Package 'teal.picks'

May 19, 2026

Type Package

Title Dataset and Variable Picker and Merge Module for 'teal' Applications

Version 0.1.0

Date 2026-05-07

Description Allows users to interactively select datasets, variables, and values within 'teal' applications using a 'tidyselect'-style interface. Selected picks can be merged and transformed into analysis-ready data within 'teal' modules.

License Apache License 2.0

URL https://github.com/insightsengineering/teal.picks/, https://insightsengineering.github.io/teal.picks/

BugReports https://github.com/insightsengineering/teal.picks/issues

Depends R (>= 4.1)

Imports bsicons, checkmate, dplyr, htmltools, logger, methods, rlang, shiny, shinyWidgets, teal, teal.code, teal.data, teal.logger, tidyselect, yaml

Suggests jsonlite, knitr, rmarkdown, rvest, shinytest2, teal.transform, testthat (>= 3.0), tibble, withr (>= 3.0.0)

VignetteBuilder knitr

Config/Needs/website insightsengineering/nesttemplate

Config/roxygen2/version 8.0.0

Config/testthat/edition 3

Encoding UTF-8

Language en-US

NeedsCompilation no

Author Dawid Kaledkowski [aut, cre] (ORCID: <https://orcid.org/0000-0001-9533-457X>), Andre Verissimo [aut] (ORCID: <https://orcid.org/0000-0002-2212-339X>),

Marcin Kosinski [aut],
Lluís Revilla Sancho [aut] (ORCID:
<https://orcid.org/0000-0001-9747-2570>),
Oriol Senan [aut] (ORCID: <https://orcid.org/0000-0002-9621-3371>),
Dony Unardi [rev],
F. Hoffmann-La Roche AG [cph, fnd]

Maintainer Dawid Kaledkowski <dawid.kaledkowski@roche.com>

Repository CRAN

Date/Publication 2026-05-19 09:10:02 UTC

Contents

`as.picks` *Convert data_extract_spec to picks*

Description

[Experimental] Helper functions to ease transition between `teal.transform::data_extract_spec()` and `picks()`.

Usage

```
as.picks(x, quiet = FALSE)

teal_transform_filter(x, label = "Filter")
```

Arguments

`x`	(`data_extract_spec`, `select_spec`, `filter_spec`) object to convert to `picks`
`quiet`	(`logical(1)`) whether to suppress warnings about non-convertible elements.
`label`	(`character(1)`) Label of the module.

Details

With introduction of picks, data_extract_spec will no longer serve a primary tool to define variable choices and default selection in teal-modules and eventually data_extract_spec will be deprecated. To ease the transition to the new tool, we provide as.picks method which can handle 1:1 conversion from data_extract_spec to picks. Unfortunately, when data_extract_spec contains filter_spec then as.picks is unable to provide reliable picks equivalent.

Value

A picks object when conversion is possible, otherwise NULL with a warning (if quiet = FALSE).

Examples

```
# convert des with eager select_spec
as.picks(
  teal.transform::data_extract_spec(
    dataname = "iris",
    teal.transform::select_spec(
      choices = c("Sepal.Length", "Sepal.Width", "Species"),
      selected = c("Sepal.Length", "Species"),
      multiple = TRUE,
      ordered = TRUE
    )
  )
)

# convert des with delayed select_spec
as.picks(
  teal.transform::data_extract_spec(
    dataname = "iris",
    teal.transform::select_spec(
      choices = teal.transform::variable_choices("iris"),
      selected = teal.transform::first_choice(),
      multiple = TRUE,
      ordered = TRUE
    )
  )
)

as.picks(
  teal.transform::data_extract_spec(
    dataname = "iris",
    teal.transform::select_spec(
      choices = teal.transform::variable_choices(
        "iris",
        subset = function(data) names(Filter(is.numeric, data))
      ),
      selected = teal.transform::first_choice(),
      multiple = TRUE,
      ordered = TRUE
    )
  )
```

```
)

# teal_transform_module build on teal.transform

teal_transform_filter(
  teal.transform::data_extract_spec(
    dataname = "iris",
    filter = teal.transform::filter_spec(
      vars = "Species",
      choices = c("setosa", "versicolor", "virginica"),
      selected = c("setosa", "versicolor")
    )
  )
)

teal_transform_filter(
  picks(
    datasets(choices = "iris", select = "iris"),
    variables(choices = "Species", "Species"),
    values(
      choices = c("setosa", "versicolor", "virginica"),
      selected = c("setosa", "versicolor")
    )
  )
)
```

`check_last_level` *Assert level*

Description

Assert level

Usage

```
check_last_level(x, class)

assert_last_level(x, class, .var.name = checkmate::vname(x), add = NULL)
```

Arguments

`x`	`picks` object
`class`	Class of the last element of picks
`.var.name`	[`character(1)`] The custom name for `x` as passed to any `assert*` function. Defaults to a heuristic name lookup.
`add`	[`AssertCollection`] Collection to store assertion messages. See `AssertCollection`.

Value

For `check_last_level` a logical value or a string. For `assert_last_level` invisibly the object checked or an error.

Examples

```
x <- picks(datasets(), variables(), values())
assert_last_level(x, "values")
```

`helper_functions_pick` *Helper functions to check pick attributes*

Description

Helper functions for pick objects generated from `datasets()`, `variables()` or `values()`:

- `is_pick_multiple()` checks if a pick has the `multiple` attribute set to `TRUE`.
- `is_pick_fixed()` checks if a pick has the `fixed` attribute set to `TRUE`.
- `is_pick_ordered()` checks if a pick has the `ordered` attribute set to `TRUE`.

Usage

```
is_pick_multiple(x)

is_pick_fixed(x)

is_pick_ordered(x)
```

Arguments

`x`	(`datasets`, `variables` or `values`) pick to check.

Value

`TRUE` if the pick has the attribute set to `TRUE`, `FALSE` otherwise.

Examples

```
p <- picks(datasets("iris"), variables(), values())

is_pick_multiple(p$variables)

is_pick_fixed(p$variables)

is_pick_ordered(p$variables)
```

`interaction_vars` *Declare interaction variable pairs for* `tidyselect`

Description

Used inside `tidyselect` expressions to declare a pair of variables that interact with each other. The pair is recorded in the selection environment and the positions of both variables within the available variables are returned.

Usage

```
interaction_vars(
  var1,
  var2,
  vars = tidyselect::peek_vars(fn = "interaction_vars")
)
```

Arguments

`var1`	An unquoted variable name.
`var2`	An unquoted variable name that interacts with `var1`.
`vars`	Character vector of available variable names, retrieved automatically via `tidyselect::peek_vars()`.

Value

An integer vector of length 2 giving the positions of `var1` and `var2` in `vars`, or `NA` where a variable is not found.

Examples

```
picks(
  datasets("ADAE"),
  variables(
    c(AGE, RACE, interaction_vars("COUNTRY", "RACE")),
    selected = "COUNTRY:RACE",
    multiple = TRUE
  ),
  values()
)
```

merge_srv *Merge Server Function for Dataset Integration*

Description

merge_srv is a powerful Shiny server function that orchestrates the merging of multiple datasets based on user selections from picks objects. It creates a reactive merged dataset (teal_data object) and tracks which variables from each selector are included in the final merged output.

This function serves as the bridge between user interface selections (managed by selectors) and the actual data merging logic. It automatically handles:

- Dataset joining based on join keys
- Variable selection and renaming to avoid conflicts
- Reactive updates when user selections change
- Generation of reproducible R code for the merge operation

Usage

```
merge_srv(
  id,
  data,
  selectors,
  output_name = "anl",
  join_fun = "dplyr::inner_join"
)
```

Arguments

id	(character(1)) Module ID for the Shiny module namespace
data	(reactive) A reactive expression returning a teal.data::teal_data object containing the source datasets to be merged. This object must have join keys defined via teal.data::join_keys() to enable proper dataset relationships.
selectors	(named list) A named list of selector objects. Each element can be: • A picks object defining dataset and variable selections • A reactive expression returning a picks object The names of this list are used as identifiers for tracking which variables come from which selector.
output_name	(character(1)) Name of the merged dataset that will be created in the returned teal_data object. Default is "anl". This name will be used in the generated R code.
join_fun	(character(1)) The joining function to use for merging datasets. Must be a qualified function name (e.g., "dplyr::left_join", "dplyr::inner_join", "dplyr::full_join"). Default is "dplyr::inner_join". The function must accept by and suffix parameters.

Value

A `list` with two reactive elements:

- `data`A `reactive` returning a teal.data::teal_data object containing the merged dataset. The merged dataset is named according to `output_name` parameter. The `teal_data` object includes:
  - The merged dataset with all selected variables
  - Complete R code to reproduce the merge operation
  - Updated join keys reflecting the merged dataset structure
- `variables` A `reactive` returning a named list mapping selector names to their selected variables in the merged dataset. The structure is: `list(selector_name_1 = c("var1", "var2"), selector_name_2 = c("var3", "var4"), ...)`. Variable names reflect any renaming that occurred during the merge to avoid conflicts.

How It Works

The `merge_srv` function performs the following steps:

1. **Receives Input Data**: Takes a reactive `teal_data` object containing source datasets with defined join keys
2. **Processes Selectors**: Evaluates each selector (whether static `picks` or reactive) to determine which datasets and variables are selected
3. **Determines Merge Order**: Uses topological sort based on the `join_keys` to determine the optimal order for merging datasets.
4. **Handles Variable Conflicts**: Automatically renames variables when:
   - Multiple selectors choose variables with the same name from different datasets
   - Foreign key variables would conflict with existing variables
   - Renaming follows the pattern `{column-name}_{dataset-name}`
5. **Performs Merge**: Generates and executes merge code that:
   - Selects only required variables from each dataset
   - Applies any filters defined in selectors
   - Joins datasets using specified join function and join keys
   - Maintains reproducibility through generated R code
6. **Updates Join Keys**: Creates new join key relationships for the merged dataset (`"anl"`) relative to remaining datasets in the `teal_data` object
7. **Tracks Variables**: Keeps track of the variable names in the merged dataset

Usage Pattern

```
# In your Shiny server function
merged <- merge_srv(
  id = "merge",
  data = shiny::reactive(my_teal_data),
  selectors = list(
    selector1 = picks(...),
```

```
    selector2 = shiny::reactive(picks(...))
  ),
  output_name = "anl",
  join_fun = "dplyr::left_join"
)

# Access merged data
merged_data <- merged$data()  # teal_data object with merged dataset
anl <- merged_data[["anl"]]   # The actual merged data.frame/tibble

# Get variable mapping
vars <- merged$variables()
# Returns: list(selector1 = c("VAR1", "VAR2"), selector2 = c("VAR3", "VAR4_ADSL"))

# Get reproducible code
code <- teal.code::get_code(merged_data)
```

Merge Logic Details

Dataset Order: Datasets are merged in topological order based on join keys. The first dataset acts as the "left" side of the join, and subsequent datasets are joined one by one.

Join Keys: The function uses join keys from the source `teal_data` object to determine:

- Which datasets can be joined together
- Which columns to use for joining (the by parameter)
- Whether datasets need intermediate joins (not yet implemented)

Variable Selection: For each dataset being merged:

- Selects user-chosen variables from selectors
- Includes foreign key variables needed for joining (even if not explicitly selected)
- Removes duplicate foreign keys after join (they're already in the left dataset)

Conflict Resolution: When variable names conflict:

- Variables from later datasets get suffixed with `_dataname`
- Foreign keys that match are merged (not duplicated)
- The mapping returned in `merge_vars` reflects the final names

Integration with Selectors

`merge_srv` is designed to work with `picks_srv()` which creates selector objects:

```
# Create selectors in server
selectors <- picks_srv(
  picks =  list(
    adsl = picks(...),
    adae = picks(...)
```

```
  ),
  data = data
)

# Pass to merge_srv
merged <- merge_srv(
  id = "merge",
  data = data,
  selectors = selectors
)
```

See Also

- picks_srv() for creating selectors
- teal.data::join_keys() for defining dataset relationships

Examples

```
# Complete example with CDISC data
library(teal.picks)
library(teal.data)
library(shiny)

# Prepare data with join keys
data <- teal_data()
data <- within(data, {
  ADSL <- teal.data::rADSL
  ADAE <- teal.data::rADAE
})
join_keys(data) <- default_cdisc_join_keys[c("ADSL", "ADAE")]

# Create Shiny app
ui <- fluidPage(
  picks_ui("adsl", picks(datasets("ADSL"), variables())),
  picks_ui("adae", picks(datasets("ADAE"), variables())),
  verbatimTextOutput("code"),
  verbatimTextOutput("vars")
)

server <- function(input, output, session) {
  # Create selectors
  selectors <- list(
    adsl = picks_srv("adsl",
      data = shiny::reactive(data),
      picks = picks(datasets("ADSL"), variables())
    ),
    adae = picks_srv("adae",
      data = shiny::reactive(data),
      picks = picks(datasets("ADAE"), variables())
    )
  )
```

```
  # Merge datasets
  merged <- merge_srv(
    id = "merge",
    data = shiny::reactive(data),
    selectors = selectors,
    output_name = "anl",
    join_fun = "dplyr::left_join"
  )

  # Display results
  output$code <- renderPrint({
    cat(teal.code::get_code(merged$data()))
  })

  output$vars <- renderPrint({
    merged$variables()
  })
}
if (interactive()) {
  shinyApp(ui, server)
}
```

---

picks *Choices/selected settings*

Description

Define choices and default selection for variables. `picks` allows app-developer to specify `datasets`, `variables` and `values` to be selected by app-user during Shiny session. Functions are based on the idea of `choices/selected` where app-developer provides `choices` and what is `selected` by default. App-user changes `selected` interactively (see `picks_module`).

Usage

```
picks(..., check_dataset = TRUE)

datasets(choices = tidyselect::everything(), selected = 1L, fixed = NULL, ...)

variables(
  choices = tidyselect::everything(),
  selected = 1L,
  multiple = NULL,
  fixed = NULL,
  ordered = FALSE,
  ...
)
```

```
values(
  choices = function(x) !is.na(x),
  selected = function(x) !is.na(x),
  multiple = TRUE,
  fixed = NULL,
  ...
)
```

Arguments

...	for picks(...): hierarchical structure that contains datasets() as first element and optionally variables() and values() for variables(...) and values(...): additional arguments delivered to pickerInput
check_dataset	(logical(1)) whether to check that the first element of picks is datasets(). This is useful to set to FALSE when creating picks objects that have a required dataset that is not selected by the user and defined in the module itself.
choices	(tidyselect::language or character) Available values to choose.
selected	(tidyselect::language or character) Choices to be selected.
fixed	(logical(1)) selection will be fixed and not possible to change interactively.
multiple	(logical(1)) if more than one selection is possible.
ordered	(logical(1)) if the selected should follow the selection order. If FALSE selected returned from srv_module_input() would be ordered according to order in choices.

Value

For picks() it returns an object of picks class, which is a list of pick objects with additional attributes for Shiny interactivity. For datasets(), variables(), and values() it returns a pick object with class corresponding to the type of selection including the choices and selected values.

tidyselect support

Both choices and selected parameters support tidyselect syntax, enabling dynamic and flexible variable selection patterns. This allows choices to be determined at runtime based on data characteristics rather than hard-coded values.

Using tidyselect for choices and selected:

When choices uses tidyselect, the available options are determined dynamically based on actually selected data:

- tidyselect::everything() - All variables/datasets
- tidyselect::starts_with("prefix") - Variables starting with a prefix
- tidyselect::ends_with("suffix") - Variables ending with a suffix
- tidyselect::contains("pattern") - Variables containing a pattern
- tidyselect::matches("regex") - Variables matching a regular expression
- tidyselect::where(predicate) - Variables/datasets satisfying a predicate function
- tidyselect::all_of(vars) - All specified variables (error if missing)

- `tidyselect::any_of(vars)` - Any specified variables (silent if missing)
- Range selectors like `Sepal.Length:Petal.Width` - Variables between two positions
- Integer indices (e.g., `1L`, `1L:3L`, `c(1L, 3L, 5L)`) - Select by position. Be careful, must be integer!

The `selected` parameter can use the same syntax but it will be applied to the subset defined in choices. This means that `choices = is.numeric, selected = is.factor` or `choices = c("a", "b", "c"), selected` will imply en empty `selected`.

Warning: Using explicit character values for `selected` with dynamic `choices` may cause issues if the selected values are not present in the dynamically determined choices. Prefer using numeric indices (e.g., `1` for first variable) when `choices` is dynamic.

Structure and element dependencies

The `picks()` function creates a hierarchical structure where elements depend on their predecessors, enabling cascading reactive updates during Shiny sessions.

Element hierarchy:

A `picks` object must follow this order:

1. `datasets()` - to select a dataset. Always the first element (required).
2. `variables()` - To select columns from the chosen dataset.
3. `values()` - To select specific values from the chosen variable(s).

Each element's choices are evaluated within the context of its predecessor's selection.

How dependencies work:

- **Fixed dataset**: When `datasets(choices = "iris")` specifies one dataset, the `variables()` choices are evaluated against that dataset columns.
- **Multiple dataset choices**: When `datasets(choices = c("iris", "mtcars"))` allows multiple options, `variables()` choices are re-evaluated each time the user selects a different dataset. This creates a reactive dependency where variable choices update automatically.
- **Dynamic dataset choices**: When using `datasets(choices = tidyselect::where(is.data.frame))`, all available data frames are discovered at runtime, and variable choices adapt to whichever dataset the user selects.
- **Variable to values**: Similarly, `values()` choices are evaluated based on the selected variable(s), allowing users to filter specific levels or values. When multiple variables are selected, then values will be a concatenation of the columns.

Best practices:

- Always start with `datasets()` - this is enforced by validation
- Use dynamic `choices` in `variables()` when working with multiple datasets to ensure compatibility across different data structures
- Prefer `tidyselect::everything()` or `tidyselect::where()` predicates for flexible variable selection that works across datasets with different schemas
- Use numeric indices for `selected` when `choices` are dynamic to avoid referencing variables that may not exist in all datasets

Important: `values()` requires type-aware configuration:

Why `values()` *is different from* `datasets()` *and* `variables()`*:*
`datasets()` and `variables()` operate on named lists of objects, meaning they work with character-based identifiers. This allows you to use text-based selectors like `starts_with("S")` or `contains("prefix")` consistently for both datasets and variable names.
`values()` is fundamentally different because it operates on the **actual data content** within a selected variable (column). The type of data in the column determines what kind of filtering makes sense:

- `numeric` **columns** (e.g., `age`, `height`, `price`) contain numbers
- `character`/`factor` **columns** (e.g., `country`, `category`, `status`) contain categorical values
- `Date`/`POSIXct` **columns** contain temporal data
- `logical` **columns** contain TRUE/FALSE values

Type-specific UI controls:
The `values()` function automatically renders different UI controls based on data type:

- `numeric` **data**: Creates a `sliderInput` for range selection
  - `choices` must be a numeric vector of length 2: `c(min, max)`
  - `selected` must be a numeric vector of length 2: `c(selected_min, selected_max)`
- **Categorical data** (`character`/`factor`): Creates a `pickerInput` for discrete selection
  - `choices` can be a character vector or predicate function
  - `selected` can be specific values or a predicate function
- `Date`/`POSIXct` **data**: Creates date/datetime range selectors
  - `choices` must be a Date or `POSIXct` vector of length 2
- `logical` **data**: Creates a checkbox or picker for TRUE/FALSE selection

Developer responsibility:
App developers must ensure `values()` **configuration matches the variable type:**

1. **Know your data**: Understand what type of variable(s) users might select
2. **Configure appropriately**: Set `choices` and `selected` to match expected data types
3. **Use predicates for flexibility**: When variable type is dynamic, use predicate functions like `function(x) !is.na(x)` (the default) to handle multiple types safely

Examples of correct usage:

```
# For a numeric variable (e.g., age)
picks(
  datasets(choices = "demographic"),
  variables(choices = "age", multiple = FALSE),
  values(choices = c(0, 100), selected = c(18, 65))
)

# For a categorical variable (e.g., country)
picks(
  datasets(choices = "demographic"),
  variables(choices = "country", multiple = FALSE),
  values(choices = c("USA", "Canada", "Mexico"), selected = "USA")
)

# Safe approach when variable type is unknown - use predicates
picks(
```

```
  datasets(choices = "demographic"),
  variables(choices = tidyselect::everything(), selected = 1L),
  values(choices = function(x) !is.na(x), selected = function(x) !is.na(x))
)
```

Common mistakes to avoid:

```
# WRONG: Using string selectors for numeric data
values(choices = starts_with("5"))  # Doesn't make sense for numeric data!

# WRONG: Providing categorical choices for a numeric variable
values(choices = c("low", "medium", "high"))  # Won't work if variable is numeric!

# WRONG: Providing numeric range for categorical variable
values(choices = c(0, 100))  # Won't work if variable is factor/character!
```

Example: Three-level hierarchy:

```
picks(
  datasets(choices = c("iris", "mtcars"), selected = "iris"),
  variables(choices = tidyselect::where(is.numeric), selected = 1L),
  values(choices = tidyselect::everything(), selected = seq_len(10))
)
```

In this example:

- User first selects a dataset (`iris` or `mtcars`)
- Variable choices update to show only numeric columns from selected dataset
- After selecting a variable, value choices show all unique values from that column

Examples

```
# Select columns from iris dataset using range selector
picks(
  datasets(choices = "iris"),
  variables(choices = Sepal.Length:Petal.Width, selected = 1L)
)

# Single variable selection from iris dataset
picks(
  datasets(choices = "iris", selected = "iris"),
 variables(choices = c("Sepal.Length", "Sepal.Width"), selected = "Sepal.Length", multiple = FALSE)
)

# Dynamic selection: any variable from iris, first selected by default
picks(
  datasets(choices = "iris", selected = "iris"),
  variables(choices = tidyselect::everything(), selected = 1L, multiple = FALSE)
)

# Multiple dataset choices: variable choices will update when dataset changes
picks(
  datasets(choices = c("iris", "mtcars"), selected = "iris"),
```

```
  variables(choices = tidyselect::everything(), selected = 1L, multiple = FALSE)
)

# Select from any dataset, filter by numeric variables
picks(
  datasets(choices = c("iris", "mtcars"), selected = 1L),
  variables(choices = tidyselect::where(is.numeric), selected = 1L)
)

# Fully dynamic: auto-discover datasets and variables
picks(
  datasets(choices = tidyselect::where(is.data.frame), selected = 1L),
  variables(choices = tidyselect::everything(), selected = 1L, multiple = FALSE)
)

# Select categorical variables with length constraints
picks(
  datasets(choices = tidyselect::everything(), selected = 1L),
  variables(choices = is_categorical(min.len = 2, max.len = 15), selected = seq_len(2))
)
```

`picks_module` *Interactive picks*

Description

Creates UI and server components for interactive `picks()` in Shiny modules. The module is based on configuration provided via `picks()` and its responsibility is to determine relevant input values

The module supports both single and combined `picks`:

- Single `picks` objects for a single input
- Named lists of `picks` objects for multiple inputs

Usage

```
picks_ui(id, picks, container = "badge_dropdown")

## S3 method for class 'list'
picks_ui(id, picks, container)

## S3 method for class 'picks'
picks_ui(id, picks, container)

picks_srv(id = "", picks, data)

## S3 method for class 'list'
picks_srv(id, picks, data)
```

```
## S3 method for class 'picks'
picks_srv(id, picks, data)
```

Arguments

id	(character(1)) Shiny module ID
picks	(picks or list) object created by picks() or a named list of such objects
container	(character(1) or function) UI container type. Can be one of htmltools::tags functions. By default, elements are wrapped in a package-specific drop-down.
data	(reactive) Reactive expression returning the data object to be used for populating choices

Details

The module uses S3 method dispatch to handle different ways to provide picks:

- .picks methods handle single 'picks" object
- .list methods handle multiple picks objects

The UI component (picks_ui) creates the visual elements, while the server component (picks_srv) manages the reactive logic,

Value

- picks_ui(): UI elements for the input controls
- picks_srv(): Server-side reactive logic returning the processed data

See Also

picks() for creating 'picks" objects

Examples

```
library(shiny)

example_pick <- picks(
  datasets("ADSL"),
  variables(selected = c("SEX", "COUNTRY", "ARMCD"))
)
ui <- fluidPage(
  picks_ui("my_picks", picks = example_pick),
  h4("Resolved picks:"),
  verbatimTextOutput("result"),
  h4("Table:"),
  tableOutput("table")
)
server <- function(input, output, session) {
  data <- teal.data::teal_data("ADSL" = teal.data::rADSL)
  teal.data::join_keys(data) <- teal.data::default_cdisc_join_keys["ADSL"]
```

```
  selectors <- picks_srv(
    picks = list(my_picks = example_pick),
    data = reactive(data)
  )
  anl <- merge_srv("merge", data = reactive(data), selectors = selectors)
 output$result <- renderPrint(cat(gsub("\033\\[[0-9;]*m", "", format(selectors$my_picks()))))
  output$table <- renderTable(anl$data()$anl)
}

if (interactive()) {
  shinyApp(ui, server)
}
```

---

ranged *Select a range*

Description

Helper to work with ranges. Setting `choices` or `selected` to range using `ranged()` in any of them will automatically create a `numeric`, `Date` or `POSIXct` input to filter. `variables(choices)` must only refer to `numeric`, `Date`, or `POSIXct` columns. An informative error is raised if the resolved column type is unsupported.

Usage

```
ranged(min = -Inf, max = Inf)
```

Arguments

`min`	(`numeric(1)`) Minimal value.
`max`	(`numeric(1)`) Maximal value.

Value

A function that allows the use of a range in `choices` or `selected` of `values()` in `numeric`, `Date`, or `POSIXct` variables.

Examples

```
p <- picks(
  datasets(choices = "mtcars"),
  variables(choices = is.numeric, selected = 1),
  values(choices = ranged(), ranged(20, 30))
)
resolver(data = list("mtcars label" = mtcars), x = p)
```

resolver *Resolve* picks

Description

Resolve iterates through each picks element and determines values .

Usage

```
resolver(x, data)
```

Arguments

x	(picks()) settings for picks.
data	(teal_data() environment or list) any data collection supporting object extraction with [[. Used to determine values of unresolved picks.

Value

resolved picks.

Examples

```
x <- picks(datasets(tidyselect::where(is.data.frame)), variables("a", "a"))
data <- list(
  df1 = data.frame(a = as.factor(LETTERS[1:5]), b = letters[1:5]),
  df2 = data.frame(a = LETTERS[1:5], b = 1:5),
  m = matrix()
)
resolver(x = x, data = data)
```

tidyselectors tidyselect *helpers*

Description

#' **[Experimental]** Predicate functions simplifying picks specification.

Usage

```
is_categorical(min.len, max.len)
```

Arguments

min.len	(integer(1)) minimal number of unique values
max.len	(integer(1)) maximal number of unique values

Value

A tidyselector that can be used directly in choices or selected of variables() in picks().

Examples

```
# select factor column but exclude foreign keys
variables(choices = is_categorical(min.len = 2, max.len = 10))

# Supports tidyselect helpers, e.g. to select all categorical variables with 2 to 10 unique values
dplyr::select(iris, dplyr::where(is_categorical(2, 10)))

p <- picks(
  datasets(is.data.frame, 2L),
  variables(is_categorical(2, 10))
)
resolver(data = list(mtcars = mtcars, iris = iris), x = p)
```

---

tm_merge *Merge module*

Description

Example teal::module containing interactive inputs and displaying results of merge.

Usage

```
tm_merge(label = "merge-module", picks, transformators = list())
```

Arguments

label	(character(1)) Label shown in the navigation item for the module or module group. For modules() defaults to "root". See Details.
picks	(list of picks)
transformators	(list of teal_transform_module) that will be applied to transform module's data input. To learn more check vignette("transform-input-data", package = "teal").

Value

A teal::module object that merges datasets based on user selections and displays the results.

Examples

```
library(teal)

data <- within(teal.data::teal_data(), {
  iris <- iris
  mtcars <- mtcars
})

app <- init(
  data = data,
  modules = modules(
    modules(
      label = "Testing modules",
      tm_merge(
        label = "non adam",
        picks = list(
          a = picks(
            datasets("iris", "iris"),
            variables(
              choices = c("Sepal.Length", "Species"),
              selected = "Sepal.Length"
            ),
            values()
          )
        )
      )
    )
  )
)
if (interactive()) {
  shinyApp(app$ui, app$server, enableBookmarking = "server")
}
```

Index