Package 'DeclareDesign'

October 25, 2026

Type Package

Title A Tidyverse-First Research Design Declaration and Diagnosis Tool

Version 2.0.0

Description Declares, simulates, diagnoses and redesigns research designs. A design is declared as a model, an inquiry, a data strategy and an answer strategy, then simulated with 'purrr' and diagnosed with 'dplyr'. Version 2.0 is a ground-up rewrite of the package on tidyverse conventions.

License MIT + file LICENSE

Encoding UTF-8

Depends R (>= 4.1.0), randomizr (>= 1.0.1), fabricatr (>= 2.0.0), estimatr (>= 2.0.0)

Imports broom, dplyr, generics, parallel, purrr, rlang, tibble, tidyr, utils

Suggests furrr, future, knitr, MASS, progressr, rmarkdown, testthat (>= 3.0.0), withr

VignetteBuilder knitr

Config/testthat/edition 3

Config/roxygen2/version 8.0.0

URL https://declaredesign.org/r/declaredesign/, https://github.com/DeclareDesign/DeclareDesign

BugReports https://github.com/DeclareDesign/DeclareDesign/issues

NeedsCompilation no

Author Alexander Coppock [aut, cre] (ORCID: <https://orcid.org/0000-0002-5733-2386>), Graeme Blair [aut] (ORCID: <https://orcid.org/0000-0001-9164-2102>), Jasper Cooper [aut] (ORCID: <https://orcid.org/0000-0002-8639-3188>), Macartan Humphreys [aut] (ORCID: <https://orcid.org/0000-0001-7029-2326>), Neal Fultz [aut]

Maintainer Alexander Coppock <acoppock@gmail.com>

Repository CRAN

Date/Publication 2026-10-25 14:30:36 UTC

Contents

DeclareDesign-package . . . 3
+.dd . . . 4
cite_design . . . 5
compare_designs . . . 5
compare_diagnoses . . . 6
declare-defunct . . . 8
declare_assignment . . . 8
declare_diagnosands . . . 9
declare_estimator . . . 10
declare_inquiry . . . 12
declare_measurement . . . 14
declare_model . . . 15
declare_notes . . . 16
declare_parameters . . . 18
declare_population . . . 19
declare_sampling . . . 20
declare_step . . . 21
declare_test . . . 22
default_diagnosands . . . 23
delete_step . . . 24
design_notes . . . 25
design_parameters . . . 26
diagnose_design . . . 27
diagnose_simulations . . . 28
draw_data . . . 29
draw_estimands . . . 30
draw_estimates . . . 31
expand_design . . . 31
format.diagnosis . . . 32
get_diagnosands . . . 34
get_estimates . . . 35
get_simulations . . . 36
insert_step . . . 36
label_estimator . . . 37
model_handler . . . 38
pop.var . . . 39
print.compared_diagnoses . . . 39
print.design . . . 40
print.design_step . . . 41
print.diagnosis . . . 41
print.objects . . . 42
redesign . . . 43

replace_step 44
run_design 45
select_diagnosands 46
set_citation 47
set_diagnosands 48
simulate_design 49
summary.design 51
summary.diagnosis 52
tidy.diagnosis 52
tidy_try 53
vars 54

Index **55**

`DeclareDesign-package` *DeclareDesign: A Tidyverse-First Research Design Declaration and Diagnosis Tool*

Description

Declares, simulates, diagnoses and redesigns research designs. A design is declared as a model, an inquiry, a data strategy and an answer strategy, then simulated with purrr and diagnosed with dplyr. Version 2.0 is a ground-up rewrite of the package on tidyverse conventions.

Details

A design is a sequence of steps combined with +. Each step is one of the four elements of a research design:

- `declare_model()` describes the world the design runs in.
- `declare_inquiry()` records the question. Its value on a draw is the estimand.
- `declare_sampling()`, `declare_assignment()`, and `declare_measurement()` make up the data strategy: who is observed, what they are assigned, and what is measured.
- `declare_estimator()` and `declare_test()` are the answer strategy.

`diagnose_design()` simulates a design many times and reports how its estimates compare to its estimands: bias, power, coverage, and the rest. `redesign()` changes a parameter and returns a new design, so two versions can be compared before any data are collected.

```
design <-
  declare_model(N = 100, U = rnorm(N), Y_Z_0 = U, Y_Z_1 = U + 0.5) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_assignment(Z = sample(rep(0:1, length.out = N))) +
  declare_measurement(Y = Y_Z_0 * (1 - Z) + Y_Z_1 * Z) +
  declare_estimator(Y ~ Z, .method = lm, term = "Z", inquiry = "ATE")

diagnose_design(design)
```

Author(s)

Maintainer: Alexander Coppock <acoppock@gmail.com> (ORCID)

Authors:

- Alexander Coppock <acoppock@gmail.com> (ORCID)
- Graeme Blair <graeme.blair@gmail.com> (ORCID)
- Jasper Cooper <jaspercooper@gmail.com> (ORCID)
- Macartan Humphreys <macartan@gmail.com> (ORCID)
- Neal Fultz <nfultz@gmail.com>

See Also

Useful links:

- https://declaredesign.org/r/declaredesign/
- https://github.com/DeclareDesign/DeclareDesign
- Report bugs at https://github.com/DeclareDesign/DeclareDesign/issues

---

`+.dd` *Combine design steps into a design*

Description

The + operator concatenates `design_step` and `design` objects into a single `design`. `design + NULL` is a no-op that returns the design unchanged, which makes conditional step addition convenient.

Usage

```
## S3 method for class 'dd'
e1 + e2
```

Arguments

`e1`	(required) A `design_step`, a `design`, or `NULL`.
`e2`	(required) A `design_step`, a `design`, or `NULL`.

Value

A `design` (class `c("design", "dd")`): a list of the steps of `e1` followed by those of `e2`, named by their labels.

Examples

```
d <- declare_model(N = 50, U = rnorm(N), Y = U) +
  declare_inquiry(mu = mean(Y))
length(d)
```

cite_design *Retrieve a design's citation*

Description

Returns the citation `set_citation()` attached to the design, or, when none was attached, the citation for the DeclareDesign package itself.

Usage

```
cite_design(design)
```

Arguments

`design`	(required) A `design`.

Value

The design's citation, a `bibentry` or a string, or `utils::citation("DeclareDesign")` when none has been set.

See Also

Other citing a design: `set_citation()`

Examples

```
design <- set_citation(declare_model(N = 10, Y = rnorm(N)),
                       title = "Example", author = "Coppock", year = 2026)
cite_design(design)
```

compare_designs *Defunct comparison helpers*

Description

These functions existed in DeclareDesign 1.x but are not part of 2.0. Each compared two designs by printing their code, their draws, or their summaries side by side, which is a job for a diff of the two scripts. They error with a message saying so.

Usage

```
compare_designs(...)

compare_design_code(...)

compare_design_data(...)

compare_design_estimates(...)

compare_design_inquiries(...)

compare_design_summaries(...)

print_code(...)
```

Arguments

...	(optional) Ignored.

Details

compare_diagnoses(), which compares what two designs achieve rather than how they are written, is implemented.

Value

Never returns; always raises an error.

---

compare_diagnoses *Compare the diagnosands of two designs*

Description

Diagnoses both designs and reports, for every diagnosand they have in common, the value under each design and the difference between them. The bootstrap replicates behind each diagnosis supply a standard error and a confidence interval for the difference.

Usage

```
compare_diagnoses(
  design1,
  design2,
  sims = 500,
  bootstrap_sims = 100,
  diagnosands = NULL,
  merge_by_estimator = TRUE,
  alpha = 0.05
)
```

Arguments

design1, design2	(required) A design or a diagnosis.
sims	(optional) Number of simulations, used only for arguments that are designs. Defaults to 500.
bootstrap_sims	(optional) Number of bootstrap replicates, used only for arguments that are designs. Defaults to 100. With 0, differences are reported without standard errors or intervals.
diagnosands	(optional) A diagnosands design_step applied to both designs, so the two sides are compared on the same footing by construction. NULL (the default) uses what diagnose_design() would: diagnosands attached with set_diagnosands(), else default_diagnosands(). Used only for arguments that are designs; a diagnosis passed in keeps the diagnosands it was computed with.
merge_by_estimator	(optional) Match estimators by label. TRUE by default; FALSE compares every pair of estimators within an inquiry.
alpha	(optional) One minus the coverage of the reported interval. Defaults to 0.05.

Details

Rows are matched on whichever of inquiry, estimator, and term both diagnoses carry. With merge_by_estimator = FALSE, estimators are not matched to each other and every pair within an inquiry is compared, which is how to compare two estimators that carry different labels.

mean_difference is the difference of the two point estimates. The bootstrap enters only through se_difference and the interval. The two designs are simulated independently, so the replicates are paired arbitrarily; the interval is a percentile interval on that difference.

Either argument may be a design or an already computed diagnosis, so a diagnosis that took a long time to run can be reused. Passing two diagnoses computed with different diagnosands compares whatever they have in common.

Value

A compared_diagnoses object: a list with three elements. compared_diagnoses_df is a tibble with one row per diagnosand per matched group, holding the grouping columns, diagnosand, mean_1, mean_2, mean_difference, se_1, se_2, se_difference, conf.low, and conf.high (the last four are NA without bootstrap replicates). diagnosis1 and diagnosis2 are the two diagnosis objects compared.

See Also

Other simulation and diagnosis: diagnose_design(), diagnose_simulations(), get_diagnosands(), get_simulations(), simulate_design()

Examples

```
design <- declare_parameters(n_units = 100) +
  declare_model(N = n_units, U = rnorm(N), Y_Z_0 = U, Y_Z_1 = U + 0.5) +
```

```
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_assignment(Z = randomizr::complete_ra(N)) +
  declare_measurement(Y = Y_Z_1 * Z + Y_Z_0 * (1 - Z)) +
  declare_estimator(Y ~ Z, .method = lm, term = "Z", inquiry = "ATE",
                    label = "ols")
bigger <- redesign(design, n_units = 200)
compare_diagnoses(design, bigger, sims = 20, bootstrap_sims = 20)
```

`declare-defunct` *Defunct step declarations from DeclareDesign 1.x*

Description

`declare_potential_outcomes()` and `declare_reveal()` were separate steps in 1.x. In 2.0 potential outcomes belong in `declare_model()` and revealing belongs in `declare_measurement()`, so neither has a 2.0 equivalent as a step of its own.

Usage

```
declare_potential_outcomes(...)

declare_reveal(...)
```

Arguments

`...`	(optional) Ignored.

Details

They exist here only so that 1.x code fails with a message saying what to write instead. Without them R reports `could not find function`, which says nothing about the replacement.

Value

Never returns; always raises an error.

`declare_assignment` *Declare an assignment procedure*

Description

Assigns units to treatment conditions, adding the assignment variable (`Z` by convention) to the data. The randomization itself is usually a randomizr function such as `randomizr::complete_ra()`.

Usage

```
declare_assignment(..., label = "assignment", draws = 1L)
```

Arguments

...	(optional) Named arguments forwarded to fabricatr::fabricate(). Unquoted expressions are evaluated lazily in the caller's environment with access to the current data frame.
label	(optional) Step label. Defaults to "assignment".
draws	(optional) Number of nested draws for this step. Defaults to 1. When > 1, the step is re-executed draws times for each upstream draw during nested simulation.

Value

A design_step (class c("design_step", "dd", "function")): a function that takes the current data frame and returns it with the declared columns added. Add it to a design with +.

See Also

Other design declarations: declare_estimator(), declare_inquiry(), declare_measurement(), declare_model(), declare_notes(), declare_parameters(), declare_sampling(), declare_step(), declare_test()

Examples

```
design <-
  declare_model(N = 100, U = rnorm(N)) +
  declare_assignment(Z = sample(rep(0:1, length.out = N)))

table(draw_data(design)$Z)
```

---

declare_diagnosands *Declare diagnosands*

Description

Defines the summary statistics computed across simulations during diagnosis. Each named expression is evaluated against the simulations data frame, grouped by inquiry, estimator, term, and outcome where present.

Usage

```
declare_diagnosands(..., subset = NULL, alpha = 0.05, label = "diagnosands")
```

Arguments

...	(optional) Named expressions defining diagnosands, each evaluated on the simulations table of one group and returning a single value.
subset	(optional) An expression evaluated on the simulations table; only rows for which it is TRUE enter the diagnosands. NULL (the default) keeps every simulation.
alpha	(optional) Significance level. Any diagnosand expression that mentions alpha sees this value. Defaults to 0.05.
label	(optional) Step label. Defaults to "diagnosands".

Value

A diagnosands design_step: a function that takes a simulations table and returns a tibble with columns diagnosand and value, one row per diagnosand. Its dots attribute carries the diagnosand quosures. Pass it to diagnose_design() as diagnosands.

See Also

Other diagnosands: default_diagnosands(), format.diagnosis(), select_diagnosands(), set_diagnosands()

Examples

```
diags <- declare_diagnosands(
  bias = mean(estimate - estimand, na.rm = TRUE),
  rmse = sqrt(mean((estimate - estimand)^2, na.rm = TRUE))
)

design <-
  declare_model(N = 30, U = rnorm(N), Y_Z_0 = U, Y_Z_1 = U + 0.5) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_assignment(Z = sample(rep(0:1, length.out = N))) +
  declare_measurement(Y = Y_Z_0 * (1 - Z) + Y_Z_1 * Z) +
  declare_estimator(Y ~ Z, .method = lm, term = "Z", inquiry = "ATE")

diagnose_design(design, sims = 5, bootstrap_sims = 0, diagnosands = diags)

# power at the 10 percent level, computed on significant simulations only
declare_diagnosands(power = mean(p.value <= alpha), alpha = 0.1)
```

---

declare_estimator *Declare an estimator*

Description

Declares the answer strategy: a function that produces an estimate of an inquiry from the observed data. The estimate is tidied to one row per term and joined to its estimand by the inquiry label during diagnosis.

Usage

```
declare_estimator(
  ...,
  .method = NULL,
  .summary = tidy_try,
  inquiry = NULL,
  term = NULL,
  label = "estimator",
  handler = NULL,
  draws = 1L
)
```

Arguments

`...`	(optional) Arguments forwarded, as written, to `.method` (or to `handler`), which evaluates them itself with the data as `data`. Typically the formula appears first (e.g., `Y ~ Z`); without a `handler`, these must include whatever `.method` needs. A legacy `model =` argument is read as `.method` with a deprecation warning rather than forwarded.
`.method`	(optional) The model-fitting function. Defaults to `estimatr::lm_robust()`.
`.summary`	(optional) Function used to tidy `.method`'s output. Defaults to `tidy_try()`.
`inquiry`	(optional) The label of the inquiry this estimator targets, as a string (`inquiry = "ATE"`), or a character vector of labels. Passing the inquiry step object is an error. The label joins the estimate to its estimand in every table of estimates and in the diagnosis.
`term`	(optional) Which model terms to report. A character vector keeps exactly those terms, in that order, and errors if any is absent from the tidied output; `TRUE` keeps every term including the intercept. When left unset (or `FALSE`), only the first non-intercept term is reported, so `declare_estimator(Y ~ Z + X)` returns the `Z` row alone; a `handler`'s table is kept whole unless `term` names terms. Evaluated on every draw, so it may name a design parameter.
`label`	(optional) Step label. Defaults to `"estimator"`. When two estimator or test steps in one design share a label, `+` relabels them from their formulas, with a message.
`handler`	(optional) A function. When supplied, the estimator bypasses `.method` and `.summary` and instead calls `handler(data, ...)` with the dots as written, not evaluated; it must return a tidy table.
`draws`	(optional) Number of nested draws for this step. Defaults to `1`. When `> 1`, the step is re-executed `draws` times for each upstream draw during nested simulation.

Value

A `design_step` (class `c("design_step", "dd", "function")`): a function that takes a data frame and returns a tibble of estimates, one row per reported term, with `estimator` and, when `inquiry` is set, `inquiry` columns. A draw on which `.method` errors returns a row with `error = TRUE` and the message rather than stopping the run. Add it to a design with `+`.

See Also

Other design declarations: declare_assignment(), declare_inquiry(), declare_measurement(), declare_model(), declare_notes(), declare_parameters(), declare_sampling(), declare_step(), declare_test()

Examples

```
design <- declare_model(N = 50, U = rnorm(N), Y = U) +
  declare_inquiry(mu = mean(Y)) +
  declare_estimator(Y ~ 1, .method = lm, term = "(Intercept)",
                    inquiry = "mu", label = "ols")
draw_estimates(design)
```

---

declare_inquiry *Declare an inquiry*

Description

Each named argument records an inquiry: an expression evaluated on the data as it stands when the step runs. Its value on a given draw is the estimand.

Usage

```
declare_inquiry(
  ...,
  subset = NULL,
  label = "inquiry",
  handler = NULL,
  draws = 1L
)

declare_inquiries(
  ...,
  subset = NULL,
  label = "inquiry",
  handler = NULL,
  draws = 1L
)

declare_estimand(
  ...,
  subset = NULL,
  label = "inquiry",
  handler = NULL,
  draws = 1L
)
```

```
declare_estimands(
  ...,
  subset = NULL,
  label = "inquiry",
  handler = NULL,
  draws = 1L
)
```

Arguments

`...`	(optional) Named expressions. Each expression becomes an inquiry whose numeric value is recorded in the simulation output. An unnamed expression takes the step's `label` as its name.
`subset`	(optional) Unquoted expression evaluated on the data; only rows where it is `TRUE` enter the estimands.
`label`	(optional) Step label. Defaults to `"inquiry"`. A step that declares exactly one named inquiry takes that name as its label instead, overriding `label`.
`handler`	(optional) A function. When supplied, `...` arguments are evaluated against the data and passed to `handler()` rather than being treated as named scalar inquiries; useful for vectorised inquiry sets (for example, `handler = tibble`).
`draws`	(optional) Number of nested draws for this step. Defaults to `1`. When `> 1`, the step is re-executed `draws` times for each upstream draw during nested simulation.

Value

A `design_step` (class `c("design_step", "dd", "function")`): a function that takes a data frame and returns a tibble with columns `inquiry` and `estimand`, one row per inquiry. Add it to a design with `+`.

See Also

Other design declarations: `declare_assignment()`, `declare_estimator()`, `declare_measurement()`, `declare_model()`, `declare_notes()`, `declare_parameters()`, `declare_sampling()`, `declare_step()`, `declare_test()`

Examples

```
design <-
  declare_model(N = 100, U = rnorm(N), Y_Z_0 = U, Y_Z_1 = U + 0.5) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0))

draw_estimands(design)

# Placed before declare_sampling(), the question is about the population;
# placed after, about the sample.
design <-
  declare_model(N = 500, Y = rnorm(N)) +
  declare_inquiry(population_mean = mean(Y)) +
```

```
  declare_sampling(S = sample(rep(0:1, length.out = N))) +
  declare_inquiry(sample_mean = mean(Y))

draw_estimands(design)
```

`declare_measurement` *Declare measurement*

Description

Records what the study observes: the measured outcome, typically `Y` revealed from the potential outcomes given the assignment `Z`. Columns are added to the data the way `declare_model()` adds them, without a new ID column.

Usage

```
declare_measurement(..., label = "measurement", draws = 1L)
```

Arguments

`...`	(optional) Named arguments forwarded to `fabricatr::fabricate()`. Unquoted expressions are evaluated lazily in the caller's environment with access to the current data frame.
`label`	(optional) Step label. Defaults to `"measurement"`.
`draws`	(optional) Number of nested draws for this step. Defaults to `1`. When `> 1`, the step is re-executed `draws` times for each upstream draw during nested simulation.

Value

A `design_step` (class `c("design_step", "dd", "function")`): a function that takes the current data frame and returns it with the declared columns added. Add it to a design with `+`.

See Also

Other design declarations: `declare_assignment()`, `declare_estimator()`, `declare_inquiry()`, `declare_model()`, `declare_notes()`, `declare_parameters()`, `declare_sampling()`, `declare_step()`, `declare_test()`

Examples

```
design <-
  declare_model(N = 100, U = rnorm(N), Y_Z_0 = U, Y_Z_1 = U + 0.5) +
  declare_assignment(Z = sample(rep(0:1, length.out = N))) +
  declare_measurement(Y = Y_Z_0 * (1 - Z) + Y_Z_1 * Z)

head(draw_data(design))
```

declare_model *Declare the data-generating model*

Description

The model describes the world the design runs in. It does not have to be correct: declaring one you doubt is how you find out whether the design still works when you are wrong about it.

Usage

```
declare_model(..., label = "model", draws = 1L)
```

Arguments

`...`	(optional) Named arguments forwarded to `fabricatr::fabricate()`. Unquoted expressions are evaluated lazily in the caller's environment with access to the current data frame.
`label`	(optional) Step label. Defaults to `"model"`.
`draws`	(optional) Number of nested draws for this step. Defaults to `1`. When `> 1`, the step is re-executed `draws` times for each upstream draw during nested simulation.

Details

Declares a step that builds (or augments) the population data via `fabricatr::fabricate()`. The first model step in a design receives `data = NULL`, so it must specify `N` (and any variables) directly. Subsequent model steps add columns to the existing data.

Value

A `design_step` (class `c("design_step", "dd", "function")`): a function that takes the current data frame and returns a tibble with the declared columns added (or built from scratch, for the first step). Add it to a design with `+`.

See Also

Other design declarations: `declare_assignment()`, `declare_estimator()`, `declare_inquiry()`, `declare_measurement()`, `declare_notes()`, `declare_parameters()`, `declare_sampling()`, `declare_step()`, `declare_test()`

Examples

```
design <- declare_model(N = 50, U = rnorm(N), Y = U + 1)

head(draw_data(design))

# A second model step adds columns to the data the first one made.
design <-
```

```
  declare_model(N = 50, U = rnorm(N)) +
  declare_model(Y_Z_0 = U, Y_Z_1 = U + 0.5)

head(draw_data(design))
```

`declare_notes` *Declare quantities computed while a design runs*

Description

Records a named quantity at a point in the pipeline and makes it available to every step that follows. A note is computed when the design runs, at the position it is declared, from whatever is in scope there: the parameters, the notes already taken, and the data as they stand at that moment.

```
declare_model(N = 100, Y = rnorm(N)) +
  declare_notes(tallest = max(Y)) +
  declare_sampling(S = complete_rs(N, n = 10)) +
  declare_inquiry(shortfall = mean(tallest - Y))
```

`tallest` is the largest value in the population, and the inquiry reads it after sampling has thrown most of that population away. Notes are what carry a quantity across a step that destroys the data it was computed from.

Usage

```
declare_notes(..., label = "notes")
```

Arguments

`...`	(optional) Named quantities. Each is evaluated when the design runs, in order, and may read the parameters, the notes taken before it, and the current data.
`label`	(optional) Step label. Defaults to `"notes"`.

Details

A note is fixed from the moment it is taken until the end of that run. It is not recomputed by the steps that read it, so `tallest` above stays the population maximum rather than becoming the sample maximum. It is redrawn on the next run, because the data are. Declaring the same name again later overwrites it from that point on and leaves the earlier steps alone.

A note that reads no data is the ordinary way to write a quantity derived from a parameter:

```
declare_parameters(m_arms = 3) +
  declare_notes(ks = seq_len(m_arms)[-1],
                term_names = paste0("factor(Z)", ks))
```

`m_arms` is the one thing a caller sets: `redesign(design, m_arms = 4)`. `ks` and `term_names` follow from it, and cannot be set independently without contradicting it, so `redesign()` refuses them and `design_parameters()` does not list them. That is the difference between a note and a parameter. A `declare_parameters()` step names what a caller may change; a `declare_notes()` step names what the design works out for itself.

Within one call, each note may read the ones declared before it. A note may not share a name with a declared parameter.

The step generates no data and adds no rows to a design's output.

Value

A `design_step` (class `c("design_step", "dd", "function")`) whose `dots` attribute holds the note expressions. Called on a data frame, it returns the data unchanged. Add it to a design with `+`.

See Also

`declare_parameters()` for the quantities a caller may change, `design_notes()` for the notes a design takes.

Other design declarations: `declare_assignment()`, `declare_estimator()`, `declare_inquiry()`, `declare_measurement()`, `declare_model()`, `declare_parameters()`, `declare_sampling()`, `declare_step()`, `declare_test()`

Examples

```
# A quantity that outlives the data it is computed from.
design <-
  declare_model(N = 100, Y = rnorm(N)) +
  declare_notes(population_mean = mean(Y)) +
  declare_sampling(S = complete_rs(N, n = 10)) +
  declare_inquiry(gap = mean(Y) - population_mean)
draw_estimands(design)

# A quantity derived from a parameter, so `m_arms` is the only name a
# redesign can set.
design <-
  declare_parameters(m_arms = 3) +
  declare_notes(inquiry_names = paste0("ate_", seq_len(m_arms)[-1])) +
  declare_model(N = 60, u = rnorm(N)) +
  declare_inquiry(handler = function(data) {
    data.frame(inquiry = inquiry_names, estimand = 0)
  })
design_parameters(design)
design_notes(design)
draw_estimands(redesign(design, m_arms = 4))
```

`declare_parameters` *Declare the parameters of a design*

Description

Names the quantities a design is built from, so that `redesign()` changes them and nothing else. A parameter declared here is bound for every step that follows, and takes precedence over a binding of the same name in the workspace.

Parameters are ordinary R expressions evaluated once, when the design is built, and each may read the ones declared before it: `declare_parameters(m_arms = 3, ks = seq_len(m_arms)[-1])`. They are not redrawn on each simulation, which is what separates a parameter from a random variable: `declare_parameters(u = rnorm(1))` fixes one number for the life of the design.

Usage

```
declare_parameters(..., label = "parameters")
```

Arguments

`...`	(optional) Named parameters. Each is evaluated once, in order, and may read the parameters declared before it.
`label`	(optional) Step label. Defaults to `"parameters"`.

Details

Without a `declare_parameters()` step, a design's parameters are whatever its expressions read out of the environment they were written in, and `redesign()` finds them by name. That works and it stays supported: `declare_model(N = n_units, ...)` with `n_units` defined outside the design needs no declaration.

What does need a name is a number written into the design. `declare_model(N = 500)` puts 500 in the design, and nothing outside it names that number, so `redesign(design, N = 1000)` errors rather than rewriting the argument because its name matched. Giving the value a name outside the design fixes that on its own, and is the usual answer; declaring it here earns its place when several steps read the value, or when a column shares its name. A redesign changes the declared parameter, and a column of the same name is left alone, because it belongs to the data rather than to the design.

The step generates no data and is skipped when the design runs.

Value

A `design_step` (class `c("design_step", "dd", "function")`) that carries the declared parameters and their values. Called on a data frame, it returns the data unchanged. Add it to a design with `+`.

See Also

Other design declarations: declare_assignment(), declare_estimator(), declare_inquiry(), declare_measurement(), declare_model(), declare_notes(), declare_sampling(), declare_step(), declare_test()

Examples

```
design <-
  declare_parameters(n_units = 100, effect = 0.5) +
  declare_model(N = n_units, U = rnorm(N), Y_Z_0 = U, Y_Z_1 = U + effect) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_assignment(Z = sample(rep(0:1, length.out = N))) +
  declare_measurement(Y = Y_Z_0 * (1 - Z) + Y_Z_1 * Z)

nrow(draw_data(design))
nrow(draw_data(redesign(design, n_units = 40)))

# A later parameter may read an earlier one.
design <- declare_parameters(m_arms = 3, ks = seq_len(m_arms)[-1]) +
  declare_model(N = 60) +
  declare_assignment(Z = complete_ra(N, conditions = seq_len(m_arms)))
design_parameters(design)
```

---

declare_population *Deprecated alias for* declare_model()

Description

declare_population() is retained for compatibility with older scripts but will be removed in a future version. Use declare_model() instead.

Usage

```
declare_population(...)
```

Arguments

...	(optional) Forwarded to declare_model().

Value

A design_step, as declare_model() returns.

declare_sampling *Declare a sampling procedure*

Description

Draws the sample from the population, keeping the rows where the sampling indicator S is 1. Name the indicator S, as in declare_sampling(S = complete_rs(N, n = 50)), or say which rows to keep with filter = ; a sampling step that produces neither keeps every row and warns.

Usage

```
declare_sampling(..., filter = NULL, label = "sampling", draws = 1L)
```

Arguments

...	(optional) Named arguments forwarded to fabricatr::fabricate(). Unquoted expressions are evaluated lazily in the caller's environment with access to the current data frame.
filter	(optional) Unquoted expression evaluated against the data; rows where the expression is TRUE are retained, and S is then not consulted.
label	(optional) Step label. Defaults to "sampling".
draws	(optional) Number of nested draws for this step. Defaults to 1. When > 1, the step is re-executed draws times for each upstream draw during nested simulation.

Value

A design_step (class c("design_step", "dd", "function")): a function that takes the current data frame and returns the rows sampled, with any declared columns added. Add it to a design with +.

See Also

Other design declarations: declare_assignment(), declare_estimator(), declare_inquiry(), declare_measurement(), declare_model(), declare_notes(), declare_parameters(), declare_step(), declare_test()

Examples

```
design <-
  declare_model(N = 200, Y = rnorm(N)) +
  declare_sampling(S = sample(rep(0:1, length.out = N)))

nrow(draw_data(design))
```

declare_step *Declare a custom data-handling step*

Description

Wraps an arbitrary handler function as a step in the design. The handler must accept `data` as its first argument and return a data frame.

Usage

```
declare_step(handler, ..., label = "custom_step", draws = 1L)
```

Arguments

`handler`	(required) A function whose first argument is `data`.
`...`	(optional) Additional arguments passed to `handler` **as written**, so the handler evaluates them itself. A bare column name arrives as a name, which is what lets `tidyr::pivot_wider(id_cols = pair)` and the dplyr verbs select and mask as they normally do. This is the same rule `declare_estimator()` follows. A handler that does no evaluation of its own and wants a value computed from the data takes a closure instead: `declare_step(handler = function(data) f(data, cutoff = mean(data$a)))`.
`label`	(optional) Step label. Defaults to `"custom_step"`.
`draws`	(optional) Number of nested draws for this step. Defaults to `1`. When `> 1`, the step is re-executed `draws` times for each upstream draw during nested simulation.

Value

A `design_step` (class `c("design_step", "dd", "function")`): a function that takes the current data frame and returns what `handler` returns for it. Add it to a design with `+`.

See Also

Other design declarations: `declare_assignment()`, `declare_estimator()`, `declare_inquiry()`, `declare_measurement()`, `declare_model()`, `declare_notes()`, `declare_parameters()`, `declare_sampling()`, `declare_test()`

Examples

```
step <- declare_step(handler = function(data, k) {
  data$X2 <- data$X * k
  data
}, k = 2)
df <- data.frame(X = 1:5)
step(df)
```

```
# A tidyselect handler receives the column names, not their contents.
wide <- declare_step(
  id_cols = pair, names_from = role, values_from = c(ID, a),
  handler = tidyr::pivot_wider
)
long <- data.frame(pair = rep(1:3, each = 2), role = rep(c("A", "B"), 3),
                   ID = sprintf("%03d", 1:6), a = 1:6)
wide(long)
```

`declare_test` *Declare a hypothesis test*

Description

Like `declare_estimator()` but the result table does not gain an `inquiry` column; intended for tests not tied to an estimand.

Usage

```
declare_test(
  ...,
  .method = NULL,
  .summary = tidy_try,
  term = NULL,
  label = "test",
  handler = NULL,
  draws = 1L
)
```

Arguments

`...`	(optional) Arguments forwarded, as written, to `.method` (or to `handler`), which evaluates them itself with the data as `data`. Typically the formula appears first (e.g., `Y ~ Z`); without a `handler`, these must include whatever `.method` needs. A legacy `model =` argument is read as `.method` with a deprecation warning rather than forwarded.
`.method`	(optional) The model-fitting function. Defaults to `estimatr::lm_robust()`, as in `declare_estimator()`.
`.summary`	(optional) Function used to tidy `.method`'s output. Defaults to `tidy_try()`.
`term`	(optional) Which model terms to report. A character vector keeps exactly those terms, in that order, and errors if any is absent from the tidied output; `TRUE` keeps every term including the intercept. When left unset (or `FALSE`), only the first non-intercept term is reported, so `declare_estimator(Y ~ Z + X)` returns the `Z` row alone; a `handler`'s table is kept whole unless `term` names terms. Evaluated on every draw, so it may name a design parameter.
`label`	(optional) Step label. Defaults to `"test"`. When two estimator or test steps in one design share a label, `+` relabels them from their formulas, with a message.

handler (optional) A function. When supplied, the test bypasses `.method` and `.summary` and instead calls `handler(data, ...)` with the dots as written, not evaluated; it must return a tidy table.

draws (optional) Number of nested draws for this step. Defaults to `1`. When `> 1`, the step is re-executed `draws` times for each upstream draw during nested simulation.

Value

A `design_step`, as `declare_estimator()` returns, whose estimates carry no `inquiry` column.

See Also

Other design declarations: `declare_assignment()`, `declare_estimator()`, `declare_inquiry()`, `declare_measurement()`, `declare_model()`, `declare_notes()`, `declare_parameters()`, `declare_sampling()`, `declare_step()`

Examples

```
design <- declare_model(N = 30, Z = rep(0:1, 15), Y = Z + rnorm(N)) +
  declare_test(Y ~ Z, term = "Z", label = "diff")
draw_estimates(design)
```

---

default_diagnosands *Default diagnosands*

Description

Returns the standard set of diagnosands. A diagnosand is a summary of a design's behaviour across simulations, so each one is computed over the whole set of draws rather than within any single draw.

Usage

```
default_diagnosands()
```

Details

`mean_estimand` Average of the estimand across draws. The estimand varies from draw to draw whenever the model is stochastic.

`mean_estimate` Average of the estimator's estimate across draws.

`bias` Average of estimate minus estimand. Zero means the estimator lands on the estimand on average, not that it is close on any one draw.

`sd_estimate` Standard deviation of the estimates, which is the true sampling variability of the design.

`rmse` Root mean squared error, the average distance between estimate and estimand. It counts bias and variability together, so it is the one to read when an estimator trades one against the other.

`power` Share of draws in which the p-value is at or below 0.05. Read it as power only when the effect is real; under a true null the same number is the false positive rate.

`coverage` Share of draws in which the confidence interval contains the estimand. A 95 percent interval should give 0.95. Materially lower means the intervals are too narrow and the design overstates its own precision.

Value

A diagnosands `design_step`, as `declare_diagnosands()` returns, holding the seven diagnosands listed under Details.

See Also

Other diagnosands: `declare_diagnosands()`, `format.diagnosis()`, `select_diagnosands()`, `set_diagnosands()`

Examples

```
default_diagnosands()
```

`delete_step` *Delete a step from a design (deprecated)*

Description

Deprecated in 2.0, and warns once per session. Reconstruct the design with + instead.

Usage

```
delete_step(design, step)
```

Arguments

`design`	(required) A `design`.
`step`	(required) A label, integer, or `design_step` naming the step to delete.

Value

A `design` without that step.

See Also

Other modifying a design: `design_notes()`, `design_parameters()`, `expand_design()`, `insert_step()`, `redesign()`, `replace_step()`

Examples

```
design <- declare_model(N = 30, Y = rnorm(N)) +
  declare_inquiry(mu = mean(Y))
delete_step(design, "mu") |> names()
```

`design_notes` *Report the notes a design takes*

Description

Lists the names a `declare_notes()` step declares and the expression each is computed from. Unlike `design_parameters()`, no value is reported: a note has one only while the design is running, and a note computed from data has a different one on every draw.

The two functions divide a design's names between them. What `design_parameters()` lists can be changed with `redesign()`; what `design_notes()` lists cannot, and follows from what can.

Usage

```
design_notes(design)
```

Arguments

`design`	(required) A `design` or a `design_step`.

Value

A data frame with one row per note and columns `name`, `expr` (the expression, as a character label), and `step` (the position of the step that declares it).

See Also

`design_parameters()`.

Other modifying a design: `delete_step()`, `design_parameters()`, `expand_design()`, `insert_step()`, `redesign()`, `replace_step()`

Examples

```
design <- declare_parameters(m_arms = 3) +
  declare_notes(ks = seq_len(m_arms)[-1]) +
  declare_model(N = 60)
design_notes(design)
```

`design_parameters` *Report the parameters a design can be redesigned on*

Description

Lists the names `redesign()` can change: the parameters a `declare_parameters()` step declares, and the objects the design's expressions read out of the environments they were written in.

Symbols that resolve to a package (`rnorm`, `complete_ra`) are not parameters and are left out, and neither are names that resolve to nothing, because those are columns an earlier step created.

Usage

```
design_parameters(design)
```

Arguments

`design`	(required) A `design` or a `design_step`.

Value

A data frame of class `objects` with one row per name per step: `name`, `value` (a display snippet of the value), `kind` (`scalar`, `vector`, `list`, `data`, `function`, or `other`), `declared` (whether a `declare_parameters()` step declares it), and `step` (the position of the step that reads it), plus the list columns `quosure` and `env` that locate the binding. Rows are in step order. It prints aggregated to one row per name.

See Also

Other modifying a design: `delete_step()`, `design_notes()`, `expand_design()`, `insert_step()`, `redesign()`, `replace_step()`

Examples

```
design <- declare_parameters(n_units = 100) +
  declare_model(N = n_units, Y = rnorm(N))
design_parameters(design)
```

`diagnose_design` *Diagnose a design*

Description

Simulates the design and computes diagnosands (bias, RMSE, power, coverage, and so on) across simulations. If `bootstrap_sims > 0`, bootstrap standard errors for each diagnosand are also reported.

Usage

```
diagnose_design(
  ...,
  sims = NULL,
  bootstrap_sims = 100,
  diagnosands = NULL,
  progress = FALSE
)

diagnose_designs(
  ...,
  sims = NULL,
  bootstrap_sims = 100,
  diagnosands = NULL,
  progress = FALSE
)
```

Arguments

`...`	(required) One or more `design` objects, or lists of them. A simulations data frame from `simulate_design()` in first position is diagnosed directly, as `diagnose_simulations()` does.
`sims`	(optional) Number of simulations. Defaults to `NULL`. When `NULL`, designs with step-level `draws` run in nested mode; otherwise the design runs `500` flat simulations. When supplied alongside step-level `draws`, the `draws` are ignored and a warning is emitted.
`bootstrap_sims`	(optional) Number of bootstrap replicates for diagnosand standard errors. Defaults to `100`; `0` skips the bootstrap.
`diagnosands`	(optional) A diagnosands `design_step` (e.g., from `declare_diagnosands()`). `NULL` (the default) uses diagnosands attached with `set_diagnosands()`, else `default_diagnosands()`.
`progress`	(optional) If `TRUE`, display a progress bar for this call by wrapping it in `progressr::with_progress()`. Defaults to `FALSE`. See `simulate_design()`.

Value

A diagnosis object: a list whose main elements are simulations_df (the simulations, as simulate_design() returns them), diagnosands_df (a tibble with one row per group of design, redesign parameters, inquiry, estimator, outcome, and term, holding n_sims, a column per diagnosand, and, when bootstrapped, an se(<diagnosand>) column for each), diagnosand_names, bootstrap_sims, and bootstrap_replicates. When the simulation was nested, variance_decomposition holds a tibble decomposing the variance of each quantity by draw level; otherwise it is NULL.

See Also

Other simulation and diagnosis: compare_diagnoses(), diagnose_simulations(), get_diagnosands(), get_simulations(), simulate_design()

Examples

```
design <- declare_model(N = 40, U = rnorm(N), Z = rep(0:1, 20), Y = U + Z) +
  declare_inquiry(ATE = 1) +
  declare_estimator(Y ~ Z, .method = lm, term = "Z", inquiry = "ATE",
                    label = "ols")
diagnose_design(design, sims = 5, bootstrap_sims = 0)
```

---

diagnose_simulations *Diagnose a precomputed simulations table*

Description

Diagnose a precomputed simulations table

Usage

```
diagnose_simulations(simulations_df, diagnosands = NULL, bootstrap_sims = 100)
```

Arguments

simulations_df	(required) A tibble produced by simulate_design(). Grouping set with dplyr::group_by() adds to the groups diagnosed.
diagnosands	(optional) A diagnosands design_step (e.g., from declare_diagnosands()). NULL (the default) uses default_diagnosands().
bootstrap_sims	(optional) Number of bootstrap replicates. Defaults to 100; 0 skips the bootstrap.

Value

A diagnosis object, as diagnose_design() returns.

See Also

Other simulation and diagnosis: compare_diagnoses(), diagnose_design(), get_diagnosands(), get_simulations(), simulate_design()

Examples

```
design <- declare_model(N = 40, U = rnorm(N), Z = rep(0:1, 20), Y = U + Z) +
  declare_inquiry(ATE = 1) +
  declare_estimator(Y ~ Z, .method = lm, term = "Z", inquiry = "ATE",
                    label = "ols")
sims <- simulate_design(design, sims = 5)
diagnose_simulations(sims, bootstrap_sims = 0)
```

`draw_data` *Draw the realized data from a design*

Description

Runs the data-generating steps only. The inquiries and estimators are not run, so a design whose estimator fails still draws its data, and drawing data does not pay to fit models it would throw away.

Usage

```
draw_data(design, ...)
```

Arguments

`design`	(required) A `design`.
`...`	(optional) Named parameter values, as in `redesign()`. Each supplies one value: use `redesign()` itself to sweep over several. A parameter named `design`, or any prefix of it, cannot be supplied this way and needs the `redesign()` route.

Details

Named parameter values in `...` are applied first, so `draw_data(design, theta = 0.5)` is `draw_data(redesign(design, theta = 0.5))`: a convenience for trying a value without naming it above the design or rewriting the declaration. The names reach the design by `redesign()`'s rule, so a name the design does not read warns and is ignored, and a number written inside a step is refused with the two ways to name it. `data`, `start` and `end`, 1.x's arguments for running part of a design on supplied data, are refused with `get_estimates()` and `design[start:end]` as the replacements, unless the design reads a parameter by that name.

Value

A tibble: the data as the last data-generating step leaves it.

See Also

Other drawing from a design: `draw_estimands()`, `draw_estimates()`, `get_estimates()`, `run_design()`

Examples

```
design <- declare_model(N = 25, X = rnorm(N))
df <- draw_data(design)
nrow(df)

# A value the declaration leaves free, supplied at draw time
model <- declare_model(N = 25, D = rbinom(N, 1, 0.5), Y = theta * D + rnorm(N))
head(draw_data(model, theta = 0.5))
```

---

draw_estimands *Draw the realized estimands*

Description

Runs the data-generating and inquiry steps only, not the estimators.

Usage

```
draw_estimands(design, ...)

draw_estimand(design, ...)
```

Arguments

design	(required) A design.
...	(optional) Named parameter values, as in redesign(). Each supplies one value: use redesign() itself to sweep over several. A parameter named design, or any prefix of it, cannot be supplied this way and needs the redesign() route.

Value

A tibble with columns inquiry and estimand, one row per inquiry.

See Also

Other drawing from a design: draw_data(), draw_estimates(), get_estimates(), run_design()

Examples

```
design <- declare_model(N = 25, U = rnorm(N), Y = U) +
  declare_inquiry(mu = mean(Y))
draw_estimands(design)
```

draw_estimates *Draw the realized estimates*

Description

Runs the design once and returns its estimates joined to inquiries.

Usage

```
draw_estimates(design, ...)
```

Arguments

design	(required) A design.
...	(optional) Named parameter values, as in redesign(). Each supplies one value: use redesign() itself to sweep over several. A parameter named design, or any prefix of it, cannot be supplied this way and needs the redesign() route.

Value

The estimates from one run: a tibble with one row per estimate: the tidied columns (term, estimate, std.error, p.value, conf.low, conf.high, and so on), estimator, and, where an estimator names an inquiry, inquiry and the realized estimand. A design with no estimator returns its estimands instead.

See Also

Other drawing from a design: draw_data(), draw_estimands(), get_estimates(), run_design()

Examples

```
design <- declare_model(N = 30, U = rnorm(N), Y = U) +
  declare_inquiry(mu = mean(Y)) +
  declare_estimator(Y ~ 1, .method = lm, term = "(Intercept)",
                    inquiry = "mu", label = "ols")
draw_estimates(design)
```

expand_design *Build a family of designs from a designer function (deprecated)*

Description

Deprecated in 2.0. Call the designer once and redesign() the result: redesign(designer(), N = c(100, 200)) does what expand_design(designer, N = c(100, 200)) did, and reaches the same names, since a designer's argument is a name the design reads. expand_design() still works and warns once per session.

Usage

```
expand_design(.designer, ..., .expand = TRUE)
```

Arguments

.designer	(required) A function returning a design.
...	(optional) Named parameter values to vary, passed to .designer as its arguments.
.expand	(optional) If TRUE (the default), expand the grid; if FALSE, zip parallel vectors.

Details

The one thing it did that redesign() cannot is rebuild a design whose *shape* depends on an argument, such as a designer that adds a step when blocked = TRUE. For that, call the designer with each value.

Value

A design or a named list of designs, as redesign() returns.

See Also

Other modifying a design: delete_step(), design_notes(), design_parameters(), insert_step(), redesign(), replace_step()

Examples

```
designer <- function(N = 50) {
  declare_model(N = N, Y = rnorm(N)) + declare_inquiry(mu = mean(Y))
}
# deprecated
designs <- suppressWarnings(expand_design(designer, N = c(10, 20)))
# instead
designs <- redesign(designer(), N = c(10, 20))
length(designs)
```

---

format.diagnosis *Format a diagnosis for display*

Description

Rounds the diagnosands, puts each bootstrap standard error in parentheses on a second row beneath its estimate, and gives the columns display names (mean_estimand becomes Mean Estimand). This is what print.diagnosis() shows and what you want in front of knitr::kable().

Usage

```
## S3 method for class 'diagnosis'
format(x, digits = 2, ...)

reshape_diagnosis(diagnosis, digits = 2)
```

Arguments

x	(required) A diagnosis object.
digits	(optional) Number of decimal places. Defaults to 2.
...	(optional) Ignored.
diagnosis	(required) A diagnosis object.

Details

The standard-error rows appear only when the diagnosis was bootstrapped. At bootstrap_sims = 0 there are no replicates, so the table is one row per group and there are no parentheses.

It is a format() method because that is what it does. R's format() is the generic for "the character representation this object prints as", and print() is documented as calling it. The three views of a diagnosis are worth keeping straight: get_diagnosands() is wide and numeric with se(bias) as a sibling column, tidy() is long and numeric with the standard error and interval as fields of each row, and this is wide and character with the standard error rendered beneath its estimate. Only the first two can be computed on. This one is a rendering of the first, a pure function of it and digits, and it carries strictly less information than either: the interval is not in it and the precision is gone.

reshape_diagnosis() is DeclareDesign's name for the same thing and calls this method, so code written against DeclareDesign keeps working.

It returns a data.frame rather than a tibble because every column is a formatted string, and a tibble would print a <chr> row under a display table.

Redesign parameter columns keep their own names, since prob_each is the argument the reader passed and not a phrase to be title-cased.

DeclareDesign's select and exclude arguments are deliberately absent: the result is a data frame, so select() already chooses columns from it, and an argument that duplicates a verb is the wart this package exists to remove.

Value

A data.frame of character columns: the grouping columns and one column per diagnosand under its display name, with each bootstrap standard error in parentheses on the row beneath its estimate.

See Also

Other diagnosands: declare_diagnosands(), default_diagnosands(), select_diagnosands(), set_diagnosands()

Examples

```
design <- declare_model(N = 30, Y = rnorm(N), Z = rep(0:1, 15)) +
  declare_inquiry(ATE = 0) +
  declare_estimator(Y ~ Z, .method = lm, term = "Z", inquiry = "ATE",
                    label = "ols")
d <- diagnose_design(design, sims = 5, bootstrap_sims = 0)
format(d)

# DeclareDesign's name for the same table
reshape_diagnosis(d)

# choose columns with select(), on the data frame it returns
format(d) |> dplyr::select(Term, Bias, Power)
```

---

get_diagnosands *Get the diagnosands table from a diagnosis*

Description

Get the diagnosands table from a diagnosis

Usage

```
get_diagnosands(diagnosis)
```

Arguments

diagnosis	(required) A diagnosis object.

Value

The diagnosis's diagnosands_df: a tibble with one row per group (design, inquiry, estimator, outcome, and term), holding n_sims, a column per diagnosand, and, when bootstrapped, an se(<diagnosand>) column for each.

See Also

Other simulation and diagnosis: compare_diagnoses(), diagnose_design(), diagnose_simulations(), get_simulations(), simulate_design()

Examples

```
design <- declare_model(N = 30, Y = rnorm(N), Z = rep(0:1, 15)) +
  declare_inquiry(ATE = 0) +
  declare_estimator(Y ~ Z, .method = lm, term = "Z", inquiry = "ATE",
                    label = "ols")
d <- diagnose_design(design, sims = 5, bootstrap_sims = 0)
get_diagnosands(d)
```

`get_estimates` *Get estimates for a fixed dataset*

Description

Runs only the estimator steps of a design against a supplied data frame.

Usage

```
get_estimates(
  design,
  data = draw_data(design),
  start = 1L,
  end = length(design)
)
```

Arguments

`design`	(required) A `design`.
`data`	(optional) A data frame. Defaults to `draw_data(design)`, a fresh draw.
`start`	(optional) Integer; first step index to consider. Defaults to `1`.
`end`	(optional) Integer; last step index to consider. Defaults to the last step.

Value

A tibble with one row per estimate from the estimator and test steps between `start` and `end`, as `draw_estimates()` returns but without estimands. An empty tibble when that range holds no estimator.

See Also

Other drawing from a design: `draw_data()`, `draw_estimands()`, `draw_estimates()`, `run_design()`

Examples

```
design <- declare_model(N = 30, U = rnorm(N), Z = rep(0:1, 15), Y = U + Z) +
  declare_estimator(Y ~ Z, .method = lm, term = "Z", label = "ols")
df <- draw_data(design)
get_estimates(design, df)
```

get_simulations *Get the simulations table from a diagnosis*

Description

Get the simulations table from a diagnosis

Usage

```
get_simulations(diagnosis)
```

Arguments

diagnosis	(required) A diagnosis object.

Value

The diagnosis's simulations_df: a tibble with one row per estimate per simulation, as simulate_design() returns it.

See Also

Other simulation and diagnosis: compare_diagnoses(), diagnose_design(), diagnose_simulations(), get_diagnosands(), simulate_design()

Examples

```
design <- declare_model(N = 30, Y = rnorm(N), Z = rep(0:1, 15)) +
  declare_inquiry(ATE = 0) +
  declare_estimator(Y ~ Z, .method = lm, term = "Z", inquiry = "ATE",
                    label = "ols")
d <- diagnose_design(design, sims = 5, bootstrap_sims = 0)
get_simulations(d)
```

insert_step *Insert a step into a design (deprecated)*

Description

Deprecated in 2.0, and warns once per session. Reconstruct the design with + instead.

Usage

```
insert_step(design, new_step, before = NULL, after = NULL)
```

Arguments

design	(required) A design.
new_step	(required) A design_step to insert.
before, after	(required) A label, integer, or design_step indicating the anchor point. Provide exactly one of the two; if both are given, after is used. before comes first, as in DeclareDesign 1.x, so a script that passed the anchor by position keeps its meaning.

Value

A design with new_step inserted at the anchor.

See Also

Other modifying a design: delete_step(), design_notes(), design_parameters(), expand_design(), redesign(), replace_step()

Examples

```
design <- declare_model(N = 30, Y = rnorm(N)) +
  declare_inquiry(mu = mean(Y))
new_design <- insert_step(design,
  declare_measurement(Y2 = Y * 2),
  after = "model")
names(new_design)
```

---

label_estimator *Wrap a custom function as a labeled estimator*

Description

label_estimator() returns a function suitable for use as .method in declare_estimator() or declare_test(): it fits the model, tidies the fit, tags the result with estimator and inquiry columns, and keeps the terms asked for. label_test() is the same function under a second name, kept so that 1.x scripts run; a test is an estimator whose tidied output carries a p-value, and nothing in the wrapper depends on which name was called. 1.x's fn = argument is .method = in both.

Usage

```
label_estimator(
  .method,
  label = NULL,
  inquiry = NULL,
  term = NULL,
  .summary = tidy_try
)
```

```
label_test(
  .method,
  label = NULL,
  inquiry = NULL,
  term = NULL,
  .summary = tidy_try
)
```

Arguments

`.method`	(required) A function that fits a model from `data` and `...`.
`label`	(optional) Estimator label, written to the `estimator` column. `NULL` (the default) adds no column.
`inquiry`	(optional) Inquiry label (or list of labels); the first is written to the `inquiry` column. `NULL` (the default) adds no column.
`term`	(optional) Character vector of terms to keep. `NULL` (the default) keeps every term.
`.summary`	(optional) Function for tidying the model fit. Defaults to `tidy_try()`.

Value

A function of `data` and `...` that calls `.method(data = data, ...)`, tidies the fit with `.summary`, and returns the resulting tibble with the `estimator` and `inquiry` columns set and its rows restricted to `term`.

Examples

```
my_est <- label_estimator(
  function(data, ...) lm(Y ~ Z, data = data),
  label = "lm", inquiry = "ATE", term = "Z"
)
df <- data.frame(Y = rnorm(20), Z = rep(0:1, 10))
my_est(df)
```

---

model_handler *Deprecated alias for* `label_estimator()`

Description

`model_handler()` and `tidy_estimator()` are retained for compatibility but will be removed in a future version. Use `label_estimator()` instead.

Usage

```
model_handler(...)

tidy_estimator(...)
```

Arguments

...	(required) Forwarded to label_estimator(), which needs at least .method.

Value

A function, as label_estimator() returns.

---

pop.var *Population variance*

Description

Computes the (uncorrected) variance: the mean squared deviation from the mean. Useful for design-based standard errors. Missing values are dropped.

Usage

```
pop.var(x)
```

Arguments

x	(required) A numeric vector.

Value

A numeric scalar: the mean of (x - mean(x))^2 over the non-missing values of x.

Examples

```
pop.var(c(1, 2, 3, 4, 5))
```

---

print.compared_diagnoses *Print a diagnosand comparison*

Description

Print a diagnosand comparison

Usage

```
## S3 method for class 'compared_diagnoses'
print(x, ...)
```

Arguments

x	(required) A compared_diagnoses object, as returned by compare_diagnoses().
...	(optional) Passed to the print() method of compared_diagnoses_df.

Value

x, invisibly.

Examples

```
design <- declare_parameters(n_units = 100) +
  declare_model(N = n_units, Y = rnorm(N), Z = randomizr::complete_ra(N)) +
  declare_inquiry(ATE = 0) +
  declare_estimator(Y ~ Z, .method = lm, term = "Z", inquiry = "ATE",
                    label = "ols")
print(compare_diagnoses(design, redesign(design, n_units = 200),
                        sims = 20, bootstrap_sims = 20))
```

---

print.design *Print a design*

Description

Lists the steps as the calls that declared them, then the parameters and objects the design's expressions refer to: the names redesign() can change, the value each one currently holds, its kind, and which steps would respond to a change. A design with no such names prints its steps alone. summary() runs the design once as well and says what each step did to the data.

Usage

```
## S3 method for class 'design'
print(x, ...)
```

Arguments

x	(required) A design.
...	(optional) Ignored.

Value

x, invisibly.

Examples

```
d <- declare_model(N = 30, Y = rnorm(N)) + declare_inquiry(mu = mean(Y))
print(d)
```

print.design_step *Print a design step*

Description

Print a design step

Usage

```
## S3 method for class 'design_step'
print(x, ...)
```

Arguments

x	(required) A design_step.
...	(optional) Ignored.

Value

x, invisibly.

Examples

```
s <- declare_model(N = 10, Y = rnorm(N))
print(s)
```

print.diagnosis *Print a diagnosis*

Description

Prints the table format.diagnosis() builds: one row of diagnosand estimates per group, with bootstrap standard errors in parentheses on the row beneath.

Usage

```
## S3 method for class 'diagnosis'
print(x, digits = 2, ...)
```

Arguments

x	(required) A diagnosis.
digits	(optional) Number of decimal places. Defaults to 2.
...	(optional) Ignored.

Value

The data.frame of formatted strings that format.diagnosis() returns, invisibly.

Examples

```
design <- declare_model(N = 30, Y = rnorm(N), Z = rep(0:1, 15)) +
  declare_inquiry(ATE = 0) +
  declare_estimator(Y ~ Z, .method = lm, term = "Z", inquiry = "ATE",
                    label = "ols")
d <- diagnose_design(design, sims = 5, bootstrap_sims = 0)
print(d)
```

---

print.objects *Print the objects table, one row per name*

Description

Prints the table design_parameters() returns, and that print(design) shows under "Parameters and objects the design refers to", aggregated to one row per name with five columns. name is the name redesign() can change. value is a display snippet of what it currently holds. kind is how redesign() reads a bare replacement: scalar, vector, and list supply one design per element, while data, function, and other are single values. declared is TRUE when a declare_parameters() step declares the name and FALSE when the design reads it from the environment it was written in. steps lists the positions of the steps that read the name, which are the steps that would respond to a change.

Usage

```
## S3 method for class 'objects'
print(x, ...)
```

Arguments

x	(required) An objects table, as returned by design_parameters().
...	(optional) Ignored.

Details

The underlying table has one row per name per step and carries the quosure and env columns that locate each binding; those are left out of the print.

Value

x, invisibly.

Examples

```
tau <- 0.3
design <-
  declare_parameters(n_units = 100) +
  declare_model(N = n_units, U = rnorm(N), Y_Z_0 = U, Y_Z_1 = U + tau) +
  declare_inquiry(ATE = mean(Y_Z_1 - Y_Z_0)) +
  declare_assignment(Z = complete_ra(N))
design_parameters(design)
```

`redesign` *Redesign*

Description

Returns a copy of the design with one or more parameters changed, so that versions can be diagnosed side by side. Several values for a parameter give several designs: with `.expand = TRUE` (the default) the cross-product of the values is taken, and with `.expand = FALSE` they are zipped position-wise.

Usage

```
redesign(.design, ..., .expand = TRUE)
```

Arguments

`.design`	(required) A `design`. Named with a dot, like `.method` and `.summary` in `declare_estimator()`, because everything else here is a parameter of the user's design: a plain `design` would partially match and swallow a parameter named `d`, `de`, `des`, `desi`, or `desig`, and designs with a parameter named `d` exist.
`...`	(optional) Named parameter values. A bare atomic vector, and a bare list, supply one design per element; a data frame, a matrix, a function, or any classed object is one value. To sweep over such values, pass a list of them, and to supply one list as a single value wrap it: `list(x)`.
`.expand`	(optional) If `TRUE` (the default), expand the parameter grid; if `FALSE`, zip parallel vectors. Dotted for the same reason as `.design`: an undotted `expand` after the dots would collide exactly with a parameter of that name.

Details

A parameter that no step responds to draws a warning and is otherwise ignored. `summary()` on a design lists the names that are available.

An atomic vector always supplies one value per design, so a parameter that is itself a vector has to be wrapped: `prob_each = list(c(0, 0.5, 0.5))` is one design, where `prob_each = c(0, 0.5, 0.5)` is three. Handing a bare vector to a parameter that currently holds one warns.

A bare list is split the same way, one design per element, so a parameter whose value is a list is wrapped by the same rule: `levels_list = list(levels)` is one design and `levels_list =`

`levels` is one design per element of `levels`. The two are told apart by what the elements are, since a sweep over lists holds a list in every element, and a bare list handed to a list-valued parameter whose elements are not lists warns.

Only bare vectors and bare lists are read that way. A data frame, a matrix, and anything carrying a class are single replacement values, so a design written as `declare_model(data = dataset, ...)`, with `dataset <- pilot` in the workspace, swaps its data with `redesign(design, dataset = real)` and needs no wrapping. Naming the parameter `dataset` rather than after the data it starts with keeps the name true once it is redesigned. A redesign reaches the name the design reads the data under, not `data`, which names fabricate's argument and belongs to the declaration.

Value

A single `design` if one combination is supplied, otherwise a list of designs named by the values that distinguish them: `redesign(design, N = c(10, 20))` gives `N = 10` and `N = 20`, so a simulation's `design` column, and `bind_rows(.id = "design")`, carry the values that vary. Designs are named `design_1`, `design_2`, etc. when a parameter's value cannot be written down, such as a function or a data frame.

See Also

Other modifying a design: `delete_step()`, `design_notes()`, `design_parameters()`, `expand_design()`, `insert_step()`, `replace_step()`

Examples

```
designer <- function(N) {
  declare_model(N = N, Y = rnorm(N)) +
    declare_inquiry(mu = mean(Y))
}
design <- designer(50)
redesigned <- redesign(design, N = c(10, 20))
length(redesigned)

# a function-valued parameter
summarizer <- function(x) mean(x)
design <- declare_model(N = 50, Y = rnorm(N)) +
  declare_inquiry(mu = summarizer(Y))
redesign(design, summarizer = stats::median)
```

---

replace_step *Replace a step in a design (deprecated)*

Description

Deprecated in 2.0, and warns once per session. Reconstruct the design with + instead.

Usage

```
replace_step(design, step, new_step)
```

Arguments

design	(required) A design.
step	(required) A label, integer, or design_step naming the step to replace.
new_step	(required) The replacement design_step.

Value

A design with new_step in place of step.

See Also

Other modifying a design: delete_step(), design_notes(), design_parameters(), expand_design(), insert_step(), redesign()

Examples

```
design <- declare_model(N = 30, Y = rnorm(N)) +
  declare_inquiry(mu = mean(Y))
replace_step(design, "mu", declare_inquiry(med = median(Y)))
```

---

run_design *Execute a design end-to-end*

Description

Runs the design once and returns one row per estimate, with the realized estimands joined on. This is one simulation: simulate_design(sims = 1) without the sim_ID column.

Usage

```
run_design(design, ...)
```

Arguments

design	(required) A design.
...	(optional) Named parameter values, as in redesign(). Each supplies one value: use redesign() itself to sweep over several. A parameter named design, or any prefix of it, cannot be supplied this way and needs the redesign() route.

Details

Use draw_data() for the realized data frame, draw_estimands() for the inquiries alone, and draw_estimates() for the estimates alone.

Value

The result of one run: a tibble with one row per estimate: the tidied columns (`term`, `estimate`, `std.error`, `p.value`, `conf.low`, `conf.high`, and so on), `estimator`, and, where an estimator names an inquiry, `inquiry` and the realized `estimand`. A design with no estimator returns its estimands instead, as `draw_estimands()` does.

See Also

Other drawing from a design: `draw_data()`, `draw_estimands()`, `draw_estimates()`, `get_estimates()`

Examples

```
design <- declare_model(N = 30, U = rnorm(N), Y = U) +
  declare_inquiry(mu = mean(Y))
run_design(design)
```

---

select_diagnosands *Build a diagnosands step from a set of named diagnosands*

Description

Two ways to call this, told apart by whether the first argument is a `design_step`.

Usage

```
select_diagnosands(..., alpha = 0.05, subset = NULL, na.rm = FALSE)
```

Arguments

`...`	(required) Diagnosand names to keep, as strings, optionally preceded by a diagnosands `design_step` to subset.
`alpha`	(optional) Significance level used by `power`, `type_s_rate`, `exaggeration_ratio`, and `prop_pos_sig`. Defaults to `0.05`.
`subset`	(optional) An expression evaluated on the simulations table; only the rows for which it is `TRUE` enter the diagnosands. `NULL` (the default) keeps every simulation.
`na.rm`	(optional) Passed to the library diagnosands. Defaults to `FALSE`.

Details

Given names only, `select_diagnosands("bias", "power")` builds a fresh diagnosands step out of the built-in library, which is what DeclareDesign does. `alpha`, `subset`, and `na.rm` shape the expressions it writes.

Given a diagnosands step first, `select_diagnosands(my_diagnosands, "bias")` keeps those entries of that step and nothing else. The expressions are yours and are copied unchanged, so `alpha`,

subset, and na.rm have nothing to act on and are refused rather than ignored: set them on the declare_diagnosands() call that made the set.

A consequence worth knowing: the same name can mean different things in the two forms. select_diagnosands("bias") writes mean(estimate - estimand, na.rm = FALSE), following DeclareDesign's default, while select_diagnosands(defa "bias") returns the na.rm = TRUE version default_diagnosands() declares.

The library is a superset of default_diagnosands(): mean_estimand, mean_estimate, bias, sd_estimate, rmse, power, coverage, mean_se, type_s_rate, exaggeration_ratio, var_estimate, mean_var_hat, prop_pos_sig, and mean_ci_length.

Value

A diagnosands design_step, as declare_diagnosands() returns, holding the named diagnosands in the order given.

See Also

Other diagnosands: declare_diagnosands(), default_diagnosands(), format.diagnosis(), set_diagnosands()

Examples

```
select_diagnosands("sd_estimate", "mean_se")

# subset a custom set instead of the library
select_diagnosands(default_diagnosands(), "bias", "rmse")
```

---

set_citation *Attach a citation to a design*

Description

Records how the design should be cited, as a utils::bibentry() built from the fields given, or as the text in citation. cite_design() returns it.

Usage

```
set_citation(
  design,
  title = NULL,
  author = NULL,
  year = NULL,
  description = "Unpublished research design declaration",
  citation = NULL
)
```

Arguments

design	(required) A design.
title	(optional) Title of the design.
author	(optional) Author or authors, as a string or a utils::person().
year	(optional) Year.
description	(optional) Note on the entry. Defaults to "Unpublished research design declaration".
citation	(optional) A ready-made citation, as a string or a bibentry, used in place of the fields above.

Value

design, with the citation stored in its citation attribute.

See Also

Other citing a design: cite_design()

Examples

```
design <- declare_model(N = 10, Y = rnorm(N))
design <- set_citation(design, title = "Example", author = "Coppock",
                       year = 2026)
cite_design(design)
```

---

set_diagnosands *Attach diagnosands to a design (deprecated)*

Description

Deprecated in 2.0. Stores diagnosands as an attribute on design, which diagnose_design() uses in place of the defaults when present. Pass the diagnosands to the diagnosis instead: diagnose_design(design, diagnosands = ...).

Usage

```
set_diagnosands(design, diagnosands = default_diagnosands())
```

Arguments

design	(required) A design.
diagnosands	(optional) A diagnosands design_step. Defaults to default_diagnosands().

Details

Two reasons, beyond diagnosis not being a feature of a design. The attribute does not survive `redesign()`, so anything attached this way is silently gone the first time a parameter moves, and the defaults that replace it may not be computable from the estimator's output. And a diagnosands object replaces the default set rather than adding to it, so declaring one diagnosand drops the other six without saying so.

Value

`design`, with `diagnosands` stored in its `diagnosands` attribute.

See Also

Other diagnosands: `declare_diagnosands()`, `default_diagnosands()`, `format.diagnosis()`, `select_diagnosands()`

Examples

```
design <-
  declare_model(N = 30, Y = rnorm(N)) +
  declare_inquiry(mu = mean(Y)) +
  declare_estimator(Y ~ 1, .method = lm, term = "(Intercept)", inquiry = "mu")

# instead of set_diagnosands(design, ...)
diagnose_design(
  design,
  diagnosands = declare_diagnosands(mean_estimate = mean(estimate, na.rm = TRUE)),
  sims = 5,
  bootstrap_sims = 0
)
```

`simulate_design` *Simulate one or more designs*

Description

Runs each design `sims` times, collecting estimands and estimates into a single long tibble suitable for diagnosis. A `design` column names the design each row came from, whether one design is supplied or several.

Usage

```
simulate_design(..., sims = NULL, progress = FALSE)

simulate_designs(..., sims = NULL, progress = FALSE)
```

Arguments

...	(required) One or more design objects, or lists of them.
sims	(optional) Number of simulations per design. Defaults to NULL, which means 500 flat simulations for a design with no step-level draws. A design with step-level draws runs in nested mode whether or not sims is supplied; supplying it warns and is otherwise ignored.
progress	(optional) If TRUE, display a progress bar for this call by wrapping it in progressr::with_progress(). The better habit is to opt in once per session with progressr::handlers(global = TRUE), which covers every call and lets you choose how progress is shown. Nothing is displayed by default.

Details

If any step in a design has draws > 1, the simulation runs in nested mode: each step with draws > 1 fans out, and the total simulation count equals the product of all draws values across steps. Declared draws take priority over sims, so supplying both runs the nested simulation and warns, printing the plan it is running instead.

Steps upstream of the first fan-out run once and are held fixed across the draws below them, which is what makes declare_assignment(..., draws = 50) mean fifty randomizations of one population.

Parallelism is handled through the future ecosystem. Call future::plan(multisession, workers = 4) before simulate_design() and, if the furrr package is installed, simulations run in parallel with no other change. Every draw runs on its own L'Ecuyer-CMRG random-number stream whichever plan is active, so set.seed() before the call gives the same table sequentially and in parallel. The streams are not those of DeclareDesign 1.x, so a 1.x seed does not reproduce a 1.x table here.

Value

A tibble with one row per estimate per simulation: design, sim_ID, the estimate columns (term, estimate, std.error, p.value, conf.low, conf.high, and so on), estimator, inquiry, estimand, and a column for each redesign parameter. A nested simulation adds a <label>_draw column for each step with draws > 1.

See Also

Other simulation and diagnosis: compare_diagnoses(), diagnose_design(), diagnose_simulations(), get_diagnosands(), get_simulations()

Examples

```
design <- declare_model(N = 30, U = rnorm(N), Z = rep(0:1, 15), Y = U + Z) +
  declare_inquiry(ATE = 1) +
  declare_estimator(Y ~ Z, .method = lm, term = "Z", inquiry = "ATE",
                    label = "ols")
simulate_design(design, sims = 3)
```

summary.design *Summarize a design*

Description

Runs the design once and reports what each step did: the rows a data step produced or kept and the columns it added, dropped, or changed; the value each inquiry took; the estimate each estimator returned. The run itself, estimates joined to estimands, comes after the steps, then the parameters and objects the design refers to. Everything the printout shows is on the returned object, so a document or an app can use the pieces.

Usage

```
## S3 method for class 'design'
summary(object, run = TRUE, ...)
```

Arguments

object	(required) A design.
run	(optional) Whether to run the design once. Defaults to TRUE.
...	(optional) Ignored.

Details

The run is one random draw, so the numbers differ from call to call. A design whose estimator is slow can be summarized without running it: run = FALSE gives the steps and the parameters alone, which is what print() shows.

Value

A summary.design: a list with steps (a tibble with one row per step: step, label, type, call, and one_run, the account of that step on this run), parameters and notes (as design_parameters() and design_notes() return them), ran (logical), and when the design was run, data (the final data), inquiries and estimates (as run_design() returns them).

Examples

```
n_units <- 30
design <- declare_model(N = n_units, U = rnorm(N), Y = U) +
  declare_inquiry(mu = mean(Y)) +
  declare_estimator(Y ~ 1, .method = lm, term = "(Intercept)",
                    inquiry = "mu", label = "ols")
summary(design)
summary(design, run = FALSE)
summary(design)$steps
```

summary.diagnosis *Summarize a diagnosis*

Description

The same report `print.diagnosis()` gives.

Usage

```
## S3 method for class 'diagnosis'
summary(object, digits = 2, ...)
```

Arguments

`object`	(required) A `diagnosis`.
`digits`	(optional) Number of decimal places. Defaults to `2`.
`...`	(optional) Ignored.

Value

The `data.frame` of formatted strings that `format.diagnosis()` returns, invisibly.

Examples

```
design <- declare_model(N = 30, Y = rnorm(N), Z = rep(0:1, 15)) +
  declare_inquiry(ATE = 0) +
  declare_estimator(Y ~ Z, .method = lm, term = "Z", inquiry = "ATE",
                    label = "ols")
summary(diagnose_design(design, sims = 5, bootstrap_sims = 0))
```

tidy.diagnosis *Tidy a diagnosis*

Description

Returns the diagnosands in long form, one row per group per diagnosand, with the bootstrap standard error and percentile confidence interval alongside. This is the shape to plot from: `ggplot(aes(estimate, diagnosand)) + geom_errorbarh(aes(xmin = conf.low, xmax = conf.high))`.

Usage

```
## S3 method for class 'diagnosis'
tidy(x, conf.int = TRUE, conf.level = 0.95, ...)
```

Arguments

x	(required) A diagnosis.
conf.int	(optional) Whether to include the confidence interval. Defaults to TRUE.
conf.level	(optional) Confidence level for the interval. Defaults to 0.95.
...	(optional) Ignored.

Details

std.error, conf.low, and conf.high come from the bootstrap replicates, so they appear only when the diagnosis was run with bootstrap_sims > 0.

Value

A tibble with one row per group per diagnosand: the grouping columns (design, inquiry, estimator, outcome, term, and any redesign parameters), diagnosand, estimate, and, when bootstrapped, std.error, conf.low, and conf.high.

Examples

```
design <- declare_model(N = 30, Y = rnorm(N), Z = rep(0:1, 15)) +
  declare_inquiry(ATE = 0) +
  declare_estimator(Y ~ Z, .method = lm, term = "Z", inquiry = "ATE",
                    label = "ols")
d <- diagnose_design(design, sims = 5, bootstrap_sims = 0)
tidy(d)
```

---

tidy_try *Tidy a model fit, with a fallback*

Description

Tries broom::tidy() with confidence intervals; if that fails (because the model class has no tidy() method), assembles a minimal tidy table from the coefficient summary.

Usage

```
tidy_try(fit, ...)
```

Arguments

fit	(required) A fitted model. A data frame is returned as a tibble unchanged.
...	(optional) Passed to broom::tidy().

Value

A tibble with one row per term and columns including term, estimate, std.error, statistic, p.value, conf.low, and conf.high. When neither broom::tidy() nor coef(summary(fit)) works, a tibble with those columns and no rows.

Examples

```
fit <- lm(mpg ~ wt, data = mtcars)
tidy_try(fit)
```

`vars` *Capture variables as quosures*

Description

Convenience alias for `rlang::quos()` used inside design declarations to pass through bare expressions.

Usage

```
vars(...)
```

Arguments

`...`	(optional) Expressions, captured unevaluated.

Value

A list of quosures of class `quosures`, one per expression.

Examples

```
qs <- vars(x, y, z)
length(qs)
```

Index

∗ **citing a design**
cite_design, 5
set_citation, 47
∗ **design declarations**
declare_assignment, 8
declare_estimator, 10
declare_inquiry, 12
declare_measurement, 14
declare_model, 15
declare_notes, 16
declare_parameters, 18
declare_sampling, 20
declare_step, 21
declare_test, 22
∗ **diagnosands**
declare_diagnosands, 9
default_diagnosands, 23
format.diagnosis, 32
select_diagnosands, 46
set_diagnosands, 48
∗ **drawing from a design**
draw_data, 29
draw_estimands, 30
draw_estimates, 31
get_estimates, 35
run_design, 45
∗ **modifying a design**
delete_step, 24
design_notes, 25
design_parameters, 26
expand_design, 31
insert_step, 36
redesign, 43
replace_step, 44
∗ **simulation and diagnosis**
compare_diagnoses, 6
diagnose_design, 27
diagnose_simulations, 28
get_diagnosands, 34
get_simulations, 36
simulate_design, 49
+.dd, 4

broom::tidy(), *53*

cite_design, 5
cite_design(), *47, 48*
compare_design_code *(*compare_designs*)*, 5
compare_design_data *(*compare_designs*)*, 5
compare_design_estimates *(*compare_designs*)*, 5
compare_design_inquiries *(*compare_designs*)*, 5
compare_design_summaries *(*compare_designs*)*, 5
compare_designs, 5
compare_diagnoses, 6
compare_diagnoses(), *6*, *28*, *34*, *36*, *40*, *50*

declare-defunct, 8
declare_assignment, 8
declare_assignment(), *3*, *12–15*, *17*, *19–21*, *23*
declare_diagnosands, 9
declare_diagnosands(), *24*, *27, 28*, *33*, *47*, *49*
declare_estimand *(*declare_inquiry*)*, 12
declare_estimands *(*declare_inquiry*)*, 12
declare_estimator, 10
declare_estimator(), *3*, *9*, *13–15*, *17*, *19–23*, *37*, *43*
declare_inquiries *(*declare_inquiry*)*, 12
declare_inquiry, 12
declare_inquiry(), *3*, *9*, *12*, *14, 15*, *17*, *19–21*, *23*
declare_measurement, 14
declare_measurement(), *3*, *8, 9*, *12, 13*, *15*, *17*, *19–21*, *23*
declare_model, 15

declare_model(), *3*, *8, 9*, *12–14*, *17*, *19–21*, *23*
declare_notes, 16
declare_notes(), *9*, *12–15*, *19–21*, *23*, *25*
declare_parameters, 18
declare_parameters(), *9*, *12–15*, *17*, *20, 21*, *23*, *26*, *42*
declare_population, 19
declare_potential_outcomes *(*declare-defunct*)*, 8
declare_reveal *(*declare-defunct*)*, 8
declare_sampling, 20
declare_sampling(), *3*, *9*, *12–15*, *17*, *19*, *21*, *23*
declare_step, 21
declare_step(), *9*, *12–15*, *17*, *19, 20*, *23*
declare_test, 22
declare_test(), *3*, *9*, *12–15*, *17*, *19–21*, *37*
DeclareDesign *(*DeclareDesign-package*)*, 3
DeclareDesign-package, 3
default_diagnosands, 23
default_diagnosands(), *7*, *10*, *27, 28*, *33*, *47–49*
delete_step, 24
delete_step(), *25, 26*, *32*, *37*, *44, 45*
design_notes, 25
design_notes(), *17*, *24*, *26*, *32*, *37*, *44, 45*, *51*
design_parameters, 26
design_parameters(), *17*, *24, 25*, *32*, *37*, *42*, *44, 45*, *51*
diagnose_design, 27
diagnose_design(), *3*, *7*, *10*, *28*, *34*, *36*, *48*, *50*
diagnose_designs *(*diagnose_design*)*, 27
diagnose_simulations, 28
diagnose_simulations(), *7*, *27, 28*, *34*, *36*, *50*
dplyr::group_by(), *28*
draw_data, 29
draw_data(), *30, 31*, *35*, *45, 46*
draw_estimand *(*draw_estimands*)*, 30
draw_estimands, 30
draw_estimands(), *29*, *31*, *35*, *45, 46*
draw_estimates, 31
draw_estimates(), *29, 30*, *35*, *45, 46*

estimatr::lm_robust(), *11*, *22*
expand_design, 31
expand_design(), *24–26*, *37*, *44, 45*

fabricatr::fabricate(), *9*, *14, 15*, *20*
format.diagnosis, 32
format.diagnosis(), *10*, *24*, *41, 42*, *47*, *49*, *52*

get_diagnosands, 34
get_diagnosands(), *7*, *28*, *33*, *36*, *50*
get_estimates, 35
get_estimates(), *29–31*, *46*
get_simulations, 36
get_simulations(), *7*, *28*, *34*, *50*

insert_step, 36
insert_step(), *24–26*, *32*, *44, 45*

label_estimator, 37
label_estimator(), *38, 39*
label_test *(*label_estimator*)*, 37

model_handler, 38

pop.var, 39
print(), *40*, *51*
print.compared_diagnoses, 39
print.design, 40
print.design_step, 41
print.diagnosis, 41
print.diagnosis(), *32*, *52*
print.objects, 42
print_code *(*compare_designs*)*, 5
progressr::with_progress(), *27*, *50*

randomizr::complete_ra(), *8*
redesign, 43
redesign(), *3*, *17, 18*, *24–26*, *29–32*, *37*, *40*, *42*, *45*, *49*
replace_step, 44
replace_step(), *24–26*, *32*, *37*, *44*
reshape_diagnosis *(*format.diagnosis*)*, 32
rlang::quos(), *54*
run_design, 45
run_design(), *29–31*, *35*, *51*

select_diagnosands, 46
select_diagnosands(), *10*, *24*, *33*, *49*
set_citation, 47
set_citation(), *5*
set_diagnosands, 48
set_diagnosands(), *7*, *10*, *24*, *27*, *33*, *47*
simulate_design, 49

simulate_design(), *7*, *27*, *28*, *34*, *36*
simulate_designs *(*simulate_design*)*, 49
summary(), *40*, *43*
summary.design, 51
summary.diagnosis, 52

tidy(), *33*
tidy.diagnosis, 52
tidy_estimator *(*model_handler*)*, 38
tidy_try, 53
tidy_try(), *11*, *22*, *38*

utils::bibentry(), *47*
utils::person(), *48*

vars, 54