transformations in a row, since it avoids the need for deeply nesting operations or keeping many temporary variables.

Returns a TableData object that results from applying all the operations in sequence.

```
#let td = TableData(data: (a: (1, 2, 3),
b: (4, 5, 6)))
#to-table(chain(td,
  filter.with(expression: "a > 1"),
  sort-values.with(by: "b", descending:
true)
))
```

a	b
3	6
2	5

Parameters

```
chain(
  td: TableData,
  ..operations: TableData
) -> TableData
```

td `TableData`

The initial table to which the operations will be applied.

..operations `TableData`

A list of table operations. Each operation is applied to the table in sequence. Operations must be compatible with TableData.

filter

Filters rows in a table based on a given expression, returning a new TableData object containing only the rows for which the expression evaluates to true. This function filters rows in the table based on a boolean expression. The expression is evaluated for each row, and only rows for which the expression evaluates to true are retained in the output table.

```
#let td = TableData(data: (a: (1, 2, 3),
b: (4, 5, 6)))
#to-table(filter(td, expression: "a > 1
and b > 5"))
```

a	b
3	6

Parameters

```
filter(
  td: TableData,
  expression: string
) -> TableData
```

td `TableData`

The table to filter.

expression `string`

A boolean expression used to filter rows. The expression can reference fields in the table and must result in a truthy output.

Default: `none`

group-by

Creates a list of (value, group-table) pairs, one for each unique value in the given field. This list is optionally condensed into one table using specified aggregation functions.

```
#let td = TableData(data: (
  a: (1, 1, 1, 2, 3, 3),
  b: (4, 5, 6, 7, 8, 9),
  c: (10, 11, 12, 13, 14, 15)
))
#let first-group = group-by(td, by:
"a").at(0)
Group identity: #repr(first-group.at(0))
#to-table(first-group.at(1))
Aggregated:
#to-table(group-by(td, by: "a", aggs:
(count: "a.len()")))
```

Group identity: 1

a	b	c
1	4	10
1	5	11
1	6	12

Aggregated:

a	count
1	3
2	1
3	2

Parameters

```
group-by(
  td: TableData ,
  by: string ,
  aggs: dictionary ,
  field-info: dictionary
) -> array TableData
```

td `TableData`

The table to group

by `string`

The field whose values are used for grouping

Default: `none`

aggs `dictionary`

A dictionary of aggregations to apply to each group. The keys are the field names for the resulting table, and the values are the aggregation functions or expressions.

Default: `(:)`

field-info `dictionary`

Optional overrides to the initial table's field info. This is useful in case an aggregation function changes the field's type or needs a new display function.

Default: `(:)`

sort-values

Sorts the rows of a table based on the values of a specified field, returning a new TableData object with rows sorted based on the specified field.

Parameters

```
sort-values(
  td: TableData,
  by: string,
  key: function,
  descending: bool
) -> TableData
```

td `TableData`

The table to sort

by `string`

The field name to sort by

Default: `none`

key `function`

A function that transforms the values of the field before sorting. Defaults to the identity function if not provided.

Default: `values => values`

descending `bool`

Specifies whether to sort in descending order. Defaults to false for ascending order.

Default: `false`

Functions in `display.typ`

format-float

Converts a float to a string where the comma, decimal, and precision can be customized.

```
#format-float(123456, precision: 2, pad:
true)\
#format-float(123456.1121, precision: 1,
hundreds-separator: "_")
```

```
123,456.00
123_456.1
```

Parameters

```
format-float(
  number: number,
  hundreds-separator: str auto,
  decimal: str auto,
  precision: int none,
  pad: bool
) -> str
```

number `number`

The number to convert

hundreds-separator `str` or `auto`

The character to use to separate hundreds

Default: `auto`

decimal `str` or `auto`

The character to use to separate the integer and fractional portions

Default: `auto`

precision `int` or `none`

The number of digits to show after the decimal point. If `none`, then no rounding will be done.

Default: `none`

pad `bool`

If true, then the fractional portion will be padded with zeros to match the precision if needed.

Default: `false`

format-usd

Converts a float to a United States dollar amount.

```
#format-usd(12.323)\
#format-usd(-12500.29)
```

```
$12.32
−$12,500.29
```

Parameters

```
format-usd(
  number: float int,
  ..args: any
) -> str
```

number `float` or `int`

The number to convert

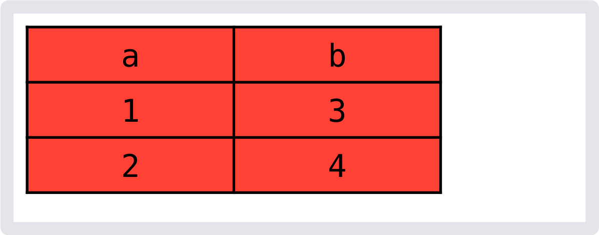
..args `any`

Passed to `format-float()`()

to-table

Converts a `TableData()` into a displayed `table`. This is the main (and only intended) way of rendering `tada` data. Most keywords can be overridden for customizing the output.

```
#let td = TableData(
  data: (a: (1, 2), b: (3, 4)),
// Tables can carry their own kwargs, too
  table-kwargs: (inset: (x: 3em, y: 0.5em))
)
#to-table(td, fill: red)
```



a	b
1	3
2	4

Parameters

```
to-table(
  td: TableData,
  ..kwargs: any
) -> content
```

td `TableData`

The data to render

..kwargs `any`

Passed to `table`